Programming Embedded Systems
with C and GNU Development Tools

Michael Barr and Anthony Massa

O'REILLY

Contents

Foreword

If you mention the word *embedded* to most people, they'll assume you're talking about reporters in a war zone. Few dictionaries—including the canonical *Oxford English Dictionary*—link *embedded* to computer systems. Yet embedded systems underlie nearly all of the electronic devices used today, from cell phones to garage door openers to medical instruments. By now, it's nearly impossible to build anything electronic without adding at least a small microprocessor and associated software.

Vendors produce some nine billion microprocessors every year. Perhaps 100 or 150 million of those go into PCs. That's only about one percent of the units shipped. The other 99 percent go into embedded systems; clearly, this stealth business represents the very fabric of our highly technological society.

And use of these technologies will only increase. Solutions to looming environmental problems will surely rest on the smarter use of resources enabled by embedded systems. One only has to look at the network of 32-bit processors in Toyota's hybrid Prius to get a glimpse of the future.

Though prognostications are difficult, it is absolutely clear that consumers will continue to demand ever-brainier products requiring more microprocessors and huge increases in the corresponding software. Estimates suggest that the firmware content of most products doubles every 10 to 24 months. While the demand for more code is increasing, our productivity rates creep up only slowly. So it's also clear that the industry will need more embedded systems people in order to meet the demand.

What skills will these people need? In the PC world, one must be a competent C/C++ programmer. But embedded developers must have a deep understanding of both the programming languages and the hardware itself; no one can design, code, and test an interrupt service routine, for instance, without knowing where the interrupts come from, how the hardware prioritizes them, the tricks behind servicing that hardware, and machine-level details about saving and preserving the system's context. A firmware developer must have detailed insight into the hardware implementation of his system's peripherals before he can write a single line of driver code.

In the PC world, the magic of the hardware is hidden behind an extensive API. In an embedded system, that API is always written by the engineers that are developing the product.

In this book, Michael Barr and Anthony Massa show how the software and hardware form a synergistic gestalt. They don't shy away from the intricacies of interrupts and I/O, or priority inversion and mutexes.

The authors appropriately demonstrate building embedded systems using a variety of open source tools, including the GNU compiler suite, which is a standard tool widely used in this industry. eCos and Linux, both free/open source products, are used to demonstrate small and large operating systems.

The original version of this book used an x86 target board, which has been replaced in this edition by an ARM-based product. Coincidently, as this volume was in production, Intel made an end-of-life announcement for all of its embedded x86 processors. Readers can be assured that the ARM will be around for a very long time, as it's supported by an enormous infrastructure of vendors.

The hardware is inexpensive and easily available; the software is free. Together they represent the mainstream of embedded systems development. Readers can be sure they'll use these tools in the future.

Buy the development kit, read the book, and execute the examples. You'll get the hands-on experience that employers demand: building and working with real embedded applications.

—Jack Ganssle

Preface

*First figure out why you want the students to learn the
subject and what you want them to know, and the
method will result more or less by common sense.*
—Richard Feynman

Embedded software is in almost every electronic device in use today. There is software hidden away inside our watches, DVD players, mobile phones, antilock brakes, and even a few toasters. The military uses embedded software to guide missiles, detect enemy aircraft, and pilot UAVs. Communication satellites, deep-space probes, and many medical instruments would've been nearly impossible to create without it.

Someone has to write all that software, and there are tens of thousands of electrical engineers, computer scientists, and other professionals who actually do. We are two of them, and we know from our personal experiences just how hard it can be to learn the craft.

Each embedded system is unique, and the hardware is highly specialized to the application domain. As a result, embedded systems programming can be a widely varying experience and can take years to master. However, one common denominator across almost all embedded software development is the use of the C programming language. This book will teach you how to use C in any embedded system.

Even if you already know how to write embedded software, you can still learn a lot from this book. In addition to learning how to use C more effectively, you'll also benefit from the detailed explanations and source code associated with common embedded software problems. Among the advanced topics covered in the book are memory testing and verification, device driver design and implementation, real-time operating system internals, and code optimization techniques.

Why We Wrote This Book

Each year, globally, approximately one new processor is manufactured per person. That's more than six billion new processors each year, fewer than two percent of

which are the Pentiums and PowerPCs at the heart of new personal computers. You may wonder whether there are really that many computers surrounding us. But we bet that within five minutes you can probably spot dozens of products in your own home that contain processors: televisions, stereos, MP3 players, coffee makers, alarm clocks, VCRs, DVD players, microwaves, dishwashers, remote controls, bread machines, digital watches, and so on. And those are just the personal possessions—many more such devices are used at work. The fact that every one of those products contains not only a processor, but also software, is the impetus for this book.

One of the hardest things about this subject is knowing when to stop writing. Each embedded system is unique, and we have therefore learned that there is an exception to every rule. Nevertheless, we have tried to boil the subject down to its essence and present the things that programmers definitely need to know about embedded systems.

Intended Audience

This is a book about programming embedded systems in C. As such, it assumes that the reader already has some programming experience and is at least familiar with the syntax of the C language. It also helps if you have some familiarity with basic data structures, such as linked lists. The book does not assume that you have a great deal of knowledge about computer hardware, but it does expect that you are willing to learn a little bit about hardware along the way. This is, after all, a part of the job of an embedded programmer.

While writing this book, we had two types of readers in mind. The first reader is a beginner—much as we were once. He has a background in computer science or engineering and a few years of programming experience. The beginner is interested in writing embedded software for a living but is not sure just how to get started. After reading the first several chapters, he will be able to put his programming skills to work developing simple embedded programs. The rest of the book will act as a reference for the more advanced topics encountered in the coming months and years of his career.

The second reader is already an embedded systems programmer. She is familiar with embedded hardware and knows how to write software for it but is looking for a reference book that explains key topics. Perhaps the embedded systems programmer has experience only with assembly language programming and is relatively new to C. In that case, the book will teach her how to use the C language effectively in an embedded system, and the later chapters will provide advanced material on real-time operating systems, peripherals, and code optimizations.

Whether you fall into one of these categories or not, we hope this book provides the information you are looking for in a format that is friendly and easily accessible.

Organization

The book contains 14 chapters and 5 appendixes. The chapters can be divided quite nicely into two parts. The first part consists of Chapters 1 through 5 and is intended mainly for newcomers to embedded systems. These chapters should be read in their entirety and in the order that they appear. This will bring you up to speed quickly and introduce you to the basics of embedded software development. After completing Chapter 5, you will be ready to develop small pieces of embedded software on your own.

The second part of the book consists of Chapters 6 through 14 and discusses advanced topics that are of interest to inexperienced and experienced embedded programmers alike. These chapters are mostly self-contained and can be read in any order. In addition, Chapters 6 through 12 contain example programs that might be useful to you on a future embedded software project.

Chapter 1, *Introduction*
> Explains the field of embedded programming and lays out the parameters of the book, including the reference hardware used for examples

Chapter 2, *Getting to Know the Hardware*
> Shows how to explore the documentation for your hardware and represent the components you need to interact with in C

Chapter 3, *Your First Embedded Program*
> Creates a simple blinking light application that illustrates basic principles of embedded programming

Chapter 4, *Compiling, Linking, and Locating*
> Goes over the ways that embedded systems differ from conventional computer systems during program building steps, covering such issues as cross-compilers

Chapter 5, *Downloading and Debugging*
> Introduces the tools you'll need in order to iron out problems in both hardware and software

Chapter 6, *Memory*
> Describes the different types of memory that developers choose for embedded systems and the issues involved in using each type

Chapter 7, *Peripherals*
> Introduces the notion of a device driver, along with other coding techniques for working with devices

Chapter 8, *Interrupts*
> Covers this central area of working with peripherals

Chapter 9, *Putting It All Together*
> Combines the concepts and code from the previous chapter with convenience functions and a main program, to create a loadable, testable application

Chapter 10, *Operating Systems*
> Introduces common operating system concepts, including tasks (or threads) and synchronization mechanisms, along with the reasons for adding a real-time operating system

Chapter 11, *eCos Examples*
> Shows how to use some features of the eCos real-time operating system

Chapter 12, *Embedded Linux Examples*
> Accomplishes the same task as the previous chapter, but for the embedded Linux operating system

Chapter 13, *Extending Functionality*
> Describes options for adding buses, networking, and other communication features to a system

Chapter 14, *Optimization Techniques*
> Describes ways to decrease code size, reduce memory use, and conserve power

Appendix A, *The Arcom VIPER-Lite Development Kit*
> Describes the board used for the examples in this book and how to order one for yourself

Appendix B, *Setting Up Your Software Development Environment*
> Gives instructions for loading the software described in this book on your host Windows or Linux computer

Appendix C, *Building the GNU Software Tools*
> Shows you how to compile the GNU development tools

Appendix D, *Setting Up the eCos Development Environment*
> Shows you how to build an eCos library appropriate for your embedded system so you can compile programs to run on your system

Appendix E, *Setting Up the Embedded Linux Development Environment*
> Describes how to install the embedded Linux tools for your Arcom system and build and run a program on it

Throughout the book, we have tried to strike a balance between specific examples and general information. Whenever possible, we have eliminated minor details in the hope of making the book more readable. You will gain the most from the book if you view the examples, as we do, primarily as tools for understanding important concepts. Try not to get bogged down in the details of any one circuit board or chip. If you understand the general C programming concepts, you should be able to apply them to any embedded system you encounter.

To focus the book's example code on specific concepts, we intentionally left it incomplete—for example, by eliminating certain include files and redundant variable declarations. For complete details about the code, refer to the full example source code on the book's web site.

Conventions, Typographical and Otherwise

The following typographical conventions are used throughout the book:

Italic

> Indicates names of files, programs, methods, and options when they appear in the body of a paragraph. Italic is also used for emphasis and to introduce new terms.

`Constant Width`

> In examples, indicates the contents of files and the output of commands. In regular text, this style indicates keywords, functions, variable names, classes, objects, parameters, and other code snippets.

`Constant Width Bold`

> Indicates commands and options to be typed literally. This style is used in examples only.

`Constant Width Bold Italic`

> Indicates text to be replaced with user values; for example, a filename on your system. This style is used in examples only.
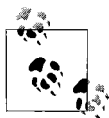
> This symbol is used to indicate a tip, suggestion, or general note.
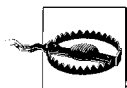
> This symbol is used to indicate a warning.

Other conventions relate to gender and roles. With respect to gender, we have purposefully used both "he" and "she" throughout the book. With respect to roles, we have occasionally distinguished between the tasks of hardware engineers, embedded software engineers, and application programmers. But these titles refer only to roles played by individual engineers, and it should be noted that it can and often does happen that a single individual fills more than one of these roles on an embedded-project team.

Obtaining the Examples Online

This book includes many source code listing, and all but the most trivial snippets are available online. These examples are organized by chapter number and include build instructions (makefiles) to help you recreate each of the executables. The complete archive is available at *http://examples.oreilly.com/embsys2*.

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: "*Programming Embedded Systems with C and GNU Development Tools,* Second Edition, by Michael Barr and Anthony Massa. Copyright 2007 O'Reilly Media, Inc., 978-0-596-00983-0."

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*.

Introduction

I think there is a world market for maybe five computers.
—Thomas Watson, Chairman of IBM, 1943

There is no reason anyone would want a computer in their home.
—Ken Olson, President of Digital Equipment Corporation, 1977

One of the more surprising developments of the last few decades has been the ascendance of computers to a position of prevalence in human affairs. Today there are more computers in our homes and offices than there are people who live and work in them. Yet many of these computers are not recognized as such by their users. In this chapter, we'll explain what embedded systems are and where they are found. We will also introduce the subject of embedded programming and discuss what makes it a unique form of software programming. We'll explain why we have selected C as the language for this book and describe the hardware used in the examples.

What Is an Embedded System?

An *embedded system* is a combination of computer hardware and software—and perhaps additional parts, either mechanical or electronic—designed to perform a dedicated function. A good example is the microwave oven. Almost every household has one, and tens of millions of them are used every day, but very few people realize that a computer processor and software are involved in the preparation of their lunch or dinner.

The design of an embedded system to perform a dedicated function is in direct contrast to that of the personal computer. It too is comprised of computer hardware and software and mechanical components (disk drives, for example). However, a personal computer is not designed to perform a specific function. Rather, it is able to do many different things. Many people use the term *general-purpose computer* to make

this distinction clear. As shipped, a general-purpose computer is a blank slate; the manufacturer does not know what the customer will do with it. One customer may use it for a network file server, another may use it exclusively for playing games, and a third may use it to write the next great American novel.

Frequently, an embedded system is a component within some larger system. For example, modern cars and trucks contain many embedded systems. One embedded system controls the antilock brakes, another monitors and controls the vehicle's emissions, and a third displays information on the dashboard. Some luxury car manufacturers have even touted the number of processors (often more than 60, including one in each headlight) in advertisements. In most cases, automotive embedded systems are connected by a communications network.

It is important to point out that a general-purpose computer interfaces to numerous embedded systems. For example, a typical computer has a keyboard and mouse, each of which is an embedded system. These peripherals each contain a processor and software and is designed to perform a specific function. Another example is a modem, which is designed to send and receive digital data over an analog telephone line; that's all it does. And the specific function of other peripherals can each be summarized in a single sentence as well.

The existence of the processor and software in an embedded system may be unnoticed by a user of the device. Such is the case for a microwave oven, MP3 player, or alarm clock. In some cases, it would even be possible to build a functionally equivalent device that does not contain the processor and software. This could be done by replacing the processor-software combination with a custom *integrated circuit* (*IC*) that performs the same functions in hardware. However, the processor and software combination typically offers more flexibility than a hardwired design. It is generally much easier, cheaper, and less power intensive to use a processor and software in an embedded system.

History and Future

Given the definition of embedded systems presented earlier in this chapter, the first such systems could not possibly have appeared before 1971. That was the year Intel introduced the world's first single-chip microprocessor. This chip, the 4004, was designed for use in a line of business calculators produced by the Japanese company Busicom. In 1969, Busicom asked Intel to design a set of custom integrated circuits, one for each of its new calculator models. The 4004 was Intel's response. Rather than design custom hardware for each calculator, Intel proposed a general-purpose circuit that could be used throughout the entire line of calculators. This general-purpose processor was designed to read and execute a set of instructions—software—stored in an external memory chip. Intel's idea was that the software would give each calculator its unique set of features and that this design style would drive demand for its core business in memory chips.

The microprocessor was an overnight success, and its use increased steadily over the next decade. Early embedded applications included unmanned space probes, computerized traffic lights, and aircraft flight control systems. In the 1980s and 1990s, embedded systems quietly rode the waves of the microcomputer age and brought microprocessors into every part of our personal and professional lives. Most of the electronic devices in our kitchens (bread machines, food processors, and microwave ovens), living rooms (televisions, stereos, and remote controls), and workplaces (fax machines, pagers, laser printers, cash registers, and credit card readers) are embedded systems; over 6 billion new microprocessors are used each year. Less than 2 percent (or about 100 million per year) of these microprocessors are used in general-purpose computers.

It seems inevitable that the number of embedded systems will continue to increase rapidly. Already there are promising new embedded devices that have enormous market potential: light switches and thermostats that are networked together and can be controlled wirelessly by a central computer, intelligent air-bag systems that don't inflate when children or small adults are present, medical monitoring devices that can notify a doctor if a patient's physiological conditions are at critical levels, and dashboard navigation systems that inform you of the best route to your destination under current traffic conditions. Clearly, individuals who possess the skills and the desire to design the next generation of embedded systems will be in demand for quite some time.

Real-Time Systems

One subclass of embedded systems deserves an introduction at this point. A *real-time system* has timing constraints. The function of a real-time system is thus partly specified in terms of its ability to make certain calculations or decisions in a timely manner. These important calculations or activities have deadlines for completion.

The crucial distinction among real-time systems lies in what happens if a deadline is missed. For example, if the real-time system is part of an airplane's flight control system, the lives of the passengers and crew may be endangered by a single missed deadline. However, if instead the system is involved in satellite communication, the damage could be limited to a single corrupt data packet (which may or may not have catastrophic consequences depending on the application and error recovery scheme). The more severe the consequences, the more likely it will be said that the deadline is "hard" and thus, that the system is a *hard real-time system*. Real-time systems at the other end of this continuum are said to have "soft" deadlines—a *soft real-time system*. Figure 1-1 shows some examples of hard and soft real-time systems.

Real-time system design is not simply about speed. Deadlines for real-time systems vary; one deadline might be in a millisecond, while another is an hour away. The main concern for a real-time system is that there is a guarantee that the hard deadlines of the system are always met. In order to accomplish this the system must be predictable.
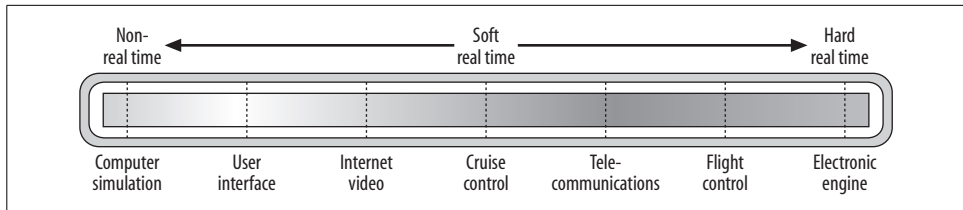
Figure 1-1. A range of example real-time systems

The architecture of the embedded software, and its interaction with the system hardware, play a key role in ensuring that real-time systems meet their deadlines. Key software design issues include whether polling is sufficient or interrupts should be used, and what priorities should be assigned to the various tasks and interrupts. Additional forethought must go into understanding the worst-case performance requirements of the specific system activities.

All of the topics and examples presented in this book are applicable to the designers of real-time systems. The designer of a real-time system must be more diligent in his work. He must guarantee reliable operation of the software and hardware under all possible conditions. And, to the degree that human lives depend upon the system's proper execution, this guarantee must be backed by engineering calculations and descriptive paperwork.

Variations on a Theme

Unlike software designed for general-purpose computers, embedded software cannot usually be run on other embedded systems without significant modification. This is mainly because of the incredible variety of hardware in use in embedded systems. The hardware in each embedded system is tailored specifically to the application, in order to keep system costs low. As a result, unnecessary circuitry is eliminated and hardware resources are shared wherever possible.

In this section, you will learn which hardware features are common across all embedded systems and why there is so much variation with respect to just about everything else. Later in the book, we will look at some techniques that can be used to minimize the impact of software changes so they are not needed throughout all layers of the software.

Common System Components

By definition, all embedded systems contain a processor and software, but what other features do they have in common? Certainly, in order to have software, there must be a place to store the executable code and temporary storage for runtime data manipulation. These take the form of *read-only memory* (ROM) and *random access memory* (RAM), respectively; most embedded systems have some of each. If only a small

amount of memory is required, it might be contained within the same chip as the processor. Otherwise, one or both types of memory reside in external memory chips.

All embedded systems also contain some type of inputs and outputs. For example, in a microwave oven, the inputs are the buttons on the front panel and a temperature probe, and the outputs are the human-readable display and the microwave radiation. The outputs of the embedded system are almost always a function of its inputs and several other factors (elapsed time, current temperature, etc.). The inputs to the system usually take the form of sensors and probes, communication signals, or control knobs and buttons. The outputs are typically displays, communication signals, or changes to the physical world. See Figure 1-2 for a general example of an embedded system.
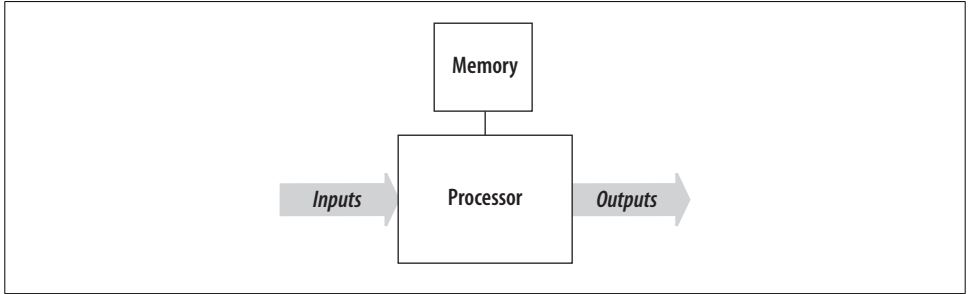
Figure 1-2. A generic embedded system

With the exception of these few common features, the rest of the embedded hardware is usually unique and, therefore, requires unique software. This variation is the result of many competing design criteria.

The software for the generic embedded system shown in Figure 1-2 varies depending on the functionality needed. The hardware is the blank canvas, and the software is the paint that we add in order to make the picture come to life. Figure 1-3 gives just a couple of possible high-level diagrams that could be implemented on such a generic embedded system.

Both the basic embedded software diagram in Figure 1-3(a) and the more complex embedded software diagram in Figure 1-3(b) contain very similar blocks. The hardware block is common in both diagrams.

The device drivers are embedded software modules that contain the functionality to operate the individual hardware devices. The reason for the device driver software is to remove the need for the application to know how to control each piece of hardware. Each individual device driver would typically need to know only how to control its hardware device. For instance, for a microwave oven, separate device drivers control the keypad, display, temperature probe, and radiation control.

If more functionality is required, it is sometimes necessary to include additional layers in the embedded software to assist with this added functionality. In this example,

Figure 1-3. (a) Basic embedded software diagram and (b) a more complex embedded software diagram
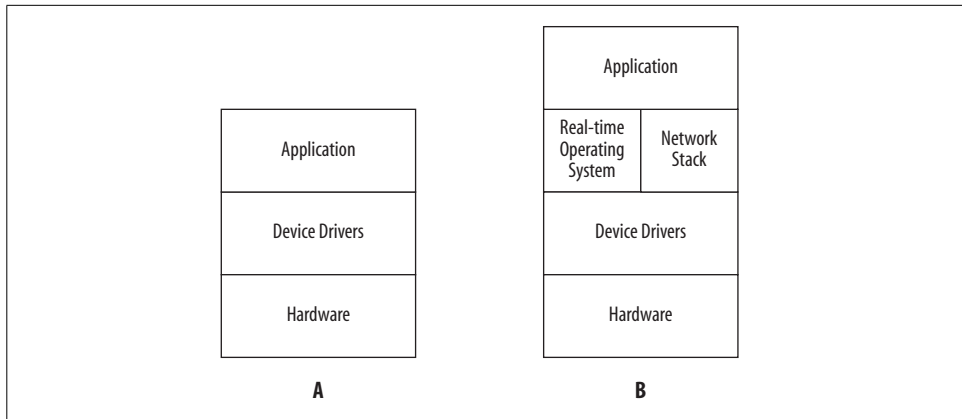
the complex diagram includes a *real-time operating system* (*RTOS*) and a networking stack. The RTOS can help the programmer separate the application's functionality into distinct tasks for better organization of the application software and a more responsive system. We will investigate the use of an RTOS later in this book. The network stack also adds to the functionality of the basic embedded system; a microwave oven might use it to pop up a message on your desktop computer when your lunch is ready.

The responsibilities of the application software layer is the same in both the basic and the complex embedded software diagrams. In a microwave oven, the application processes the different inputs and controls the outputs based on what the user commands it to do.

You'll notice that the software in Figure 1-3 is represented by discrete blocks stacked on top of one another with fixed borders. This is done deliberately, to indicate the separation of the different software functional layers that make up the complete embedded software system. Later, we will break down these blocks further to show you how you can keep your embedded software clean, easy to read, and portable. Keeping these software layers distinct, with well-defined methods that neighboring layers can use to communicate, helps you write good embedded software.

Requirements That Affect Design Choices

Each embedded system must meet a completely different set of requirements, any or all of which can affect the compromises and trade-offs made during the development of the product. For example, if the system must have a production cost of less than $10, other desirable traits—such as processing power and system reliability—might need to be sacrificed in order to meet that goal.

Of course, production cost is only one of the possible constraints under which embedded hardware designers work. Other common design requirements include:

Processing power

The workload that the main chip can handle. A common way to compare processing power is the *millions of instructions per second* (MIPS) rating. If two otherwise similar processors have ratings of 25 MIPS and 40 MIPS, the latter is said to be the more powerful. However, other important features of the processor need to be considered. One is the register width, which typically ranges from 8 to 64 bits. Today's general-purpose computers use 32- and 64-bit processors exclusively, but embedded systems are still mainly built with less costly 4-, 8-, and 16-bit processors.

Memory

The amount of memory (ROM and RAM) required to hold the executable software and the data it manipulates. Here the hardware designer must usually make his best estimate up front and be prepared to increase or decrease the actual amount as the software is being developed. The amount of memory required can also affect the processor selection. In general, the register width of a processor establishes the upper limit of the amount of memory it can access (e.g., a 16-bit address register can address only 64 KB (2^{16}) memory locations).[*]

Number of units

The expected production run. The trade-off between production cost and development cost is affected most by the number of units expected to be produced and sold. For example, it rarely makes sense to develop custom hardware components for a low-volume product.

Power consumption

The amount of power used during operation. This is extremely important, especially for battery-powered portable devices. A common metric used to compare the power requirements of portable devices is mW/MIPS (milliwatts per MIPS); the greater this value, the more power is required to get work done. Lower power consumption can also lead to other favorable device characteristics, such as less heat, smaller batteries, less weight, smaller size, and simpler mechanical design.

Development cost

The cost of the hardware and software design processes, known as *nonrecurring engineering* (NRE). This is a fixed, one-time cost, so on some projects, money is no object (usually for high-volume products), whereas on other projects, this is the only accurate measure of system cost (for the production of a small number of units).

---

[*] The narrower the register width, the more likely it is that the processor employs tricks such as multiple address spaces to support more memory. There are still embedded systems that do the job with a few hundred bytes. However, several thousand bytes is a more likely minimum, even on an 8-bit processor.

Lifetime

   How long the product is expected to stay in use. The required or expected lifetime affects all sorts of design decisions, from the selection of hardware components to how much system development and production is allowed to cost. How long must the system continue to function (on average)? A month, a year, or a decade?

Reliability

   How reliable the final product must be. If it is a children's toy, it may not have to work properly 100 percent of the time, but if it's an antilock braking system for a car, it had sure better do what it is supposed to do each and every time.

In addition to these general requirements, each system has detailed functional requirements. These are the things that give the embedded system its unique identity as a microwave oven, pacemaker, or pager.

Table 1-1 illustrates the range of typical values for each of the previous design requirements. The "low," "medium," and "high" labels are meant for illustration purposes and should not be taken as strict deliniations. An actual product has one selection from each row. In some cases, two or more of the criteria are linked. For example, increases in required processing power could lead to increased production costs. Conversely, we might imagine that the same increase in processing power would have the effect of decreasing the development costs—by reducing the complexity of the hardware and software design. So the values in a particular column do not necessarily go together.

Table 1-1. Common design requirements for embedded systems

Criterion	Low	Medium	High
Processor	4- or 8-bit	16-bit	32- or 64-bit
Memory	< 64 KB	64 KB to 1 MB	> 1 MB
Development cost	< $100,000	$100,000 to $1,000,000	> $1,000,000
Production cost	< $10	$10 to $1,000	> $1,000
Number of units	< 100	100 to 10,000	> 10,000
Power consumption	> 10 mW/MIPS	1 to 10 mW/MIPS	< 1 mW/MIPS
Lifetime	Days, weeks, or months	Years	Decades
Reliability	May occasionally fail	Must work reliably	Must be fail-proof

Embedded Design Examples

To demonstrate the variation in design requirements from one embedded system to the next, as well as the possible effects of these requirements on the hardware, we will now take some time to describe three embedded systems in some detail. Our goal is to put you in the system designer's shoes for a few moments before narrowing our discussion to embedded software development.

Digital Watch

At the current peak of the evolutionary path that began with sundials, water clocks, and hourglasses is the digital watch. Among its many features are the presentation of the date and time (usually to the nearest second), the measurement of the length of an event to the nearest hundredth of a second, and the generation of an annoying little sound at the beginning of each hour. As it turns out, these are very simple tasks that do not require very much processing power or memory. In fact, the only reason to employ a processor at all is to support a range of models and features from a single hardware design.

The typical digital watch contains a simple, inexpensive 4-bit processor. Because processors with such small registers cannot address very much memory, this type of processor usually contains its own on-chip ROM. And, if there are sufficient registers available, this application may not require any RAM at all. In fact, all of the electronics— processor, memory, counters, and real-time clocks—are likely to be stored in a single chip. The only other hardware elements of the watch are the inputs (buttons) and outputs (display and speaker).

A digital watch designer's goal is to create a reasonably reliable product that has an extraordinarily low production cost. If, after production, some watches are found to keep more reliable time than most, they can be sold under a brand name with a higher markup. For the rest, a profit can still be made by selling the watch through a discount sales channel. For lower-cost versions, the stopwatch buttons or speaker could be eliminated. This would limit the functionality of the watch but might require few or even no software changes. And, of course, the cost of all this development effort may be fairly high, because it will be amortized over hundreds of thousands or even millions of watch sales.

In the case of the digital watch, we see that software, especially when carefully designed, allows enormous flexibility in response to a rapidly changing and highly competitive market.

Video Game Player

When you pull the Sony PlayStation 2 out from your entertainment center, you are preparing to use an embedded system. In some cases, these machines are more powerful than personal computers of the same generation. Yet video game players for the home market are relatively inexpensive compared with personal computers. It is the competing requirements of high processing power and low production cost that keep video game designers awake at night.

The companies that produce video game players don't usually care how much it costs to develop the system as long as the production costs of the resulting product are low—typically around a hundred dollars. They might even encourage their engineers to design custom processors at a development cost of millions of dollars each.

So, although there might be a 64-bit processor inside your video game player, it is probably not the same processor that would be found in a general-purpose computer. In all likelihood, the processor is highly specialized for the demands of the video games it is intended to play.

Because production cost is so crucial in the home video game market, the designers also use tricks to shift the costs around. For example, one tactic is to move as much of the memory and other peripheral electronics as possible off of the main circuit board and onto the game cartridges.* This helps to reduce the cost of the game player but increases the price of every game. So, while the system might have a powerful 64-bit processor, it might have only a few megabytes of memory on the main circuit board. This is just enough memory to bootstrap the machine to a state from which it can access additional memory on the game cartridge.

We can see from the case of the video game player that in high-volume products, a lot of development effort can be sunk into fine-tuning every aspect of a product.

Mars Rover

In 1976, two unmanned spacecrafts arrived on the planet Mars. As part of their mission, they were to collect samples of the Martian surface, analyze the chemical makeup of each, and transmit the results to scientists back on Earth. Those Viking missions were amazing. Surrounded by personal computers that must be rebooted occasionally, we might find it remarkable that more than 30 years ago, a team of scientists and engineers successfully built two computers that survived a journey of 34 million miles and functioned correctly for half a decade. Clearly, reliability was one of the most important requirements for these systems.

What if a memory chip had failed? Or the software had contained bugs that had caused it to crash? Or an electrical connection had broken during impact? There is no way to prevent such problems from occurring, and on other space missions, these problems have proved ruinous. So, all of these potential failure points and many others had to be eliminated by adding redundant circuitry or extra functionality: an extra processor here, special memory diagnostics there, a hardware timer to reset the system if the software got stuck, and so on.

More recently, NASA launched the Pathfinder mission. Its primary goal was to demonstrate the feasibility of getting to Mars on a budget. Of course, given the advances in technology made since the mid-70s, the designers didn't have to give up too much to accomplish this. They might have reduced the amount of redundancy somewhat, but they still gave Pathfinder more processing power and memory than Viking. The Mars Pathfinder was actually two embedded systems: a landing craft and a rover. The landing craft had a 32-bit processor and 128 MB of RAM; the rover, on the other

---

* For example, Atari and Nintendo have designed some of their systems this way.

hand, had only an 8-bit processor and 512 KB of RAM. These choices reflect the different functional requirements of the two systems. Production cost probably wasn't much of an issue in either case; any investment would have been worth an improved likelihood of success.

Life As an Embedded Software Developer

Let's now take a brief look at some of the qualities of embedded software that set embedded developers apart from other types of software developers. An embedded software developer is the one who gets her hands dirty by getting down close to the hardware.

Embedded software development, in most cases, requires close interaction with the physical world—the hardware platform. We say "in most cases" because there are very large embedded systems that require individuals to work solely on the application-layer software for the system. These application developers typically do not have any interaction with the hardware. When designed properly, the hardware device drivers are abstracted away from the actual hardware so that a developer writing software at the application level doesn't know how a string gets output to the display, just that it happens when a particular routine is called with the proper parameters.

Hardware knowledge

The embedded software developer must become intimately familiar with the integrated circuits, the boards and buses, and the attached devices used in order to write solid embedded software (also called *firmware*). Embedded developers shouldn't be afraid to dive into the schematics, grab an oscilloscope probe, and start poking around the circuit to find out what is going on.

Efficient code

Because embedded systems are typically designed with the least powerful and most cost-effective processor that meets the performance requirements of the system, embedded software developers must make every line of code count. The ability to write efficient code is a great quality to possess as a firmware developer.

Peripheral interfaces

At the lowest level, firmware is very specialized, because each component or circuit has its own activity to perform and, furthermore, its own way of performing that activity. Embedded developers need to know how to communicate with the different devices or *peripherals* in order to have full control of the devices in the system. Reacting to stimuli from external peripherals is a large part of embedded software development.

For example, in one microwave oven, the firmware might get the data from a temperature sensor by reading an 8-bit register in an external analog-to-digital converter; in another system, the data might be extracted by controlling a serial bus that interfaces to the external sensor circuit via a single wire.

Robust code

There are expectations that embedded systems will run for years in most cases. This is not a typical requirement for software applications written for a PC or Mac. Now, there are exceptions. However, if you had to keep unplugging your microwave in order to get it to heat up your lunch for the proper amount of time, it would probably be the last time you purchased a product from that company.

Minimal resources

Along the same lines of creating a more robust system, another large differentiator between embedded software and other types of software is resource constraints. The rules for writing firmware are different from the rules for writing software for a PC. Take memory allocation, for instance. An application for a modern PC can take for granted that it will have access to practically limitless resources. But in an embedded system, you will run out of memory if you do not plan ahead and design the software properly.

An embedded software developer must closely manage resources, from memory to processing power, so that the system operates up to specification and so failures don't occur. For example, using standard dynamic memory allocation functions can cause fragmentation, and eventually the system may cease to operate. This requires a reboot since you have no place to store incoming data.

Quite often, in embedded software, a developer will allocate all memory needed by the system at initialization time. This is safer than using dynamic memory allocation, though it cannot always be done.

Reusable software

As we mentioned before, *code portability* or *code reuse*—writing software so that it can be moved from hardware platform to hardware platform—is very useful to aid transition to new projects. This cannot always be done; we have seen how individual each embedded system is. Throughout this book, we will look at basic methods to ensure that your embedded code can be moved more easily from project to project. So if your next project uses an LCD for which you've previously developed a driver, you can drop in the old code and save some precious time in the schedule.

Development tools

The tools you will use throughout your career as an embedded developer will vary from company to company and often from project to project. This means you will need to learn new tools as you continue in your career. Typically, these tools are not as powerful or as easy to use as those used in PC software development.

The debugging tools you might come across could vary from a simple LED to a full-blown *in-circuit emulator* (ICE). This requires you, as the firmware developer, and the one responsible for debugging your code, to be very resourceful and have a bag of techniques you can call upon when the debug environment is lacking. Throughout the book, we will present different "low-level software tools" you can implement with little impact on the hardware design.

These are just a few qualities that separate embedded software developers from the rest of the pack. We will investigate these and other techniques that are specific to embedded software development as we continue.

The C Language: The Lowest Common Denominator

One of the few constants across most embedded systems is the use of the C programming language. More than any other, C has become the language of embedded programmers. This has not always been the case, and it will not continue to be so forever. However, at this time, C is the closest thing there is to a standard in the embedded world. In this section, we'll explain why C has become so popular and why we have chosen it as the primary language of this book.

Because successful software development so frequently depends on selecting the best language for a given project, it is surprising to find that one language has proven itself appropriate for both 8-bit and 64-bit processors; in systems with bytes, kilobytes, and megabytes of memory; and for development teams that range from one to a dozen or more people. Yet this is precisely the range of projects in which C has thrived.

The C programming language has plenty of advantages. It is small and fairly simple to learn, compilers are available for almost every processor in use today, and there is a very large body of experienced C programmers. In addition, C has the benefit of processor-independence, which allows programmers to concentrate on algorithms and applications rather than on the details of a particular processor architecture. However, many of these advantages apply equally to other high-level languages. So why has C succeeded where so many other languages have largely failed?

Perhaps the greatest strength of C—and the thing that sets it apart from languages such as Pascal and FORTRAN—is that it is a very "low-level" high-level language. As we shall see throughout the book, C gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages. The "low-level" nature of C was a clear intention of the language's creators. In fact, Brian W. Kernighan and Dennis M. Ritchie included the following comment in the opening pages of their book *The C Programming Language* (Prentice Hall):

> C is a relatively "low level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

Few popular high-level languages can compete with C in the production of compact, efficient code for almost all processors. And, of these, only C allows programmers to interact with the underlying hardware so easily.

Other Embedded Languages

Of course, C is not the only language used by embedded programmers. At least four other languages—assembly, C++, Forth, and Ada—are worth mentioning in greater detail.

In the early days, embedded software was written exclusively in the assembly language of the target processor. This gave programmers complete control of the processor and other hardware, but at a price. Assembly languages have many disadvantages, not the least of which are higher software development costs and a lack of code portability. In addition, finding skilled assembly programmers has become much more difficult in recent years. Assembly is now used primarily as an adjunct to the high-level language, usually only for startup system code or those small pieces of code that must be extremely efficient or ultra-compact, or cannot be written in any other way.

Forth is efficient but extremely low-level and unusual; learning to get work done with it takes more time than with C.

C++ is an object-oriented superset of C that is increasingly popular among embedded programmers. All of the core language features are the same as C, but C++ adds new functionality for better data abstraction and a more object-oriented style of programming. These new features are very helpful to software developers, but some of them reduce the efficiency of the executable program. So C++ tends to be most popular with large development teams, where the benefits to developers outweigh the loss of program efficiency.

Ada is also an object-oriented language, though substantially different from C++. Ada was originally designed by the U.S. Department of Defense for the development of mission-critical military software. Despite being twice accepted as an international standard (Ada83 and Ada95), it has not gained much of a foothold outside of the defense and aerospace industries. And it has been losing ground there in recent years. This is unfortunate because the Ada language has many features that would simplify embedded software development if used instead of C or C++.

Choosing a Language for the Book

A major question facing the authors of a book such as this one is which programming language or languages to discuss. Attempting to cover too many languages might confuse the reader or detract from more important points. On the other hand, focusing too narrowly could make the discussion unnecessarily academic or (worse for the authors and publisher) limit the potential market for the book.

Certainly, C must be the centerpiece of any book about embedded programming, and this book is no exception. All of the sample code is written in C, and the discussion will focus on C-related programming issues. Of course, everything that is said

about C programming applies equally to C++. We will use assembly language only when a particular programming task cannot be accomplished in any other way.

We feel that this focus on C with a brief introduction to assembly most accurately reflects the way embedded software is actually developed today and the way it will continue to be developed in the near term. This is why examples in this edition do not use C++. We hope that this choice will keep the discussion clear, provide information that is useful to people developing actual systems, and include as large a potential audience as possible. However, we do cover the impact of C++ on embedded software in Chapter 14.

Consistent Coding Practices

Whatever language is selected for a given project, it is important to institute some basic coding guidelines or styles to be followed by all developers on a project. Coding guidelines can make reading code easier, both for you and for the next developer that has to inherit your code. Understanding exactly what a particular software routine is doing is difficult enough without having to fight through several changes in coding style that emerged because a number of different developers touched the same routine over the years, each leaving his own unique mark. Stylistic issues, such as how variables are named or where the curly brace should reside, can be very personal to some developers.

There are a number of decent coding standards floating around on the Internet. One standard we like is located online at *http://www.ganssle.com* and was developed by Jack Ganssle. Another that we like, by Miro Samek, is located online at *http://www. quantum-leaps.com*.

These standards give you guidelines on everything from directory structures to variable names and are a great starting point; you can incorporate into them the styles that you find necessary and helpful. If a coding standard for the entire team is not something you can sell your company on, use one yourself and stick to it.

A Few Words About Hardware

It is the nature of programming that books about the subject must include examples. Typically, these examples are selected so that interested readers can easily experiment with them. That means readers must have access to the very same software development tools and hardware platforms used by the authors. Unfortunately, it does not make sense to run any of the example programs on the platforms available to most readers—PCs, Macs, and Unix workstations.

Fixed Width Integers: Sometimes Size Matters

Computer programmers don't always care how wide an integer is when held by the processor. For example, when we write:

```
int i;

for (i = 0; i < N; i++)
{
    ...
}
```

we generally expect our compiler to generate the most efficient code possible, whether that makes the loop counter an 8-, 16-, 32-, or even 64-bit quantity.

As long as the integer is wide enough to hold the maximum value (N, in the example just shown), we want the processor to be used in the most efficient way. And that's precisely what the ISO C and C++ standards tell the compiler writer to do: choose the most efficient integer size that will fulfill the specific request. Because of the variable size of integers on different processors and the corresponding flexibility of the language standards, the previous code may result in a 32-bit integer with one compiler but a 16-bit integer with another—possibly even when the very same processor is targeted.

But in many other programming situations, integer size matters. Embedded programming, in particular, often involves considerable manipulation of integer data of fixed widths.

In hindsight, it sure would've been nice if the authors of the C standard had defined some standard names and made compiler providers responsible for providing the appropriate typedef for each fixed-size integer type in a library header file. Alternatively, the C standard could have specified that each of the types short, int, and long has a standard width on all platforms; but that might have had an impact on performance, particularly on 8-bit processors that must implement 16- and 32-bit additions in multi-instruction sequences.

Interestingly, it turns out the 1999 update to the International Organization for Standardization's (ISO) C standard (also referred to as *C99*) did just that. The ISO has finally put the weight of its standard behind a preferred set of names for signed and unsigned fixed-size integer data types. The newly defined type names are:

8-bit: int8_t, uint8_t
16-bit: int16_t, uint16_t
32-bit: int32_t, uint32_t
64-bit: int64_t, uint64_t

According to the updated standard, this required set of typedefs (along with some others) is to be defined by compiler vendors and included in the new header file *stdint.h*.

If you're already using a C99-compliant compiler, this new language feature makes that declaration of a fixed-width integer variable or a register as straightforward as using one of the new type names.

Even if you don't have an updated compiler, the inclusion of these names in the C99 standard suggests that it's time to update your coding standards and practices. Love them or hate them, at least these new names are part of an accepted international standard. As a direct result, it will be far easier in the future to port C programs that require fixed-width integers to other compilers and target platforms. In addition, modules that are reused or sold with source can be more easily understood when they conform to standard naming and typing conventions such as those in C99.

If you don't have a C99-compliant compiler yet, you'll have to write your own set of typedefs, using compiler-specific knowledge of the char, short, and long primitive widths.

For the examples in this book, we use the C99 style for variable types that require specific widths. We have generated our own *stdint.h* that is specific to the *gcc* variant targeting the ARM XScale processor. Our file may not work in other build environments.

Even selecting a standard embedded platform is difficult. As you have already learned, there is no such thing as a "typical" embedded system. Whatever hardware is selected, the majority of readers will not have access to it. But despite this rather significant problem, we do feel it is important to select a reference hardware platform for use in the examples. In so doing, we hope to make the examples consistent and, thus, the entire discussion more clear—whether you have the chosen hardware in front of you or not.

In choosing an example platform, our first criterion was that the platform had to have a mix of peripherals to support numerous examples in the book. In addition, we sought a platform that would allow readers to carry on their study of embedded software development by expanding on our examples with more advanced projects. Another criterion was to find a development board that supported the GNU software development tools; with their open source licensing and coverage on a wide variety of embedded processors, the GNU development tools were an ideal choice.

The chosen hardware consists of a 32-bit processor (the XScale ARM),[*] a hefty amount of memory (64 MB of RAM and 16 MB of ROM), and some common types of inputs, outputs, and peripheral components. The board we've chosen is called the VIPER-Lite and is manufactured and sold by Arcom. A picture of the Arcom VIPER-Lite development board (along with the add-on module and other supporting hardware) is shown in Figure 1-4. Additional information about the Arcom board and instructions for obtaining one can be found in Appendix A.

---

[*] The processor on the VIPER-Lite board is the PXA255 XScale processor, which is based on the ARM v.5TE architecture. The XScale processor was developed by an Intel Corporation embedded systems division that was sold to Marvell Technology Group in July 2006.
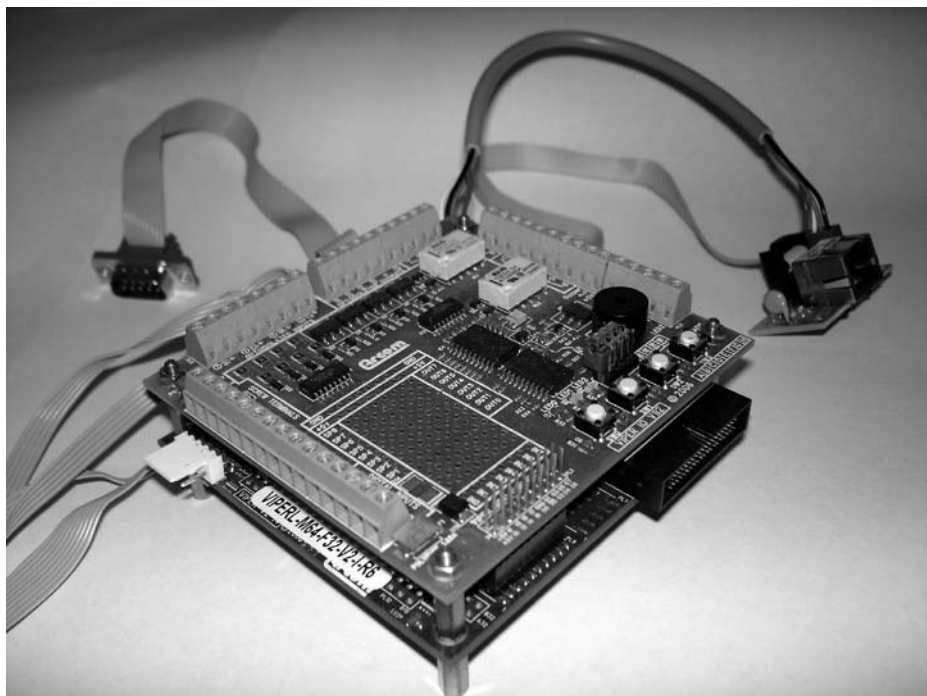
Figure 1-4. The Arcom VIPER-Lite development boards

If you have access to the reference hardware, you will be able to work through the examples in the book as they are presented. Otherwise, you will need to port the example code to an embedded platform that you do have access to. Toward that end, we have made every effort to make the example programs as portable as possible. However, the reader should bear in mind that the hardware is different in each embedded system and that some of the examples might be meaningless on hardware different from the hardware we have chosen here. For example, it wouldn't make sense to port our flash memory driver to a board that had no flash memory devices.

Although we will get into some basic details about hardware, the main focus of this book is embedded software. We recommend that you take a look at *Designing Embedded Systems* by John Catsoulis (O'Reilly). John has an extensive background on the subject and does a wonderful job presenting often difficult material in a very understandable way. It makes a great companion for this book.

Getting to Know the Hardware

hard·ware *n. The part of a computer system that can be kicked.*

As an embedded software engineer, you'll have the opportunity (and challenge) to work with many different pieces of hardware in your career. In this chapter, we will begin by taking a look at the basics in understanding a schematic. We will also teach you a simple procedure that we use to familiarize ourselves with any new board. In the process, we'll guide you through the creation of a C-language header file that describes the board's most important features and a piece of software that initializes the hardware to a known state.

Understanding the Big Picture

Before writing software for an embedded system, you must first be familiar with the hardware on which it will run. At first, you just need to understand the general operation of the system, such as what the board's main function is and what the inputs and outputs are. Initially, you do not need to understand every little detail of the hardware—how every component or peripheral operates and what registers need to be programmed for particular functions.

Whenever you receive a new board, you should take some time to read the main documents provided with it. If the board is an off-the-shelf product, it might arrive with a "User's Guide" or "Programmer's Manual" that has been written for software developers. (The Arcom development kit, for example, includes this information as well as datasheets for all major components on the board.) However, if the board was custom designed for your project, the documentation might be more cryptic or may have been written mainly for the reference of the hardware designers. Either way, this is the single best place to start.

While you are reading the documentation, set the board itself aside. This will help you to focus on the big picture. There will be plenty of time to examine the actual

board more closely when you have finished reading. Before picking up the board, you should be able to answer two basic questions about it:

- What is the overall purpose of the board?
- How does data flow through it?

For example, imagine that you are a software developer on a design team building a print server. You have just received an early prototype board from the hardware designers. The purpose of the board is to share a printer among several computers. The hardware reads data from a network connection and sends that data to a printer for output. The print server must mediate between the computers and decide which computer from the network gets to send data to the printer. Status information also flows in the opposite direction to the computers on the network.

Though the purpose of most systems is self-explanatory, the flow of the data might not be. We often find that a block diagram is helpful in achieving rapid comprehension. If you are lucky, the documentation provided with your hardware will contain a block diagram. However, you might also find it useful to create your own block diagram. That way, you can leave out hardware components that are unrelated to the basic flow of data through the system.

In the case of the Arcom board, the hardware was designed for demonstration purposes rather than with one specific application in mind. However, we'll imagine that it has a purpose. The user of the device connects the computers to the Ethernet port and a printer to the parallel port. Any computer on the network can then send documents to the printer, though only one of them can do so at a given time.

The diagram in Figure 2-1 illustrates the flow of data through the print server. (Only those hardware devices involved in this application of the Arcom board are shown.) By looking at the block diagram, you should be able to quickly visualize the flow of the data through the system. Data to be printed is accepted from the Ethernet controller, held in RAM until the printer is ready for more data, and delivered to the printer via the parallel port. Status information is fed back to the various computers requesting output on the printer. The software that makes all of this happen is stored in ROM. Note that the PC/104 bus includes buffered signals of the address and data buses in addition to other signals.

In order to get a better idea of how the block diagram relates to the actual hardware on the Arcom board for our print server device, examine Figure 2-2, which shows the diagram overlaid on top of the Arcom board. This figure gives you a better idea of the ICs involved in the print server device and how the data is routed through the actual hardware.

We recommend creating a project notebook or binder. Once you've created a block diagram, place it as the first page in your project notebook. You need it handy so you can refer to it throughout the project. As you continue working with this piece of hardware, write down everything you learn about it in your notebook. If you get a
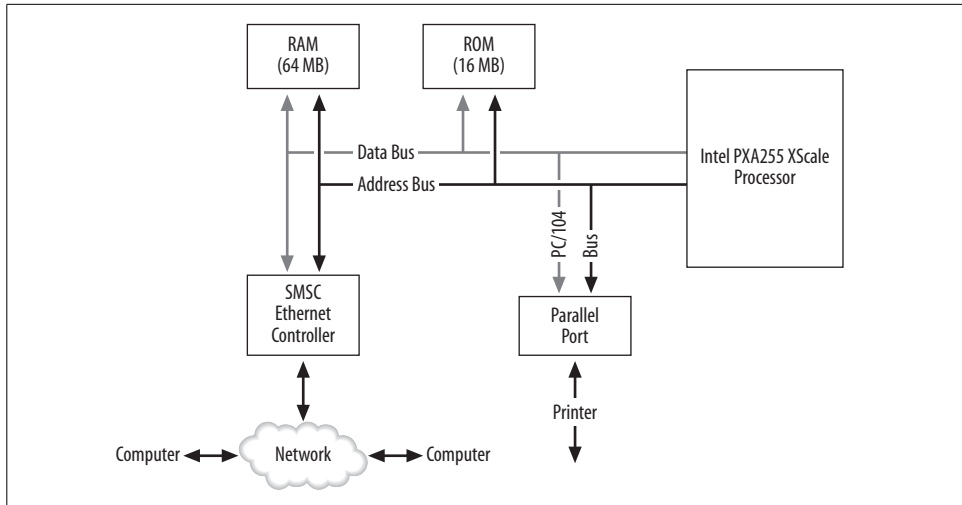
Figure 2-1. Block diagram for the print server

useful handout at a meeting, put it into your notebook. Put tabs in there so you can quickly jump to important information that you refer to all the time. You might also want to keep notes about the software design and implementation. It is very useful to refer back to your notes to refresh your memory about why a particular decision was made for the software. A project notebook is valuable not only while you are developing the software, but also once the project is complete. You will appreciate the extra effort you put into keeping a notebook in case you need to make changes to your software, or work with similar hardware, months or years later.

If you still have any big-picture questions after reading the hardware documents, ask a hardware engineer for some help. If you don't already know the hardware's designer, take a few minutes to introduce yourself. If you have some time, take him out to lunch, or buy him a beer after work. (You don't even have to talk about the project the whole time!) We have found that many software engineers have difficulty communicating with hardware engineers, and vice versa. In embedded systems development, it is especially important that the hardware and software teams be able to communicate with one another. This chapter will give you a solid foundation so that you can speak the language of the hardware engineer.

Before moving on, let's take a brief detour to better understand the basics of hardware and schematics.

Hardware Basics

The next step in understanding the hardware is to take a look at the *schematic*. A schematic is a drawing comprised of standardized symbols to represent all of a circuit's components and connections. The schematic gives the details of the hardware,
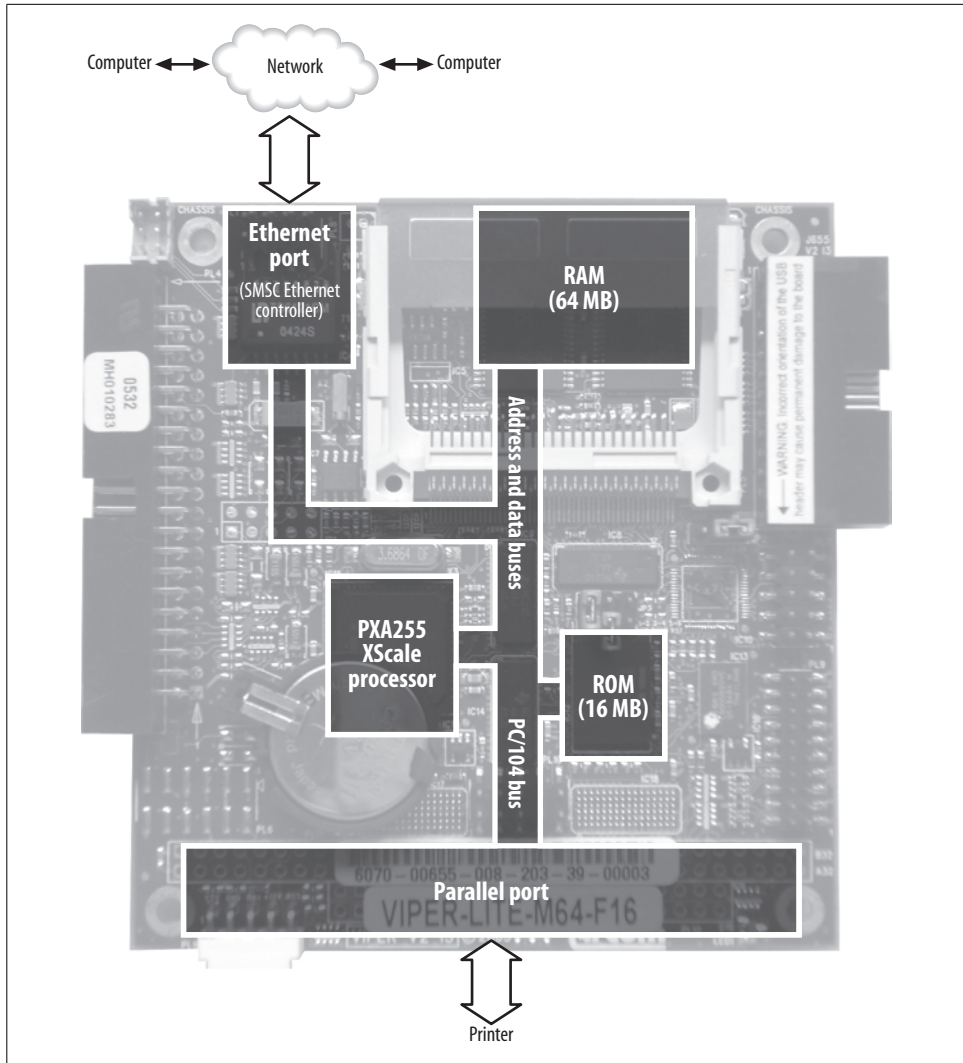
Figure 2-2. Block diagram for the print server on Arcom board

showing the individual components represented in the block diagram, how the components are interconnected, and, most importantly, where to put the oscilloscope probe to see what's going on with a particular circuit. On most projects, it is not necessary for you to understand how every electrical circuit on the board operates, but you do need to understand the basic operation of the hardware.

Along with the user's guides or manuals for the board, it is also helpful to collect the *datasheets* for all major components on your board. The datasheet is a complete specification of a particular hardware component, including electrical, timing, and interface parameters.

Often the hardware engineer has already collected the datasheets; if so, your work is partly complete. You might want to take a look at the other information available for a particular component, because there are often separate hardware and software documents, especially for more complex devices. For example, a processor often includes a Programmer's Guide in addition to the other literature. These documents can give you valuable information for using various features of the processor; they occationally even provide code snippets.

There are also application notes that address particular issues associated with a specific component. It is a good idea to look for any *errata* documents for all devices. The device's errata will give you a heads-up on any issues regarding the way a device operates, and, more importantly, workarounds for these issues.

It's a good idea to periodically check for updates of the board components' information. This will save you the frustration of chasing a problem that was fixed in the latest datasheet. All of this information is an asset when you are trying to understand a new piece of hardware. You can generally find these documents on the manufacturer's web site.

Schematic Fundamentals

Before we take a look at a schematic, let's go over some of the basics of schematics. Figure 2-3 shows some of the basic schematic symbols you will come across, although there might be some variations in symbols from schematic to schematic. The first column gives the name of the particular component; the second column shows the reference designator prefix or component name; and the third column shows the symbols for the related component.

You may notice that two symbols are shown for the diode component. The symbol on the right is for a light emitting diode (LED), which we will take a look at shortly.

The symbols for ground and power can also vary from schematic to schematic; two symbols for power and ground are included in Figure 2-3. In addition to VCC, the reference designator commonly used for power is VDD. Since many circuits use multiple voltage levels, you may also come across power symbols that are labeled with the actual voltage, such as +5 or +3.3, instead of VCC or VDD. The power and ground symbols are typically placed vertically, as shown, whereas the other symbols in Figure 2-3 might show up in any orientation.

A *reference designator* is a combination of letters, numbers, or both, which are used to identify components on a schematic. Reference designators typically include a number to aid in identifying a specific part in the schematic. For example, three resistors in a schematic might have reference designators R4, R21, and R428. The reference designators are normally silkscreened (a painted overlay) on the circuit board for part identification.
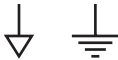
Component	Reference Designator Prefix	Symbol
Resistor	R	
Capacitor	C	
Diode	D	
Crystal	X	
Inductor	L	
Power	VCC	
Ground	GND	

Figure 2-3. Basic schematic symbols

Along with the reference designators, certain components (such as capacitors, inductors, and resistors) are marked by their associated values. For example, in Figure 2-4, resistor R39 has a value of 680 Ω.
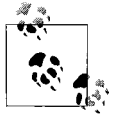
> The values for some components on a schematic are written in a way to aid clarification. For example, a resistor with a value of 4.7 kΩ has its value written as 4K7. A resistor with a value of 12.4 Ω is written as 12R4. Using this method, it is easier to understand the value of the component should the decimal fail to print properly.

You will also notice that integrated circuit symbols are not included in this figure. That is because IC schematic representations vary widely. Typically, a hardware engineer needs to create his own schematic symbol for the ICs used in the design. It is then up to the hardware engineer to use the clearest method possible to capture and represent the IC's symbol.

The reference designator for ICs can vary as well. Typically, the letter U is used. The Arcom board schematic, however, uses the reference designator IC.

IC symbols also include a *component type* or part number used by the manufacturer. The component type is often helpful when you need to hunt for the datasheets for the parts of a particular design. Descriptive text might save you the trouble of deciphering the markings and codes on the top of a specific IC.

Now that we have an introduction to the symbols used in a schematic, let's take a look at a schematic. Figure 2-4 is a partial schematic of a fictional board. In this figure, we show some of the connections on the processor.
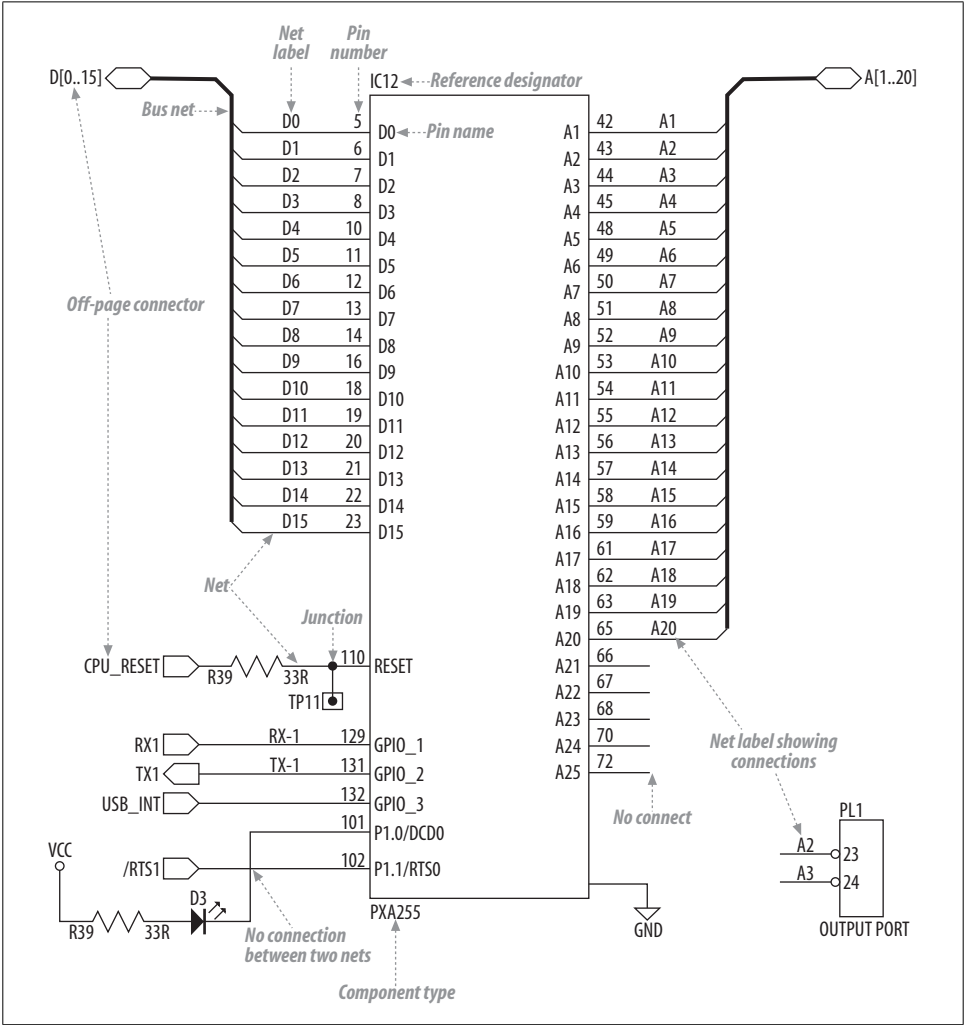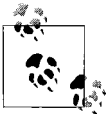

Figure 2-4. Example schematic

> The italic labels and associated arrows are not part of the original schematic. These are included to point out particular aspects of the schematic. We wanted to note this because quite often text notes are included in the schematic for clarification or layout concerns. One such clarification note in Figure 2-4 is the label OUTPUT PORT on port PL1.

The processor is the main component in this schematic. The symbol for the processor has a reference designator IC12, which is located at the top of the symbol on this schematic. The component type of the processor is PXA255 and is located at the bottom of the symbol.

The processor's pins and their associated *pin numbers* run along the sides of the symbol. For example, bit 0 of the data bus is named D0 and is located on pin number 5 of the processor.

You will also notice that some pins, such as P1.1/$\overline{\text{RTS0}}$ pin number 102, have a bar over the *pin name*. This indicates that the signal is *active low*. This means a logic level of 0 will activate the funtionality of this signal, whereas a logic level of 1 will deactivate the function. The opposite type of operation is *active high*.

Active low functionality can also be indicated by a forward slash (/) or tilde (~) placed either before or after the signal name. The signal is then typically pronounced "not RTS0" or "RTS0 bar." Some component manufacturers represent an active-low signal with the prefix "n" in front of the signal name, such as nRESET.

The wire connecting the different components together is called a *net*. Two nets that connect create a *junction*. On the schematic, a junction point is indicated by a dot, as you can see in Figure 2-4 on the RESET pin of the processor. In contrast, when two nets cross, but are not connected, there is no junction. This is shown where the net connected to the LED D3 crosses the net /RTS1.

We say that pins not connected to any nets are *no connects* or *open*. No connects are often represented on a schematic with a small cross at the end of the net. Examples of no connect pins are shown on the processor pins A21 through A25. Sometimes IC manufacturers will label no connect pins NC.

Related signals, such as data signals or address signals, are represented on a schematic by a thicker line called a *bus net*. For example, the data bus is labeled D[0..15] (in other schematics the data bus might be labeled [D0:D15]), which means the data bus net is made up of the signals D0 through D15. The individual signals are shown connecting to the processor data pins. Similarly, the address bus net is labeled A[1..20] and is made up of the signals A1 through A20. Nets connected to a bus net still need to be individually labeled.

If each net in a schematic were connected to the desired location, after a while it could create quite a rat's nest.[*] Having nets cross over each other is typically avoided in order to maintain clarity in the schematic. To facilitate this clarity, the hardware engineer can use *net labels* to assign names to the nets. The convention is that nets

---

[*] Incidentally, "rats nest" is the term used to describe the connection of nets made during layout. Once you see the initial stage of a layout, you'll understand how this name was derived.

marked with the same net label are connected, even if the engineer did not actually draw a line connecting them.

For example, in Figure 2-4, a portion of the connector with the reference designator PL1 is shown. (Incidentally, connectors and jumpers often use the reference designator J.) Because the net connected to pin number 23 of the connector PL1 is labeled A2, and the net connected to the processor's pin number 43 is labeled A2, they are connected even though the hardware engineer did not run a line to represent the A2 net connected from the processor over to PL1.

In order to aid in testing the hardware and software, hardware engineers often include *test points*. A test point is a metallic area on the finished board that provides access to a particular signal to aid in the debugging or monitoring of that signal. One test point, with the reference designator TP11, is shown in Figure 2-4 on the RESET pin of the processor. With the move to smaller and smaller IC packages and smaller pins, test points are a necessity for debugging and also aid in production testing. Also, it is impossible to probe on any pins of a ball grid array (BGA) package part, because all of the pins are contained under the IC. In this case, a test point helps greatly.

In cases where a schematic cannot fit onto a single page, there must be a way to interconnect nets from one page to another. This is the job of the *off-page connector*. Off-page connectors can be used for individual nets or bus nets. For example, the off-page connector of the data bus is D[0..15]. This is the exact same off-page connector name used on the memory page to connect the processor's data bus to the RAM's data bus.

We have found a couple of ideas to be useful for off-page connectors. These might be useful mainly for the hardware engineer working on the schematic, but other people on the team should know about them, too.

First, it is helpful if the signal type (input, output, or bidirectional) is properly represented by the appropriate off-page connector. Thus, in Figure 2-4, the data bus off-page connector indicates that these are bidirectional signals; the CPU_RESET off-page connector indicates that this signal is an input to the processor; and the TX1 off-page connector indicates that this signal is an output from the processor.

Another helpful idea is to add a little text note next to each off-page connector with the page number(s) where that particular net is used. This might not make sense for a 5-page schematic, but flipping through 20 pages of schematics can be a nightmare.

Additional tips can be found in the December 2002 *Embedded Systems Programming** article "Design for Debugability," which can be found online at *http://www.embedded.com*.

---

* *Embedded Systems Programming* magazine has changed its name to *Embedded Systems Design*.

When you take a look at the full set of schematics, you will notice that there is a block at the lower-righthand corner of each page. This is the title block; every schematic we have come across has some version of this. The title block has useful information about the schematic, such as the date, the designer's name, the revision, a page number and description of the schematic on that page, and often a list of changes made.

At this point, we have a solid understanding of the system components that make up our platform and how they are interconnected. Let's next see how to get to know the hardware.

Examine the Landscape

It is often useful to put yourself in the processor's shoes for a while. Imagine what it is like to be the processor. What does the processor's world look like? Who is connected to it? How does it talk to these other devices?

If you think about it from this perspective, one thing you quickly realize is that the processor has a lot of compatriots. These are the other pieces of hardware on the board, with which the processor communicates. The processor has different methods for communicating with these other pieces of hardware. In this section, you will learn to recognize their names and addresses.

The first thing to notice is that there are two basic types of hardware to which processors connect: memories and peripherals. Memories are for the storage and retrieval of data and code. *Peripherals* are specialized hardware devices that either coordinate interaction with the outside world or perform a specific hardware function. For example, two of the most common peripherals in embedded systems are serial ports and timers.

Members of Intel's 80x86 and some other processor families have two distinct address spaces through which they can communicate with these memories and peripherals. The first address space is called the *memory space* and is intended mainly for memory devices; the second is reserved exclusively for peripherals and is called the *I/O space*. However, peripherals can also be located within the memory space, at the discretion of the hardware designer. When that happens, we say that those peripherals are memory-mapped and that system has *memory-mapped I/O*. Some processors support only a memory space, in which case all peripherals are memory-mapped.

From the processor's point of view, memory-mapped peripherals look and act very much like memory devices. However, the function of a peripheral is quite different from that of a memory device. Instead of simply storing the data that is provided to it, a peripheral might instead interpret it as a command or as data to be processed in some way.

The designers of embedded hardware often prefer to use memory-mapped I/O exclusively, because it has advantages for both the hardware and software developers. It is attractive to the hardware developer because she might be able to eliminate the I/O space, and some of its associated wires, altogether. This might not significantly lower the production cost of the board, but it might reduce the complexity of the hardware design. Memory-mapped peripherals make life easier for the programmer, who can use C-language pointers, structs, and unions to interact with the peripherals more easily.

Memory Map

All processors store their programs and data in memory. This memory may reside on the same chip as the processor or in external memory chips. Memory is located in the processor's memory space, and the processor communicates with it by way of two sets of electrical wires called the address bus and the data bus. To read or write a particular location in memory, the processor first writes the desired address onto the address bus. Some logic (either on the processor or in an external circuit), known as an *address decoder*, interprets the upper address bits on this bus and selects the appropriate memory or peripheral chip. The data is then transferred over the data bus. The address decoder can be an external IC, but today many processors include this logic on-chip.

There are also control signals for reading and writing to various devices in a processor's memory space that are commonly referred to as the *control bus*. These control bus signals include read, write, and chip-select (or chip-enable). Some processors combine the read and write signals into a single read/write signal. On these processors, a read operation is performed by setting the signal to one level and a write operation is performed by setting the signal to the opposite level. For example, if the signal name is RD/$\overline{\text{WR}}$ (pronounced "read write bar"), the signal is set to a 1 for a read operation and set to 0 for a write operation.

The chip-select signal is set to its active level when the address for a particular device falls within the device's address range. For example, let's say a RAM device occupies the address range from 0x0000 to 0x0FFF. When the software instruction accesses the variable located at address 0x01F2, the chip-select for the RAM is set at its active level.

The read and write signals are set to their active levels by the processor based on the type of memory transaction. Figure 2-5 is an example of a *timing diagram*, a graphical representation of the timing relationship between the various signals for a given operation. The entire diagram in Figure 2-5 shows one read *cycle*. In this case, the cycle is reading 16 bits of data from memory. Typically, a table of *timing requirements* accompanies a timing diagram. The timing requirements detail the minimum and maximum acceptable times for each of the various signals and the timing relationships among the signals.
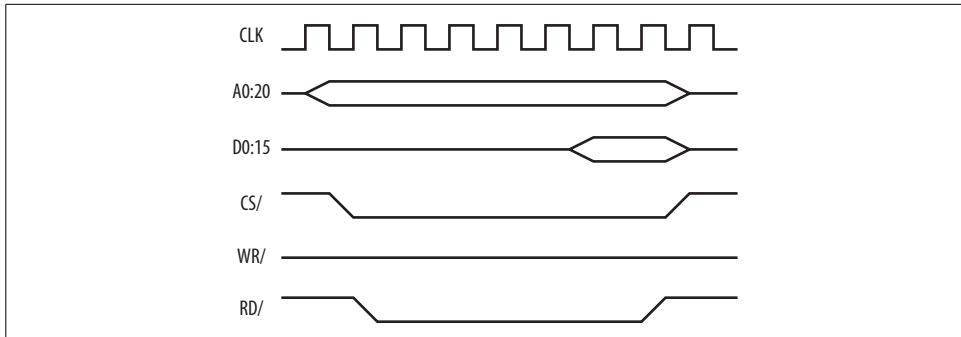
Figure 2-5. Example timing diagram

The clock signal (CLK) is the basis for all operations of the processor and is shown as the top signal in the timing diagram of Figure 2-5. The processor *clock* is generally a square wave that sequences all operations of the processor.

The next group of signals are the address bus, A[0:20], followed by the data bus, D[0:15]. Such buses are depicted in timing diagrams as shown in Figure 2-5, where a single entry represents the entire range of signals rather than each signal having its own entry. A bus is typically stable (meaning it contains a valid address or data) during the period of time when the single line splits into two lines. In hardware terms, the bus goes from being tristate (single line) to having real information present (dual line), and then back to being tristate again.

The next signal is active low chip select (CS/), and after that is the write (WR/) signal, which is also active low. Since this is a timing diagram for a read operation, the write signal stays inactive during the entire cycle. The last signal is the read (RD/). It goes low (active) after the address is set by the processor.

Additional examples of timing diagrams for the PXA255 processor can be found in the Memory Controller section of the *PXA255 Processor Developer's Manual* as well as the *PXA255 Processor Electrical, Mechanical, and Thermal Specification* datasheet.
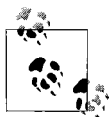
> Some processors might also include other types of control signals to help access various types of peripheral devices. These signals can be named Ready, Hold, Hold Acknowledge, Wait, and other things. A hardware engineer can use these signals to access a wide range of devices, notably those that operate slower than the processor. For example, a slower ROM can use the processor's Hold signal to tell the processor it needs more time to complete the read of a particular memory address. The processor can then wait until the ROM is able to finish getting the data for the processor.

While you are reading about a new board, create a table that shows the name and address range of each memory device and peripheral that is located in the memory space. This table is called a *memory map*. Organize the table so that the lowest

address is at the bottom and the highest address is at the top. Each time you add a device to the memory map, place it in its appropriate location in memory and label the starting and ending addresses in hexadecimal. After you have finished inserting all of the devices into the memory map, be sure to label any unused memory regions as such.

If you look back at the block diagram of the Arcom board in Figure 2-1, you will see that there are three devices attached to the address and data buses. (The PC/104 bus is connected to the address and data buses through buffers.) These devices are the RAM, ROM, and SMSC Ethernet controller. The RAM is located at the bottom of the memory address range. The ROM is located toward the top of the range.
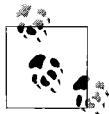
> Sometimes a single address range, particularly for memory devices, is comprised of multiple ICs. For example, the hardware engineer might use two ROM chips, each of which has a storage capacity of 1 MB. This gives the processor a total of 2 MB of ROM. Furthermore, the hardware engineer is able to set up the two individual ROM chips so the processor does not know which chip it is actually accessing; the division of the two chips is transparent to the processor, and it sees the memory as one contiguous block.

The memory map in Figure 2-6 shows what the memory devices in Figure 2-1 look like to the processor. Also included in Figure 2-6 are the processor's internal peripheral registers, labeled PXA255 Peripherals, which are mapped into the processor's memory space. In a sense, this is the processor's "address book." Just as you maintain a list of names and addresses in your personal life, you must maintain a similar list for the processor. The memory map contains one entry for each of the memories and peripherals that are accessible from the processor's memory space. This diagram is arguably the most important piece of information about the system for an embedded software engineer and should be kept up to date and maintained as part of the permanent records associated with the project.

For each new board, you should create a C-language header file that describes its most important features. This file provides an abstract interface to the hardware. In effect, it allows you to refer to the various devices on the board by name rather than by address. This has the added benefit of making your application software more portable. If the memory map ever changes—for example, if the 64 MB of RAM is moved—you need only change the affected lines of the board-specific header file and recompile your application.

Abstracting the hardware into a file (for smaller projects) or a directory of files (for larger projects) also allows you to reuse certain portions of your code as you move from project to project. Because the hardware engineer will likely reuse components in new designs, you too can reuse drivers for these components in these new designs.
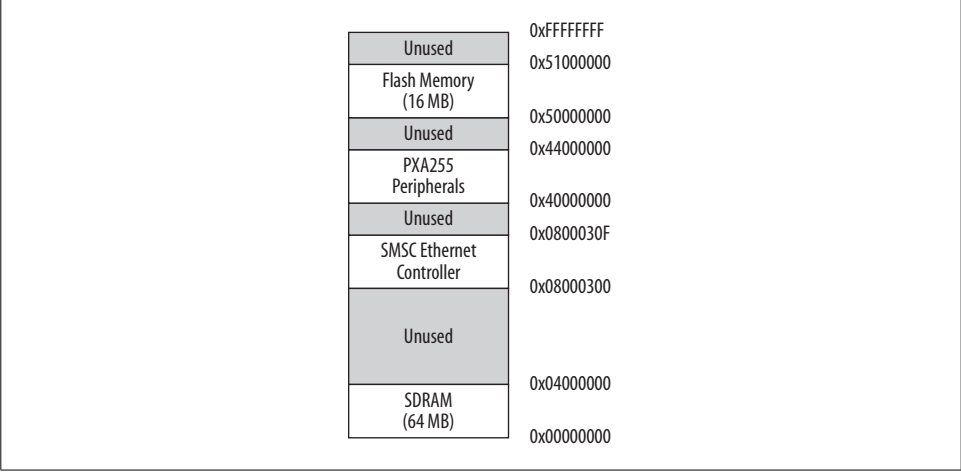
Figure 2-6. Memory map for the Arcom board

As this chapter progresses, we will show you how to create a header file for the Arcom board; the following code is the first section of this file. This part of the header file describes the memory map:

```
/**********************************************************************
 *
 *   Memory Map
 *
 *           Base Address   Size   Description
 *           -------------- -----  ----------------------------------
 *           0x00000000     64M    SDRAM
 *           0x08000300     N/A    Ethernet controller
 *           0x50000000     16M    Flash
 *
 **********************************************************************/

#define SDRAM_BASE              (0x00000000)
#define ETHERNET_BASE           (0x08000300)
#define FLASH_BASE              (0x50000000)
```

Learn How to Communicate

Now that you know the names and addresses of the memory and peripherals attached to the processor, it is time to learn how to communicate with the peripherals. There are two basic communication techniques: *polling* and *interrupts*. In either case, the processor usually issues some sort of command to the device by writing—by way of the memory or I/O space—particular data values to particular addresses within the device, and then waits for the device to complete the assigned task. For example, the processor might ask a timer to count down from 1,000 to 0. Once the

countdown begins, the processor is interested in just one thing: is the timer finished counting yet?

If polling is used, the processor repeatedly checks to see whether the task has been completed. This is analogous to the small child who repeatedly asks, "Are we there yet?" throughout a long trip. Like the child, the processor spends a large amount of otherwise useful time asking the question and getting a negative response. To implement polling in software, you need only create a loop that reads the status register of the device in question. Here is an example:

```
do
{
    /* Play games, read, listen to music, etc. */
    ...

    /* Poll to see if we're there yet. */
    status = areWeThereYet( );

} while (status == NO);
```

The second communication technique uses interrupts. An interrupt is an asynchronous electrical signal from a peripheral to the processor. Interrupts can be generated from peripherals external or internal to the processor, as well as by software.

When interrupts are used, the processor issues commands to the peripheral exactly as before, but then waits for an interrupt to signal completion of the assigned work. While the processor is waiting for the interrupt to arrive, it is free to continue working on other things. When the interrupt signal is asserted, the processor finishes its current instruction, temporarily sets aside its current work, and executes a small piece of software called the *interrupt service routine* (ISR) or *interrupt handler*. When the ISR completes, the processor returns to the work that was interrupted.

Of course, this isn't all automatic. The programmer must write the ISR himself and "install" and enable it so that it will be executed when the relevant interrupt occurs. The first few times you do this, it will be a significant challenge. But, even so, the use of interrupts generally decreases the complexity of one's overall code by giving it a better structure. Rather than device polling being embedded within an unrelated part of the program, the two pieces of code remain appropriately separate.

On the whole, interrupts are a much more efficient use of the processor than polling. The processor is able to use a larger percentage of its waiting time performing useful work. However, there is some overhead associated with each interrupt. It takes a good bit of time—relative to the length of time it takes to execute an opcode—to put aside the processor's current work and transfer control to the interrupt service routine. Many of the processor's registers must be saved in memory.

In practice, both interrupts and polling are used frequently. Interrupts are used when efficiency is paramount or when multiple devices must be monitored simultaneously. Polling is typically used when the processor must respond to some event

more quickly than is possible using interrupts or when large amounts of data are expected to arrive at particular intervals, such as during real-time data acquisition. We will take a closer look at interrupts in Chapter 8.

Getting to Know the Processor

If you haven't worked with the processor on your board before, you should take some time to get familiar with it now. This shouldn't take very long if you do all of your programming in a high-level language such as C. You need to dig in and find out how particular peripherals of the processor work. Generally, to the user of a high-level language, most processors look and act pretty much the same. However, if you'll be doing any assembly language programming, you need to familiarize yourself with the processor's register architecture and instruction set.

Everything you need to know about the processor can be found in the databooks provided by the manufacturer. If you don't have a databook or programmer's guide for your processor already, you should obtain one immediately. If you are going to be a successful embedded systems programmer, you must be able to read databooks and get something out of them. Processor databooks are usually well written—as databooks go—so they are an ideal place to start. Begin by flipping through the databook and noting sections that are most relevant to the tasks at hand. Then go back and begin reading the processor overview section.

Things you'll want to learn about the processor from its databook are:

- What address does the processor jump to after a reset?
- What is the state of the processor's registers and peripherals at reset?
- What is the proper sequence to program a peripheral's registers?
- Where should the interrupt vector table be located? Does it have to be located at a specific address in memory? If not, how does the processor know where to find it?
- What is the format of the interrupt vector table? Is it just a table of pointers to ISR functions?
- Are there any special interrupts—sometimes called *traps*—that are generated within the processor itself? Must an ISR be written to handle each of these?
- How are interrupts enabled and disabled? Globally and individually?
- How are interrupts acknowledged or cleared?

In addition to the processor databook, the Internet contains a wealth of information for embedded software developers. The manufacturer's site is a great place to start. In addition, a search for a particular processor can yield oodles of useful information from fellow developers, including code snippets giving you exact details on how to write your software. Several newsgroups are also targeted toward embedded software development and toward specific processors.
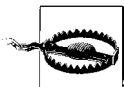
You need to take care to fully understand any licensing issues of the software you find on the Internet should you decide to use someone else's code. You might have to get your company's legal department involved in order to avoid any problems.

Another useful tool for understanding the processor is a development board. Once the processor is selected, you can search for your options for a development board. You need to consider the peripherals and software tools included on the development board. For example, if your application is going to include an Ethernet port, it would be a good idea to select a development board that also includes an Ethernet port. There is typically example software included with the development board as well. If the project uses a processor that you have not worked with before, the example software can get you up the learning curve a lot faster. The development board will assist you in getting a jump-start on the embedded software development.

Another benefit of a development board is that if you are seeing some oddities related to your project's hardware, you can always go back to the development board (where the hardware should be stable) and run some tests to see whether the problem is specific to the new design.

Processors in General

Many of the most common processors are members of families of related devices. In some cases, the members of such a processor family represent points along an evolutionary path. The most obvious example is Intel's 80x86 family, which spans from the original 8086 to the Pentium 4 and beyond. In fact, the 80x86 family has been so successful that it has spawned an entire industry of imitators.

As it is used in this book, the term *processor* refers to any of three types of devices known as *microprocessors*, *microcontrollers*, and *digital signal processors*. The name microprocessor is usually reserved for a chip that contains a powerful *central processing unit* (CPU) that has not been designed with any particular computation in mind. These chips are usually the foundation of personal computers and high-end workstations, although microprocessors are used in embedded systems as well. Other widely known microprocessors are members of Freescale's[*] 68K—found in older Macintosh computers—and the ubiquitous 80x86 families.

A microcontroller is very much like a microprocessor, except that it has been designed specifically for use in embedded systems. Microcontrollers typically include a CPU, memory (a small amount of RAM, ROM, or both), and other peripherals in the same integrated circuit. If you purchase all of these items on a single chip, it is possible to reduce the cost of an embedded system substantially. Among the most

---

[*] This is Motorola's new semiconductor division.

popular microcontrollers are the 8051 and its many imitators and the 68HCxx series. It is also common to find microcontroller versions of popular microprocessors. For example, Intel's 386 EX is a microcontroller version of the 80386 microprocessor.

The final type of processor is a *digital signal processor*, or DSP. The CPU within a DSP is specially designed to perform discrete-time signal processing calculations—like those required for audio and video communications—extremely fast. Because DSPs can perform these types of calculations much faster than other processors, they offer a powerful, low-cost microprocessor alternative for designers of cell phones and other telecommunications and multimedia equipment. Analog Devices, Freescale, and TI are each vendors of common DSP devices.

The PXA255 XScale Processor

The processor on the Arcom board is a PXA255 that incorporates the XScale core. XScale is based on the ARM Version 5TE architecture. In order to find out more about the ARM processor, a great book is David Seal's *ARM Architecture Reference Manual* (Addison-Wesley); this book is commonly referred to as the "ARM ARM."

In addition to the CPU, the PXA255 contains an interrupt control unit, a memory controller, several general-purpose I/O pins, four timer/counters, an I2C bus interface unit, four serial ports, 16 direct memory access (DMA) channels, a memory controller that supports several memory types including DRAM, a USB client, an LCD controller, two pulse width modulators, a real-time clock, a watchdog timer unit, and a power management unit. These extra hardware devices are located within the same chip and are referred to as *on-chip peripherals*. The CPU is able to communicate with and control the on-chip peripherals directly, via internal buses.

Although the on-chip peripherals are distinct hardware devices, they act like little extensions of the PXA255 CPU. The software can control them by reading and writing to the peripheral specific registers. The control and status registers for each of the on-chip peripherals are located at fixed addresses in memory space. The exact addresses of each register can be found in the *PXA255 Processor Developer's Manual*. To isolate these details from your application software and to aid in the readability of your software, it is good practice to include the addresses of any registers you will be using in the header file for your board. You can see from the following code snippet that it is more difficult to understand what is going on when addresses are used directly:

```
if (bLedEnable == TRUE)
{
    *((uint32_t *)0x40E00018) = 0x00400000;
}
```

Although comments could clarify what is going on in the above code, it is better to use more descriptive names in your software that will make your code self-documenting, but do add comments as well to aid in understanding.

In addition to a header file describing the board's features, a C-language header file that describes the processor's registers should also be created. An example of descriptive names for some of the registers in the PXA255 processor follows. It is also helpful to define descriptive names for particular bits in a register if they will be addressed individually.[*]

```
/*******************************************************
 * PXA255 XScale ARM Processor On-Chip Peripherals
 *******************************************************/

/* Timer Registers */
#define TIMER_0_MATCH_REG           (*((uint32_t volatile *)0x40A00000))
#define TIMER_1_MATCH_REG           (*((uint32_t volatile *)0x40A00004))
#define TIMER_2_MATCH_REG           (*((uint32_t volatile *)0x40A00008))
#define TIMER_3_MATCH_REG           (*((uint32_t volatile *)0x40A0000C))
#define TIMER_COUNT_REG             (*((uint32_t volatile *)0x40A00010))
#define TIMER_STATUS_REG            (*((uint32_t volatile *)0x40A00014))
#define TIMER_INT_ENABLE_REG        (*((uint32_t volatile *)0x40A0001C))

/* Timer Interrupt Enable Register Bit Descriptions */
#define TIMER_0_INTEN               (0x01)
#define TIMER_1_INTEN               (0x02)
#define TIMER_2_INTEN               (0x04)
#define TIMER_3_INTEN               (0x08)

/* Timer Status Register Bit Descriptions */
#define TIMER_0_MATCH               (0x01)
#define TIMER_1_MATCH               (0x02)
#define TIMER_2_MATCH               (0x04)
#define TIMER_3_MATCH               (0x08)

/* Interrupt Controller Registers */
#define INTERRUPT_PENDING_REG       (*((uint32_t volatile *)0x40D00000))
#define INTERRUPT_ENABLE_REG        (*((uint32_t volatile *)0x40D00004))
#define INTERRUPT_TYPE_REG          (*((uint32_t volatile *)0x40D00008))

/* Interrupt Enable Register Bit Descriptions */
#define GPIO_0_ENABLE               (0x00000100)
#define UART_ENABLE                 (0x00400000)
#define TIMER_0_ENABLE              (0x04000000)
#define TIMER_1_ENABLE              (0x08000000)
#define TIMER_2_ENABLE              (0x10000000)
#define TIMER_3_ENABLE              (0x20000000)
```

---

[*] We will discuss the use of the keyword `volatile` in Chapter 7.

```
/* General Purpose I/O (GPIO) Registers */
#define GPIO_0_LEVEL_REG            (*((uint32_t volatile *)0x40E00000))
#define GPIO_1_LEVEL_REG            (*((uint32_t volatile *)0x40E00004))
#define GPIO_2_LEVEL_REG            (*((uint32_t volatile *)0x40E00008))
#define GPIO_0_DIRECTION_REG        (*((uint32_t volatile *)0x40E0000C))
#define GPIO_1_DIRECTION_REG        (*((uint32_t volatile *)0x40E00010))
#define GPIO_2_DIRECTION_REG        (*((uint32_t volatile *)0x40E00014))
#define GPIO_0_SET_REG              (*((uint32_t volatile *)0x40E00018))
#define GPIO_1_SET_REG              (*((uint32_t volatile *)0x40E0001C))
#define GPIO_2_SET_REG              (*((uint32_t volatile *)0x40E00020))
#define GPIO_0_CLEAR_REG            (*((uint32_t volatile *)0x40E00024))
#define GPIO_1_CLEAR_REG            (*((uint32_t volatile *)0x40E00028))
#define GPIO_2_CLEAR_REG            (*((uint32_t volatile *)0x40E0002C))
#define GPIO_0_FUNC_LO_REG          (*((uint32_t volatile *)0x40E00054))
#define GPIO_0_FUNC_HI_REG          (*((uint32_t volatile *)0x40E00058))
```

Let's take a look at the earlier code snippet written to use a register definition from the example header file:

```
if (bLedEnable == TRUE)
{
    GPIO_0_SET_REG = 0x00400000;
}
```

This code is a lot easier to read and understand, even without a comment. Defining registers in a header file, as we have shown in the preceding code, also prevents you or another team member from running to the databook every other minute to look up a register address.

Study the External Peripherals

At this point, you've studied every aspect of the new hardware except the external peripherals. These are the hardware devices that reside outside the processor chip and communicate with it by way of interrupts and I/O or memory-mapped registers.

Begin by making a list of the external peripherals. Depending on your application, this list might include LCD or keyboard controllers, analog-to-digital (A/D) converters, network interface chips, or custom *application-specific integrated circuits* (ASICs). In the case of the Arcom board, the list contains just two items: the SMSC Ethernet controller and the parallel port.

You should obtain a copy of the user's manual or datasheet for each device on your list. At this early stage of the project, your goal in reading these documents is to understand the basic functions of the device. What does the device do? What registers are used to issue commands and receive the results? What do the various bits and larger fields within these registers mean? When, if ever, does the device generate interrupts? How are interrupts acknowledged or cleared at the device?

When you are designing the embedded software, you should try to break the program down along device lines. It is usually a good idea to associate a software

module called a *device driver* with each of the external peripherals. A device driver is nothing more than a collection of software routines that control the operation of a specific peripheral and isolate the application software from the details of that particular hardware device. We'll have a lot more to say about device drivers later on.

Initialize the Hardware

The final step in getting to know your new hardware is to write some initialization software. This is your best opportunity to develop a close working relationship with the hardware, especially if you will be developing the remainder of the software in a high-level language.

During hardware initialization, it may be impossible to avoid using assembly language. However, after completing this step, you will be ready to begin writing small programs.[*]
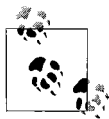
> If you are one of the first software engineers to work with a new board—especially a prototype—the hardware might not work as advertised. All processor-based boards require some amount of software testing to confirm the correctness of the hardware design and the proper functioning of the various peripherals. This puts you in an awkward position when something is not working properly. How do you know whether the hardware or your software is causing the problem? If you happen to be good with hardware or have access to a simulator, you might be able to construct some experiments to answer this question. Otherwise, you should probably ask a hardware engineer to join you in the lab for a joint debugging session.

The hardware initialization should be executed before the startup code described in Chapter 4. The code described there assumes that the hardware has already been initialized and concerns itself only with creating a proper runtime environment for high-level language programs. Figure 2-7 provides an overview of the entire initialization process, from processor reset through hardware initialization and C startup code to `main`.

The first stage of the initialization process is the reset code. This is a small piece of assembly language (usually only two or three instructions) that the processor executes immediately after it is powered on or reset. The sole purpose of this code is to transfer control to the hardware initialization routine. The first instruction of the reset code must be placed at a specific location in memory, usually called the *reset*

---

[*] In order to make the example in Chapter 3 a little easier to understand, we didn't show any of the initialization code there. However, it is necessary to get the hardware initialization code working before you can write even simple programs such as Blinking LED. The Arcom board includes a debug monitor that handles all of the assembly language hardware initialization.
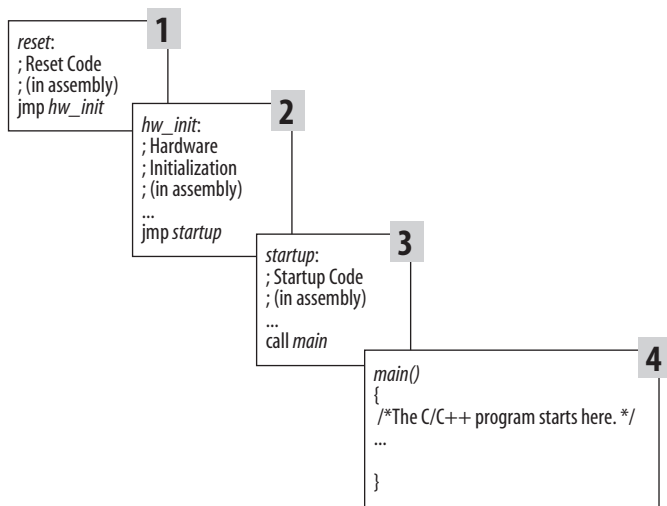
Figure 2-7. The hardware and software initialization process

address or *reset vector*, which is specified in the processor databook. The reset address for the PXA255 is 0x00000000.

Most of the actual hardware initialization takes place in the second stage. At this point, we need to inform the processor about its environment. This is also a good place to initialize the interrupt controller and other critical peripherals. Less critical hardware devices can be initialized when the associated device driver is started, usually from within main.

The PXA255 has boot select pins that allow you to specify the type and width of the memory device from which the processor attempts to execute the initial instructions. The memory device that the processor boots from typically contains the code to program several internal registers of the PXA255 that must be programmed before any useful work can be done with the processor. These internal registers are responsible for setting up the memory map and are part of the processor's internal memory controller. By programming the memory interface configuration registers, you are essentially waking up each of the memory and peripheral devices that are connected to the processor.

The PXA255 contains six chip-selects for interfacing to various types of external memory. These chip-selects are active for accesses in particular memory ranges. Each chip-select is associated with a single "chip enable" wire that runs from the processor to some other chip. In many systems, the circuitry to do this is external to the processor. The association between particular chip-selects and hardware devices must be established by the hardware designer. All you need to do is get a list of chip-select settings from her and load those settings into the memory configuration registers.

Upon reset, the PXA255 jumps to the address 0x00000000, which activates chip select 0 (the nCS[0] pin). This is the processor's "fetal position," and it implies that this chip-select is used to access some type of nonvolatile memory, such as flash in the Arcom board's case. Since there are no other chip-selects configured at this point, the software must also not require the use of any RAM. This includes making a subroutine call, as this will require access to the stack (discussed in a moment), which lives in RAM; function calls are verboten. Until the RAM is active and the stack pointer initialized, your code must be linear.

The hardware initialization routine, hw_init, should start by initializing the memory interface configuration registers to inform the processor about the other memory and peripheral devices that are installed on the board. By the time this task is complete, the entire range of ROM and RAM addresses will be enabled, so the remainder of your software can be located at any convenient address in either ROM or RAM.

The third initialization stage contains the *startup code*. Its job is to prepare the way for code written in a high-level language. One of the important tasks of this code is to set up the stack for the system. The *stack* is the area of RAM that the processor uses for temporary storage during execution. The stack operates on the *last-in-first-out* (LIFO) principle in which items of data are pushed onto and popped off of the stack: typically, local variables (a.k.a. *automatic variables*) and return addresses from function calls during program execution. After initializing the stack, the startup code calls main. From that point forward, all of your other software can be written in a high-level language.

Hopefully, you are starting to understand how embedded software gets from processor reset to your main program. Admittedly, the very first time you try to pull all of these components (reset code, hardware initialization, high-level language startup code, and application) together on a new board, there will be problems. So expect to spend some time debugging each of them. We'll take a look at debugging in Chapter 5.

With a new hardware platform, some hardware problems may pop up as well. These might lead to a problem where the processor simply doesn't do anything. Sometimes the problem is a basic issue that was overlooked. Some of the basic things to do are:

- Make sure the processor and ROM are receiving the proper voltage required to operate the parts.
- Check to make sure the clock signal is running. The processor won't do anything without a clock.
- Verify that the processor is coming out of reset properly. You can check the address a processor is fetching using a logic analyzer. This will validate that the processor is trying to fetch the first instruction from the location you expect.
- Make sure that a watchdog timer isn't resetting the processor.
- Ensure that input pins on the processor are pulled high or low. This is particularly important for interrupt pins. An input pin in an unknown state (commonly called a *floating* pin) can wreak all sorts of havoc for a processor.

The hardware engineer might handle these tasks for you, but don't be afraid to jump right in and look over the schematics yourself. Or better yet, see whether you can sit in the lab with the hardware engineer while he performs his initial check-out of the board.

Expect that the initial hardware bring-up will be the hardest part of the project. You will soon see that once you have a basic program operating that you can fall back on, the work just gets easier and easier—or at least more similar to other types of computer programming.

Your First Embedded Program

ACHTUNG! Das machine is nicht fur gefingerpoken und mittengrabben. Ist easy schnappen der springenwerk, blowenfusen und corkenpoppen mit spitzensparken. Ist nicht fur gewerken by das dummkopfen. Das rubbernecken sightseeren keepen hands in das pockets. Relaxen und vatch das blinkenlights!

—Electronics Laboratory Sign

In this chapter, we'll dive right into embedded programming by way of an example. Our example is similar in spirit to the "Hello, World!" example found in the beginning of most other programming books. We'll discuss why we picked this particular program and point out the parts of it that are dependent on the target hardware. This chapter contains only the source code for the program. We'll discuss how to create the executable and how to actually run it in the chapters that follow.

Hello, World!

It seems that every programming book ever written begins with the same example—a program that prints "Hello, World!" on the user's screen. An overused example such as this might seem a bit boring. Among other things, the example helps readers quickly assess the ease or difficulty with which simple programs can be written in the programming environment at hand. In that sense, "Hello, World!" serves as a useful benchmark for users of programming languages and computer platforms.

Based on the "Hello, World!" benchmark, embedded systems are among the most challenging computer platforms for programmers to work with. In some embedded systems, it might even be impossible to implement the "Hello, World!" program. And in those systems that are capable of supporting it, the printing of text strings is usually more of an endpoint than a beginning.

A principal assumption of the "Hello, World!" example is that there is some sort of output device on which strings of characters can be printed. A text window on the

user's monitor often serves that purpose. But most embedded systems lack a monitor or analogous output device. And those that do have one typically require a special piece of embedded software, called a display driver, to be implemented first—a rather challenging way to begin one's embedded programming career.

It would be much better to begin with a small, easily implemented, and highly portable embedded program in which there is little room for programming mistakes. After all, the reason our book-writing counterparts continue to use the "Hello, World!" example is that implementing it is a no-brainer. This eliminates one of the variables in the case that the user's program doesn't work correctly the first time: it isn't a bug in his code; rather, it is a problem with the development tools or process he used to create the executable program.

Embedded programmers must be self-reliant. They must always begin each new project with the assumption that nothing works—that all they can rely on is the basic syntax of their programming language. Even the standard library routines might not be available to them. These are the auxiliary functions—such as `printf` and `memcpy`—that most other programmers take for granted. In fact, library routines are often as much a part of the C-language standard as the basic syntax. However, the library part of the standard is more difficult to support across all possible computing platforms and is occasionally ignored by the makers of compilers for embedded systems.

So, you won't find an actual "Hello, World!" program in this chapter. Instead, we'll write the simplest C-language program we can, without assuming you have specialized hardware (which would require a device driver) or any library with functions such as `printf`. As we progress through the book, we will gradually add standard library routines and the equivalent of a character output device to our repertoire. By that time, you'll be well on your way to becoming an expert in the field of embedded systems programming.

The Blinking LED Program

Almost every embedded system that we've encountered in our respective careers has had at least one LED that could be controlled by software. If the hardware designer plans to leave the LED out of the circuit, lobby hard for getting one attached to a general-purpose I/O (GPIO) pin. As we will see later, this might be the most valuable debugging tool you have.

A popular substitute for the "Hello, World!" program is one that blinks an LED at a rate of 1 Hz (one complete on-off cycle per second).* Typically, the code required to

---

* Of course, the rate of blink is completely arbitrary. But one of the good things about the 1 Hz rate is that it's easy to confirm with a stopwatch. Simply start the stopwatch, count off a number of blinks, stop the stopwatch, and see whether the number of elapsed seconds is the same as the the number of blinks you counted. Need greater accuracy? Simply count off more blinks.

turn an LED on and off is limited to a few lines of code, so there is very little room for programming errors to occur. And because almost all embedded systems have LEDs, the underlying concept is extremely portable.

Our first step is to learn how to control the green LED we want to toggle. On the Arcom board, the green LED is located on the add-on module shown in Figure 3-1. The green LED is labeled "LED2" on the add-on module. The Arcom board's *VIPER-Lite Technical Manual* and the *VIPER-I/O Technical Manual* describe how the add-on module's LEDs are connected to the processor. The schematics can also be used to trace the connection from the LED back to the processor, which is typically the method you need to use once you have your own hardware.
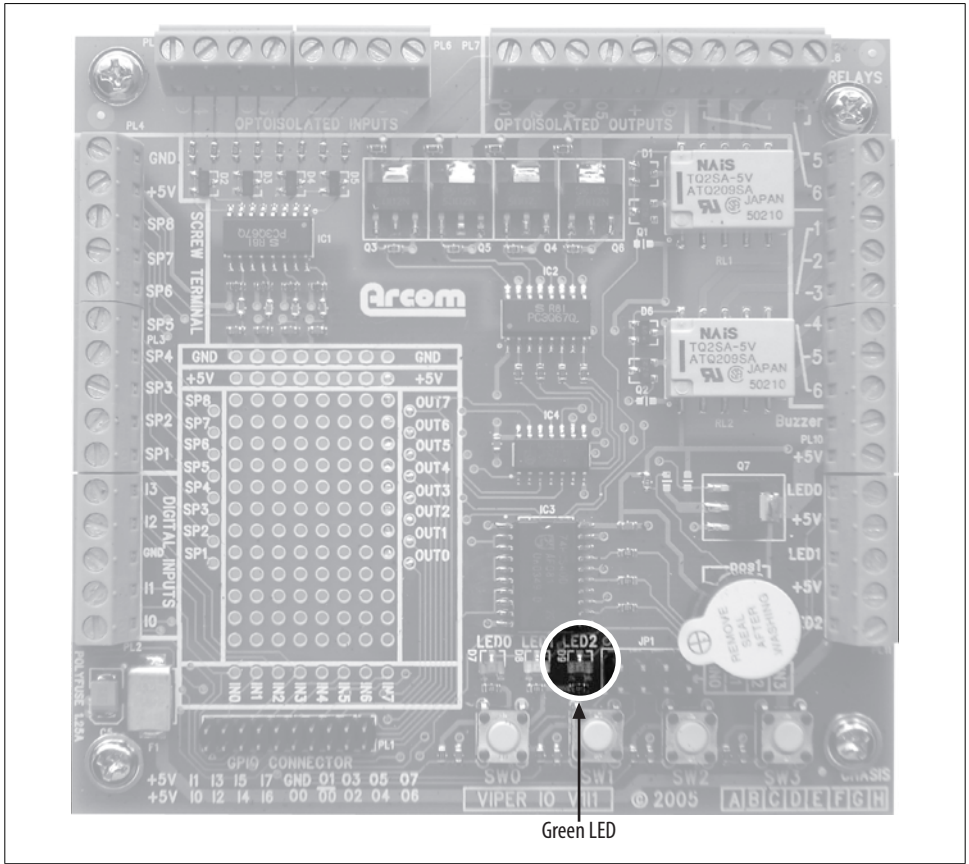
Green LED

Figure 3-1. Arcom board add-on module containing the green LED

LED2 is controlled by the signal OUT2, as described in the LEDs section in the Arcom board's *VIPER-I/O Technical Manual*. This text also informs us that the signals to the LEDs are inverted; therefore, when the output is high, the LEDs are off, and vice versa. The general-purpose I/O section of the *VIPER Technical Manual*

shows that the OUT2 signal is controlled by the processor's GPIO pin 22. Therefore, we will need to be able to set GPIO pin 22 alternately high and low to get our blinker program to function properly.

The superstructure of the Blinking LED program is shown next. This part of the program is hardware-independent. However, it relies on the hardware-dependent functions ledInit, ledToggle, and delay_ms to initialize the GPIO pin controlling the LED, change the state of the LED, and handle the timing, respectively. These functions are described in the following sections, where we'll really get a sense of what it's like to do embedded systems programming.

```c
#include "led.h"

/************************************************************************
 *
 * Function:    main
 *
 * Description: Blink the green LED once a second.
 *
 * Notes:
 *
 * Returns:     This routine contains an infinite loop.
 *
 ************************************************************************/
int main(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    while (1)
    {
        /* Change the state of the green LED. */
        ledToggle();

        /* Pause for 500 milliseconds. */
        delay_ms(500);
    }

    return 0;
}
```

The ledInit Function

Before we start to use a particular peripheral, we first need to understand the hardware used to control that specific peripheral.* Because the LED we want to blink is connected to one of the PXA255 processor's 85 bidirectional GPIO pins, we need to

---

* All of the documentation for the Arcom board is contained on the VIPER-Lite Development Kit CD-ROM. This includes datasheets and user's manuals for the components on the board.

focus on those. Often, as is the case with the PXA255 processor, I/O pins of embedded processors have multiple functions. This allows the same pins either to be used as user-controllable I/O or to support particular peripheral functionality within the processor. Configuration registers are used to select how the application will use each specific port pin.

On the PXA255, each port pin can be configured for use by the internal peripheral (called an alternate-function pin) or by the user (called a general-purpose pin). For each GPIO pin, there are several 32-bit registers. These registers allow for configuration and control of each GPIO pin. The description of the registers for the GPIO port that contains the pin for the green LED is shown in Table 3-1. These registers are located within the PXA255 chip.

Table 3-1. PXA255 GPIO registers[a]

Register name	Type	Address	Name	Purpose
GPLR0	Read-only	0x40E00000	GPIO Pin-Level Register	Reflects the state of each GPIO pin. 0 = Pin state is low. 1 = Pin state is high.
GPDR0	Read/write	0x40E0000C	GPIO Pin Direction Register	Controls whether a pin is an input or output. 0 = Pin is configured as an input. 1 = Pin is configured as an output.
GPSR0	Write-only	0x40E00018	GPIO Pin Output Set Register	For pins configured as output, the pin is set high by writing a 1 to the appropriate bit in this register. 0 = Pin is unaffected. 1 = If configured as output, pin level is set high.
GPCR0	Write-only	0x40E00024	GPIO Pin Output Clear Register	For pins configured as output, the pin is set low by writing a 1 to the appropriate bit in this register. 0 = Pin is unaffected. 1 = If configured as output, pin level is set low.
GAFR0_U	Read/write	0x40E00058	GPIO Alternate Function Register (High)	Configures GPIO pins for general I/O or alternate functionality. 00 = GPIO pin is used as general-purpose I/O. 01 = GPIO pin is used for alternate function 1. 10 = GPIO pin is used for alternate function 2. 11 = GPIO pin is used for alternate function 3.

[a] You can view additional information about the GPIO pins of the PXA255 processor in the *PXA255 Processor Developer's Manual*.

The *PXA255 Processor Developer's Manual* states that the configuration of the GPIO pins for the LEDs are controlled by bits 20 (red), 21 (yellow), and 22 (green) in the 32-bit GPDR0 register. Figure 3-2 shows the location of the bit for GPIO pin 22 in the GPDR0 register; this bit configures the direction of GPIO pin 22 that controls the green LED.
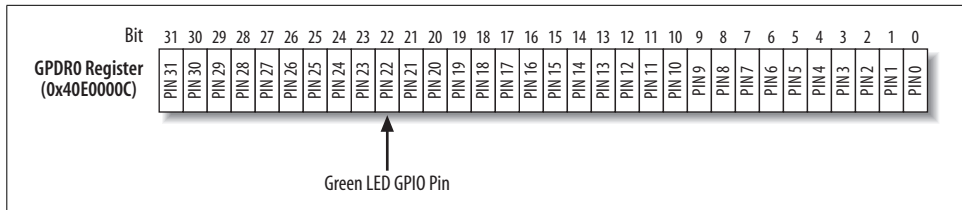
Figure 3-2. PXA255 processor GPDR0 register

The PXA255 peripheral control registers are located in memory space, as shown in Figure 2-6 in Chapter 2. The addresses of these registers are given in Table 3-1. Because the registers are memory-mapped, they are easily accessed in C in the same ways that any memory location is read or written.
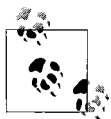
> You may notice as you read through the *PXA255 Processor Developer's Manual* that certain registers contain bits that are designated as reserved. This is typical in many registers within a processor. The processor manual will state how these bits should be read or written. In the case of the PXA255 processor, the manual states that reserved bits must be written as zeros and ignored when read. It is important that you do not use for other purposes any bits labeled as reserved.

I/O Space Register Access

If the GPIO pin registers are located in I/O space, they can be accessed only by using assembly language. The 80x86 assembly language instructions to access I/O space are in and out. The C language has no built-in support for these operations. Wrapper C functions called inport and outport are part of some 80x86-specific standard library packages.

Most registers within a CPU have a default configuration after reset. This means that before we are able to control the output on any I/O pins, we need to make sure the pin is configured properly. After reset, all GPIO pins in the PXA255 are configured as inputs. In addition, they function as general-purpose I/O pins rather than alternate-function pins.
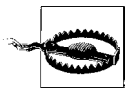
> Although the GPIO pins that control the LEDs are configured as general-purpose I/O pins upon reset, we need to ensure that the other software that is running did not change the functionality of these GPIO pins.
>
> It is a good practice always to initialize hardware you are going to use, even if you think the default behavior is fine.

In our case, we need to configure GPIO pin 22 as an output via bit 22 in the GPDR0 register. Furthermore, the GPIO pin that controls the green LED must be set to function as a general-purpose I/O pin via the same bit in the GAFR0_U register.

The bitmask for the GPIO pin that controls the green LED on the Arcom board is defined in our program as:

```
#define LED_GREEN                   (0x00400000)
```

A fundamental technique used by the ledInit routine is a read-modify-write of a hardware register. First, read the contents of the register, then modify the bit that controls the LED, and finally write the new value back into the register location. The code in ledInit performs two read-modify-write operations—one on the register GAFR0_U and one on GPDR0, in that order. These operations are done by using the C language &= and |= operators, respectively; the effect of x &= y is the same as that of x = x & y. We will take a closer look at these operators and bit manipulation in Chapter 7.

The ledInit function configures the PXA255 processor on the Arcom board to control the green LED located on the add-on module. In the following code, you may notice that we clear the GPIO pin in the GPCR0 register to ensure that the output voltage on the GPIO pin is first set to zero, as suggested in the developer's manual.

```
#define PIN22_FUNC_GENERAL          (0xFFFFCFFF)

/**********************************************************************
 *
 * Function:    ledInit
 *
 * Description: Initialize the GPIO pin that controls the LED.
 *
 * Notes:       This function is specific to the Arcom board.
 *
 * Returns:     None.
 *
 **********************************************************************/
void ledInit(void)
{
    /* Turn the GPIO pin voltage off, which will light the LED. This should
     * be done before the pins are configured. */
    GPIO_0_CLEAR_REG = LED_GREEN;

    /* Make sure the LED control pin is set to perform general
     * purpose functions. RedBoot may have changed the pin's operation. */
    GPIO_0_FUNC_HI_REG &= PIN22_FUNC_GENERAL;

    /* Set the LED control pin to operate as output. */
    GPIO_0_DIRECTION_REG |= LED_GREEN;
}
```

The ledToggle Function

This routine runs within an infinite loop and is responsible for changing the state of the LED. The state of this LED is controlled by writing to either the GPIO Pin Output Set Register (GPSR) or the GPIO Pin Output Clear Register (GPCR). The GPSR0 register allows us to set the level of the LED control GPIO pin high; the GPCR0 register allows us to set the level of the LED control GPIO pin low. Writing to bit 22 of these registers changes the voltage on the external pin and, thus, the state of the green LED. Because the GPIO pin to the LED is inverted, when bit 22 of the GPSR0 register is set, the green LED is off, whereas when bit 22 of the GPCR0 register is set, the green LED is on. The state of the LED is determined by the GPIO Pin Level Register (GPLR).

As described earlier, the PXA255 processor has separate write-only registers for setting (GPSR0) and clearing (GPCR0) the bit that controls the GPIO pin. Therefore, a read-modify-write cannot be used to toggle the state of the LED. The actual algorithm of the ledToggle routine is straightforward: determine the current state for the LED of interest and write into the GPIO register the bit that controls that LED in order to set the new state of the LED.

```
/***********************************************************************
 *
 * Function:    ledToggle
 *
 * Description: Toggle the state of one LED.
 *
 * Notes:       This function is specific to the Arcom board.
 *
 * Returns:     None.
 *
 ***********************************************************************/
void ledToggle(void)
{
    /* Check the current state of the LED control pin. Then change the
     * state accordingly. */
    if (GPIO_0_LEVEL_REG & LED_GREEN)
        GPIO_0_CLEAR_REG = LED_GREEN;
    else
        GPIO_0_SET_REG = LED_GREEN;
}
```

The delay_ms Function

We also need to implement a 500 ms delay between LED toggles. We do this by busy-waiting within the following delay_ms routine. This routine accepts the length of the requested delay, in milliseconds, as its only parameter. It then multiplies that number by the constant CYCLES_PER_MS to obtain the total number of while-loop iterations that are required in order to delay for the requested time period:

```
    /* Number of decrement-and-test cycles. */
    #define CYCLES_PER_MS           (9000)


    /**********************************************************************
     *
     * Function:    delay_ms
     *
     * Description: Busy-wait for the requested number of milliseconds.
     *
     * Notes:       The number of decrement-and-test cycles per millisecond
     *              was determined through trial and error.  This value is
     *              dependent upon the processor type, speed, compiler, and
     *              the level of optimization.
     *
     * Returns:     None.
     *
     **********************************************************************/
    void delay_ms(int milliseconds)
    {
        long volatile cycles = (milliseconds * CYCLES_PER_MS);

        while (cycles != 0)
            cycles--;
    }
```

The hardware-specific constant CYCLES_PER_MS represents the number of times the processor can get through the while loop in a millisecond. To determine this number, we used trial and error. We will see later how to use a hardware counter to achieve better timing accuracy.

The four functions main, ledInit, ledToggle, and delay_ms do the whole job of the Blinking LED program. Of course, we still need to talk about how to build and execute this program. We'll examine those topics in the next two chapters. But first, we have a little something to say about infinite loops and their role in embedded systems.

The Role of the Infinite Loop

One of the most fundamental differences between programs developed for embedded systems and those written for other computer platforms is that the embedded programs almost always have an infinite loop. Typically, this loop surrounds a significant part of the program's functionality, as it does in the Blinking LED program. The infinite loop is necessary because the embedded software's job is never done. It is intended to be run until either the world comes to an end or the board is reset, whichever happens first.

In addition, most embedded systems run only one piece of software. Although hardware is important, the system is not a digital watch or a cellular phone or a microwave oven without that software. If the software stops running, the hardware is rendered useless. So the functional parts of an embedded program are almost always surrounded by an infinite loop that ensures that they will run forever.

If we had forgotten the infinite loop in the Blinking LED program, the LED would have simply changed state once.

Compiling, Linking, and Locating

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free.
—Richard Stallman, Founder of the GNU Project
The GNU Manifesto

In this chapter, we'll examine the steps involved in preparing your software for execution on an embedded system. We'll also discuss the associated development tools and see how to build the Blinking LED program shown in Chapter 3.

But before we get started, we want to make it clear that embedded systems programming is not substantially different from the programming you've done before. The only thing that has really changed is that you need to have an understanding of the target hardware platform. Furthermore, each target hardware platform is unique— for example, the method for communicating over a serial interface can vary from processor to processor and from platform to platform. Unfortunately, this uniqueness among hardware platforms leads to a lot of additional software complexity, and it's also the reason you'll need to be more aware of the software build process than ever before.

We focus on the use of open source software tools in this edition of the book. It's wonderful that software developers have powerful operating systems and tools that are totally free and are available for exploring and altering. Open source solutions are very popular and provide tough competition for their commercial counterparts.

The Build Process

When build tools run on the same system as the program they produce, they can make a lot of assumptions about the system. This is typically not the case in embedded software development, where the build tools run on a *host* computer that differs from the *target* hardware platform. There are a lot of things that software development tools can do automatically when the target platform is well defined.[*] This automation is possible because the tools can exploit features of the hardware and operating system on which your program will execute. For example, if all of your programs will be executed on IBM-compatible PCs running Windows, your compiler can automate—and, therefore, hide from your view—certain aspects of the software build process. Embedded software development tools, on the other hand, can rarely make assumptions about the target platform. Instead, the user must provide some of her own knowledge of the system to the tools by giving them more explicit instructions.

The process of converting the source code representation of your embedded software into an executable binary image involves three distinct steps:

1. Each of the source files must be compiled or assembled into an object file.
2. All of the object files that result from the first step must be linked together to produce a single object file, called the relocatable program.
3. Physical memory addresses must be assigned to the relative offsets within the relocatable program in a process called relocation.

The result of the final step is a file containing an executable binary image that is ready to run on the embedded system.

The embedded software development process just described is illustrated in Figure 4-1. In this figure, the three steps are shown from top to bottom, with the tools that perform the steps shown in boxes that have rounded corners. Each of these development tools takes one or more files as input and produces a single output file. More specific information about these tools and the files they produce is provided in the sections that follow.

Each of the steps of the embedded software build process is a transformation performed by software running on a general-purpose computer. To distinguish this development computer (usually a PC or Unix workstation) from the target embedded system, it is referred to as the host computer. The compiler, assembler, linker, and locator run on a host computer rather than on the embedded system itself. Yet,

---

[*] Used this way, the term "target platform" is best understood to include not only the hardware but also the operating system that forms the basic runtime environment for your software. If no operating system is present, as is sometimes the case in an embedded system, the target platform is simply the processor on which your program runs.
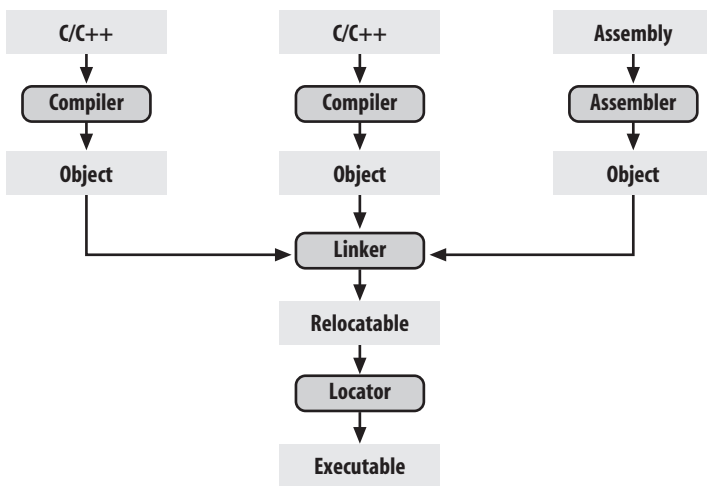
Figure 4-1. The embedded software development process

these tools combine their efforts to produce an executable binary image that will execute properly only on the target embedded system. This split of responsibilities is shown in Figure 4-2.


```
> compile foo.c
> assemble bar.asm
> link foo.o bar.o
> locate foo.exe
```

Host

The development tools that build the embedded software run on a general-purpose computer

Embedded System

Target

The embedded software that is built by those tools runs on the embedded system
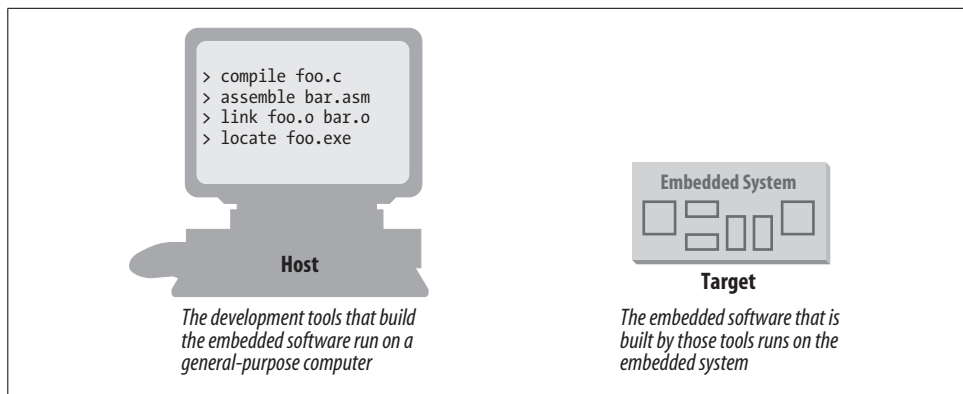
Figure 4-2. The split between host and target

In this book, we'll be using the GNU tools (compiler, assembler, linker, and debugger) for our examples. These tools are extremely popular with embedded software developers because they are freely available (even the source code is free) and support many of the most popular embedded processors. We will use features of these specific tools as illustrations for the general concepts discussed. Once understood, these same basic concepts can be applied to any equivalent development tool. The manuals for all of the GNU software development tools can be found online at *http:// www.gnu.org/manual*.

Compiling

The job of a *compiler* is mainly to translate programs written in some human-readable language into an equivalent set of opcodes for a particular processor. In that sense, an *assembler* is also a compiler (you might call it an "assembly language compiler"), but one that performs a much simpler one-to-one translation from one line of human-readable mnemonics to the equivalent opcode. Everything in this section applies equally to compilers and assemblers. Together these tools make up the first step of the embedded software build process.

Of course, each processor has its own unique machine language, so you need to choose a compiler that produces programs for your specific target processor. In the embedded systems case, this compiler almost always runs on the host computer. It simply doesn't make sense to execute the compiler on the embedded system itself. A compiler such as this—that runs on one computer platform and produces code for another—is called a *cross-compiler*. The use of a cross-compiler is one of the defining features of embedded software development.

The GNU C compiler (*gcc*) and assembler (*as*) can be configured as either native compilers or cross-compilers. These tools support an impressive set of host-target combinations. The *gcc* compiler will run on all common PC and Mac operating systems. The target processor support is extensive, including AVR, Intel x86, MIPS, PowerPC, ARM, and SPARC. Additional information about *gcc* can be found online at *http://gcc.gnu.org*.

Regardless of the input language (C, C++, assembly, or any other), the output of the cross-compiler will be an object file. This is a specially formatted binary file that contains the set of instructions and data resulting from the language translation process. Although parts of this file contain executable code, the object file cannot be executed directly. In fact, the internal structure of an object file emphasizes the incompleteness of the larger program.

The contents of an object file can be thought of as a very large, flexible data structure. The structure of the file is often defined by a standard format such as the Common Object File Format (COFF) or Executable and Linkable Format (ELF). If you'll be using more than one compiler (i.e., you'll be writing parts of your program in different source languages), you need to make sure that each compiler is capable of producing object files in the same format; *gcc* supports both of the file formats previously mentioned. Although many compilers (particularly those that run on Unix platforms) support standard object file formats such as COFF and ELF, some others produce object files only in proprietary formats. If you're using one of the compilers in the latter group, you might find that you need to get all of your other development tools from the same vendor.

Most object files begin with a header that describes the sections that follow. Each of these sections contains one or more blocks of code or data that originated within the

source file you created. However, the compiler has regrouped these blocks into related sections. For example, in *gcc* all of the code blocks are collected into a section called `text`, initialized global variables (and their initial values) into a section called `data`, and uninitialized global variables into a section called `bss`.

There is also usually a symbol table somewhere in the object file that contains the names and locations of all the variables and functions referenced within the source file. Parts of this table may be incomplete, however, because not all of the variables and functions are always defined in the same file. These are the symbols that refer to variables and functions defined in other source files. And it is up to the linker to resolve such unresolved references.

Linking

All of the object files resulting from the compilation in step one must be combined. The object files themselves are individually incomplete, most notably in that some of the internal variable and function references have not yet been resolved. The job of the linker is to combine these object files and, in the process, to resolve all of the unresolved symbols.

The output of the linker is a new object file that contains all of the code and data from the input object files and is in the same object file format. It does this by merging the `text`, `data`, and `bss` sections of the input files. When the linker is finished executing, all of the machine language code from all of the input object files will be in the `text` section of the new file, and all of the initialized and uninitialized variables will reside in the new `data` and `bss` sections, respectively.

While the linker is in the process of merging the section contents, it is also on the lookout for unresolved symbols. For example, if one object file contains an unresolved reference to a variable named `foo`, and a variable with that same name is declared in one of the other object files, the linker will match them. The unresolved reference will be replaced with a reference to the actual variable. For example, if `foo` is located at offset 14 of the output data section, its entry in the symbol table will now contain that address.

The GNU linker (*ld*) runs on all of the same host platforms as the GNU compiler. It is a command-line tool that takes the names of all the object files, and possibly libraries, to be linked as arguments. With embedded software, a special object file that contains the compiled startup code, which is covered later in this section, must also be included within this list. The GNU linker also has a scripting language that can be used to exercise tighter control over the object file that is output.

If the same symbol is declared in more than one object file, the linker is unable to proceed. It will likely complain to the programmer (by displaying an error message) and exit.

On the other hand, if a symbol reference remains unresolved after all of the object files have been merged, the linker will try to resolve the reference on its own. The reference might be to a function, such as `memcpy`, `strlen`, or `malloc`, that is part of the standard C library, so the linker will open each of the libraries described to it on the command line (in the order provided) and examine their symbol tables. If the linker thus discovers a function or variable with that name, the reference will be resolved by including the associated code and data sections within the output object file.[*] Note that the GNU linker uses *selective linking*, which keeps other unreferenced functions out of the linker's output image.

Unfortunately, the standard library routines often require some changes before they can be used in an embedded program. One problem is that the standard libraries provided with most software development tool suites arrive only in object form. You only rarely have access to the library source code to make the necessary changes yourself. Thankfully, a company called Cygnus (which is now part of Red Hat) created a freeware version of the standard C library for use in embedded systems. This package is called *newlib*. You need only download the source code for this library from the Web (currently located at *http://sourceware.org/newlib*), implement a few target-specific functions, and compile the whole lot. The library can then be linked with your embedded software to resolve any previously unresolved standard library calls.

After merging all of the code and data sections and resolving all of the symbol references, the linker produces an object file that is a special "relocatable" copy of the program. In other words, the program is complete except for one thing: no memory addresses have yet been assigned to the code and data sections within. If you weren't working on an embedded system, you'd be finished building your software now.

But embedded programmers aren't always finished with the build process at this point. The addresses of the symbols in the linking process are relative. Even if your embedded system includes an operating system, you'll probably still need an absolutely located binary image. In fact, if there is an operating system, the code and data of which it consists are most likely within the relocatable program too. The entire embedded application—including the operating system—is frequently statically linked together and executed as a single binary image.

Startup code

One of the things that traditional software development tools do automatically is insert *startup code*: a small block of assembly language code that prepares the way for the execution of software written in a high-level language. Each high-level language has its own set of expectations about the runtime environment. For example,

---

[*] We are talking only about static linking here. When dynamic linking of libraries is used, the code and data associated with the library routine are not inserted into the program directly.

programs written in C use a stack. Space for the stack has to be allocated before software written in C can be properly executed. That is just one of the responsibilities assigned to startup code for C programs.

Most cross-compilers for embedded systems include an assembly language file called *startup.asm*, *crt0.s* (short for C runtime), or something similar. The location and contents of this file are usually described in the documentation supplied with the compiler.

Startup code for C programs usually consists of the following series of actions:

1. Disable all interrupts.
2. Copy any initialized data from ROM to RAM.
3. Zero the uninitialized data area.
4. Allocate space for and initialize the stack.
5. Initialize the processor's stack pointer.
6. Call main.

Typically, the startup code will also include a few instructions after the call to main. These instructions will be executed only in the event that the high-level language program exits (i.e., the call to main returns). Depending on the nature of the embedded system, you might want to use these instructions to halt the processor, reset the entire system, or transfer control to a debugging tool.

Because the startup code is often not inserted automatically, the programmer must usually assemble it himself and include the resulting object file among the list of input files to the linker. He might even need to give the linker a special command-line option to prevent it from inserting the usual startup code. Working startup code for a variety of target processors can be found in a GNU package called *libgloss*.

Locating

The tool that performs the conversion from relocatable program to executable binary image is called a *locator*. It takes responsibility for the easiest step of the build process. In fact, you have to do most of the work in this step yourself, by providing information about the memory on the target board as input to the locator. The locator uses this information to assign physical memory addresses to each of the code and data sections within the relocatable program. It then produces an output file that contains a binary memory image that can be loaded into the target.

Whether you are writing software for a general-purpose computer or an embedded system, at some point the sections of your relocatable program must be assigned actual addresses. Sometimes software that is already in the target does this for you, as RedBoot does on the Arcom board.

In some cases, there is a separate development tool, called a *locator*, to assign addresses. However, in the case of the GNU tools, this feature is built into the linker (*ld*).

The memory information required by the GNU linker can be passed to it in the form of a *linker script*. Such scripts are sometimes used to control the exact order of the code and data sections within the relocatable program. But here, we want to do more than just control the order; we also want to establish the physical location of each section in memory.

What follows is an example of a linker script for the Arcom board. This linker script file is used to build the Blinking LED program covered in Chapter 3:

```
ENTRY (main)

MEMORY
{
    ram : ORIGIN = 0x00400000, LENGTH = 64M
    rom : ORIGIN = 0x60000000, LENGTH = 16M
}

SECTIONS
{
    data :                          /* Initialized data. */
    {
        _DataStart = . ;
        *(.data)
        _DataEnd   = . ;
```

```
    } >ram

    bss :                          /* Uninitialized data. */
    {
        _BssStart = . ;
        *(.bss)
        _BssEnd   = . ;
    } >ram

    text :                         /* The actual instructions. */
    {
        *(.text)
    } >ram
}
```

This script informs the GNU linker's built-in locator about the memory on the target board, which contains 64 MB of RAM and 16 MB of flash ROM.[*] The linker script file instructs the GNU linker to locate the data, bss, and text sections in RAM starting at address 0x00400000. The first executable instruction is designated with the ENTRY command, which appears on the first line of the preceding example. In this case, the entry point is the function main.

Names in the linker command file that begin with an underscore (e.g., _DataStart) can be referenced similarly to ordinary variables from within your source code. The linker will use these symbols to resolve references in the input object files. So, for example, there might be a part of the embedded software (usually within the startup code) that copies the initial values of the initialized variables from ROM to the data section in RAM. The start and stop addresses for this operation can be established symbolically by referring to the addresses as _DataStart and _DataEnd.

A linker script can also use various commands to direct the linker to perform other operations. Additional information and options for GNU linker script files can be found at *http://www.gnu.org*.

The output of this final step of the build process is a binary image containing physical addresses for the specific embedded system. This executable binary image can be downloaded to the embedded system or programmed into a memory chip. You'll see how to download and execute such memory images in the next chapter.

Building the Blinking LED Program

In this section, we show an example build procedure for the Arcom VIPER-Lite development board. If another hardware platform is used, a simlar process should be followed using the tools and conventions that accompany that hardware.

---

[*] There is also a version of the Arcom board that contains 32 MB of flash. If you have this version of the board, change the linker script file as follows:

```
rom : ORIGIN = 0x60000000, LENGTH = 32M
```

The installation procedure for the software development tools is provided in Appendix B. Once the tools are installed, the commands covered in the following sections are entered into a command shell. For Windows users, the command shell is a Cygwin bash shell (Cygwin is a Unix environment for Windows); for Linux users, it is a regular command shell.
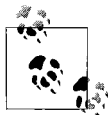
> In this and subsequent chapters, commands entered in a shell environment are indicated by the number sign (#) prompt. Commands entered in the RedBoot environment are indicated by the RedBoot prompt (RedBoot>).

We will next take a look at the individual commands in order to manually perform the three separate tasks (compiling, linking, and locating) described earlier in this chapter. Then we will learn how to automate the build procedure with makefiles.

Compile

As we have implemented it, the Blinking LED example consists of two source modules: *led.c* and *blink.c*. The first step in the build process is to compile these two files. The basic structure for the *gcc* compiler command is:

```
arm-elf-gcc [options] file...
```

The command-line options we'll need are:

-g
   To generate debugging info in default format

-c
   To compile and assemble but not link

-Wall
   To enable most warning messages

-I../include
   To look in the directory *include* for header files

Here are the actual commands for compiling the C source files:

```
# arm-elf-gcc –g -c –Wall -I../include led.c
# arm-elf-gcc -g –c –Wall -I../include blink.c
```

We broke up the compilation step into two separate commands, but you can compile the two files with one command. To use a single command, just put both of the source files after the options. If you wanted different options for one of the source files, you would need to compile it separately as just shown. For additional information about compiler options, take a look at *http://gcc.gnu.org*.

Running these commands will be a good way to verify that the tools were set up properly. The result of each of these commands is the creation of an object file that has the same prefix as the *.c* file, and the extension *.o*. So if all goes well, there will now be two additional files—*led.o* and *blink.o*—in the working directory. The compilation procedure is shown in Figure 4-3.
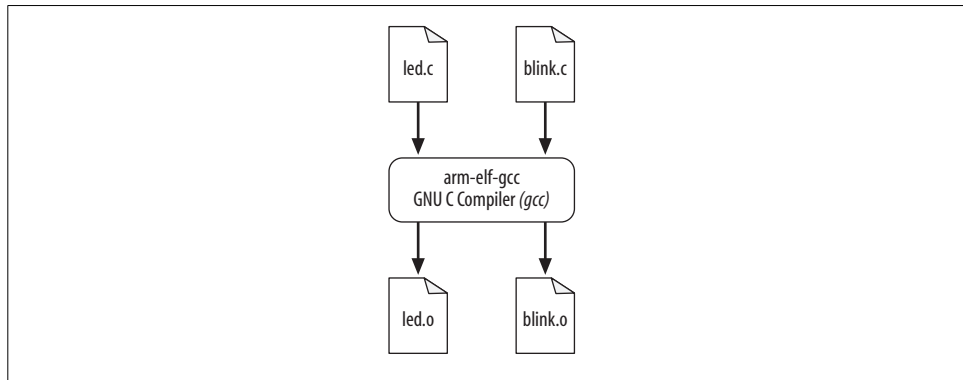
Figure 4-3. Compiling the Blinking LED program

Link and Locate

We now have the two object files—*led.o* and *blink.o*—that we need in order to perform the second step in the build process. As we discussed earlier, the GNU linker performs the linking and locating of the object files.

For the third step, locating, there is a linker script file named *viperlite.ld* that we input to *ld* in order to establish the location of each section in the Arcom board's memory. The basic structure for the linker and locater *ld* command is:

```
arm-elf-ld [options] file...
```

The command-line options we'll need for this step are:

-Map blink.map

   To generate a map file and use the given filename

-T viperlite.ld

   To read the linker script

-N

   To set the text and data sections to be readable and writable

-o blink.exe

   To set the output filename (if this option is not included, *ld* will use the default output filename *a.out*)

The actual command for linking and locating is:

```
# arm-elf-ld –Map blink.map –T viperlite.ld -N –o blink.exe led.o blink.o
```

The order of the object files determines their placement in memory. Because we are not linking in any startup code, the order of the object files is irrelevant. If startup code were included, you would want that object file to be located at the proper address. The linker script file can be used to specify where you want the startup routine (and other code) to reside in memory. Furthermore, you can also use the linker script file to specify exact addresses for code or data, should you find it necessary to do so.

As you can see in this command, the two object files—*led.o* and *blink.o*—are the last arguments on the command line for linking. The linker script file, *viperlite.ld*, is also passed in for locating the data and code in the Arcom board's memory. The result of this command is the creation of two files—*blink.map* and *blink.exe*—in the working directory. The linking and locating procedure is shown in Figure 4-4.
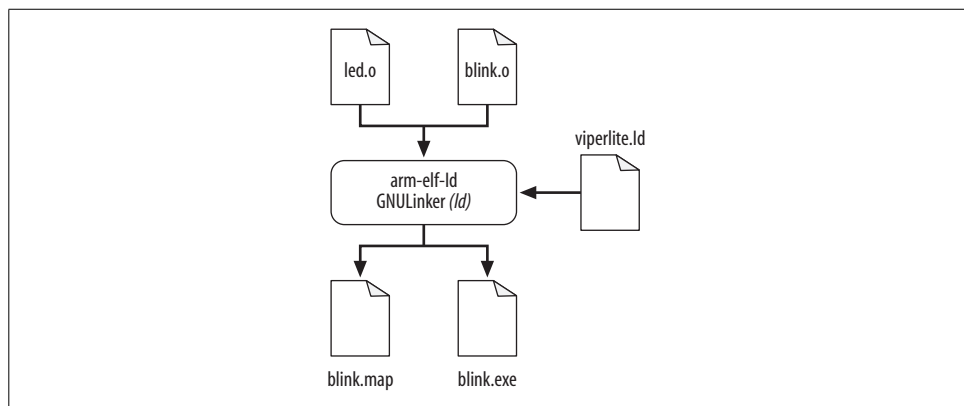
Figure 4-4. Linking and locating the Blinking LED program

The *.map* file gives a complete listing of all code and data addresses for the final software image. If you have never seen such a map file before, be sure to take a look at this one before reading on. It provides information similar to the contents of the linker script described earlier. However, these are results rather than instructions and therefore include the actual lengths of the sections and the names and locations of the public symbols found in the relocatable program. We'll see later how this file can be used as a debugging aid.

Format the Output File

The last step of the previous section creates an image of the Blinking LED program that we can load onto the Arcom board. In certain cases, you might need to format the image from the build procedure for your specific target platform.

One tool included with the GNU toolset that can assist with formatting images is the *strip* utility, which is part of the binary utilities package called *binutils* (pronounced

"bin-you-tills"). The *strip* utility can remove particular sections from an object file. The basic command structure for the *strip* utility is:

```
arm-elf-strip [options] input-file... [-o output-file]
```

The build procedure for subsequent chapters in the book generates two executable files: one with debug information and one without. The executable that contains the debug information includes *dbg* in its filename. The debug image should be used with *gdb*. If an image is downloaded with RedBoot, the nondebug image should be used.

The command used to strip symbol information is:

```
# arm-elf-strip --remove-section=.comment blinkdbg.exe -o blink.exe
```

This removes the section named .comment from the image *blinkdbg.exe* and creates the new output file *blink.exe*.

There might be another time when you need an image file that can be burned into ROM or flash. The GNU toolset has just what you need for this task. The utility *objcopy* (object copy) is able to copy the contents of one object file into another object file. The basic structure for the *objcopy* utility is:

```
arm-elf-objcopy [options] input-file [output-file]
```

For example, let's suppose we want to convert our Blinking LED program from ELF format into an Intel Hex Format file.[*] The command line we use for this is:

```
# arm-elf-objcopy -O ihex blink.exe blink.hex
```

This command uses the *–O ihex* option to generate an Intel Hex Format file. The input file is *blink.exe* (the *objcopy* utility determines the input file type). Finally, the output file is named *blink.hex*.

---

[*] Intel Hex format is an ASCII file format devised by Intel for storing and downloading binary images.
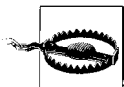
If no output filename is given, the *strip* and *objcopy* utilities overwrite the original input file with the generated file.

Some of the other GNU tools are useful for providing other information about the image you have built. For example, the *size* utility, which is part of the *binutils* package, lists the section sizes and total size for a given object file. Here is the command for using the size utility:

```
# arm-elf-size blink.exe
```

The resulting output is:

```
   text    data     bss     dec     hex filename
    328       0       0     328     148 blink.exe
```

The top row consists of column headings and shows the sections `text`, `data`, and `bss`. The Blinking LED program contains 328 bytes in the `text` section, no bytes in the `data` section, and no bytes in the `bss` section. The `dec` column shows the total image size in decimal, and the `hex` column shows it in hexadecimal (decimal 328 = hexadecimal 0x148). These total sizes are in bytes. The last column, `filename`, contains the filename of the object file.

You will notice that the size of the section, 328 bytes, is much smaller than the approximately 3 KB file size of our *blink.exe*. This is because debugging information is located also in the *blink.exe* file.

Additional information about the other GNU *binutils* can be found online at *http://www.gnu.org*.

We're now ready to download the program to our development board, which we'll do in the next chapter. To wrap up our discussion of building programs, let's take a quick look at another useful tool in the build process.

A Quick Look at Makefiles

You can imagine how tedious the build process could be if you had a large number of source code files for a particular project. Manually entering individual compiler and linker commands on the command line becomes tiresome very quickly. In order to avoid this, a *makefile* can be used. A makefile is a script that tells the *make* utility how to build a particular program. (The *make* utility is typically installed with the other GNU tools.) The *make* utility follows the rules in the makefile in order to automatically generate output files from a set of input source files.

Makefiles might be a bit of a pain to set up, but they can be a great timesaver and a very powerful tool when building project files over and over (and over) again. Having a sample available can reduce the pain of setting up a makefile.

The basic layout for a makefile build rule is:

```
target:    prerequisite
           command
```

The target is what is going to be built, the prerequisite is a file that must exist before the target can be created, and the command is a shell command used to create the target. There can be multiple prerequisites on the target line (separated by white space) and/or multiple command lines. But be sure to put a tab, not spaces, at the beginning of every line containing a command.

Here's a makefile for building our Blinking LED program:

```
XCC     = arm-elf-gcc
LD      = arm-elf-ld
CFLAGS  = -g -c -Wall \
          -I../include
LDFLAGS = -Map blink.map -T viperlite.ld -N

all: blink.exe

led.o: led.c led.h
    $(XCC) $(CFLAGS) led.c

blink.o: blink.c led.h
    $(XCC) $(CFLAGS) blink.c

blink.exe: blink.o led.o viperlite.ld
    $(LD) $(LDFLAGS) -o $@ led.o blink.o

clean:
    -rm -f blink.exe *.o blink.map
```

The first four statements in this makefile contain variables for use in the makefile. The variable names are on the left side of the equal sign. In this makefile, the respective variables do the following:

XCC

Defines the compiler executable program

LD

Defines the linker executable program

CFLAGS

Defines the flags for the compiler

LDFLAGS

Defines the flags for the linker

Variables in a makefile are used to eliminate some of the duplication of text as well as to ease portability. In order to use a variable in the code, the syntax $( ) is used with the variable name enclosed in the parentheses.

Note that if a line in a makefile gets too long, you can continue it on the following line by using the backslash (\), as shown with the CFLAGS variable.

Now for the build rules. The build targets in this file are all, led.o, blink.o, and blink.exe. Unless you specify a target when invoking the *make* utility, it searches for the first target (in this case, the first target is all) and tries to build it; this, in turn, can lead to it finding and building other targets. The *make* utility creates (or re-creates, as the case may be) the target file if it does not exist or if the prerequisite files are more recent than the target file.

At this point, it might help to look at the makefile from the bottom up. In order for blink.exe to be created, blink.o and led.o need to be built as shown in the prerequisites. However, since these files don't exist, the *make* utility will need to create them first. It will search for ways to create these two files and will find them listed as targets in the makefile. The *make* utility can create these files because the prerequisites (the source files) for these two targets exist.

Because the targets led.o and blink.o are handled similarly, let's focus on just one of them. The prerequisites for the target led.o are led.c and led.h. As stated above, the command tells the *make* utility how to create the target. The first part of the command for led.o is a reference to the variable XCC, as indicated by the syntax $(XCC), and the next part of the command is a reference to the variable CFLAGS, as indicated by the syntax $(CFLAGS). The *make* utility simply replaces variable references with the text assigned to them in the makefile. The final part of the command is the source file led.c. Strung together, these elements construct the command that the *make* utility executes. This generates a command on the shell command line as follows:

```
arm-elf-gcc -g -c -Wall -I../include led.c
```

This is the same command we entered by hand in order to compile the *led.c* file earlier in this chapter, in the section "Building the Blinking LED Program." The *make* utility compiles *blink.c* in the same way.

At this point, the *make* utility has all of the prerequisites needed to generate the target blink.exe default target. The command that the *make* utility executes (the same command we entered by hand to link and locate the Blinking LED program) to build blink.exe is:

```
arm-elf-ld -Map blink.map -T viperlite.ld -N -o blink.exe led.o blink.o
```

You may notice that in this makefile the linker is invoked directly. Instead, *gcc* could have been used to invoke the linker indirectly with the following line:

```
arm-elf-gcc -Wl,-Map,blink.map -T viperlite.ld -N -o blink.exe led.o blink.o
```

When invoking the linker indirectly, the special option *−Wl* is used so that *gcc* passes the request to generate a linker map file to the linker rather than trying to parse the argument itself. While this simple Blinking LED program does not need to link using

gcc, you should remember that more complex C programs may need special runtime library support from *gcc* and will need to be linked in this way.

The last part of the makefile is the target `clean`. However, because it was not needed for the default target, the command was not executed.

To execute the makefile's build instructions, simply change to the directory that contains the makefile and enter the command:

```
# make
```

The *make* utility will search the current directory for a file named *makefile*. If your makefile has a different name, you can specify that on the command line following the *-f* option.

With the previous command, the *make* utility will make the first target it finds. You can also specify targets on the command line for the *make* utility. For example, because `all` is the default target in the preceding makefile, you can just as easily use the following command:

```
# make all
```

A target called `clean` is typically included in a makefile, with commands for removing old object files and executables, in order to allow you to create a fresh build. The command line for executing the `clean` target is:

```
# make clean
```

Keep in mind that we've presented a very basic example of the *make* utility and makefiles for a very basic project. The *make* utility contains very powerful tools within its advanced features that can benefit you when executing large and more complex projects.
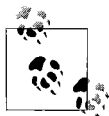
> It is important to keep the makefile updated as your project changes. Remember to incorporate new source files and keep your prerequisites up to date. If prerequisites are not set up properly, you might change a particular source file, but that source file will not get incorporated into the build. This situation can leave you scratching your head.

Additional information about the GNU *make* utility can be found online at *http://www.gnu.org* as well as in the book *Managing Projects with GNU make,* by Robert Mecklenburg (O'Reilly). These resources will give you a deeper understanding of both the *make* utility and makefiles and allow you to use their more powerful features.

Downloading and Debugging

*I can remember the exact instant when I realized that
a large part of my life from then on was going to be
spent in finding mistakes in my own programs.*

—Maurice Wilkes, Head of the Computer Laboratory of
  the University of Cambridge, 1959

Once you have an executable binary image stored as a file on the host computer, you will need a way to download that image to the embedded system and execute it. The executable binary image is usually loaded into a memory device on the target board and executed from there. And if you have the right tools at your disposal, it will be possible to set breakpoints in the program or to observe its execution in less intrusive ways. This chapter describes various techniques for downloading, executing, and debugging embedded software in general, as well as focuses on the techniques available on our development environment.

Downloading the Blinking LED Program

With most embedded systems, there are several means to get an image onto the target and run the program, some more challenging than others. In this section, we investigate the methods available for downloading the Blinking LED program onto the Arcom board, as well as some other methods that may be useful for other projects.

The software development cycle for a PC and an embedded system include many of the same stages. Figure 5-1 is a general diagram of the embedded software development cycle.

As shown in Figure 5-1, the software development cycle begins with design and the first implementation of the code. After that, there are usually iterations of the build, download and debug, and bug-fixing stages. Because a lot of time is spent in these three stages, it helps to eliminate any kinks in this process so that the majority of time can be spent on debugging and testing the software. (This diagram does not
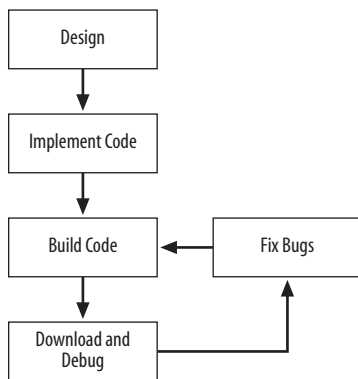
Figure 5-1. Software development cycle

take into account the handling of feature creep inevitably inflicted by the marketing department.)

Because this is a very basic diagram, other stages that may be necessary are *profiling* and *optimization*. Profiling allows a developer to determine various metrics about a program, such as where the processor is spending most of its time. Optimization is the process by which the developer tries to eliminate bottlenecks in software using various techniques, such as implementing time-critical code in assembly language. Very often, optimization techniques are compiler-, processor-, or system-specific.

Another task at this stage is *integration*. Once the development cycle is complete, system-level testing is typically done. And after a product ships, the software enters its maintenance phase for the duration of the product's life cycle, when the code must be supported and sustained. The debugging techniques shown in this chapter apply to the maintenance stage as well.

Because code must be repeatedly tested on the target hardware, a quick and straightforward method for loading software onto the target is ideal. Let's take a look at the techniques and tools we can use for this task.

Debug Monitors

A *debug monitor*, also called a *ROM monitor*, is a small program that resides in nonvolatile memory on the target hardware that facilitates various operations needed during development. One of the tasks that a debug monitor handles is basic hardware initialization. A debug monitor allows you to download and run software in RAM to debug the program.

Most debug monitors include many other useful features to assist in the development cycle. For example, most debug monitors incorporate some kind of *command-line interface* (CLI) where commands can be issued to the debug monitor for

execution on the target hardware—e.g., via serial port. These commands include downloading software, running the program, *peeking* (reading) and *poking* (writing) memory and processor registers, comparing or displaying blocks of memory, and setting initialization configurations for the hardware.

In some systems, a debug monitor is incorporated in the production units in the field as well. This can be used to update the firmware to add new features or fix bugs after the unit is deployed.

Some processors include a program similar to a debug monitor in on-chip memory. For example, the TMS320C5000 series DSPs from Texax Instruments include a program called a *bootloader*. This bootloader is used to transfer software from an external source (off-chip) to internal memory (on-chip), allowing code to reside in slower memory for storage and be transferred into faster memory prior to execution. The bootloader determines where to load code based upon the boot mode setting, which is determined by sampling particular DSP I/O pins during power up.
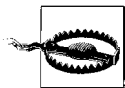
> Be aware that the execution speeds of RAM and ROM typically are very different. Code running from RAM typically executes much faster than code running from ROM, which could cause the software to behave differently. Certain types of bugs, such as timing errors, may only reveal themselves when run from just one type of memory.

RedBoot

The Arcom board includes a debug monitor called RedBoot, which is described in the sidebar "Debug Monitors" in Chapter 4. RedBoot resides in the bootloader flash on the Arcom board and uses the board's serial port COM1 for its command-line interface. The *VIPER Technical Manual* and *VIPER-I/O Technical Manual* describe how to connect the various modules to the Arcom board.

Once you have the Arcom board connected properly, you need to connect the Arcom board's COM1 port to a serial port on your computer using the cable included in the development kit.

To communicate with RedBoot's CLI, you need to run a terminal program (minicom in Linux or HyperTerminal in Windows will do just fine) on your computer. Set the serial port settings as follows:

- Baud rate: 115200
- Data bits: 8
- Parity: None
- Stop bits: 1
- Flow control: None

Now you are ready to power on the Arcom board.
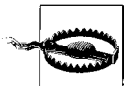
Redboot executes a script after it runs to automatically boot embedded Linux (if present on your version of the Arcom board). In order to prevent the script from running, hit the keys Ctrl and C together when RedBoot outputs its initialization message.

If all connections are properly made, an initialization message is output from the Arcom board's COM1 port once power is applied, along with the RedBoot prompt, which looks like this:

```
Ethernet eth0: MAC address 00:80:12:1c:89:b6
No IP info for device!

RedBoot(tm) bootstrap and debug environment [ROM]
Non-certified release, version W468 V3I7 - built 10:11:20, Mar 15 2006

Platform: VIPER (XScale PXA255)
Copyright (C) 2000, 2001, 2002, 2003, 2004 Red Hat, Inc.

RAM: 0x00000000-0x04000000, [0x00400000-0x03fd1000] available
FLASH: base 0x60000000, size 0x02000000, 256 blocks of 0x00020000 bytes each.
== Executing boot script in 1.000 seconds - enter ^C to abort
^C
RedBoot>
```

Because we have not entered an Internet Protocol (IP) address for the Arcom board, RedBoot outputs the message: No IP info for device! This message can be ignored for now. Another thing to notice is that we have stopped the boot script from running (and loading Linux) by entering Ctrl-C (shown in the preceding code as ^C) when RedBoot is started.

The RedBoot initialization message contains information regarding the build and version of the RedBoot image. The available memory is also listed before the prompt, including the RAM and flash memory address ranges.

You can enter help at the prompt to get a list of the supported commands. A description of the RedBoot commands can be found online at *http://ecos.sourceware.org*.

Downloading with RedBoot

Now that RedBoot is up and running, we are ready to download and run the Blinking LED program. RedBoot is able to load and run ELF files. Therefore, we use the *blink.exe* file built in Chapter 4 as the program image to run on the Arcom board.

To initiate the download, enter the following load command at the RedBoot prompt:

```
RedBoot> load -m xmodem
```

This tells RedBoot to load an image using the *xmodem* protocol as the method. After you press the Enter key, RedBoot begins to output the character C while waiting for the file to be sent over.

To begin the file transfer using Windows HyperTerminal, select Transfer → Send File... from the menu (use a similar command if you have a different terminal program). This brings up the Send File dialog box; select Xmodem for the protocol. Browse to the location of the *blink.exe* program and select it. Then click Send. A transfer statistics dialog box will be displayed showing the status of the file transfer.

Once the transfer has successfully completed, RedBoot outputs a message similar to the following:

```
Entry point: 0x00400110, address range: 0x00000024-0x0040014c
xyzModem - CRC mode, 24(SOH)/0(STX)/0(CAN) packets, 2 retries
```

This shows the entry point of the program—in this case, 0x00400110. If you refer to the map file generated by the linker during the build process, *blink.map*, the entry point address should look familiar, as shown in this portion of the map:

```
Name            Origin          Length

.text           0x004000b0      0x9c        blink.o
                0x00400110                      main
```

The map file shows that the routine main resides at 0x00400110, which is the entry point for execution of the Blinking LED program. The value 0x9C is the total length of the object file *blink.o*.

Running programs with RedBoot

Now we can run the program. Enter the following command at the RedBoot prompt:

```
RedBoot> go
```

After you press Enter, RedBoot hands control of the Arcom board over to the Blinking LED program. If everything is successful, you should now see the green LED blinking on the add-on board.

You have just successfully completed your first pass through the embedded software development cycle.

RedBoot Networking Support

You may notice that the Arcom board contains an Ethernet port. RedBoot includes a networking stack to allow communications over this port. For example, you can open up a Telnet session and communicate with RedBoot over the Ethernet port. RedBoot also supports downloading software using the *Trivial File Transfer Protocol* (TFTP). We leave the investigation of these advanced RedBoot features as an exercise for you.

When in ROM…

Another way to download embedded software is to load the binary image into a ROM device and physically insert that chip into a socket on the target board. Obviously, the contents of a truly read-only memory device could not be overwritten. However, as you'll see in Chapter 6, embedded systems commonly employ special read-only memory devices that can be programmed (or reprogrammed) with the help of a special piece of equipment called a *device programmer* or *burner*. A device programmer is a computer system that has one or more IC sockets on the top—of varying shapes and sizes—and is capable of programming memory devices of all sorts.

In an ideal development scenario, the device programmer would be connected to the same network as the host computer. That way, files that contain executable binary images could be easily transferred to it for ROM programming. After the binary image has been transferred to the device programmer, the memory chip is placed into a socket of the appropriate size and shape, and the device type is selected from an on-screen menu. The actual device programming process can take anywhere from a few seconds to several minutes, depending on the size of the binary image, the type of memory device you are using, and the quality and speed of your device programmer.

After you program the ROM, it is ready to be inserted into its socket on the board. Of course, this shouldn't be done while the embedded system is still powered on. The power should be turned off and then reapplied only after the chip has been carefully inserted.
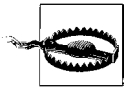
> Care should be taken when removing and inserting any part that is socketed. Pins can become bent with surprising ease, and a bent or broken pin can cause all sorts of problems that are difficult to debug.

As soon as power is applied, a processor will begin to fetch and execute the code that is stored inside the ROM. However, be aware that each type of processor has its own rules about the location of its first instruction. For example, when the ARM processor is reset, it begins by fetching and executing whatever is stored at physical address 0x00000000. This is called the *reset address*, and the instructions located there are collectively known as the *reset code*. In the case of the Arcom development board, the reset code is part of the RedBoot debug monitor.

If your program doesn't appear to be working, something could be wrong with your reset code. You must always ensure that the binary image you've loaded into the ROM satisfies the target processor's reset rules. During product development, we often find it useful to turn on one of the board's LEDs just after the reset code has been completed. That way, we know at a glance that any new code either does or doesn't satisfy the processor's most basic requirements.

Managing ROM with RedBoot

The Arcom board includes a type of memory called *flash*, which is in-circuit pro-
grammable. Even when socketed for easy removal, flash memory does not have to be
removed from the board to be reprogrammed. The RedBoot debug monitor includes
software that can perform the device programming function.

RedBoot also contains several commands to manage a flash filesystem, called the
Flash Image System (FIS). The FIS allows you to specify regions in flash, similar to a
filesystem on a hard disk drive. Using the FIS, you can create, write, and erase loca-
tions of flash based on "filenames" you select.
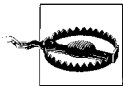
> Be extremely careful not to corrupt the existing images or configura-
> tion data residing in flash on the Arcom board. If this happens, you
> could render the board unusable.

To see what is contained in the FIS, enter the following command:

```
RedBoot> fis list
```

which will output a listing similar to this one:

```
Name            FLASH addr  Mem addr    Length      Entry point
FIS directory   0x00000000  0x00000000  0x0001F000  0x00000000
RedBoot config  0x0001F000  0x00000000  0x00001000  0x00000000
filesystem      0x00020000  0x00000000  0x01FE0000  0x00000000
```

This list command shows the images currently available in the RedBoot FIS. There
are a few other FIS commands supported by RedBoot. For details on the other FIS-
related commands and options, refer to the RedBoot User's Guide online at *http://
ecos.sourceware.org*.

You have some decisions to make when deciding how and where to download a pro-
gram to the hardware. The biggest disadvantage of using flash memory for down-
loads is that there is no easy way to debug software that is executing out of flash
memory or ROM. When single-stepping or executing to a breakpoint, the debugger
replaces the subsequent instruction with a software interrupt, which is used to halt
the processor's execution. Thus, a debugger doesn't work in any form of read-only
memory, such as flash. Of course, you can still examine the state of the LEDs and
other externally visible hardware, but this will never provide as much information
and feedback as a debugger. So, flash might be fine once you know that your soft-
ware works and you're ready to deploy the system, but it's not very helpful during
software development.

Some processors can work around the issue of executing out of flash or ROM. In
some cases, the processor includes a TRACE instruction that executes a single
instruction and then automatically vectors to an interrupt. On other processors, a
breakpoint register gets you back to the debug monitor.

Remote Debuggers

If available, a *remote debugger* can be used to download, execute, and debug embedded software over a serial port or network connection between the host and target (also called *cross-platform debugging*). The program running on the host of a remote debugger has a user interface that looks just like any other debugger that you might have used. The main debugger screen is usually either a command-line interface or graphical user interface (GUI). GUI debuggers typically contain several smaller windows to simultaneously show the active part of the source code, current register contents, and other relevant information about the executing program.

Note that in the case of embedded systems, the debugger and the software being debugged are executing on two different computer systems. Remote debugger software runs on the host computer and provides the user interface just described. But there is also a backend component that runs on the target processor and communicates with the host debugger frontend over a communications link. The debugger backend provides low-level control of the target processor. Figure 5-2 shows how these two components work together.
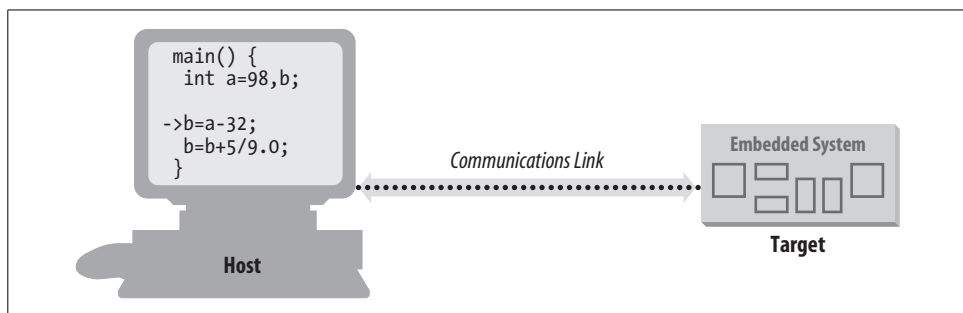
Figure 5-2. Components of a remote debug session

The debug monitor resides in ROM—having been placed there either by you or at the factory—and is automatically started whenever the target processor is reset. It monitors the communications link to the host computer and responds to requests from the remote debugger host software. Of course, these requests and the monitor's responses must conform to some predefined communications protocol and are typically of a very low-level nature. Examples of requests the host software can make are "read register x," "modify register y," "read n bytes of memory starting at address z," and "modify the data at address a." The remote debugger combines sequences of these low-level commands to accomplish complex debugging tasks such as downloading a program, single-stepping, and setting breakpoints.

It is helpful to build the program being tested to include symbolic debug information, which we did with the –g option during the compilation step of the build procedure in Chapter 4. The –g option causes the compiler to place additional information in the object file for use by the debugger. This debug information allows the debugger to relate between the executable program and the source code.

One such debugger is the GNU debugger (*gdb*). Like the other GNU tools, it was originally designed for use as a native debugger and was later given the ability to perform remote debugging. The *gdb* debug monitor that runs on the target hardware is called a *gdb stub*. Additional information about *gdb* can be found online at *http://sources.redhat.com/gdb*.

The GNU software tools include *gdb*. The version installed is CLI-based, so there are a few commands to learn in order to run the debugger properly. There are several GUIs available for *gdb*, such as Insight (*http://sources.redhat.com/insight*) and DataDisplay Debugger (*http://www.gnu.org/software/ddd*).

RedBoot contains a *gdb*-compatible debug monitor. Therefore, once a host attempts to communicate with the target using the *gdb* protocol, RedBoot turns control of the target over to the *gdb* stub for the debug session.

As described earlier, the host and target use a predefined protocol. For *gdb*, this protocol is the ASCII-based Remote Serial Protocol. To learn more about the *gdb* Remote Serial Protocol, go to *http://sources.redhat.com/gdb*. Another good resource for information about *gdb* is *Debugging with GDB: The GNU Source-Level Debugger*, by Richard Stallman, Roland Pesch, and Stan Shebs (Free Software Foundation).[*]

Remote debuggers are one of the most commonly used downloading and testing tools during development of embedded software. This is mainly because of their low cost. Embedded software developers already have the requisite host computer. In addition, the price of a remote debugger does not add significantly to the cost of a suite of cross-development tools (compiler, linker, locator, etc.).

However, there are some disadvantages to using a debug monitor, including the inability to debug startup code. Another disadvantage is that code must execute from RAM. Furthermore, when using a debug monitor, a communication channel must exist to interface the target to the host computer.

Debugging on the Arcom Board

gdb is able to operate over serial or TCP/IP network ports. RedBoot also supports *gdb* debug sessions over either of these ports. For the example that follows, we use the serial port. We then cover some of the basic *gdb* commands that are in the example.

---

[*] This document is included in electronic form on the Arcom VIPER-Lite Development Kit CD-ROM.

In order to demonstrate some additional debug capabilities, we have added a global variable, gChapter, to the Blinking LED program.

To prepare for the debugging examples, cycle power on the Arcom board and halt the RedBoot boot script by pressing Ctrl-C. Once the RedBoot initialization message is output, you're ready to start.

Invoke *gdb*, passing the name of the program to debug as an argument, by using the following command:

```
# arm-elf-gdb blink.exe
```

gdb outputs a message similar to this one:

```
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=arm-elf"...
```

If you use the wrong executable that does not contain debugging information with *gdb*, the following message is output:

```
(no debugging symbols found)
```

There should also be a (*gdb*) prompt waiting for input. Next, issue the command to have *gdb* connect to the Arcom board. The following command assumes that the computer's serial port that is connected to the target board is COM1 (if a different PC serial port is used, modify the command accordingly):

```
(gdb) target remote /dev/ttyS0
```

GDB Connection Problems

Because the same computer serial port is being used for *gdb* and RedBoot communications, make sure another program (such as the terminal program you used to download the code) has not opened the port. If another program has control over the computer's serial port, *gdb* will not be able to connect to the target Arcom board. You should also verify that the host computer is connected to the correct serial port on the Arcom board.

After *gdb* successfully connects to the target, a response similar to this one will follow:

```
Remote debugging using /dev/ttyS0
```

The host computer running the *gdb* command-line interface is now connected to the *gdb* stub residing on the target hardware within RedBoot.

Next download the blink.exe program onto the target with the command:

```
(gdb) load blink.exe
```

When program loading completes successfully, a message similar to this one is output from *gdb*:

```
Loading section data, size 0x4 lma 0x400000
Loading section text, size 0x148 lma 0x400004
Start address 0x400110, load size 332
Transfer rate: 2656 bits in <1 sec, 166 bytes/write.
```

Now you are ready to start debugging!

Let's start by setting a *breakpoint* in the code. A breakpoint is an address or condition where the debugger will halt execution of the program. The debugger sets a breakpoint by replacing a given instruction with a software interrupt (that gets sent back to the debugger). Removing a breakpoint removes the software interrupt and restores the preplaced instruction.

Use the following command to set a breakpoint that is hit when the routine ledToggle is called:

```
(gdb) b ledToggle
```
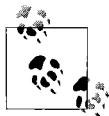
> *gdb* commands are not case-sensitive (though symbols are) and can be abbreviated to the shortest unique string. For example, you can set a breakpoint with any of these commands:
>
> ```
> (gdb) breakpoint ledToggle
> (gdb) break ledToggle
> (gdb) br ledToggle
> (gdb) b ledToggle
> ```
>
> Each command accomplishes the same goal—i.e., setting a breakpoint at the routine ledToggle. The RedBoot CLI commands operate similarly.

After successfully setting the breakpoint, *gdb* responds with information about the breakpoint as follows:

```
Breakpoint 1 at 0x400070: file led.c, line 66.
```

The response shows the breakpoint number (1 in this case), the address of the breakpoint (0x400070), the file where the function is located (*led.c*), and the line number in that file where the breakpoint is set (66).

If you need to check which breakpoints are set within a program for a *gdb* session, you can use the command:

```
(gdb) info b
```

The response from *gdb* is something like this:

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00400070 in ledToggle at led.c:66
```

Next, run the program by entering the continue command:

```
(gdb) c
```

Once *gdb* hits the breakpoint, it will halt execution and output the source code line that is to be executed next:

```
Breakpoint 1, ledToggle () at led.c:66
66              if (GPIO_0_LEVEL_REG & LED_GREEN)
```

The first line of output shows the breakpoint that was hit, along with the description. The second line shows the line number in the file, 66 in this case, with the source code for that line. The green LED should be lit now because that is its initial state in the program.

To have *gdb* show the source code where the current program is stopped, use the list command:

```
(gdb) l
```

This will dump, by default, 10 source code lines. To dump the next 10 source code lines, simply enter the list command again.

> To repeat the last command in a *gdb* session, simply press Enter.

A useful feature of the remote debugger is that it can check symbol values. The command to show the value of the gChapter variable is:

```
(gdb) print /x gChapter
```

The */x* option formats the output for hexadecimal. The response from *gdb* is:

```
$1 = 0x5
```

The $1 is a special history number that *gdb* assigns, and *gdb* allows reference back to this value at a later time. The current value of gChapter is 5.

The print command can also be used to change the value of a symbol. In the following command, the variable gChapter is changed to 12:

```
(gdb) p/x gChapter=12
```

The response from *gdb* is:

```
$2 = 0xc
```

This shows that the new value of gChapter is 0xC (12 decimal).

Debug Tip: Using the Memory Map for Symbol Value Lookup

Debug symbols associate variable and function names with their addresses, as well as include type information about the symbol. This allows you to reference a particular variable by using its symbol name.

There may be a time when you are unable to use these symbols for one reason or another; for example, perhaps you need to debug code released by another party and there is no debug information contained in that program file.

When symbol information is not available, hope is not lost for obtaining additional information using a debugger. The map file can be used to manually look up the symbol addresses. For example, the *blink.map* file shows the address of the variable gChapter as 0x00400000 with a length of 4 bytes:

```
    .data        0x00400000      0x4 blink.o
                 0x00400000              gChapter
```

Given the address of a symbol, you can use *gdb* to find out the current value of that symbol by using the command:

```
(gdb) x/d 0x400000
```

The command x stands for "examine." It can be followed immediately by an option specifying the format in which to display the data (here, */d* for decimal) and then the address of memory to read (here 0x400000). In this case, *gdb* responds with the following output:

```
0x400000 <gChapter>:    12
```

This shows the current value of gChapter as 12 (this value was set in the previous command). This allows you to peek and poke variables without having the symbol information.

Another very useful feature of a remote debugger is the ability to step through lines of source code, an action commonly called *single-stepping*. There are several different types of single-step commands, such as stepping a single machine instruction and stepping a single source code line. The following command, next, steps a single source code line:

```
(gdb) n
```

When debugging, it is also important to realize that using compiler optimization can affect the behavior of the code in the debug session. Most compilers have an option for enabling optimization. For example, with *gcc*, the *–O* option invokes optimization. This option has various levels, which are indicated by a number (0, 1, 2, or 3) that follows the option switch—for example, *–O2*. The optimization switch *–Os* optimizes for size. By default, no optimization is selected. Details about the specific optimizations for each level can be found in the *gcc* documentation located online at *http://gcc.gnu.org/onlinedocs*.

The reason compiler optimization needs to be considered when debugging is because the compiler can reorder code and remove entire sections of code and/or variables without telling you. It is for this reason that most debugging is done with optimization turned off. Keep this in mind when single-stepping source code.

The source code line to be executed next is output by *gdb* as shown here:

```
69              GPIO_0_SET_REG = LED_GREEN;
```

gdb provides two commands for stepping one line at a time: step and next. The difference between them is that when you reach the start of a function call, step enters the function and runs the first statement within the function, whereas next runs the whole function.

Now run the program again with the continue command:

```
(gdb) c
```

You might notice that the LED is now off.

Execution will halt once again when *gdb* encounters the breakpoint at ledToggle.

gdb allows you to check which functions have executed by using the backtrace or bt command. The backtrace command shows how your program got to where it currently is. Enter the backtrace command as follows:

```
(gdb) bt
```

The *gdb* output looks something like this:

```
#0  ledToggle () at led.c:66
#1  0x00400140 in main () at blink.c:75
```

The response from *gdb* shows the most recently executed function (indicated by #0), followed by the function that called it (indicated by #1), and so on. The preceding response shows that the routine main called the routine ledToggle.

With *gdb* you can also view the processor's register values:

```
(gdb) info registers
```

To print the value of a specific register, use the command:

(gdb) `p/x $pc`

This command outputs the current value of the program counter register in hexadecimal format.

You should now have a good understanding of how to use *gdb*. We've examined the most important commands, but it may be helpful to play around with some others at this point.

In order to remove the breakpoint set earlier, use the `delete` command:

(gdb) `d`

Then confirm the removal of all breakpoints. You can now continue running the program and watch the LED blink.

To halt execution, press Ctrl-C. However, the program may not respond to this command. If this is the case, *gdb* will ask if you want to stop debugging the program.

One disadvantage of a command-line interface debugger is that commands need to be learned. However, once you get the hang of it, you'll find it's quite easy to maneuver around.

Additional *gdb* commands can be found by using the *gdb* `help` command. Assistance with a specific command can be obtained by entering `help` followed by the command name (e.g., `help breakpoint`). There is also a *gdb* quick-reference guide available at several sites online.

Emulators

An *in-circuit emulator* (*ICE*) provides a lot more functionality than a remote debugger. In addition to providing the features available with a remote debugger, an ICE allows you to debug startup code and programs running from ROM, set breakpoints for code running from ROM, and even run tests that require more RAM than the system contains.

An ICE typically takes the place of—or emulates—the processor on your target board. (Some emulators come with an adapter that clips over the processor on the target.) The ICE is itself an embedded system, with its own copy of the target processor, RAM, ROM, and embedded software. In-circuit emulators are usually pretty expensive. But they are powerful tools, and in a tight spot, nothing else will help you get the debug job done better.

Like a debug monitor, an emulator uses a remote debugger for its host interface. In some cases, it is even possible to use the same debugger frontend for both. But because the emulator has its own copy of the target processor, it is possible to monitor and control the state of the processor in real time. This allows the emulator to

support such powerful debug features as hardware breakpoints and real-time tracing. Additional information about in-circuit emulators can be found in the November 2001 *Embedded Systems Programming* article "Introduction to In-Circuit Emulators," which can be found online at *http://www.embedded.com*.

With a debug monitor, you can set breakpoints in your program. However, these software breakpoints are restricted to instruction fetches—the equivalent of the command "stop execution if this instruction is about to be fetched." Emulators, by contrast, also support *hardware breakpoints*. Hardware breakpoints allow you to stop execution in response to a wider variety of events—not only instruction fetches, but also interrupts and reads and writes of memory. For example, you might set a hardware breakpoint on the event "address bus = 0x2034FF00 and data bus = 0x20310000."

Another useful feature of an in-circuit emulator is *real-time tracing*. Typically, an emulator incorporates a large block of special-purpose RAM that is dedicated to storing information about each processor cycle executed. This feature allows you to see in exactly what order things happened, so it can help you answer questions such as, "Did the timer interrupt occur before or after the variable bar became 12?" In addition, real-time trace features are often able to either restrict the information stored or post-process the data prior to viewing it, to cut down on the amount of memory wasted.

Another type of debug tool similar to an ICE is a *background debug mode* (*BDM*), or *JTAG* (pronounced "jay-tag") debugger. JTAG debuggers are typically less expensive than in-circuit emulators but offer much of the same functionality. These tools rely on a debug interface (the JTAG interface) and on-chip test circuitry found in modern processors. Additional information about JTAG emulators can be found in the February 2003 *Embedded Systems Programming* article "Introduction to On-Chip Debug," located online at *http://www.embedded.com*.

The article "How to choose an in-circuit emulator" in the July 2002 issue of *Embedded Systems Programming* is useful for learning how to select an ICE.

Another type of device that emulates a read-only memory device is a *ROM emulator*. Like an ICE, it is an embedded system that connects to the target and communicates with the host. However, with this device, the target connection is via a ROM socket. To the embedded processor, it looks like any other read-only memory device. But to the remote debugger, it looks like a debug monitor.

ROM emulators have some advantages over debug monitors. First, no one has to port the debug monitor code to your particular target hardware. Second, the ROM emulator supplies its own serial or network connection to the host, so it is not necessary to use the target's own, usually limited, resources. And finally, the ROM emulator is a true replacement for the original ROM, so none of the target's memory is used up by the debug monitor code.

Some disadvantages of using a ROM emulator are that it does not provide general debugging capabilities (i.e., reading processor register values) and is useless in systems that lack external memory.

Other Useful Tools

This section is an introduction to other tools that many software developers find useful.

Simulators

A *simulator* is a completely host-based program that simulates (hence the catchy name) the functionality and instruction set of the target processor. The user interface is usually the same as or similar to that of the remote debugger. In fact, it might be possible to use one debugger host for the simulator target as well, as shown in Figure 5-3. Although simulators have many disadvantages, they are quite valuable in the earlier stages of a project when there is not yet any actual hardware for the programmers to experiment with. If you cannot get your hands on a development board, a simulator is the best tool for getting a jump-start on the software development.
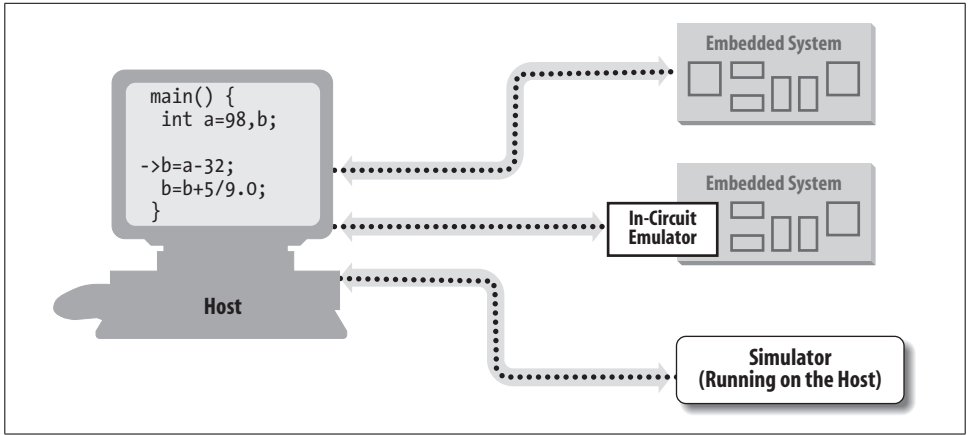
Figure 5-3. A common debugger frontend

By far, the biggest disadvantage of a simulator is that it simulates only the processor. Embedded systems frequently contain one or more important peripherals. Interaction with these devices can sometimes be imitated with simulator scripts or other workarounds, but such workarounds are often more trouble to create than the simulation is worth. So you probably won't do too much with the simulator once the actual embedded hardware is available.

Debug Tip: Hardware Verification Using a Simulator

If you ever encounter a situation in which you—after having read the databook—think the target processor is behaving differently from how it should, try running the same software in a simulator. If your program works fine there, you'll know it's a problem related to your hardware. But if the simulator exhibits the same weirdness as the actual chip, you'll know you've been misinterpreting the processor documentation all along.

Hardware Tools

As mentioned before, one of the key aspects that differentiates the embedded developer from the typical software developer is "closeness" to the hardware. Several tools are available to assist you with finding out what is going on with the hardware. A basic understanding of how to use these tools is essential to developing good debugging skills, particularly since these same tools are very useful for low-level software debugging.

Once you have access to your target hardware—and especially during hardware debug—logic analyzers and oscilloscopes can be indispensable debug tools. A logic analyzer and an oscilloscope are most useful for debugging the interactions between the processor and other chips on the board. Because they can view only signals that lie outside the processor, however, these tools cannot control the flow of execution of software. This lack of software execution control makes them significantly less useful by themselves, but coupled with a software debug tool such as a remote debugger or an emulator, they can be extremely valuable.

A *logic analyzer* is a piece of laboratory equipment designed specifically for troubleshooting digital hardware. It can have dozens or even hundreds of inputs, each capable of detecting only one thing: whether the electrical signal it is attached to is currently at logic level 1 or 0. Any subset of the inputs that you select can be displayed against a timeline as illustrated in Figure 5-4. Most logic analyzers will also let you begin capturing data, or *trigger*, on a particular pattern. For example, you might make this request: "Display the values of input signals 1 through 10, but don't start recording what happens until inputs 2 and 5 are both zero at the same time."

An *oscilloscope* is another piece of laboratory equipment for hardware debugging. But this one is used to examine any electrical signal—analog or digital—on any piece of hardware. Oscilloscopes are sometimes useful for quickly observing the voltage or signal waveform on a particular pin, or, in the absence of a logic analyzer, for something slightly more complex. However, the number of inputs is much smaller (there are usually two to four), and advanced triggering logic is not often available.
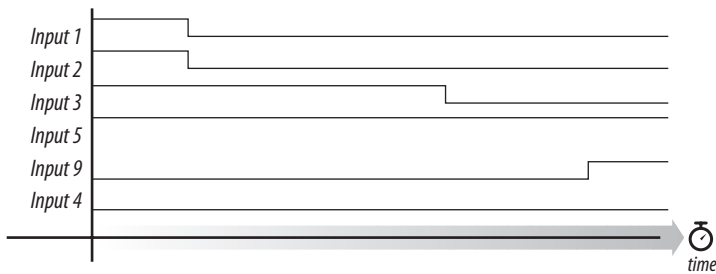
Figure 5-4. A typical logic analyzer display

Debug Tip: External Triggering

Occasionally, it is desirable to coordinate your observation of some set of electrical signals on the target with the embedded software that is running there. For example, you might want to observe the bus interaction between the processor and one of the peripherals attached to it. A handy trick is to add an output statement to the software just prior to the start of the interaction you're interested in. This output statement should cause a unique logic pattern to appear on one or more processor pins. For example, you might cause a spare I/O pin to change from a zero to a one. A logic analyzer can then be set up to trigger on the occurrence of that event and to capture everything that follows.
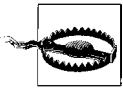
> When using an oscilloscope, be sure to connect the probe's ground lead to your target hardware's ground. Failing to do so will give you an incorrect picture of what is happening.

One of the most primitive debug techniques available is the use of an LED as an indicator of success or failure. The basic idea is to slowly walk the LED enable code through the larger program. In other words, first begin with the LED enable code at the reset address. If the LED turns on, you can edit the program—moving the LED enable code to just after the next execution milestone—and then rebuild and test. Because this technique gives you very little information about the state of the processor, it is most appropriate for very simple, linearly executed programs such as the startup code. But if you don't have access to a remote debugger or any of the other debug tools, this type of debugging might be your only choice.

If an LED is not present on your hardware platform, you can still use this debug technique with an I/O signal and an oscilloscope. In this case, set the I/O signal to a specific level once you reach a particular execution milestone. Using the oscilloscope, you can then probe that I/O pin to determine whether the code has set it

appropriately. If so, you know that the code executed successfully up to that point, and you can now move the I/O signal code to the next milestone.

The method of using an I/O signal and an oscilloscope can also be used as a basic performance measurement tool. An I/O pin can be used to measure how long a program is spending in a given routine, or how long it takes to execute a particular fragment of code. This can show potential bottlenecks in the program.

For example, to precisely measure the length of time spent in the delay_ms routine (when passed in a parameter of 1), we could set an I/O pin high when we enter the routine and then set the same I/O pin low before exiting. We could then attach an oscilloscope lead to this I/O pin to measure the amount of time that the I/O pin is high, which is the time spent in the delay_ms routine. The oscilloscope screen should look similar to the image in Figure 5-5.
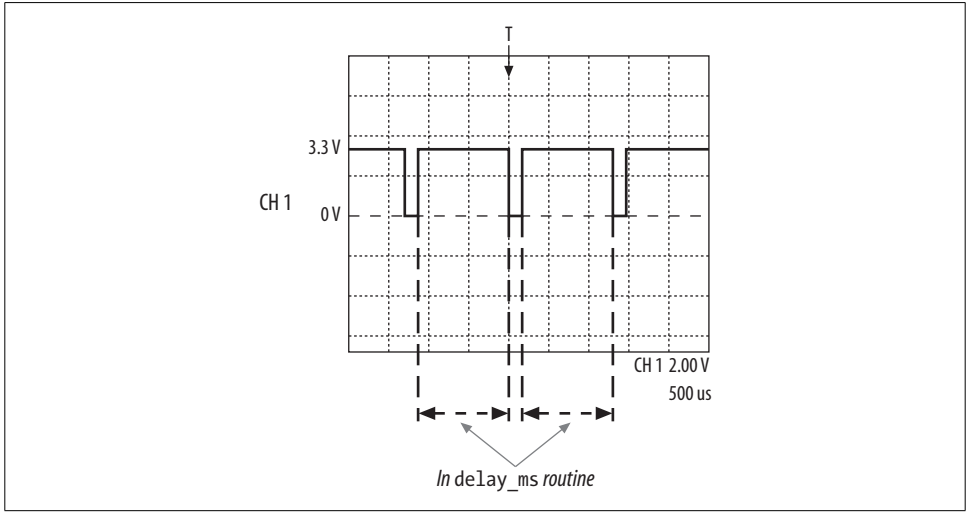
Figure 5-5. Using I/O signals for debug and performance measurements

As shown in Figure 5-5, channel 1 (CH 1) is the probe that captured the signal. The dotted horizontal lines indicate voltage increments—in this case, 2 volts per division; the dotted vertical lines indicate time increments—in this case, 500 microseconds per division. The "T" at the top of the screen, along with the arrow, indicate when the oscilloscope triggered on the falling edge of the I/O pin. The I/O signal goes from (approximately) 0 to 3.3 volts.

Incidentally, this test shows that the actual delay_ms routine that is supposed to delay for 1 millisecond is a little off, because the time from setting the I/O pin high to setting the I/O pin low is a bit longer than two divisions.

If I/O pins are available and several inputs are supported by the oscilloscope, you can use multiple I/O signals simultaneously in order to get a snapshot of the entire

system. In more complex systems, you can move the I/O set calls around in the various routines and measure how each routine is performing.

Finding Pin 1

Before (carefully) probing around the circuit board, let's learn how to identify particular pins on an IC. Figure 5-6 shows several common methods used to identify pin 1 on an IC. As shown in this figure, a square pad is often used for pin 1, whereas the other pads are typically circles. This is typically the case when the IC is in a *dual inline package* (DIP). Another indicator of pin 1 is a silkscreened circle next to pin 1.

IC manufacturers also indicate pin 1 by putting either a circular indentation next to it or an arc indentation on the top of the IC, in which case pin 1 is located to the left of this indentation. On some smaller chips, the pin 1 side is *chamfered* (a grove is cut). The other pin numbers almost always increase as you move counterclockwise from pin 1 around the chip.

A combination of these may be used in some cases. You might want to take a look at the Arcom board to get a better idea of what some of these pin-1 indicators look like.
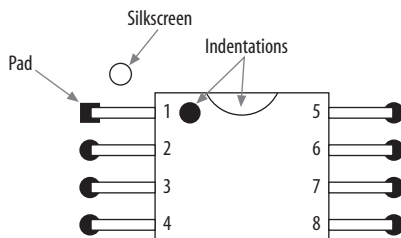
Figure 5-6. Identifying pin 1 on an IC

Lint

A *lint* program is a tool for statically checking source code for portability problems and common coding syntax errors, such as ignored return values and type inconsistencies. A compiler provides some of this error checking, but a lint program verifies these areas of a program much more carefully and therefore aids in the development of more robust software.

Setting up *lint* is similar to setting up a compiler, where different options are passed into the program to control the type of output produced. In fact, you can augment your build procedure to include a lint check that sends its output to a file for you to review at a later time.

A good introduction to using *lint* is the article "Introduction to Lint" from the archives of *Embedded Systems Programming*. This article can be found online at *http://www.embedded.com*. For additional information, pick up *Checking C Programs with Lint*, by Ian F. Darwin (O'Reilly).

Several commercial and open source *lint* programs are available. One such program is called *Splint*. Additional information about Splint as well as the latest release of it can be found online at *http://www.splint.org*.

Version Control

Management of source code is an important part of any development project. This is especially true when multiple developers are working on the same source code at the same time.

Version control software allows for storage of source code in a repository that can be updated as the project progresses through the different development stages. Various version control programs offer several features such as logging, file comparisons, and tagging releases, as well as tracking bug fixes and code updates for new features. Version control software can also assist in finding bugs that were introduced after changes were made to a stable code release.

Version control software can be useful not just for source code but for all files associated with a specific project or product, including programs, the tools used to build the software, and documentation. There are several open source version control programs available. Here are a few:

Concurrent Versions System (CVS) (http://ximbiot.com/cvs/cvshome)
 A system that allows many people to work on large sets of files simultaneously and can even combine changes from different people to a single file. *Essential CVS*, by Jennifer Vesperman (O'Reilly) covers CVS. There are also several graphical host applications for CVS, such as the Windows-based program WinCVS (*http://www.wincvs.org*).

Subversion (http://subversion.tigris.org)
 A follow-on to CVS that solves some of CVS's problems for large projects and is gaining adherents. *Version Control with Subversion*, by Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato (O'Reilly) covers Subversion version control software.[*]

Revision Control System (RCS) (http://www.gnu.org/software/rcs)
 A free software (GNU project) version of a traditional Unix source control program adequate for small projects.

---

[*] The Subversion book is also available online at *http://svnbook.red-bean.com*.

Dig into the Hardware

Most of the debugging tools described in this chapter will be used at one point or another in every embedded project. Oscilloscopes and logic analyzers are most often used to debug hardware problems; simulators to test software before the hardware is available; and debug monitors and emulators during the integration and software debug. *Lint* and version control software are typically used throughout the entire project. To be most effective, you should understand what each tool is for and when and where to apply it for the greatest impact.

On many occasions, software engineers don't want anything to do with the hardware, but this attitude lessens the software engineer's usefulness. Most projects are successful because the team members have a variety of skills and can assist in areas other than the discipline in which they are trained. Don't look at a hardware problem as something that can be solved only by a hardware engineer. Look at it as something that you can learn from and help solve.

Don't be afraid to get in there—alone or with the hardware engineer—and find out what is going on. If you don't understand the issue, sit in the lab with the hardware engineer for a while to get a better idea of what the problem is. You might even be able to write a piece of code that exacerbates the problem and, as a result, uncovers its cause.

Memory

Tyrell: *If we give them a past, we create a cushion for their emotions and, consequently, we can control them better.*
Deckard: *Memories. You're talking about memories.*
—the movie *Blade Runner*

In this chapter, you will learn everything you need to know about memory in embedded systems. In particular, you will learn about the types of memory you are likely to encounter, how to test memory devices to see whether they are working properly, and how to use flash memory.

Types of Memory

Many types of memory devices are available for use in modern computer systems. As an embedded software engineer, you must be aware of the differences between them and understand how to use each type effectively. In our discussion, we will approach these devices from a software viewpoint. As you are reading, try to keep in mind that the development of these devices took several decades. The names of the memory types frequently reflect the historical nature of the development process.

Most software developers think of memory as being either RAM or ROM. But, in fact, there are subtypes of each class, and even a third class of hybrid memories that exhibit some of the characteristics of both RAM and ROM. In a RAM device, the data stored at each memory location can be read or written, as desired. In a ROM device, the data stored at each memory location can be read at will, but never written. The hybrid devices offer ROM-like permanence, but under some conditions it is possible to overwrite their data. Figure 6-1 provides a classification system for the memory devices that are commonly found in embedded systems.
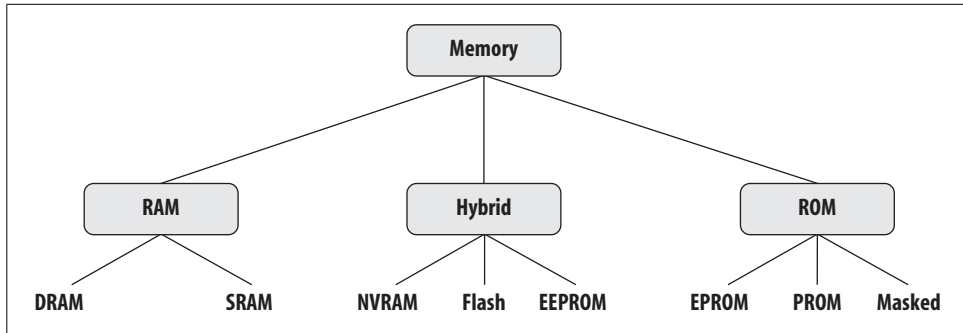
Figure 6-1. Common memory types in embedded systems

Types of RAM

There are two important memory devices in the RAM family: SRAM and DRAM. The main difference between them is the duration of the data stored. *Static RAM* (*SRAM*) retains its contents as long as electrical power is applied to the chip. However, if the power is turned off or lost temporarily, its contents will be lost forever. *Dynamic RAM* (*DRAM*), on the other hand, has an extremely short data lifetime—usually less than a quarter of a second. This is true even when power is applied continuously.

In short, SRAM has all the properties of the memory you think of when you hear the word RAM. Compared with that, DRAM sounds kind of useless. What good is a memory device that retains its contents for only a fraction of a second? By itself, such a volatile memory is indeed worthless. However, a simple piece of hardware called a *DRAM controller* can be used to make DRAM behave more like SRAM (see the sidebar "DRAM Controllers," in this section for more information). The job of the DRAM controller, often included within the processor, is to periodically refresh the data stored in the DRAM. By refreshing the data several times a second, the DRAM controller keeps the contents of memory alive for as long as they are needed. So, DRAM is as useful as SRAM after all.

When deciding which type of RAM to use, a system designer must consider access time and cost. SRAM devices offer extremely fast access times (approximately four times faster than DRAM) but are much more expensive to produce. Generally, SRAM is used only where access speed is crucial. However, if a system requires only a small amount of memory, SRAM may make more sense because you could avoid the cost of a DRAM controller.

A much lower cost-per-byte makes DRAM attractive whenever large amounts of RAM are required. DRAM is also available in much larger capacities than SRAM. Many embedded systems include both types: a small block of SRAM (a few hundred kilobytes) along a critical data path and a much larger block of DRAM (in the megabytes)

for everything else. Some small embedded systems get by without any added memory: they use only the microcontroller's on-chip memory.
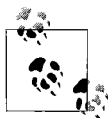
There are quite a few variations of DRAM you may encounter, including Synchronous DRAM (SDRAM), Double Data Rate SDRAM (DDR SDRAM), and Rambus DRAM (RDRAM).

DRAM Controllers

If your embedded system includes DRAM, there is probably a DRAM controller onboard (or on-chip) as well. The PXA255 has a DRAM controller on-chip. The DRAM controller is an extra piece of hardware placed between the processor and the memory chips. Its main purpose is to perform the refresh operations required to keep your data alive in the DRAM. However, it cannot do this properly without some help from you.

One of the first things your software must do is initialize the DRAM controller. If you do not have any other RAM in the system, you must do this before creating the stack or heap, because those areas of memory would then be located in the DRAM. This initialization code is usually written in assembly language and placed within the hardware-initialization module.

Almost all DRAM controllers require a short initialization sequence that consists of one or more setup commands. The setup commands tell the controller about the hardware interface to the DRAM and how frequently the data there must be refreshed. To determine the initialization sequence for your particular system, consult the designer of the board or read the databooks that describe the DRAM and DRAM controller. If the DRAM in your system does not appear to be working properly, it could be that the DRAM controller either is not initialized or has been initialized incorrectly.

Types of ROM

Memories in the ROM family are distinguished by the methods used to write new data to them (usually called *programming* or *burning*) and the number of times they can be rewritten. This classification reflects the evolution of ROM devices from hardwired to one-time programmable to erasable-and-programmable. A common feature across all these devices is their ability to retain data and programs forever, even when power is removed.

The very first ROMs were hardwired devices that contained a preprogrammed set of data or instructions. The contents of the ROM had to be specified before chip production, so the actual data could be used to arrange the transistors inside the chip! Hardwired memories are still used, though they are now called *masked ROMs* to distinguish them from other types of ROM. The main advantage of a masked ROM is a

low production cost. Unfortunately, the cost is low only when hundreds of thousands of copies of the same ROM are required, and no changes are ever needed.

Another type of ROM is the *programmable ROM* (*PROM*), which is purchased in an unprogrammed state. If you were to look at the contents of an unprogrammed PROM, you would see that all the bits are 1s. The process of writing your data to the PROM involves a special piece of equipment called a *device programmer*, which writes data to the device by applying a higher-than-normal voltage to special input pins of the chip. Once a PROM has been programmed in this way, its contents can never be changed. If the code or data stored in the PROM must be changed, the chip must be discarded and replaced with a new one. As a result, PROMs are also known as *one-time programmable* (OTP) devices. Many small embedded microcontrollers are also considered one-time programmable, because they contain built-in PROM.

An *erasable-and-programmable ROM* (EPROM) is programmed in exactly the same manner as a PROM. However, EPROMs can be erased and reprogrammed repeatedly. To erase an EPROM, simply expose the device to a strong source of ultraviolet light. (There is a "window" in the top of the device to let the ultraviolet light reach the silicon. You can buy an EPROM eraser containing this light.) By doing this, you essentially reset the entire chip to its initial—unprogrammed—state. The erasure time of an EPROM can be anything from 10 to 45 minutes, which can make software debugging a slow process.

Though more expensive than PROMs, their ability to be reprogrammed made EPROMs a common feature of the embedded software development and testing process for many years. It is now relatively rare to see EPROMs used in embedded systems, as they have been supplanted by newer technologies.

Hybrid Types

As memory technology has matured in recent years, the line between RAM and ROM devices has blurred. There are now several types of memory that combine the best features of both. These devices do not belong to either group and can be collectively referred to as hybrid memory devices. Hybrid memories can be read and written as desired, like RAM, but maintain their contents without electrical power, just like ROM. Write cycles to hybrid memories are similar to RAM, but they take significantly longer than writes to a RAM, so you wouldn't want to use this type for your main system memory. Two of the hybrid devices, EEPROM and flash, are descendants of ROM devices; the third, NVRAM, is a modified version of SRAM.

An *electrically-erasable-and-programmable ROM* (EEPROM) is internally similar to an EPROM, but with the erase operation accomplished electrically. Additionally, a single byte within an EEPROM can be erased and rewritten. Once written, the new data will remain in the device forever—or at least until it is electrically erased. One

tradeoff for this improved functionality is higher cost; another is that typically EEPROM is good for 10,000 to 100,000 write cycles.

EEPROMs are available in a standard (address and data bus) parallel interface as well as a serial interface. In many designs, the Inter-IC (I²C) or Serial Peripheral Interface (SPI) buses are used to communicate with serial EEPROM devices. We'll take a look at the I²C and SPI buses in Chapter 13.

Flash is the most important recent advancement in memory technology. It combines all the best features of the memory devices described thus far. Flash memory devices are high-density, low-cost, nonvolatile, fast (to read, but not to write), and electrically reprogrammable. These advantages are overwhelming, and the use of flash memory has increased dramatically in embedded systems as a direct result.

Erasing and writing data to a flash memory requires a specific sequence of writes using certain data values. From a software viewpoint, flash and EEPROM technologies are very similar. The major difference is that flash devices can be erased only one sector at a time, not byte by byte. Typical sector sizes range from 8 KB to 64 KB. Despite this disadvantage, flash is much more popular than EEPROM and is rapidly displacing many of the ROM devices as well.

The third member of the hybrid memory class is *nonvolatile RAM* (NVRAM). Nonvolatility is also a characteristic of the ROM and hybrid memories discussed earlier. However, an NVRAM is physically very different from those devices. An NVRAM is usually just an SRAM with a battery backup. When the power is on, the NVRAM operates just like any other SRAM. But when the power is off, the NVRAM draws just enough electrical power from the battery to retain its current contents. NVRAM is sometimes found in embedded systems to store system-critical information. Incidentally, the "CMOS" in an IBM-compatible PC was historically an NVRAM.

Table 6-1 summarizes the characteristics of different memory types.

Table 6-1. Memory device characteristics

Memory type	Volatile?	Writable?	Erase/ rewrite size	Erase/rewrite cycles	Relative cost	Relative speed
SRAM	Yes	Yes	Byte	Unlimited	Expensive	Fast
DRAM	Yes	Yes	Byte	Unlimited	Moderate	Moderate
Masked ROM	No	No	N/A	N/A	Inexpensive (in quantity)	Slow
PROM	No	Once, with programmer	N/A	N/A	Moderate	Slow
EPROM	No	Yes, with programmer	Entire chip	Limited (see specs)	Moderate	Slow
EEPROM	No	Yes	Byte	Limited (see specs)	Expensive	Moderate to read, slow to write

Table 6-1. Memory device characteristics (continued)

Memory type	Volatile?	Writable?	Erase/rewrite size	Erase/rewrite cycles	Relative cost	Relative speed
Flash	No	Yes	Sector	Limited (see specs)	Moderate	Fast to read, slow to write
NVRAM	No	Yes	Byte	None	Expensive	Fast

Direct Memory Access

Since we are discussing memory, this is a good time to discuss a memory transfer technique called *direct memory access* (DMA). DMA is a technique for transferring blocks of data directly between two hardware devices with minimal CPU involvement. In the absence of DMA, the processor must read the data from one device and write it to the other, one byte or word at a time. For each byte or word transferred, the processor must fetch and execute a sequence of instructions. If the amount of data to be transferred is large, or the frequency of transfers is high, the rest of the software might never get a chance to run. However, if a DMA controller is present, it can perform the entire transfer, with little assistance from the processor.

Here's how DMA works. When a block of data needs to be transferred, the processor provides the DMA controller with the source and destination addresses and the total number of bytes. The DMA controller then transfers the data from the source to the destination automatically. When the number of bytes remaining reaches zero, the block transfer ends.

In a typical DMA scenario, the block of data is transferred directly to or from memory. For example, a network controller might want to place an incoming network packet into memory as it arrives but notify the processor only once the entire packet has been received. By using DMA, the processor can spend more time processing the data once it arrives and less time transferring it between devices. The processor and DMA controller must use the same address and data buses during this time, but this is handled automatically by the hardware, and the processor is otherwise uninvolved with the actual transfer. During a DMA transfer, the DMA controller arbitrates control of the bus between the processor and the DMA operation.

Endian Issues

Endianness is the attribute of a system that indicates whether integers are represented from left to right or right to left. Why, in today's world of virtual machines and gigaHertz processors, would a programmer care about such a silly topic? Well, unfortunately, endianness must be chosen every time a hardware or software architecture is designed, and there isn't much in the way of natural law to help in the decision.

Endianness comes in two varieties: big and little. A *big-endian* representation has a multibyte integer written with its most significant byte on the left; a number represented thus is easily read by English-speaking humans. A *little-endian* representation, on the other hand, places the most significant byte on the right. Of course, computer architectures don't have an intrinsic "left" or "right." These human terms are borrowed from our written forms of communication. The following definitions are more precise:

Big-endian
> Means that the most significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field

Little-endian
> Means that the least significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field

The origin of the odd terms big-endian and little-endian can be traced to the 1726 book *Gulliver's Travels,* by Jonathan Swift. In one part of the story, resistance to an imperial edict to break soft-boiled eggs on the "little end" escalates to civil war. The plot is a satire of England's King Henry VIII's break with the Catholic Church. A few hundred years later, in 1981, Danny Cohen applied the terms and the satire to our current situation in *IEEE Computer* (vol. 14, no. 10).

Endianness in Devices

Endianness doesn't matter on a single system. It matters only when two computers are trying to communicate. Every processor and every communication protocol must choose one type of endianness or the other. Thus, two processors with different endianness will conflict if they communicate through a memory device. Similarly, a little-endian processor trying to communicate over a big-endian network will need to do software-byte reordering.

Intel's 80x86 processors and their clones are little-endian. Sun's SPARC, Motorola's 68K, and the PowerPC families are all big-endian. Some processors even have a bit in a register that allows the programmer to select the desired endianness. The PXA255 processor supports both big- and little-endian operation via bit 7 in Control Register 1 (Coprocessor 15 (CP15) register 1).

An endianness difference can cause problems if a computer unknowingly tries to read binary data written in the opposite format from a shared memory location or file. Figure 6-2(a) shows the memory contents for the data 0x12345678 (a long), 0xABCD (a word), and 0xEF (a byte) on a little-endian machine. The same data represented on a big-endian machine is shown in Figure 6-2(b).
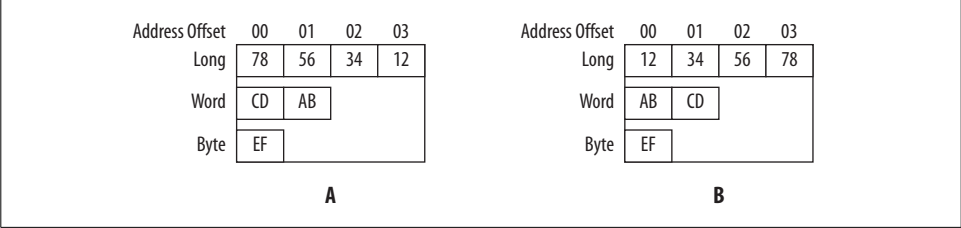
Figure 6-2. (a) Little-endian memory, (b) big-endian memory

Endianness in Networking

Another area where endianness is an issue is in network communications. Since different processor types (big-endian and little-endian) can be on the same network, they must be able to communicate with each other. Therefore, network stacks and communication protocols must also define their endianness. Otherwise, two nodes of different endianness would be unable to communicate. This is a more substantial example of endianness affecting the embedded programmer.

As it turns out, all of the protocol layers in the TCP/IP suite are defined as big-endian. In other words, any 16- or 32-bit value within the various layer headers (for example, an IP address, a packet length, or a checksum) must be sent and received with its most significant byte first.

Let's say you wish to establish a TCP socket connection to a computer whose IP address is 192.0.1.2. IPv4 uses a unique 32-bit integer to identify each network host. The dotted decimal IP address must be translated into such an integer.

The multibyte integer representation used by the TCP/IP protocols is sometimes called *network byte order*. Even if the computers at each end are little-endian, multibyte integers passed between them must be converted to network byte order prior to transmission across the network, and then converted back to little-endian at the receiving end.

Suppose an 80x86-based, little-endian PC is talking to a SPARC-based, big-endian server over the Internet. Without further manipulation, the 80x86 processor would convert 192.0.1.2 to the little-endian integer 0x020100C0 and transmit the bytes in the following order: 0x02, 0x01, 0x00, 0xC0. The SPARC would receive the bytes in the followng order: 0x02, 0x01, 0x00, 0xC0. The SPARC would reconstruct the bytes into a big-endian integer 0x020100c0, and misinterpret the address as 2.1.0.192.

Preventing this sort of confusion leads to an annoying little implementation detail for TCP/IP stack developers. If the stack will run on a little-endian processor, it will have to reorder (at runtime) the bytes of every multibyte data field within the various layers' headers. If the stack will run on a big-endian processor, there's nothing to worry about. For the stack to be portable (that is, to be able to run on processors of both

types), it will have to decide whether or not to do this reordering. The decision is typically made at compile time.

A common solution to the endianness problem is to define a set of four preprocessor macros:

htons( )
> Reorder the bytes of a 16-bit unsigned value from processor order to network order. The macro name can be read "host to network short."

htonl( )
> Reorder the bytes of a 32-bit unsigned value from processor order to network order. The macro name can be read "host to network long."

ntohs( )
> Reorder the bytes of a 16-bit unsigned value from network order to processor order. The macro name can be read "network to host short."

ntohl( )
> Reorder the bytes of a 32-bit unsigned value from network order to processor order. The macro name can be read "network to host long."

Following is an example of the implementation of these macros. We will take a look at the left shift (<<) and right shift (>>) operators in Chapter 7.

```
#if defined(BIG_ENDIAN) && !defined(LITTLE_ENDIAN)

    #define htons(A)  (A)
    #define htonl(A)  (A)
    #define ntohs(A)  (A)
    #define ntohl(A)  (A)

#elif defined(LITTLE_ENDIAN) && !defined(BIG_ENDIAN)

    #define htons(A)  ((((uint16_t)(A) & 0xff00) >> 8) | \
                       (((uint16_t)(A) & 0x00ff) << 8))
    #define htonl(A)  ((((uint32_t)(A) & 0xff000000) >> 24) | \
                       (((uint32_t)(A) & 0x00ff0000) >> 8)  | \
                       (((uint32_t)(A) & 0x0000ff00) << 8)  | \
                       (((uint32_t)(A) & 0x000000ff) << 24))
    #define ntohs     htons
    #define ntohl     htohl

#else

    #error Either BIG_ENDIAN or LITTLE_ENDIAN must be #defined, but not both.

#endif
```

If the processor on which the TCP/IP stack is to be run is itself also big-endian, each of the four macros will be defined to do nothing, and there will be no runtime performance impact. If, however, the processor is little-endian, the macros will reorder the

bytes appropriately. These macros are routinely called when building and parsing network packets and when socket connections are created.

Runtime performance penalties can occur when using TCP/IP on a little-endian processor. For that reason, it may be unwise to select a little-endian processor for use in a device with an abundance of network functionality, such as a router or gateway. Embedded programmers must be aware of the issue and be prepared to convert between their different representations as required.

Memory Testing

One of the first pieces of serious embedded software you are likely to write is a memory test. Once the prototype hardware is ready, the designer would like some reassurance that he has wired the address and data lines correctly and that the memory chips are working properly. At first this might seem like a fairly simple assignment, but as you look at the problem more closely, you will realize that it can be difficult to detect subtle memory problems with a simple test. In fact, as a result of programmer naiveté, many embedded systems include memory tests that would detect only the most catastrophic memory failures. Some of these might not even notice that the memory chips have been removed from the board!

The purpose of a memory test is to confirm that each storage location in a memory device is working. In other words, if you store the number 50 at a particular address, you expect to find that number stored there until another number is written. The basic idea behind any memory test, then, is to write some set of data values to each address in the memory device and verify the data by reading it back. If all of the values read back are the same as those that were written, then the memory device is said to pass the test. As you will see, it is only through careful selection of the set of data values that you can be sure that a passing result is meaningful.

Of course, a memory test such as the one just described is unavoidably destructive. In the process of testing the memory, you must overwrite its prior contents. Because it is generally impractical to overwrite the contents of nonvolatile memories, the tests described in this section are generally used only for RAM testing. In fact, running comprehensive memory tests on flash or EEPROM is often a bad idea because the number of writes involved can shorten the useful life of the device. However, if the contents of a hybrid memory are unimportant—as they are during the product development stage—these same algorithms can be used to test those devices as well. We address the problem of validating the contents of a nonvolatile memory in the section "Validating Memory Contents," later in this chapter.

Common Memory Problems

Before learning about specific test algorithms, you should be familiar with the types of memory problems that are likely to occur. One common misconception among

software engineers is that most memory problems occur within the chips themselves. Though a major issue at one time (a few decades ago), problems of this type are increasingly rare. The manufacturers of memory devices perform a variety of post-production tests on each batch of chips. If there is a problem with a particular batch, it is unlikely that one of the bad chips will make its way into your system.

The one type of memory chip problem you could encounter is a catastrophic failure. This is usually caused by some sort of physical or electrical damage to the chip after manufacture. Catastrophic failures are uncommon, and they usually affect large portions of the chip. Because a large area is affected, it is reasonable to assume that catastrophic failure will be detected by any decent test algorithm.

In our experience, a more common source of memory problems is the circuit board. Typical circuit board problems are:

- Problems with the wiring between the processor and memory device
- Missing memory chips
- Improperly inserted memory chips

These are the problems that a good memory test algorithm should be able to detect. Such a test should also be able to detect catastrophic memory failures without specifically looking for them. So let's discuss circuit board problems in more detail.

Electrical wiring problems

An electrical wiring problem could be caused by an error in design or production of the board or as the result of damage received after manufacture. Each of the wires that connect the memory device to the processor is one of three types:

- Address signal
- Data signal
- Control signal

The address and data signals select the memory location and transfer the data, respectively. The control signals tell the memory device whether the processor wants to read or write the location and precisely when the data will be transferred. Unfortunately, one or more of these wires could be improperly routed or damaged in such a way that it is either *shorted* (i.e., connected to another wire on the board) or *open* (not connected to anything). Shorting is often caused by a bit of solder splash, whereas an open wire could be caused by a broken trace. Both cases are illustrated in Figure 6-3.

Problems with the electrical connections to the processor will cause the memory device to behave incorrectly. Data might be corrupted when it's stored, stored at the wrong address, or not stored at all. Each of these symptoms can be explained by wiring problems on the data, address, and control signals, respectively.
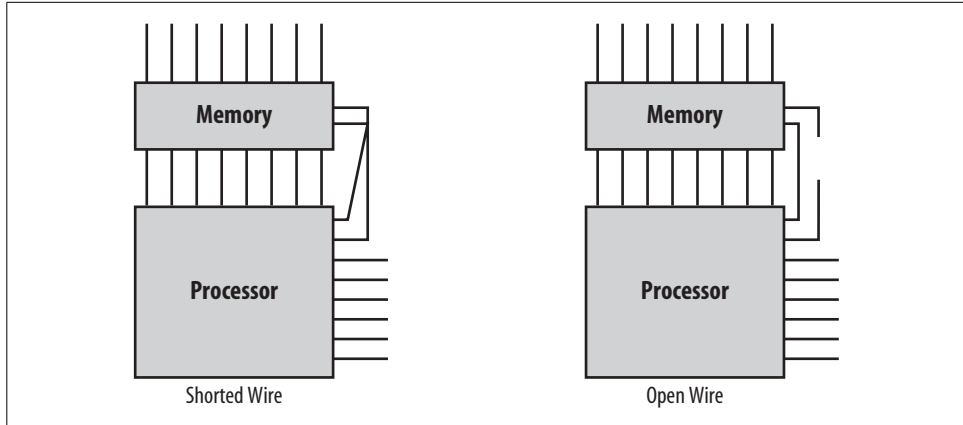
Figure 6-3. Possible wiring problems

If the problem is with a data signal, several data bits might appear to be "stuck together" (i.e., two or more bits always contain the same value, regardless of the data transmitted). Similarly, a data bit might be either "stuck high" (always 1) or "stuck low" (always 0). These problems can be detected by writing a sequence of data values designed to test that each data pin can be set to 0 and 1, independently of all the others.

If an address signal has a wiring problem, the contents of two memory locations might appear to overlap. In other words, data written to one address will instead overwrite the contents of another address. This happens because an address bit that is shorted or open causes the memory device to see an address different from the one selected by the processor.

Another possibility is that one of the control signals is shorted or open. Although it is theoretically possible to develop specific tests for control signal problems, it is not possible to describe a general test that covers all platforms. The operation of many control signals is specific to either the processor or the memory architecture. Fortunately, if there is a problem with a control signal, the memory probably won't work at all, and this will be detected by other memory tests. If you suspect a problem with a control signal, it is best to seek the advice of the board's designer before constructing a specific test.

Missing memory chips

A missing memory chip is clearly a problem that should be detected. Unfortunately, because of the capacitive nature of unconnected electrical wires, some memory tests will not detect this problem. For example, suppose you decided to use the following test algorithm: write the value 1 to the first location in memory, verify the value by reading it back, write 2 to the second location, verify the value, write 3 to the third location, verify, and so on. Because each read occurs immediately after the corresponding

write, it is possible that the data read back represents nothing more than the voltage remaining on the data bus from the previous write. If the data is read back quickly, it will appear that the data has been correctly stored in memory, even though there is no memory chip at the other end of the bus!

To detect a missing memory chip, a better test must be used. Instead of performing the verification read immediately after the corresponding write, perform several consecutive writes followed by the same number of consecutive reads. For example, write the value 1 to the first location, write the value 2 to the second location, write the value 3 to the third location, and then verify the data at the first location, the second location, and so on. If the data values are unique (as they are in the test just described), the missing chip will be detected: the first value read back will correspond to the last value written (3) rather than to the first (1).

Improperly inserted chips

If a memory chip is present but improperly inserted, some pins on the memory chip will either not be connected to the circuit board at all or will be connected at the wrong place. These pins will be part of the data bus, address bus, or control wiring. The system will usually behave as though there is a wiring problem or a missing chip. So as long as you test for wiring problems and missing chips, any improperly inserted chips will be detected automatically.

Before going on, let's quickly review the types of memory problems we must be able to detect. Memory chips only rarely have internal errors, but if they do, they are typically catastrophic in nature and should be detected by any test. A more common source of problems is the circuit board, where a wiring problem can occur or a memory chip might be missing or improperly inserted. Other memory problems can occur, but the ones described here are the most common and also the simplest to test in a generic way.

Developing a Test Strategy

By carefully selecting your test data and the order in which the addresses are tested, you can detect all of the memory problems described earlier. It is usually best to break your memory test into small, single-purpose pieces. This helps to improve the efficiency of the overall test and the readability of the code. More specific tests can also provide more detailed information about the source of the problem, if one is detected.

We have found that it is best to have three individual memory tests, which should be executed in the following order:

1. Data bus test
2. Address bus test
3. Device test

The first two test for electrical wiring problems and improperly inserted chips; the third is intended to detect missing chips and catastrophic failures. As an unintended consequence, the device test will also uncover problems with the control bus wiring, though it cannot provide useful information about the source of such a problem.

The reason the order is important is that the address bus test assumes a working data bus, and the device test results are meaningless unless both the address and data buses are known to be sound. If any of the tests fail, you should work with a hardware engineer to locate the source of the problem. By looking at the data value or address at which the test failed, she should be able to quickly isolate the problem on the circuit board.

Data bus test

The first thing we want to test is the data bus wiring. We need to confirm that any value placed on the data bus by the processor is correctly received by the memory device at the other end. The most obvious way to test that is to write all possible data values and verify that the memory device stores each one successfully. However, that is not the most efficient test available. A faster method is to test the bus one bit at a time. The data bus passes the test if each data bit can be set to 0 and 1, independently of the other data bits.

A good way to test each bit independently is to perform the so-called *walking 1's* test. Table 6-2 shows the data patterns used in an 8-bit version of this test. The name *walking 1's* comes from the fact that a single data bit is set to 1 and "walked" through the entire data word. The number of data values to test is the same as the width of the data bus. This reduces the number of test patterns from 2^n to n, where n is the width of the data bus.

Table 6-2. Consecutive data values for an 8-bit walking 1's test

00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000

Because we are testing only the data bus at this point, all of the data values can be written to the same address. Any address within the memory device will do. However, if the data bus splits as it makes its way to more than one memory chip, you will need to perform the data bus test at multiple addresses—one within each chip.

To perform the walking 1's test, simply write the first data value in the table, verify it by reading it back, write the second value, verify, and so on. When you reach the end of the table, the test is complete. This time, it is okay to do the read immediately after the corresponding write because we are not yet looking for missing chips. In fact, this test may provide meaningful results even if the memory chips are not installed!

The function memtestDataBus shows how to implement the walking 1's test. It assumes that the caller will select the test address, and tests the entire set of data values at that address. If the data bus is working properly, the function returns 1 and the parameter ppFailAddr is set to NULL. Otherwise it returns 0, and the address at which the test failed is returned in the parameter ppFailAddr.

```c
/* Set the data bus width to 32 bits. */
typedef uint32_t datum;

/**********************************************************************
 *
 * Function:    memtestDataBus
 *
 * Description: Test the data bus wiring in a memory region by
 *              performing a walking 1's test at a fixed address
 *              within that region.  The address (and hence the
 *              memory region) is selected by the caller.
 *
 * Notes:
 *
 * Returns:     0 if the test fails. The failure address is returned
 *              in the parameter ppFailAddr.
 *              1 if the test succeeds. The parameter ppFailAddr is
 *              set to NULL.
 *
 **********************************************************************/
int memtestDataBus(datum *pAddress, datum **ppFailAddr)
{
    datum pattern;

    *ppFailAddr = NULL;

    /* Perform a walking 1's test at the given address. */
    for (pattern = 1; pattern != 0; pattern <<= 1)
    {
        /* Write the test pattern. */
        *pAddress = pattern;

        /* Read it back (immediately is okay for this test). */
        if (*pAddress != pattern)
        {
            *ppFailAddr = pAddress;
            return 0;
        }
    }

    return 1;
}
```

Address bus test

After confirming that the data bus works properly, you should next test the address bus. Address bus problems lead to overlapping memory locations. There are many possible addresses that could overlap. However, it is not necessary to check every possible combination. You should instead follow the example of the previous data bus test and try to isolate each address pin during testing. You simply need to confirm that each of the address pins can be set to 0 and 1 without affecting any of the others.

The smallest set of addresses that will cover all possible combinations is the set of *power-of-two* addresses. These addresses are analogous to the set of data values used in the walking 1's test. The corresponding memory locations are 0x00000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010, 0x00000020, and so forth. In addition, address 0x00000000 must be tested. The possibility of overlapping locations makes the address bus test harder to implement. After writing to one of the addresses, you must check that none of the others has been overwritten.

It is important to note that in some cases not all of the address signals can be tested in this way. Part of the address—the most significant bits on the left end—selects the memory chip itself. Another part—one or two least significant bits on the right—might not be relevant if the data bus is wider than 8 bits. These extra bits should remain constant throughout your address bus test and will thus reduce the number of test addresses. For example, if the processor has 20 address bits, it can address up to 1 MB of memory. If you want to test a 128 KB block of memory—that is, $1/8$ of the total one-megabyte address space—the 3 most significant address bits will remain constant. In that case, only the 17 least significant bits of the address bus can actually be tested.

To confirm that no two memory locations overlap, you should first write some initial data value at each power-of-two offset within the device. Then write a new value—an inverted copy of the initial value is a good choice—to the first test offset, and verify that the initial data value is still stored at every other power-of-two offset. If you find a location (other than the one you just wrote) that contains the new data value, you have found a problem with the current address bit. If no overlapping is found, repeat the procedure for each of the remaining offsets.

The function memtestAddressBus shows how this can be done in practice. The function accepts three parameters. The first parameter is the base address of the memory block to be tested, the second is its size (in bytes), and the third is used to return the address of the failure, if one occurs. The size is used to determine which address bits should be tested. For best results, the base address should contain a 0 in each of those bits. If the address bus test fails, 0 is returned and the address at which the first error was detected is returned in the parameter ppFailAddr. Otherwise, the function returns 1 to indicate success and sets ppFailAddr to NULL.

```
/***********************************************************************
 * Function:    memtestAddressBus
 *
 * Description: Test the address bus wiring in a memory region by
 *              performing a walking 1's test on the relevant bits
 *              of the address and checking for aliasing.  The test
 *              will find single-bit address failures such as stuck
 *              high, stuck low, and shorted pins.  The base address
 *              and size of the region are selected by the caller.
 *
 * Notes:       For best results, the selected base address should
 *              have enough LSB 0's to guarantee single address bit
 *              changes.  For example, to test a 64 KB region, select
 *              a base address on a 64 KB boundary.  Also, the number
 *              of bytes must describe a power-of-two region size.
 *
 * Returns:     0 if the test fails. The failure address is returned
 *              in the parameter ppFailAddr.
 *              1 if the test succeeds. The parameter ppFailAddr is
 *              set to NULL.
 *
 ***********************************************************************/
int memtestAddressBus(datum *pBaseAddress, uint32_t numBytes, datum **ppFailAddr)
{
    uint32_t addressMask = (numBytes - 1);
    uint32_t offset;
    uint32_t testOffset;
    datum    pattern = (datum) 0xAAAAAAAA;
    datum    antipattern = (datum) ~pattern;

    *ppFailAddr = NULL;

    /* Write the default pattern at each of the power-of-two offsets. */
    for (offset = sizeof(datum); (offset & addressMask) != 0; offset <<= 1)
        pBaseAddress[offset] = pattern;

    /* Check for address bits stuck high. */
    pBaseAddress[0] = antipattern;

    for (offset = sizeof(datum); offset & addressMask; offset <<= 1)
    {
        if (pBaseAddress[offset] != pattern)
        {
            *ppFailAddr = &pBaseAddress[offset];
            return 0;
        }
    }

    pBaseAddress[0] = pattern;
```

```
        /* Check for address bits stuck low or shorted. */
        for (testOffset = sizeof(datum); testOffset & addressMask; testOffset <<= 1)
        {
            pBaseAddress[testOffset] = antipattern;

            for (offset = sizeof(datum); offset & addressMask; offset <<= 1)
            {
                if ((pBaseAddress[offset] != pattern) && (offset != testOffset))
                {
                    *ppFailAddr = &pBaseAddress[offset];
                    return 0;
                }
            }

            pBaseAddress[testOffset] = pattern;
        }

        return 1;
    }
```

Device test

Once you know that the address and data bus wiring are correct, it is necessary to test the integrity of the memory device itself. The goal is to test that every bit in the device is capable of holding both 0 and 1. This test is fairly straightforward to implement, but it takes significantly longer to execute than the previous two tests.

For a complete device test, you must write and verify every memory location twice. You are free to choose any data value for the first pass, as long as you invert that value during the second. And because there is a possibility of missing memory chips, it is best to select a set of data that changes with (but is not equivalent to) the address. A simple example is an *increment* test.

The data values for the increment test are shown in the first two columns of Table 6-3. The third column shows the inverted data values used during the second pass of this test. The second pass represents a decrement test. There are many other possible choices of data, but the incrementing data pattern is adequate and easy to compute.

Table 6-3. Data values for an 8-bit increment test

Memory offset	Binary value	Inverted value
0x00	00000001	11111110
0x01	00000010	11111101
0x02	00000011	11111100
0x03	00000100	11111011
...	...	...
0xFE	11111111	00000000
0xFF	00000000	11111111

The function memtestDevice implements just such a two-pass increment/decrement test. It accepts three parameters from the caller. The first parameter is the starting address, the second is the number of bytes to be tested, and the third is used to return the address of the failure, if one occurs. The first two parameters give the user maximum control over which areas of memory are overwritten. The function returns 1 on success, and the parameter ppFailAddr is set to NULL. Otherwise, 0 is returned and the first address that contains an incorrect data value is returned in the parameter ppFailAddr.

```
/**********************************************************************
 *
 * Function:    memtestDevice
 *
 * Description: Test the integrity of a physical memory device by
 *              performing an increment/decrement test over the
 *              entire region.  In the process, every storage bit
 *              in the device is tested as a zero and a one.  The
 *              base address and the size of the region are
 *              selected by the caller.
 *
 * Notes:
 *
 * Returns:     0 if the test fails. The failure address is returned
 *              in the parameter ppFailAddr.
 *              1 if the test succeeds. The parameter ppFailAddr is
 *              set to NULL.
 *
 **********************************************************************/
int memtestDevice(datum *pBaseAddress, uint32_t numBytes, datum **ppFailAddr)
{
    uint32_t offset;
    uint32_t numWords = numBytes / sizeof(datum);
    datum    pattern;

    *ppFailAddr = NULL;

    /* Fill memory with a known pattern. */
    for (pattern = 1, offset = 0; offset < numWords; pattern++, offset++)
        pBaseAddress[offset] = pattern;

    /* Check each location and invert it for the second pass. */
    for (pattern = 1, offset = 0; offset < numWords; pattern++, offset++)
    {
        if (pBaseAddress[offset] != pattern)
        {
            *ppFailAddr = &pBaseAddress[offset];
            return 0;
        }

        pBaseAddress[offset] = ~pattern;
    }
```

```
        /* Check each location for the inverted pattern and zero it. */
        for (pattern = 1, offset = 0; offset < numWords; pattern++, offset++)
        {
            if (pBaseAddress[offset] != ~pattern)
            {
                *ppFailAddr = &pBaseAddress[offset];
                return 0;
            }

            pBaseAddress[offset] = 0;
        }

        return 1;
    }
```

Putting it all together

To make our discussion more concrete, let's consider a practical example. Suppose that we want to test a 64 KB chunk of the DRAM starting at address 0x00500000 on the Arcom board. To do this, we would call each of the three test routines in turn. In each case, the first parameter is the base address of the memory block. The width of the data bus is 32 bits, and there are a total of 64 KB to be tested (corresponding to the right most 16 bits of the address bus).

If any of the memory test routines returns a zero, we'll immediately turn on the red LED to visually indicate the error. Otherwise, after all three tests have completed successfully, we will turn on the green LED. New LED functions have been added, which allow the LEDs to be turned on or off.

In the event of an error, the test routine that failed will return some information about the problem encountered in the parameter pFailAddr. This information can be useful when communicating with a hardware engineer about the nature of the problem. However, the information returned by the function is visible only if we are running the test program in a debugger or emulator. Later we will look at a serial driver that will allow input from and output to a serial port on the board. This can be an invaluable tool for getting debug output from a program.

The best way to proceed is to assume the best, download the test program, and let it run to completion. Then, if and only if the red LED comes on, you must examine the return codes and contents of the memory to see which test failed and why.

Following is the program's main function, which performs a few LED initializations and then executes the previously defined memory test functions:

```
    #include "memtest.h"
    #include "led.h"

    #define BASE_ADDRESS            (datum *)(0x00500000)
    #define NUM_BYTES               (0x10000)
```

```
    /*********************************************************************
     *
     * Function:    main
     *
     * Description: Test a 64 KB block of DRAM.
     *
     * Notes:
     *
     * Returns:     0 on failure.
     *              1 on success.
     *
     *********************************************************************/
    int main(void)
    {
        datum *pFailAddr;

        /* Configure the LED control pins. */
        ledInit();

        /* Make sure all LEDs are off before we start the memory test. */
        ledOff(LED_GREEN | LED_YELLOW | LED_RED);

        if ((memtestDataBus(BASE_ADDRESS, &pFailAddr) != 1) ||
            (memtestAddressBus(BASE_ADDRESS, NUM_BYTES, &pFailAddr) != 1) ||
            (memtestDevice(BASE_ADDRESS, NUM_BYTES, &pFailAddr) != 1))
        {
            ledOn(LED_RED);
            return 0;
        }
        else
        {
            ledOn(LED_GREEN);
            return 1;
        }
    }
```

Unfortunately, it is not always possible to write memory tests in a high-level language. For example, C requires the use of a stack. But a stack itself requires working memory. This might be reasonable in a system that has more than one memory device. For example, you might create a stack in an area of RAM that is already known to be working, while testing another memory device. In a common situation, a small SRAM could be tested from assembly and the stack could be created in this SRAM afterward. Then a larger block of DRAM could be tested using a test algorithm implemented in a high-level language, such as the one just shown. If you cannot assume enough working RAM for the stack and data needs of the test program, you will need to rewrite these memory test routines entirely in assembly language.

> It might be possible to use the processor cache for the stack. Or if the processor uses a link register, and variables are kept in registers, it may still be possible to write tests in C without needing a stack.

Another option is to run the memory test program from an in-circuit emulator. In this case, you could choose to place the stack in an area of the emulator's own internal memory. By moving the emulator's internal memory around in the target memory map, you could systematically test each memory device on the target.
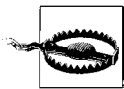
Running an emulator before you are assured that your hardware is working entails risk. If there is a physical (electrical/bus) fault in your system, the fault could destroy your expensive ICE.

You also need to be careful that the processor's cache does not fool you into thinking that the memory tests falsely succeeded. For example, imagine that the processor stores the data that you intended to write out to a particular memory location in its cache. When you read that memory location back, the processor provides the cached value. In this case, you get a valid result regardless of whether there is an actual memory error. It is best to run the memory tests with the cache (at least the data cache) disabled.

The need for memory testing is perhaps most apparent during product development, when the reliability of the hardware and its design are still unproved. However, memory is one of the most critical resources in any embedded system, so it might also be desirable to include a memory test in the final release of your software. In that case, the memory test and other hardware confidence tests should be run each time the system is powered on or reset. Together, this initial test suite forms a set of hardware diagnostics. If one or more of the diagnostics fail, a repair technician can be called in to diagnose the problem and repair or replace the faulty hardware.

Validating Memory Contents

It does not usually make sense to perform the type of memory testing described earlier when dealing with ROM or hybrid memory devices. ROM devices cannot be written at all, and hybrid devices usually contain data or programs that you can't overwrite because you'd lose the information. However, it should be clear that the same sorts of memory problems can occur with these devices. A chip might be missing, improperly inserted, or physically or electrically damaged, or there could be an electrical wiring problem. Rather than just assuming that these nonvolatile memory devices are functioning properly, you would be better off having some way to confirm that the device is working and that the data it contains is valid. That's where checksums and cyclic redundancy checks come in.

Checksums

How can we tell whether the data or program stored in a nonvolatile memory device is still valid? One of the easiest ways is to compute a *checksum* of the data when it is

known to be valid—prior to programming the ROM, for example. Then, each time you want to confirm the validity of the data, you need only recalculate the checksum and compare the result to the previously computed value. If the two checksums match, the data is assumed to be valid. By carefully selecting the checksum algorithm, we can increase the probability that specific types of errors will be detected, while keeping the size of the checksum, and the time required to check it, down to a reasonable size.

The simplest checksum algorithm is to add up all the data bytes (or—if you prefer a 16-bit checksum—words), discarding carries along the way. A noteworthy weakness of this algorithm is that if all of the data (including the stored checksum) is accidentally overwritten with 0s, this data corruption will be undetectable; the sum of a large block of zeros is also zero. The simplest way to overcome this weakness is to add a final step to the checksum algorithm: invert the result. That way, if the data and checksum are somehow overwritten with 0s, the test will fail because the proper checksum would be 0xFF.

Unfortunately, a simple sum-of-data checksum such as this one fails to detect many of the most common data errors. Clearly, if one bit of data is corrupted (switched from 1 to 0, or vice versa), the error would be detected. But what if two bits from the very same "column" happened to be corrupted alternately (the first switches from 1 to 0, the other from 0 to 1)? The proper checksum does not change, and the error would not be detected. If bit errors can occur, you will probably want to use a better checksum algorithm. We'll see one of these in the next section.

After computing the expected checksum, you'll need a place to store it. One option is to compute the checksum ahead of time and define it as a constant in the routine that verifies the data. This method is attractive to the programmer but has several shortcomings. It is possible that the data—and, as a result, the expected checksum—might change during the lifetime of the product. This is particularly likely if the data being tested is embedded software that will be periodically updated as bugs are fixed or new features added.

A better idea is to store the checksum at some fixed location in nonvolatile memory. For example, you might decide to use the very last location of the memory device being verified. This makes insertion of the checksum easy: just compute the checksum and insert it into the memory image prior to programming the memory device. When you recalculate the checksum, simply skip over the location that contains the expected result and compare the runtime checksum to the value stored there. Another good place to store the checksum is in another nonvolatile memory device. Both of these solutions work very well in practice.

Cyclic Redundancy Checks

A *cyclic redundancy check* (CRC) is a specific checksum algorithm that is designed to detect the most common data errors. The theory behind the CRC is quite mathematical and beyond the scope of this book. However, cyclic redundancy codes are frequently useful in embedded applications that require the storage or transmission of large blocks of data. What follows is a brief explanation of the CRC technique and some source code that shows how it can be implemented in C. Thankfully, you don't need to understand why CRCs detect data errors—or even how they are implemented—to take advantage of their ability to detect errors.

Here's a very brief explanation of the mathematics. When computing a CRC, think of the set of data as a very long string of 1s and 0s (called the *message*). This binary string is divided—in a rather peculiar way—by a smaller fixed binary string called the *generator polynomial*. The *remainder* of this binary long division is the CRC checksum. By carefully selecting the generator polynomial for certain desirable mathematical properties, you can use the resulting checksum to detect most (but never all) errors within the message. The strongest of these generator polynomials are able to detect all single- and double-bit errors, and all odd-length strings of consecutive error bits. In addition, greater than 99.99 percent of all burst errors—defined as a sequence of bits that has one error at each end—can be detected. Together, these types of errors account for a large percentage of the possible errors within any stored or transmitted binary message.

Generator polynomials with the best error-detection capabilities are frequently adopted as international standards. Two such standards are described in Table 6-4. Associated with each standard are its width (in bits), the generator polynomial, a binary representation of the polynomial (called the *divisor*), an initial value for the remainder, and a value to exclusive OR operation (XOR) with the final remainder.[*]

Table 6-4. International standard CRC parameters

Parameters	CRC16	CRC32
Checksum size (width)	16 bits	32 bits
Generator polynomial	$x^{16} + x^{15} + x^2 + 1$	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$
Divisor (polynomial)	0x8005	0x04C11DB7
Initial remainder	0x0000	0xFFFFFFFF
Final XOR value	0x0000	0xFFFFFFFF

---

[*] The divisor is simply a binary representation of the coefficients of the generator polynomial, each of which is either 0 or 1. To make this even more confusing, the highest-order coefficient of the generator polynomial (always a 1) is left out of the binary representation. For example, the polynomial in CRC16 has four nonzero coefficients. But the corresponding binary representation has only three 1s in it (bits 15, 2, and 0).

The following code can be used to compute any CRC formula that has a similar set of parameters. To make this as easy as possible, we have defined all of the CRC parameters as constants. To select the CRC parameters according to the desired standard, define one (and only one) of the macros CRC16 or CRC32.

```c
/* The CRC parameters.  Currently configured for CRC16. */
#define CRC_NAME                "CRC16"
#define POLYNOMIAL              0x8005
#define INITIAL_REMAINDER       0x0000
#define FINAL_XOR_VALUE         0x0000
#define REFLECT_DATA            TRUE
#define REFLECT_REMAINDER       TRUE
#define CHECK_VALUE             0xBB3D

/* The width of the CRC calculation and result. */
typedef uint16_t crc_t;

#define WIDTH                   (8 * sizeof(crc_t))
#define TOPBIT                  (1 << (WIDTH - 1))
```

The function crcCompute can be called over and over from your application to compute and verify CRC checksums.

```c
/***********************************************************************
 *
 * Function:    crcCompute
 *
 * Description: Compute the CRC of a given message.
 *
 * Notes:
 *
 * Returns:     The CRC of the message.
 *
 ***********************************************************************/
crc_t crcCompute(uint8_t const message[], uint32_t numBytes)
{
    crc_t    remainder = INITIAL_REMAINDER;
    uint32_t byte;
    int      nBit;

    /* Perform modulo-2 division, a byte at a time. */
    for (byte = 0; byte < numBytes; byte++)
    {
        /* Bring the next byte into the remainder. */
        remainder ^= (REFLECT_DATA(message[byte]) << (WIDTH - 8));

        /* Perform modulo-2 division, a bit at a time. */
        for (nBit = 8; nBit > 0; nBit--)
        {
            /* Try to divide the current data bit. */
            if (remainder & TOPBIT)
                remainder = (remainder << 1) ^ POLYNOMIAL;
            else
                remainder = (remainder << 1);
```

```
        }
    }

    /* The final remainder is the CRC result. */
    return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
}
```

A function named `crcFast` that uses a lookup table to compute a CRC more efficiently is included on this book's web site (*http://www.oreilly.com/catalog/embsys2*). Precomputing the remainders for all 256 possible bytes of data in advance (`crcInit`) substantially reduces the amount of processing done for each bit. These intermediate results are stored in a lookup table. By doing it this way, the CRC of a large message can be computed a byte at a time rather than bit by bit. This reduces the CRC calculation time significantly.

An additional benefit of splitting the computation between `crcInit` and `crcFast` is that the `crcInit` function need not be executed on the embedded system. In practice, the `crcInit` function could either be called during the target's initialization sequence (thus placing the CRC table in RAM), or it could be run ahead of time on your development computer with the results stored in the target device's ROM. The values in the table are then referenced over and over by `crcFast`.

Using Flash Memory

Flash memory offers advantages over other types of memory. Systems with flash memory can be updated in the field to incorporate new features or bug fixes discovered after the product has been shipped. This can eliminate the need to ship the unit back to the manufacturer for software upgrades. There are several issues that need to be considered when upgrading software for units in the field.

Limit downtime
    The timing of the upgrade should take place during downtime. Since the unit will probably not be able to function at its full capacity during the upgrade, you need to make sure that the unit is not performing a critical task. The customer will have to dictate the most convenient time.

Power failure
    How will the unit recover should power be removed (intentionally or otherwise) while the upgrade is taking place? If only a few bytes of the application image have been programmed into flash when the power is removed, you need a way to determine that an error occurred and prevent that code from executing. A solution may be to include a loader (similar to a debug monitor) that cannot be erased because it resides in protected flash sectors. One of the boot tasks for the loader is to check the flash memory for a valid application image (i.e., for a valid checksum). If a valid image is not present, the loader needs to know how to get a valid image onto the board, via serial port, network, or some other means.

Another solution for power failures may be to include a flash memory device that is large enough to store two application images: the current image and the old image. When new firmware is available, the old image is overwritten with the new software; the current image is left alone. Only after the image has been programmed properly and verified does it become the current image. This technique ensures that the unit always has a valid application image to execute should something bad happen during the upgrade procedure.

Upgrade code execution

From which memory chip will the software execute during the erase and programming of the new software? The software that downloads the image may be able to run from flash memory; however, the code to erase and reprogram a flash chip might need to be run from another memory device.

Device timing requirements

It is important to understand the timing requirements of the program and erase cycles for the particular flash device. It is best to make sure all data is present (and validated) before starting the programming cycle. You wouldn't want to start the programming the device and then be caught waiting for the rest of the new software to come in over a network connection. The device may have timing limits for program and erase cycles that cause the device to revert back to read mode if these limits are exceeded. The flash device driver would fail to write the data if this occurs.

Software image validity

It is important to validate the image that is written into the flash. This will ensure that the software is received into the unit correctly. The CRC algorithm presented earlier in this chapter may be sufficient to satisfy the validity of the upgrade software.

Security

If security of the image is an issue, you may need to find an algorithm to digitally sign and/or encrypt the new software. The validation and decryption of the software would then be performed prior to programming the new software into the flash memory.

Working with Flash Memory

From the programmer's viewpoint, flash is arguably the most complicated memory device ever invented. The hardware interface has improved somewhat since the original devices were introduced in 1988, but there is still a long way to go. Reading from flash memory is fast and easy, as it should be. In fact, reading data from a flash is not all that different from reading from any other memory device. The processor simply provides the address, and the memory device returns the data stored at that location. Most flash devices enter this type of "read" mode automatically whenever the system is reset; no special initialization sequence is required to enable reading.

Writing data to a flash is not as straightforward. Two factors make writes difficult. Firstly, each memory location must be erased before it can be rewritten. If the old data is not erased, the result of the write operation will be a mathematical combination of the old and new values.

The second thing that makes writes to a flash difficult is that at least one sector, or block, of the device must be erased; it is impossible to erase a single byte. The size of an individual sector varies from device to device, but each sector is usually on the order of several kilobytes. In addition, within the same device, different sector sizes may be used.

One other small difference is worth noting: the erase and write cycles take longer than the read cycle.

Flash Drivers

The process of erasing the old data and writing the new varies from one manufacturer to another and is usually rather complicated. These device programming interfaces are so awkward that it is usually best to add a layer of software to make the flash memory easier to use. If implemented, this hardware-specific layer of software is usually called the *flash driver*.

The purpose of a device driver in general is to hide the details of a specific device from the application software. In this case, the flash driver contains the specific method for writing to and erasing a specific flash device. The flash driver should present a simple *application programming interface* (API) consisting of the erase and write operations. Parts of the application software that need to modify data stored in flash memory simply call the driver and let it handle the details. This allows the application programmer to make high-level requests such as "Erase the block at address 0xD0000000" or "Write a block of data, beginning at address 0xD4000000." Distinct driver routines also keep the device-specific code separate, so it can be easily modified if another manufacturer's flash device is later used.

Flash device manufacturers typically include device drivers on their web sites. If you're looking for some example code, these web sites are a great place to start. Some of the example code may cover the very basic operations. In particular, these implementations may not handle any of the chip's possible errors. What if the erase operation never completes? You'll want to think through the problems that might arise when deploying your routines, and add error checking if necessary. More robust implementations may use a software time-out as a backup. For example, if the flash device doesn't respond within twice the maximum expected time (as stated in the device's datasheet), the routine could stop polling and indicate the error to the caller (or user) in some way.

Another feature enhancement is to include code for retries. If an erase or program cycle fails, the code could automatically retry the operation before returning a failure to the calling application.

Another thing that people sometimes do with flash memory is implement a small filesystem (similar to the FIS portion of RedBoot). Because the flash memory provides nonvolatile storage that is also rewriteable, it can be thought of as similar to any other secondary storage system, such as a hard drive. However, you must keep in mind the write cycle limitation of flash as well. In the filesystem case, the functions provided by the driver would be more file-oriented. Standard filesystem functions such as `open`, `close`, `read`, and `write` provide a good starting point for the driver's API. The underlying filesystem structure can be as simple or complex as your system requires. However, a well-understood format such as the File Allocation Table (FAT) structure, used by DOS, is good enough for most embedded projects.

Peripherals

*Each pizza glides into a slot like a circuit board into a
computer, clicks into place as the smart box interfaces
with the onboard system of the car. The address of the
customer is communicated to the car, which computes
and projects the optimal route on a heads-up display.*

—Neal Stephenson
  Snow Crash

In addition to the processor and memory, most embedded systems contain a handful of other hardware devices. Some of these devices are specific to each embedded system's application domain, while others—such as timers/counters and serial ports—are useful in a wide variety of systems. The most commonly used devices are often included within the same chip as the processor and are called *internal*, or *on-chip*, peripherals. Hardware devices that reside outside the processor chip are, therefore, said to be external peripherals. In this chapter, we'll discuss the most common software issues that arise when interfacing to a peripheral of either type.

Control and Status Registers

An embedded processor interacts with a peripheral device through a set of control and status registers. These registers are part of the peripheral hardware, and their locations, size, and individual meanings are features of the peripheral. For example, the registers within a serial controller are very different from those in a timer. In this section, we'll describe how to manipulate the contents of these control and status registers directly from your C language programs.

As discussed in Chapter 2, depending upon the design of the processor and board, peripheral devices are located either in the processor's memory space or within the I/O space. By far, the most common of the two types is memory-mapped peripherals, which are generally easier to work with.

Memory-mapped control and status registers can be made to look just like ordinary variables. To do this, you need simply declare a pointer to the register, or block of registers, and set the value of the pointer explicitly. Example code from previous chapters has already demonstrated access to peripheral registers, but let's take a closer look at the code. The GPIO registers in the PXA255 are memory-mapped so that pointers to registers look just like a pointer to any other integer variable. The following code declares the variable pGpio0Set as a pointer to a uint32_t—a 32-bit value representing the device's register—and explicitly initializes the variable to the address 0x40E00018. From that point on, the pointer to the register looks just like a pointer to any other integer variable.

```
uint32_t *pGpio0Set = (uint32_t *)(0x40E00018);
```

Note, however, one very important difference between device registers and ordinary variables in local memory. The contents of a device register can change without the knowledge or intervention of your program. That's because the register contents can also be modified by the peripheral hardware. By contrast, the contents of a variable in memory will not change unless your program modifies them explicitly. For that reason, we say that the contents of a device register are *volatile*, or subject to change without notice.

The keyword volatile should be used when declaring pointers to device registers. This warns the compiler not to make any assumptions about the data stored at that address. For example, if the compiler sees a write to the volatile location followed by another write to that same location, it will not assume that the first write is an unnecessary use of processor cycles. And in the case of reads, it will not assume that a second read of the same location will return the same result, as it would with a variable.

Here's an example that uses the keyword volatile to warn the compiler about the GPIO Pin Output Set register. The goal of this function is to write the value of the register at two different times, thereby setting two different GPIO pins high at different times:

```
uint32_t volatile *pGpio0Set = (uint32_t volatile *)(0x40E00018);

void gpioFunction(void)
{
    /* Set GPIO pin 0 high. */
    *pGpio0Set = 1;                 /* First write. */

    delay_ms(1000);

    /* Set GPIO pin 1 high. */
    *pGpio0Set = 2;                 /* Second write. */
}
```

If the volatile keyword was not used to declare the variable pGpio0Set, the optimizer would be permitted to change the operation of the code. For example, the compiler might remove the setting of pGpio0Set to 1 in the previous code because the

compiler can't see any purpose to this setting. If the compiler intervened in this manner, the GPIO pins would not operate as the software developer intended. So the volatile keyword instructs the optimization phase of the compilation to leave every change to a variable in place and to assume that the variable's contents cannot be predicted by earlier states.

It would be wrong to interpret the declaration statement of pGpio0Set to mean that the pointer itself is volatile. In fact, the value of the variable pGpio0Set will remain 0x40E00018 for the duration of the program (unless it is changed somewhere else, of course). The data that is pointed to, rather, is subject to change without notice. This is a very subtle point, and thinking about it too much may confuse you. Just remember that the location of a register is fixed, though its contents might not be. And if you use the volatile keyword, the compiler will assume the same.

You might also notice that the pointer to the GPSR0 register is declared as an unsigned integer. Registers sometimes consist of several subfields, and almost all of the values are positive by definition. For these reasons, embedded programmers typically use unsigned integer types for peripheral registers.
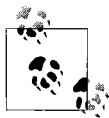
> Signed integers may be needed when reading samples from an analog-to-digital converter (A/D converter or ADC).

Bit Manipulation

The C language bitwise operators can be used to manipulate the contents of registers. These operators are & (AND), | (OR), ~ (NOT), ^ (XOR), << (left shift), and >> (right shift). The example code in the following sections shows how to test, set, clear, and toggle individual bits via a pointer to a timer status register called pTimerStatus.

In this section, we'll number the bits the way you need to think of them when creating masks. The least-significant bit is called bit 0, and it can be represented in a hexadecimal mask as 0x01; the most-significant bit in a byte is called bit 7, and it can be represented in a hexadecimal mask as 0x80.

Testing bits

The following code tests to see whether bit 3 is set in the timer status register using the & operator:

```
if (*pTimerStatus & 0x08)
{
    /* Do something here... */
}
```

In this case, we'll imagine that the value in the timer status register, contained in the variable pTimerStatus, is 0x4C; the & operator performs an AND operation with 0x08. The operation looks like this:

```
        0 1 0 0 1 1 0 0  (0x4C)
AND (&)
        0 0 0 0 1 0 0 0  (0x08)
        =======================
        0 0 0 0 1 0 0 0  (0x08)
```

Because the proper bit is set in the register, the code enters the `if` statement.

Setting bits

To set bit 4, the | operator is used as shown in the following code:

```
*pTimerStatus |= 0x10;
```

resulting in:

```
        0 1 0 0 1 1 0 0  (0x4C)
OR (|)
        0 0 0 1 0 0 0 0  (0x10)
        =================
        0 1 0 1 1 1 0 0  (0x5C)
```

Clearing bits

The code to clear bit 2 uses the & and ~ operators as follows:

```
*pTimerStatus &= ~(0x04);
```

For this operation, the inverse of 0x04 equals 0xFB. The &= operator sets bit 2 of the timer status register to 0, while leaving all other bits unchanged. The operation looks like this:

```
        0 1 0 1 1 1 0 0  (0x5C)
AND (&)
NOT (~) 1 1 1 1 1 0 1 1  (0xFB)
        =================
        0 1 0 1 1 0 0 0  (0x58)
```

Note that all bits in the register remain the same except for the bit we want to clear.

Toggling bits

It is sometimes useful to change a bit back and forth. For instance, you may want to blink an LED on and off. You may also want to toggle a bit back and forth, without having to check it first, and explicitly set or clear it. Toggling is done in C with the ^ operator. Here is the code to toggle bit 7 in the timer status register:

```
*pTimerStatus ^= 0x80;
```

This results in the following operation:

```
        0 1 0 1 1 0 0 0  (0x58)
XOR (^)
        1 0 0 0 0 0 0 0  (0x80)
        =================
        1 1 0 1 1 0 0 0  (0xD8)
```

Shifting bits

Another type of useful bitwise operation is a shift. For example, consider what happens to the value of the 8-bit unsigned integer bitCount that contains 0xAC and is shifted right by 1 bit. Code demonstrating a right shift follows:

```
bitCount >>= 1;
```

This results in:

```
        1 0 1 0 1 1 0 0  (0xAC)
>> by 1
        ==================
        0 1 0 1 0 1 1 0  (0x56)
```

In this case, a 0 is shifted in from the left. However, the C standard also allows the most significant bit to be repeated when the variable is signed. We recommend you use unsigned integers for variables on which you perform bit operations so that you will not have to worry about the different results on different compilers.

Assume the value of the 8-bit unsigned integer bitCount is again 0xAC and is shifted left by 2 bits:

```
bitCount <<= 2;
```

This results in:

```
        1 0 1 0 1 1 0 0  (0xAC)
<< by 2
        ==================
        1 0 1 1 0 0 0 0  (0xB0)
```

One reason to use a shift is if you want to perform an operation on each bit of a register in turn; you can create a bitmask (discussed in the next section) with 1 bit set or clear and shift it so you can operate on the individual bits of the register.

Bitmasks

A *bitmask* is a constant often used along with bitwise operators to manipulate one or more bits in a larger integer field. A bitmask is a constant binary pattern, such as the 16-bit hexadecimal literal 0x00FF, that can be used to mask specific bits. Bitmasks can be used with bitwise operators in order to set, test, clear, and toggle bits. Following are example bitmasks for the timer status register:

```
#define TIMER_COMPLETE          (0x08)
#define TIMER_ENABLE            (0xC0)
```

The bitmasks TIMER_COMPLETE and TIMER_ENABLE are descriptive names that correspond to specific bits in a peripheral's register. Using a symbolic (e.g., #define) bitmask allows you to write code that is more descriptive and almost self-commented. By replacing hexadecimal literals with words, the definition makes it easier for you

(or someone else) to understand the code at a later time. Here is an example of a bit-wise operation involving a bitmask:

```
if (*pTimerStatus & TIMER_COMPLETE)
{
    /* Do something here... */
}
```

Bitmask Macros

Here is a handy macro that will help you avoid typos in long hexadecimal literals:

```
#define BIT(X)                  (1 << (X))
```

To define a specific register bit in a bitmask, such as bit 22, use the macro as follows:

```
#define TIMER_STATUS            BIT(22)
```

Bitfields

A *bitfield* is a field of one or more bits within a larger integer value. Bitfields are useful for bit manipulations and are supported within a struct by C language compilers.

```
struct
{
    uint8_t  bit0   : 1;
    uint8_t  bit1   : 1;
    uint8_t  bit2   : 1;
    uint8_t  bit3   : 1;
    uint8_t  nibble : 4;
} foo;
```

Bits within a bitfield can be individually set, tested, cleared, and toggled without affecting the state of the other bits outside the bitfield.

To test bits using the bitfield, use code such as the following:

```
if (foo.bit0)
{
    /* Do other stuff. */
}
```

Here's how to test a wider field (such as two bits) using a bitfield:

```
if (foo.nibble == 0x03)
{
    /* Do other stuff. */
}
```

To set a bit using a bitfield, use this code:

```
foo.bit1 = 1;
```

And use code such as the following to set multiple bits in a bitfield:

```
foo.nibble = 0xC;
```

To clear a bit using the bitfield, use this code:

```
foo.bit2 = 0;
```

And to toggle a bit using the bitfield, use this:

```
foo.bit3 = ~foo.bit3; /* or !foo.bit3 */
```

There are some issues you must be aware of should you decide to use bitfields. Bitfields are not portable; some compilers start from the least significant bit, while others start from the most significant bit. In some cases, the compiler may require enclosing the bitfield within a union; doing this makes the bitfield code portable across ANSI C compilers.

In the following example, we use a union to contain the bitfield. In addition to making the bitfield code portable, the union provides wider register access.

```
union
{
    uint8_t byte;
    struct
    {
        uint8_t  bit0   : 1;
        uint8_t  bit1   : 1;
        uint8_t  bit2   : 1;
        uint8_t  bit3   : 1;
        uint8_t  nibble : 4;
    } bits;
} foo;
```

Instead of accessing only individual bits, the register can be written to as a whole. For example, the bitfield union, along with bitmasks, can be useful when initializing a register, as shown here:

```
foo.byte = (TIMER_COMPLETE | TIMER_ENABLE);
```

while individual bits are still accessible, as shown here:

```
foo.bits.bit2 = 1;
```

Bitmasks are more efficient than bitfields in certain instances. Specifically, a bitmask is usually a better way to initialize several bits in a register. For example, the following code initializes the timer status register by setting the two bits denoted by the macros and clearing all others:

```
*pTimerStatus = (TIMER_COMPLETE | TIMER_ENABLE);
```

Setting and clearing bits using a bitfield is no faster than using a bitmask; with some compilers, it can be slower to use a bitfield. One benefit of using bitfields is that individual bitfields may be declared volatile or const. This is useful when a register is writeable but contains a few read-only bits.

Struct Overlays

In embedded systems featuring memory-mapped I/O devices, it is sometimes useful to overlay a C struct onto each peripheral's control and status registers. Benefits of struct overlays are that you can read and write through a pointer to the struct, the register is described nicely by the struct, code can be kept clean, and the compiler does the address construction at compile time.

The following example code shows a struct overlay for a timer peripheral. If a peripheral's registers do not align correctly, reserved members can be included in the struct. Thus, in the following example, an extra field that you'll never refer to is included at offset 4 so that the control field lies properly at offset 6.

```
typedef struct
{
    uint16_t count;           /* Offset 0 */
    uint16_t maxCount;        /* Offset 2 */
    uint16_t _reserved1;      /* Offset 4 */
    uint16_t control;         /* Offset 6 */
} volatile timer_t;

timer_t *pTimer = (timer_t *)(0xABCD0123);
```
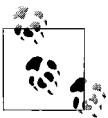
> Note that the individual fields of a struct, as well as the entire struct, can be declared volatile.

When you use a struct overlay to access registers, the compiler constructs the actual memory-mapped I/O addresses. The members of the `timer_t` struct defined in the previous example have the address offsets shown in Table 7-1.

Table 7-1. Timer peripheral struct address offsets

Struct member	Offset
count	0x00
maxCount	0x02
_reserved1	0x04
control	0x06

It is very important to be careful when creating a struct overlay to ensure that the sizes and addresses of the underlying peripheral's registers map correctly.

The bitwise operators shown earlier to test, set, clear, and toggle bits can also be used with a struct overlay. The following code shows how to access the timer peripheral's registers using the struct overlay. Here's the code for testing bits:

```
if (pTimer->control & 0x08)
{
    /* Do something here... */
}
```

Here's the code for setting bits:

```
pTimer->control |= 0x10;
```

Here's the code for clearing bits:

```
pTimer->control &= ~(0x04);
```

And here's the code for toggling bits:

```
pTimer->control ^= 0x80;
```

The Device Driver Philosophy

When it comes to designing device drivers, always focus on one easily stated goal: hide the hardware completely. This hiding of the hardware is sometimes called *hardware abstraction*. When you're finished, you want the device driver module to be the only piece of software in the entire system that reads and/or writes that particular device's control and status registers directly. In addition, if the device generates any interrupts, the interrupt service routine that responds to them should be an integral part of the device driver. The device driver can then present a generic interface to higher software levels to access the device. This eliminates the need for the application software to include any device-specific software. In this section, we'll explain why this philosophy is universally accepted and how it can be achieved.

Attempts to hide the hardware completely are difficult. Any programming interface you select will reflect the broad features of the device. That's to be expected. The goal should be to create a programming interface that would not need to be changed if the underlying peripheral were replaced with another in its general class. For example, all flash memory devices share the concepts of sectors (though the sector size may differ between chips). The following programming interface provided for a flash driver should work with any flash memory device:

```
void flashErase(uint32_t sector);
void flashWrite(uint32_t offset, uint8_t *pSrcAddr, uint32_t numBytes);
```

These two calls closely resemble the way all flash chips work in regard to reads and writes. An erase operation can be performed only on an entire sector. Once erased, individual bytes or words can be rewritten. But the interfaces here hide the specific features of the flash device and its functions from higher software levels, as desired.

Device drivers for embedded systems are quite different from their PC counterparts. In a general-purpose computer, the core of the operating system is distinct from the device drivers, which are often written by people other than the application developers. The operating system offers an interface that drivers must adhere to, while the rest of the system and applications depend on drivers doing so. For example, Microsoft's operating systems impose strict requirements on the software interface to a network card. The device driver for a particular network card must conform to this software interface, regardless of the features and capabilities of the underlying hardware. Application programs that want to use the network card are forced to use the networking API provided by the operating system and don't have direct access to the card itself. In this case, the goal of hiding the hardware completely is easily met.

By contrast, the application software in an embedded system can easily access the hardware. In fact, because all of the software is generally linked together into a single binary image, little distinction is made between the application software, operating system, and device drivers. Drawing these lines and enforcing hardware access restrictions are purely the responsibilities of the software developers. Both are design decisions that the developers must consciously make. In other words, the implementers of embedded software can more easily cheat on the software design than can their nonembedded peers.

The benefits of good device driver design are threefold:

- Because of the modularity, the structure of the overall software is easier to understand. In addition, it is easier to add or modify features of the overall application as it evolves and matures, even in deployed units.

- Because there is only one module that ever interacts directly with the peripheral's registers, the state of the hardware device can be more accurately tracked.

- Software changes that result from hardware changes are localized to the device driver, thereby making the software more portable.

Each of these benefits can and will help to reduce the total number of bugs in your embedded software and enhance the reusability of your code across systems. But you have to be willing to put in a bit of extra effort up front, at design time, in order to realize the savings.

Figure 7-1 shows the basic software layers for an embedded system. As shown in this figure, a device driver sits logically just above the hardware and contains the "knowledge" of how to operate that particular piece of hardware.
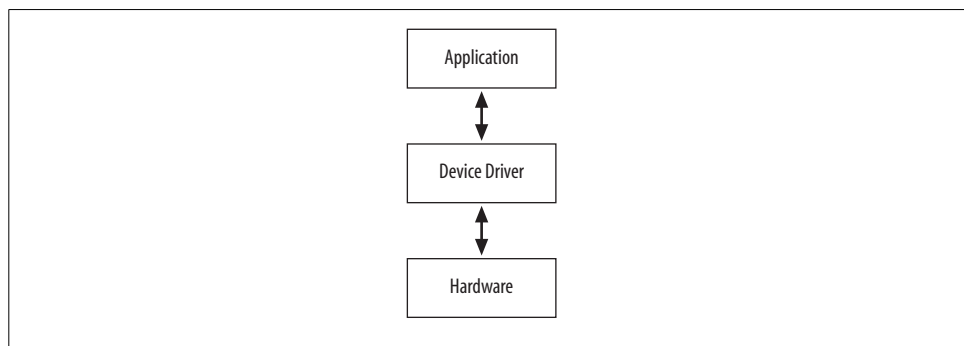
Figure 7-1. Embedded system software layers

Because the device driver contains the code to operate the hardware, the application software does not need to be complicated by these details. For example, looking back at the Blinking LED program, the file *led.c* is the LED device driver. This file contains all of the knowledge about how to initialize and operate the LED on the Arcom board. The LED device driver provides an API consisting of `ledInit` and `ledToggle`. The application in *blink.c* uses this API to toggle the LED. The application does not need to concern itself with the operation of GPIO registers in the PXA255 in order to get the LED to perform a certain task.

The philosophy of hiding all hardware specifics and interactions within the device driver usually consists of the five components in the following list. To make driver implementation as simple and incremental as possible, these elements should be developed in the order they are presented.

1. An interface to the control and status registers.

   For a commonly used memory-mapped I/O, the first step in the driver development process is to create a representation of the memory-mapped registers of your device. This usually involves studying the databook for the peripheral and creating a table of the control and status registers and their offsets. The method for representing the control and status registers can be whatever style you feel comfortable implementing.

2. Variables to track the current state of the physical (and logical) devices.

   The second step in the driver development process is to figure out what state variables you will need. For example, we'll probably need to define variables to

remind us whether the hardware has been initialized. Write-only registers are also good candidates for state variables.

Some device drivers create more than one software device for the underlying hardware. The additional instance is a purely *logical device* that is implemented over the top of the basic peripheral hardware. Think about a timer, for example. It is easy to imagine that more than one software timer could be created from a single hardware timer/counter unit. The timer/counter unit would be configured to generate a periodic clock tick, say, every millisecond, and the device driver would then manage a set of software timers of various lengths by maintaining state information for each.

3. A routine to initialize the hardware to a known state.

Once you know how you'll track the state of the physical (and logical) device, you can begin to write the functions that actually interact with and control the hardware. It is probably best to begin with the hardware initialization routine. You'll need that one first anyway, and it's a good way to get familiar with device interaction.

4. An API for users of the device driver.

After you've successfully initialized the device, you can start adding other functionality to the driver. A first step in the design for the device driver is to settle on the names and purposes of the various routines, as well as their respective parameters and return values. After this step, all that's left to do is implement and test each API function. We'll see examples of such routines in the next section.

5. Interrupt service routines.

It's best to design, implement, and test most of the device driver routines before enabling interrupts for the first time. Locating the source of interrupt-related problems can be quite challenging. If you add possible bugs present in the other driver modules to the mix, it could even become impossible. It's far better to use polling to get the guts of the driver working. That way you'll know how the device works (and that it is indeed working) when you start looking for the source of your interrupt problems—and there will almost certainly be problems.

A Serial Device Driver

The device driver example that we're about to discuss is designed to control a serial port. The hardware for this device driver uses a UART (which is pronounced "you-art" and stands for *Universal Asynchronous Receiver Transmitter*) peripheral contained within the PXA255 processor. A UART is a component that receives and transmits asynchronous serial data. *Asyncrhonous* means that data can come at unexpected intervals, similar to the input from a keyboard. A UART accepts a parallel byte from the processor. This byte is serialized, and each bit is transmitted at the appropriate time. Reception works in the reverse.

The PXA255 processor has four on-chip UARTs. For this example, we will use the Full Function UART (FFUART), which is connected to the Arcom board's COM1 port. (Note that this is the same COM port used on the Arcom board by RedBoot, eliminating the need to switch cables. The FFUART registers start at address 0x40100000.)

Before writing any software for the serial device driver, you should understand the hardware block diagram—that is, how the signals go from the peripheral to the outside world and back. This typically includes looking over the relavant portion of the schematic and gathering the datasheets for the different ICs. A block diagram for the serial port is shown in Figure 7-2.
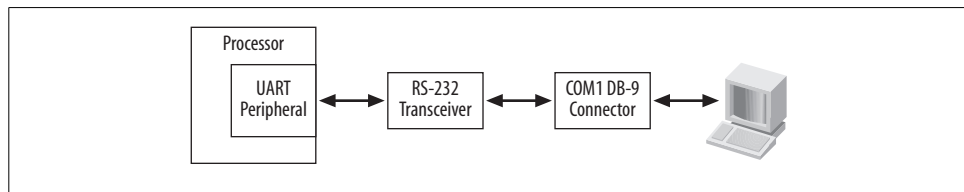
Figure 7-2. Arcom board serial port block diagram

As shown in Figure 7-2, the PXA255 UART connects to the RS-232 Transceiver, which then connects to the COM1 DB-9 connector on the Arcom board. The transceiver converts the voltage level that the Arcom board's processor uses to RS-232 voltage levels. This allows the Arcom board's UART to communicate with a PC via its serial port.

The next step is to understand how the particular peripheral works. What ICs need to be programmed in order to control the peripheral? For this serial driver, we only need to focus on the UART peripheral registers in the processor. The information about these registers is contained in the processor's documentation.

For information about the PXA255 processor UARTs, check the *PXA255 Processor Developer's Manual*—specifically, Section 10: UARTs. Information about interrupts is contained in Section 4.2: Interrupt Controller. While reading this documentation, the goal is to get an understanding of several different concepts, including:

- The register structure for controlling the peripheral—that is, how to set up communications and how to get data into and out of the peripheral
- The addresses of the control and status registers
- The method that will be used for the peripheral's operation (namely, polling or interrupts)
- If using interrupts, what conditions can cause interrupts, how the software driver is informed when an interrupt occurs, and how the interrupt is acknowledged

Get a firm grasp on what the device driver will need to do to get the peripheral to perform its task within the system. Once these initial steps are complete, you can move on to the task of writing the device driver software.

Register interface

The first step for the serial device driver is to define the register interface. For this example, we use a struct overlay for the UART registers, which are memory-mapped. The struct overlay, uart_t, is shown here:

```
typedef struct
{
    uint32_t data;
    uint32_t interruptEnable;
    uint32_t interruptStatus;
    uint32_t uartConfig;
    uint32_t pinConfig;
    uint32_t uartStatus;
    uint32_t pinStatus;
} volatile uart_t;
```

The variable pSerialPort is used to access the UART registers at address 0x40100000 and is defined as:

```
uart_t *pSerialPort = (uart_t *)(0x40100000);
```

State variables

Next, we define variables to track the current state of the hardware. A struct of serial driver parameters called serialparams_t is defined. The global variable gSerialParams is used in the serial device driver to encapsulate and help organize the configuration parameters.

The variable bInitialized is used in the serial initialization routine to keep track of the hardware configuration state. You may notice that in the following code, enumerated types are defined for the parity (parity_t), data bits (databits_t), and stop bits (stopbits_t). The enumerators are set to bitmask values in the UART configuration register. The enumerations simplify the UART configuration programming and help make the code more readable.

```
/* UART Config Register (LCR) Bit Descriptions */
#define DATABITS_LENGTH_0        (0x01)
#define DATABITS_LENGTH_1        (0x02)
#define STOP_BITS                (0x04)
#define PARITY_ENABLE            (0x08)
#define EVEN_PARITY_ENABLE       (0x10)

typedef enum {PARITY_NONE, PARITY_ODD = PARITY_ENABLE,
              PARITY_EVEN = (PARITY_ENABLE | EVEN_PARITY_ENABLE)} parity_t;
```

```
    typedef enum {DATA_5, DATA_6 = DATABITS_LENGTH_0, DATA_7 = DATABITS_LENGTH_1,
                  DATA_8 = (DATABITS_LENGTH_0 | DATABITS_LENGTH_1)} databits_t;

    typedef enum {STOP_1, STOP_2 = STOP_BITS} stopbits_t;

    typedef struct
    {
        uint32_t dataBits;
        uint32_t stopBits;
        uint32_t baudRate;
        parity_t parity;
    } serialparams_t;

    serialparams_t gSerialParams;
```

Initialization routine

The initialization routine serialInit sets up the default communication parameters for the serial device driver. The UART registers are programmed in the routine serialConfig, which gets passed in the gSerialParams variable. The variable bInitialized is used to ensure that the serial port is configured only once.

```
    /************************************************************************
     *
     * Function:    serialInit
     *
     * Description: Initialize the serial port UART.
     *
     * Notes:       This function is specific to the Arcom board.
     *              Default communication parameters are set in
     *              this function.
     *
     * Returns:     None.
     *
     ************************************************************************/
    void serialInit(void)
    {
        static int bInitialized = FALSE;

        /* Initialize the UART only once. */
        if (bInitialized == FALSE)
        {
            /* Set the communication parameters. */
            gSerialParams.baudRate = 115200;
            gSerialParams.dataBits = DATA_8;
            gSerialParams.parity = PARITY_NONE;
            gSerialParams.stopBits = STOP_1;

            serialConfig(&gSerialParams);

            bInitialized = TRUE;
        }
    }
```

Device driver API

Now additional functionality can be added by defining other serial device driver API functions. A serial device driver API should have functions to send and receive characters. For sending characters, the function serialPutChar is used; for receiving characters, serialGetChar is used.

The serial device driver API function serialPutChar waits until the transmitter is ready and then sends a single character via the serial port. Transmitting is done by writing to the UART data register. The following code shows the serialPutChar function.

```
#define TRANSMITTER_EMPTY        (0x40)

/***********************************************************************
 *
 * Function:    serialPutChar
 *
 * Description: Send a character via the serial port.
 *
 * Notes:       This function is specific to the Arcom board.
 *
 * Returns:     None.
 *
 ***********************************************************************/
void serialPutChar(char outputChar)
{
    /* Wait until the transmitter is ready for the next character. */
    while ((pSerialPort->uartStatus & TRANSMITTER_EMPTY) == 0)
        ;

    /* Send the character via the serial port. */
    pSerialPort->data = outputChar;
}
```

The serial device driver API function serialGetChar waits until a character is received and then reads the character from the serial port. To determine whether a character has been received, the data ready bit is checked in the UART status register. The character received is returned to the calling function. Here is the serialGetChar function:

```
#define DATA_READY               (0x01)

/***********************************************************************
 *
 * Function:    serialGetChar
 *
 * Description: Get a character from the serial port.
 *
 * Notes:       This function is specific to the Arcom board.
 *
 * Returns:     The character received from the serial port.
 *
 ***********************************************************************/
char serialGetChar(void)
```

```
{
    /* Wait for the next character to arrive. */
    while ((pSerialPort->uartStatus & DATA_READY) == 0)
        ;

    return pSerialPort->data;
}
```

Because this serial device driver does not use interrupts, the final step in the device driver philosophy—implementing device driver interrupt service routines—is skipped.

Testing the Serial Device Driver

Now that the serial device driver is implemented, we need to verify that it operates correctly. It is important to check the individual functions of your new API before integrating the driver into the system software.

To test the serial device driver, the Arcom board's COM1 port must be connected to a PC's serial port. After making that connection, start a terminal program, such as HyperTerminal or minicom, on the PC. (The serial port parameters should not need to be changed, because the default serial device driver parameters are the same ones used by RedBoot.)

The main function demonstrates how to exercise the serial device driver's functionality. You might notice that this software has the beginnings of a command-line interface—an indispensable tool commonly implemented in embedded systems.

First, the serial device driver is initialized by calling serialInit. Then several characters are output on the PC's serial port to test the serialPutChar function. If the serial device driver is operating properly, you should see the message start output on your PC's terminal screen.

Next, a while loop is entered that checks whether a character has been received by calling serialGetChar. If a character comes into the serial port, it is echoed back. If the user enters q in the PC's terminal program, the program exits; otherwise, the loop continues and checks for another incoming character.

```
#include "serial.h"

/**********************************************************************
 *
 * Function:    main
 *
 * Description: Exercise the serial device driver.
 *
 * Notes:
 *
 * Returns:     This routine contains an infinite loop, which can
 *              be exited by entering q.
 *
 **********************************************************************/
```

```c
int main(void)
{
    char rcvChar = 0;

    /* Configure the UART for the serial driver. */
    serialInit();

    serialPutChar('s');
    serialPutChar('t');
    serialPutChar('a');
    serialPutChar('r');
    serialPutChar('t');
    serialPutChar('\r');
    serialPutChar('\n');

    while (rcvChar != 'q')
    {
        /* Wait for an incoming character. */
        rcvChar = serialGetChar();

        /* Echo the character back along with a carriage return and line feed. */
        serialPutChar(rcvChar);
        serialPutChar('\r');
        serialPutChar('\n');
    }

    return 0;
}
```

Extending the Functionality of the Serial Device Driver

Although the serial driver is very basic, it does have core functionality that you can build upon to develop a more robust (and more useful) program. This device driver provides a platform for learning about the operation of UARTs. Following is a list of possible extensions you can use to expand the functionality of this driver. Keep these in mind for other drivers you develop as well.

Selectable configuration

You can change serialInit to take a parameter that allows the calling function to specify the initial communication parameters, such as baud rate, for the serial port.

Error checking

It is important for the device driver to do adequate error checking. Another enhancement would be to define error codes (such as parameter error, hardware error, etc.) for the device driver API. The functions in the device driver would then use these error codes to return status from the attempted operation. This allows the higher-level software to take note of failures and/or retry.

Additional APIs

Adding `serialGetStr` and `serialPutStr` (which would require buffering of the receive and transmit data) might be useful. The implementation of the string functions could make use of the `serialGetChar` and `serialPutChar` functions, if it were reasonably efficient to do so.

FIFO usage

Typically, UARTs contain FIFOs for the data received and transmitted. Using these FIFOs adds buffering to both the receive and transmit channels, making the UART driver more robust.

Interrupts

Implementing UART interrupts for reception and transmission is usually better than using polling. For example, in the function `serialGetChar`, using interrupts would eliminate the need for the driver to sit in a loop waiting for an incoming character. The application software is thus able to perform other work while waiting for data to be received.

Device Driver Design

Most embedded systems have more than one device driver. In fact, sometimes there might be dozens. As your experience grows, you will need to understand the way different devices in the system interact with each other. You will also have to consider how the application software will use the device driver so that you can provide an adequate API.

You will need to have a good understanding of the overall software design and be aware of possible issues in the system. Getting input from multiple sources can lead to a better design. Here are some areas to consider when designing a software architecture that includes various device drivers:

Interrupt priorities

If interrupts are used for the device drivers in a system, you need to determine and set appropriate priority levels.

Complete requirements

You need to be aware of the requirements of the various peripherals in the system. You don't want to design and implement your software in a manner that unknowingly handicaps the peripheral from operating as it is intended. This can cause such major problems that the product might not be usable until the system functions as specified. It's a good idea to use software design reviews to flush out any potential problems that might have been overlooked by an individual developer.

Resource usage

It is important to understand what resources are necessary for each device driver. For example, imagine designing an Ethernet device driver in a system with a very limited amount of memory. This limitation would affect the buffering scheme implemented in the Ethernet driver. The driver might accommodate the storage of only a few incoming packets, which would affect the throughput of the Ethernet interface.

Resource sharing

Beware of possible situations where multiple device drivers need to access common hardware (such as I/O pins) or common memory. This can make it difficult to track down bugs if the sharing scheme is not thoroughly thought out ahead of time. We will take a look at mechanisms for resource sharing in Chapters 8 and 10.

Interrupts

*And, as Miss [Florence] Nightingale was so
vehemently to complain—"women never have an half
hour… that they can call their own"—she was always
interrupted.*
—Virginia Woolf
  A Room of One's Own

In this chapter, we'll take a look at interrupts—a sophisticated way of managing relationships with external devices. Interrupts are an important aspect of embedded software development and one that programmers need to study carefully in order to create efficient applications. The start of this chapter gives an introduction to interrupts and different characteristics associated with them. It is important to understand what happens when an interrupt event occurs and how the interrupt is processed. Although the implementation of interrupts is processor-specific, much of the material in this chapter applies to all processors. Finally, we will expand on the Blinking LED example by using an interrupt found in practically all processors.

Overview

An interrupt can be used to signal the processor for all sorts of events—for example, because data has arrived and can be read, a user flipped a switch, or a specific amount of time has elapsed.

Interrupts allow developers to separate time-critical operations from the main program to ensure they are processed in a prioritized manner. Because interrupts are asynchronous events, they can happen at any time during the main program's execution.

Figure 8-1 shows two alternative wiring diagrams of peripherals connected to the processor interrupt pins.
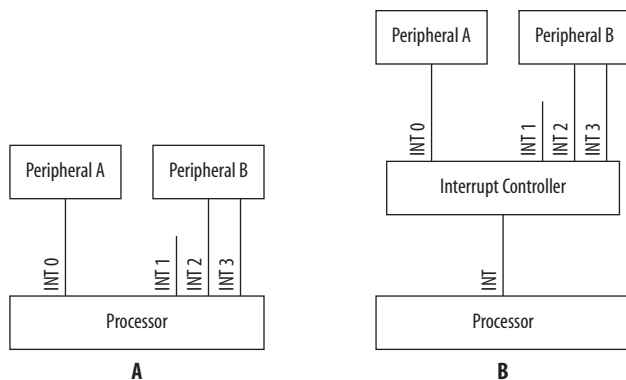
Figure 8-1. Interrupt wiring

Interrupts and Related Events

The term *interrupt* is used to cover several different hardware features. All of these mechanisms are used to divert processing of the main program in order to handle an event, but they are invoked in different circumstances and need to be treated by a programmer in different ways, so a brief listing is useful to distinguish them. Note that certain processors may define these terms differently.

Exception

A detected error condition sometimes called a software interrupt. For example, performing a divide by zero causes an exception. A software interrupt is also used by a program to perform various debug functions, such as breakpoints. Exceptions are synchronous events.

Interrupt

An asynchronous electrical signal asserted by a peripheral to the processor.

Trap

An interruption of a program that is caused by the processor's internal hardware. Traps are synchronous events.

Asynchronous events are not related in time to any other event known within the system. *Synchronous* events occur as the result of another event within the system. For instance, a signal indicating that somebody has opened a door to the room is asynchronous, whereas an exception caused by dividing by zero is synchronous.

In this chapter, we focus on interrupts in the narrower sense.

Figure 8-1(a) shows peripherals connected directly to the interrupt pins of the processor. In this case, the processor contains an interrupt controller on-chip to manage and process incoming interrupts. The PXA255 has an internal interrupt controller.

Figure 8-1(b) shows the two peripherals connected to an external interrupt controller device. An *interrupt controller* multiplexes several input interrupts into a single output interrupt. The controller also allows control over these individual input interrupts for disabling them, prioritizing them, and showing which are active.

Because many embedded processors contain peripherals on-chip, the interrupts from these peripherals are also routed to the interrupt controller within the main processor. Sometimes more interrupts are required in a system than there are interrupt pins in the processor. For these situations, peripherals can share an interrupt. The software must then determine which device caused the interrupt.

Interrupts can be either *maskable* or *nonmaskable*. Maskable interrupts can be disabled and enabled by software. Nonmaskable interrupts (*NMI*) are critical interrupts, such as a power failure or reset, that cannot be disabled by software.

A complete listing of the interrupts in your system can be constructed from information in the reference manuals for your processor and board. For example, the interrupts supported by the PXA255 processor are detailed in the *PXA255 Processor Developer's Manual*. A partial list of the supported interrupts for the PXA255 is shown in Table 8-1. We will take a look at what the interrupt number means shortly.

Table 8-1. Partial interrupt list for PXA255 processor

Interrupt number	Interrupt source
8	GPIO Pin 0
9	GPIO Pin 1
11	USB
26	Timer 0
27	Timer 1
28	Timer 2

Priorities

Because interrupts are asynchronous events, there must be a way for the processor to determine which interrupt to deal with first should multiple interrupts happen simultaneously. The processor defines an *interrupt priority* for all of the interrupts and exceptions it supports. The interrupt priorities are found in the processor's documentation.

For example, the ARM processor supports six types of interrupts and exceptions.[*] The priorities of these interrupts and exceptions are shown in Table 8-2. This table is contained in the *XScale Microarchitecture User's Manual*.

---

[*] ARMv6 also includes an imprecise abort exception priority between IRQ and prefetch abort.

Table 8-2. ARM processor exception and interrupt priorities

Priority	Exception/interrupt source
1 (highest)	Reset
2	Data abort
3	Fast Interrupt Request (FIQ)
4	Interrupt Request (IRQ)
5	Prefetch abort
6 (lowest)	Undefined instruction or software interrupt

Typically, when an interrupt occurs, a processor disables all interrupts at the same-or lower-priority levels. If multiple interrupts are waiting to be processed or are *pending*, the priority associated with the interrupts determines the order in which they are executed.

The method for handling interrupts at different priorities is very processor-specific. The following text describes some scenarios for handling interrupts at different priorities.

When a higher-priority interrupt occurs while a lower-priority interrupt is being processed, the processing of the lower-priority interrupt is suspended and the higher-priority interrupt is handled. Once the higher-priority interrupt completes, the processing of the lower-priority interrupt continues. This is called *interrupt nesting*.

If a lower-priority interrupt occurs while the processor is handling a higher-priority interrupt, the lower-priority interrupt is left pending until the higher-priority interrupt finishes executing.

When an interrupt occurs at the same priority as the interrupt currently being processed, the interrupt currently being processed is allowed to finish. Then the processing of the next interrupt starts. In this case, interrupt nesting can also occur if the currently executing interrupt reenables interrupts at its own priority level. In other words, an interrupt can allow itself to be interrupted by other interrupts that are at the same priority level.
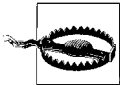
> Be careful when nesting interrupts. Additional forethought must go into the sizing of the stack, because each ISR that is interrupted must have its register state saved on the stack. This could lead to a stack overflow.

The interrupt priority is set by hardware design, by software, or, in some cases, by a combination of the two. If we look back at the wiring diagram in Figure 8-1(a), we see that the processor has four interrupt pins (INT0 through INT3). For this

example, we'll assume the processor gives INT0 the highest priority, followed by INT1, INT2, and INT3. The hardware designer must wire the interrupt pins so that the correct interrupt priorities are assigned to the various peripheral interrupts. In this case, the interrupt from Peripheral A has the highest priority.

Some interrupt controllers allow the priorities of interrupts to be set in software. In this case, the interrupt controller typically has registers that set the priorities of the various interrupts.

Levels and Edges

Level-sensitive interrupts cause the processor to respond as long as the interrupt signal is at the specified level. These interrupts are either high- or low-level-sensitive. *Edge-sensitive* interrupts cause the processor to respond when the interrupt signal goes through a transition. These interrupts are specified as rising or falling edge-sensitive. Figure 8-2 shows signals for a level-sensitive and edge-sensitive interrupt.
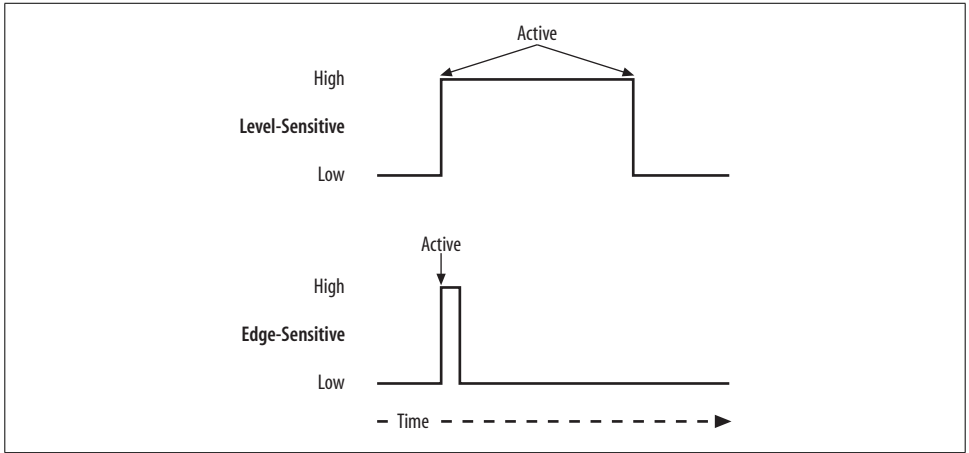
Figure 8-2. Level- and edge-sensitive interrupt signals

In Figure 8-2, the level-sensitive interrupt is active high. The time when the interrupt is active is shown in the signal diagram, which is the time when the signal is at the higher voltage. The interrupt signal on the bottom of Figure 8-2 is a rising edge-sensitive interrupt. It is active when the signal transitions from low to high, is held high for a certain minimum time (typically two or three processor clocks), and then it returns to low again.

There are issues related to both types of interrupts. For example, an edge-sensitive interrupt can be missed if a subsequent interrupt occurs before the initial interrupt is serviced. Conversely, a level-sensitive interrupt constantly interrupts the processor as long as the interrupt signal is asserted.

Most peripherals assert their interrupt until it is acknowledged. Some processors, such as the Intel386 EX, contain registers that can be programmed to support either level-sensitive or edge-sensitive interrupts on individual interrupts. Thus the sensitivity selection affects detection of new interrupts on that signal.

Acknowledging an interrupt tells the interrupting device that the processor has received the interrupt and queued it for processing. The method of acknowledging an interrupt can vary from reading an interrupt controller register to clearing an interrupt pending bit. Once the interrupt is acknowledged, the peripheral will deassert the interrupt signal. Some processors have an interrupt acknowledge signal that takes care of this automatically in hardware.

Enabling and Disabling

Maskable interrupts can be disabled and enabled either individually or globally. Nonmaskable interrupts, as the name implies, cannot be disabled. In the PXA255 processor, individual interrupts are masked in the Interrupt Controller Mask Register (ICMR). This register is shown in Figure 8-3.
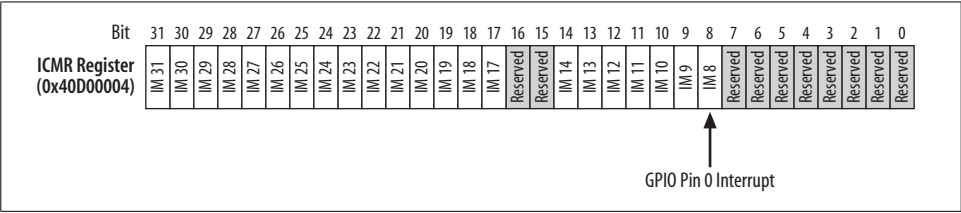
Figure 8-3. PXA255 Interrupt Controller Mask Register

The ICMR is located at address 0x40D00004. Figure 8-3 shows the Interrupt Mask (IM) for each of the interrupts supported in the PXA255. Setting the corresponding bit to 1 in the ICMR allows that particular source to generate interrupts; setting the corresponding bit to 0 masks that interrupt source.

For example, imagine that the interrupt pin from a peripheral is connected to GPIO pin 0 on the PXA255 processor. Table 8-1 shows that the GPIO pin 0 interrupt source is assigned to interrupt number 8. Therefore, to enable the GPIO pin 0 interrupt, bit 8 of the ICMR is set to 1. If an interrupt occurs when the GPIO pin 0 interrupt is enabled, it is routed to the interrupt controller for processing. To mask the GPIO pin 0 interrupt source, set bit 8 of the ICMR to 0. If an interrupt occurs while the source is disabled, the interrupt is ignored.

Each processor typically has a global interrupt enable/disable bit in one of its registers. The PXA255 has two bits in the Current Program Status Register (CPSR) that globally disable all interrupts.
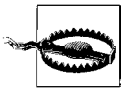
It is important to remember to reenable interrupts in your software after you have disabled them. This is a common problem that can lead to unexplained behavior in the system. If interrupts are disabled at the entry to a function, ensure that all software paths that exit the routine reenable interrupts.

You generally cannot access the global interrupt flags directly using the C language. In this case, you need to write assembly code to enable and disable global interrupts. Some compiler libraries, such as those for the x86 family of processors, contain functions to handle global interrupt enabling (with the enable function) and global interrupt disabling (with the disable function).

Interrupt Map

Now that we have an understanding of how an interrupt occurs, let's take a look at how the processor goes about dealing with an interrupt in software. After the processor is reset, either by cycling power or asserting the reset signal, interrupts are disabled. One of the jobs of the startup code is to enable global interrupts once the system is ready for them. Part of the procedure for ensuring that the system is ready for interrupts is installing software to handle them. An interrupt service routine, which carries out the basic action necessary to deal with the interrupt, is associated with each interrupt.

In order for the processor to execute the correct ISR, a mapping must exist between interrupt sources and ISR functions. This mapping usually takes the form of an *interrupt vector table*. The vector table, located at a memory address known to the hardware, is usually an array of pointers to functions. The processor uses the interrupt number (a unique number associated with each interrupt) as its index into this array. In some processors, the value stored in the vector table array is usually the address of the ISR to be executed. Other processors actually have instructions stored in the array (commonly called a *trampoline*) to jump to the ISR.

For the ARM processor, the addresses in the interrupt vector table are at fixed locations in memory. These addresses are listed in Table 8-3.

Table 8-3. ARM interrupt vector table

Exception/interrupt source	Address
Reset	0x00000000
Undefined instruction	0x00000004
Software interrupt	0x00000008
Prefetch abort	0x0000000C
Data abort	0x00000010
IRQ	0x00000018
FIQ	0x0000001C

The addresses in Table 8-3 are locations in memory used by the ARM processor to execute the ISR for a particular interrupt. Information about the interrupt vector table is contained in the documentation about the processor. Because the addresses in the ARM interrupt vector table are 32 bits apart, the code installed in the interrupt vector table is a jump to the real ISR.

It is critical for the programmer to install an ISR for all interrupts, even the interrupts that are not used in the system. If an ISR is not installed for a particular interrupt and the interrupt occurs, the execution of the program can become undefined (commonly called "going off into the weeds").

A good procedure is to install a default ISR in the interrupt vector table for any interrupt not used. The default ISR ensures that all interrupts are processed and acknowledged, allowing the main program to continue executing. Have your startup code initialize all interrupts in the vector table to the default handler to ensure there are no unhandled interrupts. Then install ISRs for specific interrupts used in the system.

The first part of initializing the interrupt vector table is to create an interrupt map that organizes the relevant information. An interrupt map is a table that contains a list of interrupt numbers and the devices to which they refer. This information should be included in the documentation provided with the board. A partial interrupt map for the Arcom board is shown in Table 8-4.

Table 8-4. Partial interrupt map for the Arcom board

Interrupt number	Interrupt source
8	Ethernet
11	USB
21	Serial Port 2
22	Serial Port 1
26	Timer 0
27	Timer 1
28	Timer 2

This table is similar to the interrupt list for the PXA255 processor shown in Table 8-1. However, this table shows the interrupt sources that are specific to the Arcom board.

Once again, our goal is to translate the information in the table into a form that is useful for the programmer. The interrupt map table should go into your project notebook for future reference. After constructing an interrupt map such as the one in Table 8-4, you should add a third section to the board-specific header file. Each line of the interrupt map becomes a single #define within the file, as shown here:

```
/*************************************************************************
 *   Interrupt Map
 *************************************************************************/
```

```
#define ETHERNET_INT            (8)
#define USB_INT                 (11)
#define SERIAL2_INT             (21)
#define SERIAL1_INT             (22)
#define TIMER0_INT              (26)
#define TIMER1_INT              (27)
#define TIMER2_INT              (28)
```

Interrupt Service Routine

Let's take a closer look at the ISR. The ISR is the function called when a particular interrupt occurs. Its central purpose is to process the interrupt and then return control to the main program. Typically, ISR functions have no arguments passed into them; they can never return a value.

In order to keep the impact of interrupts on the execution of the main program to a minimum, it is important to keep interrupt routines short. If additional processing is necessary for a particular interrupt, it is better to do this outside of the ISR. Keeping ISRs short also aids in ISR debug, which can be difficult. When it is done by a specific function, completion of the interrupt handling outside the ISR is called a *deferred service routine* (DSR).

Regardless of the specific processing required by the ISR, the ISR is responsible for doing the following things:

Saving the processor context
> Because the ISR and main program use the same processor registers, it is the responsibility of the ISR to save the processor's registers before beginning any processing of the interrupt. The processor *context* consists of the instruction pointer, registers, and any flags. Some processors perform this step automatically.

Acknowledging the interrupt
> The ISR must clear the existing interrupt, which is done either in the peripheral that generated the interrupt, in the interrupt controller, or both.

Restoring the processor context
> After interrupt processing, in order to resume the main program, the values that were saved prior to the ISR execution must be restored. Some processors perform this step automatically.

Some compilers include the keyword interrupt or something similar. This enables the compiler to automatically generate the code used to save the context when the ISR is entered, and to restore the context when the ISR is exited. An example of code that includes the interrupt keyword follows:

```
interrupt void interruptServiceRoutine(void)
{
    /* Process the interrupt. */
}
```

The documentation for the compiler shows whether the `interrupt` keyword is supported. If the compiler does not support this keyword, a compiler-specific `#pragma` may be required to declare an ISR. The GNU compiler *gcc* uses a third approach, involving the compiler-specific keyword `__attribute__`, which takes options as arguments, as shown here:

```
void interruptServiceRoutine() __attribute__ ((interrupt ("IRQ")));
```

Some processors, such as certain Microchip PICs, can have only one ISR for all interrupts. This ISR must determine the source of the interrupt by checking each potential interrupt source. In this case, it is a good idea to check the most important interrupt first. The technique used by the ISR to determine that the interrupt source is hardware-specific.
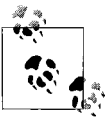
> When designing your software, it is typically best to include the ISR for a particular device in the driver for that peripheral. This keeps all the device-specific code for a particular peripheral isolated in a single module.

Figure 8-4 is a graphical representation of the interrupt process. For this example, we will assume the Ethernet network interface controller generates the interrupt, although this process is relevant for any interrupt.
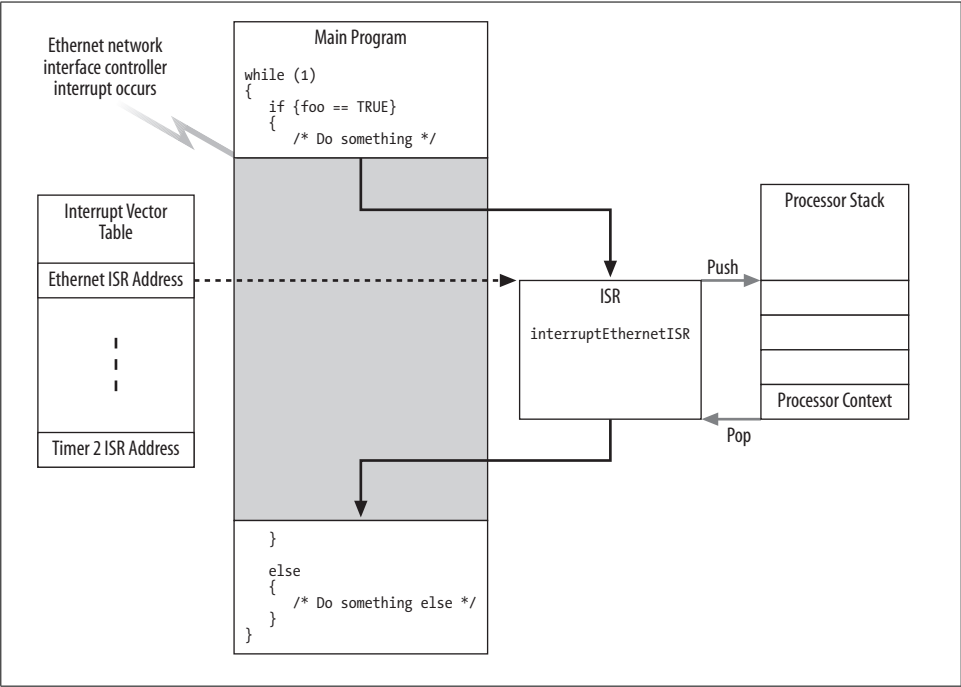
Figure 8-4. Software flow during interrupt

In Figure 8-4, the processor is executing the main program when an interrupt occurs from the Ethernet network interface controller. The processor finishes the instruction in progress before halting execution of the main program. (Some processors allow interruption of long instructions so that interrupts are not delayed for extended periods of time.)

The processor next looks up the address of the ISR for the Ethernet network interface controller, interruptEthernetISR, in the interrupt vector table, and the processor jumps to this function. The interruptEthernetISR function saves the processor context to the processor's stack.

The ISR then clears the interrupt. Once complete, the ISR restores the context and returns. The main program continues its execution from the point at which it was interrupted. Most processors have a special "return from interrupt" instruction for exiting the ISR.

One important concept associated with interrupts is *latency*. Interrupt latency is the amount of time from when an interrupt occurs to when the processor begins executing the associated interrupt service routine. Interrupt latency is a metric used by some engineers to evaluate processors and is very important in real-time systems. Disabling interrupts increases interrupt latency in an embedded system, because the latency includes the time between the occurrence of the interrupt and the moment interrupts are reenabled.

Although there is a single ISR for each interrupt, there might be a number of reasons the interrupt occurs. It is the responsibility of the ISR to determine the specific cause of the interrupt and proceed accordingly. The following code framework shows how an ISR reads the interrupt status register to determine the interrupt, acknowledges the interrupt by writing the value back to the interrupt status register, and then determines the cause of the interrupt. It is quite possible that more than one interrupt source is active during the ISR. Thus, the ISR must check every source; failure to do so may result in missing an interrupt.

```
interrupt void interruptServiceRoutine(void)
{
    uint8_t intStatus;

    /* Determine which interrupts have occurred. */
    intStatus = *pIntStatusReg;

    /* Acknowledge the interrupt. */
    *pIntStatusReg = intStatus;

    if (intStatus & INTERRUPT_SOURCE_1)
    {
        /* Do interrupt processing. */
    }

    if (intStatus & INTERRUPT_SOURCE_2)
    {
```

```
            /* Do interrupt processing. */
        }
    }
```

Shared Data and Race Conditions

A common issue when designing software that uses interrupts is how to share data between the ISR and the main program. A *race condition* is a situation where the outcome varies depending on the precise order in which the instructions of the main code and the ISR are executed; this should be strenuously avoided.

It can be extremely difficult to find race condition bugs because interrupts are asynchronous events and, to make matters worse, the race condition doesn't occur every time the code executes. Your software can run for days and pass all production tests without exhibiting this bug—but then, once the unit is shipped to the customer, it is certain to show up.

The following code example will give you a better understanding of race conditions. Imagine that the serial port ISR serialReceiveIsr is invoked when an incoming character arrives. As characters are received, gIndex is incremented to keep track of the number of characters stored in the memory buffer.

The main function also uses gIndex, by decrementing it when it processes the received characters in the memory buffer. Here is the example code:

```
int gIndex = 0;

interrupt void serialReceiveIsr(void)
{
    /* Store receive character in memory buffer. */

    gIndex++;
}

int main(void)
{
    while (1)
    {
        if (gIndex)
        {
            /* Process receive character in memory buffer. */

            gIndex--;
        }
    }
}
```

Can you spot the problem with this code?

Let's assume the variable gIndex has a value of 3 when the line of code that decrements this variable executes:

```
            gIndex--;
```

This line of code results in assembly-language instructions that do something like this:

```
LOAD gIndex into a register;
DECREMENT the register value;
STORE the register value back into gIndex;
```

The first step is to read the value of gIndex, 3, from its location in RAM into a processor register. Next, the register value is decremented, resulting in a value of 2.

Now suppose a serial port receive interrupt occurs before the new value of gIndex is stored in the memory. The processor stops executing main and executes the serial port ISR, serialReceiveIsr. The ISR increments gIndex to a value of 4.

The processor resumes execution of main after the ISR exits. At this point, main executes the line of code that stores the register value, 2, back into the variable gIndex. Now gIndex has a value of 2, as if the latest interrupt never occurred to increment gIndex.

This race condition can cause the program to lose incoming characters. Figure 8-5 shows the race condition for this example code.
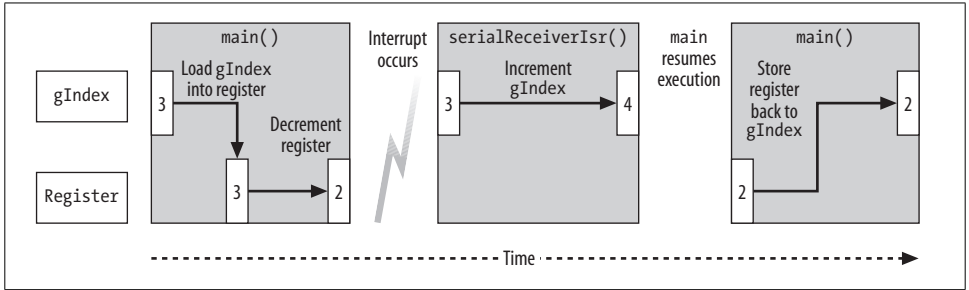
Figure 8-5. Example race condition

The decrement code in the main program is called a *critical section*. A critical section is a part of a program that must be executed in order and *atomically*, or without interruption. A line of C code (even as trivial as increment or decrement) is not necessarily atomic, as we've seen in this example.

So, how is this problem corrected? Because an interrupt can occur at any time, the only way to make such a guarantee is to disable interrupts during the critical section. In order to protect the critical section in the previous example code, interrupts are disabled before the critical section executes and then enabled after, as shown here:

```
int main(void)
{
    while (1)
    {
        interruptDisable();

        if (gIndex)
```

```
        {
            /* Process receive character in memory buffer. */

            gIndex--;
        }

        interruptEnable();
    }
}
```

In embedded systems, and especially real-time systems, it is important to keep interrupts enabled as much as possible to avoid hindering the responsiveness of the system. Try to minimize the number of critical sections and the length of critical section code.

> The safest solution is to save the state of the interrupt enable flag, disable interrupts, execute the critical section, and then restore the state of the interrupt enable flag. Enabling interrupts at the end without ensuring that they were enabled at the outset of the critical section is risky.

Race conditions can also occur when the resource shared between an ISR and the main program is a peripheral or one of the peripheral's registers. For example, suppose a main program and an ISR use the same peripheral register. The main program reads a register and stores the value. At this point, an ISR executes and modifies the value in that same register. When the main program resumes and updates the peripheral register, the ISR's value is overwritten and lost.

Critical sections are also an issue when using a real-time operating system (RTOS), because the tasks may then also share resources such as global variables or peripheral registers. We will look at this in Chapter 10.

The Improved Blinking LED Program

Now we will look at an interrupt example using a timer. For this example, we will use the Blinking LED code from Chapter 3. However, instead of using a loop to handle the timing of the LED blink, we will use a hardware timer. Most microcontrollers include up to several timers.

There are several advantages to using a timer rather than a loop for the timing: the processor is free to handle other tasks instead of sitting in a `while` loop doing nothing, a timer is more accurate for measuring a loop than a stopwatch, and you can calculate the exact time you want the timer interrupt to fire instead of using a trial-and-error approach based on the processor's clock.

In this improved Blinking LED program, the delay routine is eliminated and a timer device driver is used to handle the delay between LED toggles. The timer is used to interrupt the processor once a specific interval has elapsed.

How Timers Work

A *timer* is a peripheral that measures elapsed time, typically by counting down processor cycles or clocks. A *counter*, by contrast, measures elapsed time using external events. A timer is set up by programming an interval register in the timer peripheral, with a specific value calculated by the software engineer to determine the timer interrupt interval. The timer peripheral then uses a clock to keep count of the number of ticks that have elapsed since the timer has been started. The number of clock ticks is compared to the value in the timer interval register. Once they are equal, a timer interrupt is generated (if enabled).

The timer counts cycles either from the processor's main clock signal or from a separate clock fed into the timer peripheral from an external processor pin. In some processors, the clock used for the timer can be selected by programming the timer's configuration registers. Many processors today also include multiple internal clock sources that can be used to drive the timers.

On some processors, the calculated time interval is programmed into a timer register that is itself decremented at each clock tick. Once the value in that register hits zero, a timer interrupt is generated. The timer peripheral then reloads the timer register with the calculated interval value (stored in a separate register) and once again begins decrementing this value at each clock tick. Other processors, such as the PXA255, have timers that count up. Be sure to check your processor's documentation to understand how your timer functions.

The PXA255 processor has four timers. For this example, timer 0 is used; the 32-bit PXA255 timer registers for timer 0 are shown in Figure 8-6.
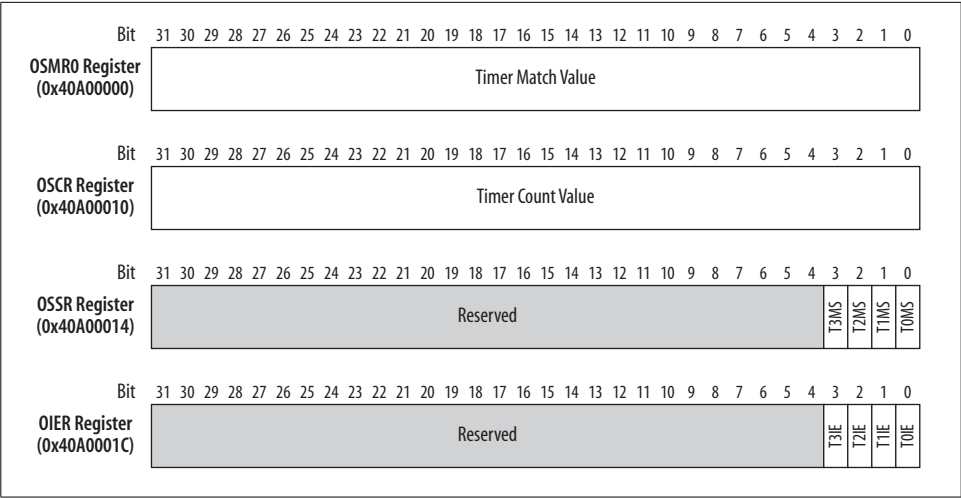
Figure 8-6. PXA255 processor timer 0 registers

On the PXA255, the timer count register (OSCR) contains a count that is incremented on rising edges of the timer clock, which operates at 3.6864 MHz. In other words, each time the clock signal goes from low to high, the OSCR is incremented by one.

The timer match register (OSMR*n*, where *n* is the timer number) contains the timer values for the four different timers. After every rising edge of the timer clock, the processor compares the value in the OSMR*n* to the OSCR. If there is a match, an interrupt is generated and the corresponding bit is set in the timer status register (OSSR). The timer interrupt enable register (OIER) determines which interrupts are enabled for the four different timers.

Watchdog Timers

Another type of timer frequently mentioned in reference to embedded systems is a *watchdog timer*. A watchdog timer is a special hardware fail-safe mechanism that intervenes if the software stops functioning properly. The watchdog timer is periodically reset (sometimes called "kicking the dog") by software. If the software crashes or hangs, the watchdog timer soon expires, causing the entire system to be reset automatically.

The inclusion and use of a watchdog timer is a common way to deal with unexpected software hangs or crashes that may occur after the system is deployed. For example, suppose your company's new product will travel into space. No matter how much testing you do before deployment, the possibility remains that there are undiscovered bugs lurking in the software and that one or more of these is capable of hanging the system altogether. If the software hangs, you won't be able to communicate with the system, so you can't issue a reset command remotely. Instead, you must build an automatic recovery mechanism into the system. And that's where the watchdog timer comes in.

One important implementation detail to remember when using a watchdog timer is that you should always implement the code that handles resetting the watchdog timer in the main processing loop. Never implement the watchdog timer reset in an ISR. The reason is that in an embedded system, the main processing loop can hang while the interrupts and ISRs continue to function. In this case, the watchdog timer would never be able to reset the system and thus allow the software to recover.

The `main` routine for the Blinking LED implementation that uses a timer instead of a delay loop is very similar to the `main` routine discussed in Chapter 3. This part of the code is hardware-independent. The `main` function starts with initialization of the LED control port with the `ledInit` function. An initialization routine for the timer device driver, `timerInit`, is called to initialize and start the timer hardware.

The infinite loop in `main` is empty in this case because there is no other processing needed for this version of the Blinking LED program. All of the processing happens

in the background with the timer interrupt, but the infinite loop is still needed in order to keep the program running. Notice here that the delay_ms function has been removed:

```c
#include "led.h"
#include "timer.h"

/************************************************************************
 *
 * Function:    main
 *
 * Description: Blink the green LED once a second.
 *
 * Notes:
 *
 * Returns:     This routine contains an infinite loop.
 *
 ************************************************************************/
int main(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    /* Configure and start the timer. */
    timerInit();

    while (1)
        ;

    return 0;
}
```

The timerInit routine initializes the registers needed for the timer device driver and then enables the timer interrupt. The global state variable bInitialized is used to ensure the timer registers are only configured once.

The first step to configure the timer is to clear any pending interrupts. This is done by writing bit 0 (defined by the bitmask TIMER_0_MATCH) to the timer status (OSSR) register (defined by the macro TIMER_STATUS_REG).

For the next step, we need to calculate the interrupt interval. The *PXA255 Processor Developer's Manual* states that the timers are incremented by a 3.6864 MHz clock. To toggle the LED every 500 ms, the following equation is used to determine the value for the timer match register:

Timer Match Register Value = Timer clock × Timer interval

For our example, the calculation is:

Timer Match Register Value = 3,686,400 Hz × 0.5 seconds
                           = 1,843,200
                           = 0x001C2000

The macro TIMER_INTERVAL_500MS is set to the interval value 0x001C2000. The *PXA255 Processor Developer's Manual* describes the algorithm for setting up a timer as follows:

1. Read the current count value in the timer count register (OSCR).
2. Add the interval offset to the current count value. This value corresponds to the amount of time before the next time-out.
3. Program the new interval value into the timer match register (OSMR0).

To set the timer interval, the OSMR0 register (defined by the macro TIMER_0_MATCH_REG) is programmed with the current value of the timer count plus the TIMER_INTERVAL_500MS timer interval.

Next, the timer interrupt is enabled in two places: the timer peripheral and the interrupt controller. In the timer peripheral, the timer interrupt enable is controlled by bit 0 (defined by the bitmask TIMER_0_INTEN) in the 32-bit OIER (defined by the macro TIMER_INT_ENABLE_REG).

Then the interrupt controller is configured to allow interrupts from the timer peripheral. As shown in Table 8-4, the timer 0 interrupt is mapped to interrupt number 26. Therefore, the program sets bit number 26 (defined by the macro TIMER_0_ENABLE) in the Interrupt Controller Mask Register (ICMR) (defined by the macro INTERRUPT_ENABLE_REG).
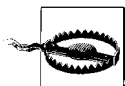
Because the timerInit function enables the timer 0 interrupt in the interrupt controller, an ISR for the timer 0 interrupt must be installed prior to calling this routine.

For the final step, initializing the timer, set the initialization state variable to TRUE, as shown here:

```
#define TIMER_INTERVAL_500MS        (0x001C2000)

/***********************************************************************
 *
 * Function:    timerInit
 *
 * Description: Initialize and start the timer.
 *
 * Notes:       This function is specific to the Arcom board.
 *              Ensure an ISR has been installed for the timer prior
 *              to calling this routine.
 *
 * Returns:     None.
 *
 ***********************************************************************/
void timerInit(void)
{
    static int bInitialized = FALSE;
```

```
        /* Initialize the timer only once. */
        if (bInitialized == FALSE)
        {
            /* Acknowledge any outstanding timer interrupts. */
            TIMER_STATUS_REG = TIMER_0_MATCH;

            /* Initialize the timer interval. */
            TIMER_0_MATCH_REG = (TIMER_COUNT_REG + TIMER_INTERVAL_500MS);

            /* Enable the timer interrupt in the timer peripheral. */
            TIMER_INT_ENABLE_REG |= TIMER_0_INTEN;

            /* Enable the timer interrupt in the interrupt controller. */
            INTERRUPT_ENABLE_REG |= TIMER_0_ENABLE;

            bInitialized = TRUE;
        }
    }
```

Prior to entering the ISR, the processor context is saved. Use the following function declaration so that the GNU compiler includes the code for the context save (and restore at the end of the ISR):

```
    void timerInterrupt() __attribute__ ((interrupt ("IRQ")));
```

Next, the ISR acknowledges the timer 0 interrupt. The processor documentation states that acknowledging a timer interrupt is accomplished by writing a 1 to the timer 0 bit (defined by the bitmask TIMER_0_MATCH) of the 32-bit OSSR (defined by the macro TIMER_STATUS). See Figure 8-6 for details of the OSSR and timer 0 match status bit 0 (T0MS).

Next, the LED state changes with a call to the function ledToggle. Then the timer match value is updated for the next timer interrupt interval. To update the timer 0 match register, first read the current timer count, add the timer interval, and then write this result into the timer match register (TIMER_0_MATCH_REG).

Finally, the processor context is restored and the ISR returns:

```
    #include "led.h"

    /**********************************************************************
     *
     * Function:    timerInterrupt
     *
     * Description: Timer 0 interrupt service routine.
     *
     * Notes:       This function is specific to the Arcom board.
     *
     * Returns:     None.
     *
     **********************************************************************/
    void timerInterrupt(void)
    {
```

```
        /* Acknowledge the timer 0 interrupt. */
        TIMER_STATUS = TIMER_0_MATCH;

        /* Change the state of the green LED. */
        ledToggle( );

        /* Set the new timer interval. */
        TIMER_0_MATCH_REG = (TIMER_COUNT_REG + TIMER_INTERVAL_500MS);
    }
```

You can now build this code and run it on the target Arcom board. If successfully built, this version of the Blinking LED program should operate the same way as the one shown in Chapter 3.

Summary of Interrupt Issues

Interrupts are an important part of most embedded systems. Here are some key points to keep in mind when dealing with interrupts:

Get the first interrupt

Focus on getting the specific interrupt you are working on to occur first. Then move on to getting that interrupt to fire subsequent times.

Interrupt blocked

Interrupts can be blocked at several points. Ensure that the specific interrupt is enabled both in the interrupt controller and at the source peripheral device. Make sure that global interrupts are enabled in the processor.

ISR installation

Verify that the ISR is installed in the interrupt vector table properly and for the correct interrupt vector. Understand the mapping of interrupts for the processor. Using the LED debug technique mentioned in Chapter 5 can be a valuable tool for tracing the execution path when an interrupt occurs.

Protect against unhandled interrupts

Ensure that there is an ISR for every interrupt in the system. It is best to install a default ISR for all interrupts at initialization time to ensure every interrupt is handled.

Processor context

Make sure the processor context is saved and restored properly in the ISR. Registers can be trampled by an ISR that will eventually wreak havoc on your main program.

Acknowledge the interrupt

The interrupt must be acknowledged so that the signal is deasserted. If this is done incorrectly or not done at all, ISRs for the same or lower-priority interrupts won't run again. Or, if it is a level-sensitive interrupt, the ISR will run repeatedly because the interrupt signal will remain asserted.

Time Sharing

In some embedded systems, you will have more tasks that need to occur at a specific interval than there are timers to use for each task. Or the processor you must use will only have a single timer. Don't worry, there are ways around this predicament—you can share a timer among several tasks.

For example, imagine that you have the following tasks that must occur in your embedded system:

1. Read a temperature sensor every 5 ms.
2. Write a character out a serial port every 12 ms.
3. Toggle an I/O pin every 100 ms.

Furthermore, in this example, the processor has only a single timer to use. What are you to do?

The timer interval is set to the greatest common factor (in this case, 1 ms) of the desired times. Next, the ISR counts the number of timer intervals that have elapsed. Once the appropriate number of intervals has occurred for the specific task, a flag is set for that job to be performed.

There are several ways to implement this type of timer sharing. One way is to have a separate static counter variable for each interval of which you need to keep track. The following is a code snippet that would be used in the timer ISR to handle the three different intervals for the jobs listed:

```
timer1Count++;
timer2Count++;
timer3Count++;

/* Set the flag to read the temperature sensor every 5 ms
   and then reset the counter. */
if (timer1Count >= 5)
{
    gbReadTemperatureSensor = TRUE;
    timer1Count = 0;
}

/* Set the flag to write the character out the serial port
   every 12 ms and then reset the counter. */
if (timer2Count >= 12)
{
    gbWriteSerialCharacter = TRUE;
    timer2Count = 0;
}
```

```
    /* Set the flag to toggle the I/O pin every 100 ms and
       then reset the counter. */
    if (timer3Count >= 100)
    {
        gbTogglePin = TRUE;
        timer3Count = 0;
    }
```
The main program would then regularly check whether any global flag is set, perform the necessary action, and reset the flag. As discussed earlier in this chapter, care must be taken to avoid a race condition since the global flags are shared between the ISR and the main program.

Avoid race conditions

A lot of forethought must go into designing your software if the embedded system you are working on uses interrupts. The communication mechanisms between the main program and the interrupt service routines must be carefully thought out. Race conditions are dangerous errors that can be extremely difficult to find.

Enabling and disabling

Keep disabling of interrupts to a minimum to reduce interrupt latency; this is very important in real-time systems. Be careful in the implementation of the disable and enable code around critical sections. Always use a variable to keep track of the enable state of interrupts so that interrupts are properly restored, and to avoid potential interrupt problems.

Putting It All Together

La commedia è finita! (The comedy is finished!)
—Ruggero Leoncavallo
   Pagliacci

In this chapter, we will bring all of the elements discussed so far together into a complete embedded application. Working hard to understand each piece of a system and application is necessary, but certain aspects may remain confusing until all of the pieces are combined. You should leave this chapter with a complete understanding of the example program and the ability to develop useful embedded applications of your own.

Application Overview

The application we're going to discuss is comprised of the components we have developed thus far, along with some additional functionality. It is a testament to the complexity of embedded software development that this example comes toward the end of this book rather than at its beginning. We've had to gradually build our way up to the computing platform that most books, and even high-level language compilers, take for granted.

The Monitor and Control application provides a means for you to exercise different aspects of an embedded system (hardware and software) by adding to the basic command set we provide in this book. You will quickly see, as your project progresses from design to production, how valuable this tool is to everyone working on the project.

Once you're able to write the Monitor and Control program, your embedded platform starts to look a lot like any other programming environment. The hardest parts of the embedded software development process—familiarizing yourself with the hardware, establishing a software development process for it, and interfacing to the individual hardware devices—are behind you. You are finally able to focus your efforts on the algorithms and user interfaces that are specific to the product you're

developing. In many cases, these higher-level aspects of the program can be developed on another computer platform—in parallel with the lower-level embedded software development we've been discussing—and merely ported to the embedded system once both are complete. Once the application level code is debugged and robust, you can port that code to future projects.

Figure 9-1 contains a high-level representation of the Monitor and Control application. This application includes three device drivers and a module for the command-line interface. An infinite loop is used for the main processing in the system. An RTOS can be incorporated into this application should you decide to use one.
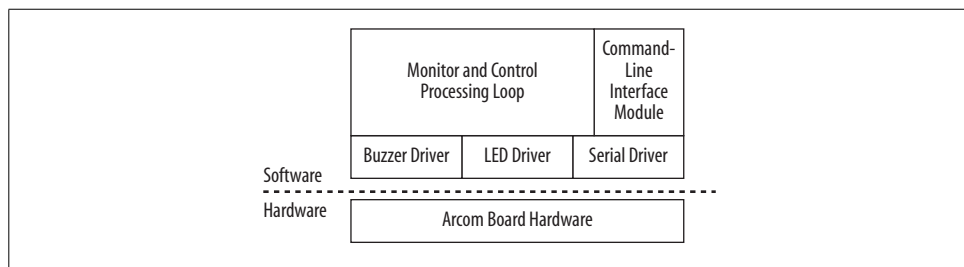
Figure 9-1. The Monitor and Control application

In addition to the Monitor and Control program's processing loop and the CLI module, three device drivers are shown in the figure. These control one of the Arcom board's LEDs, a buzzer, and a serial port. This layered design approach allows the drivers to be changed when the program is ported to a new hardware platform, with the application software remaining unchanged.

The main function that follows contains the primary processing loop for the Monitor and Control program, which includes functionality that we have explored in previous chapters (such as sending characters to and receiving characters from a serial port). Additional functionality includes a driver for the Arcom board's buzzer, the ability to assemble incoming characters into a command, and the ability to process commands. See the online example code for details about the buzzer driver. Because the Monitor and Control program accepts user input, a prompt (>) is output when the program is waiting for the user to enter a command.

```
#include "serial.h"
#include "buzzer.h"
#include "led.h"
#include "cli.h"

/**********************************************************************
 *
 * Function:    main
 *
 * Description: Monitor and control command-line interface program.
 *
```

```
 * Notes:
 *
 * Returns:      This routine contains an infinite loop.
 *
 **********************************************************************/
int main(void)
{
    char rcvChar;
    int  bCommandReady = FALSE;

    /* Configure the green LED control pin. */
    ledInit( );

    /* Configure the buzzer control pin. */
    buzzerInit( );

    /* Configure the serial port. */
    serialInit( );

    serialPutStr("Monitor and Control Program\r\n");
    serialPutChar('>');

    while (1)
    {
        /* Wait for a character. */
        rcvChar = serialGetChar( );

        /* Echo the character back to the serial port. */
        serialPutChar(rcvChar);

        /* Build a new command. */
        bCommandReady = cliBuildCommand(rcvChar);

        /* Call the CLI command processing routine to verify the command entered
         * and call the command function; then output a new prompt. */
        if (bCommandReady == TRUE)
        {
            bCommandReady = FALSE;
            cliProcessCommand( );

            serialPutChar('>');
        }
    }

    return 0;
}
```

Working with Serial Ports

We looked at a serial driver in Chapter 7. The Monitor and Control program uses the same serial driver with some additional functionality.

One change to the serial driver is the addition of a "put string" function, serialPutStr. This function, which follows, allows strings to be sent out the serial port through a single call rather than having to specify each character in order. This is similar to the standard C printf function, which calls the serialPutChar function repeatedly until the entire string has been transmitted out the serial port.

```
/***********************************************************************
 *
 * Function:    serialPutStr
 *
 * Description: Outputs a string to the serial port.
 *
 * Notes:
 *
 * Returns:     None.
 *
 ***********************************************************************/
void serialPutStr(char const *pOutputStr)
{
    char const *pChar;

    /* Send each character of the string. */
    for (pChar = pOutputStr; *pChar != '\0'; pChar++)
        serialPutChar(*pChar);
}
```

Command-Line Interface Processing

The command-line interface module is responsible for building the incoming command, parsing a completed command, and executing the function associated with the command. The command-line interface module contains two functions to handle these tasks: cliBuildCommand and cliProcessCommand.

The command_t struct has two members: the command name (defined by the pointer name), and the function to execute when the command is entered (defined by the pointer to a function function).

The command table, gCommandTable, is a command_t type array. The last command name and function in the table are set to NULL in order to aid the command lookup procedure. Additional commands can be added by following the format shown, but new commands must be added before the terminating last command.

You may notice that a macro called MAX_COMMAND_LEN is defined. Command names must be less than or equal to the maximum command length. The following code shows the command struct type and the command table:

```
#define MAX_COMMAND_LEN         (10)
#define COMMAND_TABLE_SIZE      (4)

typedef struct
```

```
    {
        char const    *name;
        void          (*function)(void);
    } command_t;

    command_t const gCommandTable[COMMAND_TABLE_LEN] =
    {
        {"HELP",    commandsHelp,},
        {"LED",     commandsLed, },
        {"BUZZER",  commandsBuzzer, },
        {NULL,      NULL }
    };
```

As characters are received from the serial port, the main processing loop calls the cliBuildCommand function, as we saw earlier in this chapter. Once a completed command is received, indicated by a carriage return (\r) character, the buffer index is reset and the function returns TRUE. Otherwise, the buffer index is incremented and FALSE is returned.

cliBuildCommand stores the incoming characters in a buffer called gCommandBuffer. The gCommandBuffer has enough room for a single command of MAX_COMMAND_LEN size plus one additional byte for the string-terminating character. Certain characters are not stored in the command buffer, including line feeds (\n), tabs (\t), and spaces.

Notice that the incoming characters are converted to uppercase (with the macro TO_UPPER) prior to insertion into the buffer. This makes the command-line interface a bit more user-friendly; the user does not have to remember to use capitalization to enter valid commands.

The local static variable idx keeps track of the characters inserted into the command buffer. As new characters are stored in the buffer, the index is incremented. If too many characters are received for a command, the index is set to zero and TRUE is returned to start processing the command.

```
    #define TO_UPPER(x)        (((x >= 'a') && (x <= 'z')) ? ((x) - ('a' - 'A')) : (x))

    static char gCommandBuffer[MAX_COMMAND_LEN + 1];

    /***********************************************************************
     *
     * Function:    cliBuildCommand
     *
     * Description: Put received characters into the command buffer. Once
     *              a complete command is received return TRUE.
     *
     * Notes:
     *
     * Returns:     TRUE if a command is complete, otherwise FALSE.
     *
     ***********************************************************************/
    int cliBuildCommand(char nextChar)
    {
```

```c
        static uint8_t idx = 0;

        /* Don't store any new line characters or spaces. */
        if ((nextChar == '\n') || (nextChar == ' ') || (nextChar == '\t'))
            return FALSE;

        /* The completed command has been received. Replace the final carriage
         * return character with a NULL character to help with processing the
         * command. */
        if (nextChar == '\r')
        {
            gCommandBuffer[idx] = '\0';
            idx = 0;
            return TRUE;
        }

        /* Convert the incoming character to uppercase. This matches the case
         * of commands in the command table. Then store the received character
         * in the command buffer. */
        gCommandBuffer[idx] = TO_UPPER(nextChar);
        idx++;

        /* If the command is too long, reset the index and process
         * the current command buffer. */
        if (idx > MAX_COMMAND_LEN)
        {
            idx = 0;
            return TRUE;
        }

        return FALSE;
    }
```

Once a completed command is assembled, the cliProcessCommand function, which follows, is called from the main processing loop. This function loops through the command table searching for a matching command name.

The variable idx is initialized to zero to start searching at the beginning of the command table and then keeps track of the command currently being checked. The function strcmp is used to compare the user command with the commands in the table. If the command is found, the flag bCommandFound is set to TRUE. This causes the search loop to exit and the associated function to execute. If the command is not found, an error message is sent out the serial port.

```c
    #include <string.h>

    /*********************************************************************
     *
     * Function:    cliProcessCommand
     *
     * Description: Look up the command in the command table. If the
     *              command is found, call the command's function. If the
     *              command is not found, output an error message.
```

```
 *
 * Notes:
 *
 * Returns:     None.
 *
 **********************************************************************/
void cliProcessCommand(void)
{
    int bCommandFound = FALSE;
    int idx;

    /* Search for the command in the command table until it is found or
     * the end of the table is reached. If the command is found, break
     * out of the loop. */
    for (idx = 0; gCommandTable[idx].name != NULL; idx++)
    {
        if (strcmp(gCommandTable[idx].name, gCommandBuffer) == 0)
        {
            bCommandFound = TRUE;
            break;
        }
    }

    /* If the command was found, call the command function. Otherwise,
     * output an error message. */
    if (bCommandFound == TRUE)
    {
        serialPutStr("\r\n");
        (*gCommandTable[idx].function)();
    }
    else
        serialPutStr("\r\nCommand not found.\r\n");
}
```

All functions in the command table are contained in one file; this keeps the entry point for all commands in a single location. The commandsLed function toggles the green LED, as shown in the following code. This function uses the same ledToggle function covered in Chapter 3.

```
#include "led.h"

/**********************************************************************
 *
 * Function:    commandsLed
 *
 * Description: Toggle the green LED command function.
 *
 * Notes:
 *
 * Returns:     None.
 *
 **********************************************************************/
void commandsLed(void)
{
```

```c
        ledToggle();
    }
```

The commandsBuzzer function toggles the buzzer on the Arcom board add-on module by calling the function buzzerToggle, as shown here:

```c
    #include "buzzer.h"

    /*********************************************************************
     *
     * Function:    commandsBuzzer
     *
     * Description: Toggle the buzzer command function.
     *
     * Notes:
     *
     * Returns:     None.
     *
     *********************************************************************/
    void commandsBuzzer(void)
    {
        buzzerToggle();
    }
```

The help command function, commandsHelp, loops through the gCommandTable and sends the command name out the serial port. This gives the user a listing of all commands supported by the command-line interface.

```c
    #include "cli.h"
    #include "serial.h"

    /*********************************************************************
     *
     * Function:    commandsHelp
     *
     * Description: Help command function.
     *
     * Notes:
     *
     * Returns:     None.
     *
     *********************************************************************/
    void commandsHelp(void)
    {
        int idx;

        /* Loop through each command in the table and send out the command
         * name to the serial port. */
        for (idx = 0; gCommandTable[idx].name != NULL; idx++)
        {
            serialPutStr(gCommandTable[idx].name);
            serialPutStr("\r\n");
        }
    }
```

The Monitor and Control program gives you a baseline of functionality for the development of a useful command-line interface. The functionality of the CLI can be extended by adding new commands or enabling users to input parameters for commands. To accommodate input parameters, the `command_t` structure can be expanded to contain the maximum and minimum values. When parsing the command, you will need to parse the input parameters and validate the parameter ranges with those contained in the command table.

Operating Systems

o·s·o·pho·bi·a n. A common fear among embedded
systems programmers.

Many embedded systems today incorporate an operating system. This can range from a small kernel to a full-featured operating system—typically called a *real-time operating system* or *RTOS* (pronounced "are-toss"). Either way, you'll need to know what features are the most important and how they are used with the rest of your software. At the very least, you need to understand what a real-time operating system looks like on the outside. In this chapter, we take a detailed look at the mechanisms found in most operating systems and how to use them.

The information in this chapter is very general and does not include specific code examples. That's because the features and APIs that implement the activities in this chapter are different on each operating system. Subsequent chapters show you what to do on Linux and eCos, two popular operating systems used in embedded environments.

History and Purpose

In the early days of computing, there was no such thing as an operating system. Application programmers were completely responsible for controlling and monitoring the state of the processor and other hardware. In fact, the purpose of the first operating systems was to provide a virtual hardware platform that made application programs easier to write. To accomplish this goal, operating system developers needed only provide a loose collection of routines—much like a modern software library—for resetting the hardware to a known state, reading the state of the inputs, and changing the state of the outputs.

Modern operating systems add to this the ability to execute multiple software *tasks* simultaneously on a single processor—a feature called *multitasking*. Each such task (also commonly called a *thread*) is a piece of the software that can be separated from the rest of the operating system and run independently. A set of embedded software requirements can usually be broken down into a small number of such independent

pieces. For example, the printer server device described in Chapter 2 contains two main software tasks:

- Receiving data from the computers attached to the Ethernet port.
- Formatting and sending the data to the printer attached to the parallel port.

Tasks provide a key software abstraction that makes the design and implementation of embedded software easier, and the resulting source code simpler to understand and maintain. By breaking the larger program into smaller pieces, the programmer can more easily concentrate her energy and talents on the unique features of the system under development.

Strictly speaking, an operating system is not a required component of any computer system—embedded or otherwise. It is always possible to perform the same functions from within the application program itself. Indeed, all of the examples so far in this book have done just that. There is one path of execution—starting at `main`—that runs on the system. This is the equivalent of having only one task. But as the complexity of the application expands beyond the simple task of blinking an LED, the benefits of an operating system far outweigh the associated costs.

One common part of all operating systems is the *kernel*. In most operating systems, the kernel consists of the scheduler, the routine to handle switching between the different tasks running in the system, and the mechanisms of communication between tasks.

The Scheduler

We have already talked about multitasking and the idea that an operating system makes it possible to execute multiple "programs" at the same time. But what does that mean? How is it possible to execute several tasks concurrently? In actuality, the tasks are not executed at the same time. Rather, they are executed in pseudoparallel. They merely take turns using the processor. This is similar to the way several people might read the same copy of a book. Only one person can actually use the book at a given moment, but each person can read it if everyone takes turns.

An operating system is responsible for deciding which task gets to use the processor at a particular moment. The piece of the operating system that decides which of the tasks has the right to use the processor at a given time is called the *scheduler*. The scheduler is the heart and soul of any operating system. Some of the more common scheduling algorithms are:

First-in-first-out (FIFO)
> This scheduling (also called *cooperative multitasking*) allows each task to run until it is finished, and only after that is the next task started.

Shortest job first
> This algorithm allows each task to run until it completes or suspends itself; the next task selected is the one that will require the least amount of processor time to complete.

Priority

This algorithm is typically used in real-time operating systems. Each task is assigned a priority that is used to determine when the task executes once it is ready. Priority scheduling can be either preemptive or nonpreemptive. *Preemptive* means that any running task can be interrupted by the operating system if a task of higher priority becomes ready.

Round robin

In this scheduling algorithm, each task runs for some predetermined amount of time called a *time slice*. After that time interval has elapsed, the running task is preempted by the operating system, and the next task in line gets its chance to run. The preempted task doesn't get to run again until each of the other tasks in that round has had its chance.

Real-Time Scheduling

The scheduler in some operating systems is real-time. While the schedulers in the previous section take only priorities and time slices into account, a real-time scheduler takes into account that some tasks have deadlines. Some tasks in a real-time system may not have deadlines or might have low penalties for missing deadlines. Tasks such as these can use background processing time to do their work.

In this type of operating system, the real-time scheduler should know the deadlines of all the tasks. The scheduler should base decisions on a comparison of the deadlines of tasks that are in the ready queue.

There are several real-time schedulers, including:

Real-time executive

Each task is assigned a unique timeslot in a periodically repeating pattern. A real-time executive is more static than an operating system and assumes that the programmer knows how long each task takes. If the executive knows the task can complete its work in the allotted time, its deadlines can all be met. This won't work if you need to create and exit tasks on the fly or run tasks at irregular intervals.

Earliest deadline first

The operating system tracks the time of the next deadline for all tasks. At each scheduling point, the scheduler selects the task with the closest deadline. This is a priority-based scheduler, but with the addition that it calculates the deadlines of real-time tasks at each timer tick and adjusts priorities as appropriate. If new priorities are assigned, this might mean new tasks are run and, therefore, that a context switch (which we discuss later in this chapter in the section "Context switch") must take place. One disadvantage to this scheduler is that it has a large computational overhead.

Minimal laxity first

> The operating system tracks deadlines and the time to complete the remaining work for each task. The scheduler selects the task with least *laxity*, where laxity is the available time minus the remaining work time, and the available time is the deadline time minus the current time. The theory behind this scheduling algorithm is that the task with the least "time to spare" can least afford to yield the processor. The exact order of which tasks run may differ from the earliest-deadline-first scheduler.

Resource reservation

> When a task is created, it states its requirements in terms of deadlines, processor usage, and other relevant resources. The operating system should admit the new task only if the system can guarantee those resources without making any other (already admitted) tasks fail.

The scheduling algorithm you choose depends on your goals. Different algorithms yield different results. Let's suppose you're given 10 jobs, and each will take a day to finish. In 10 days, you will have all of them done. But what if one or more has a deadline? If the ninth task given to you has a deadline in three days, doing the tasks in the order you receive them will cause you to miss that deadline.

The purpose of a real-time scheduling algorithm is to ensure that critical timing constraints, such as deadlines and response time, are met. When necessary, decisions are made that favor the most critical timing constraints, even at the cost of violating others.

To demonstrate the importance of a scheduling algorithm, consider a system with only two tasks, which we'll call Task 1 and Task 2. Assume these are both periodic tasks with periods T1 and T2, and for each task, the deadline is the beginning of its next cycle. Task 1 has T1 = 50 ms and a worst-case execution time of C1 = 25 ms. Task 2 has T2 = 100 ms and C2 = 40 ms.

Does the processor have enough time to handle both tasks? We can start answering this question by looking at how much of the processor time each task needs in its worst case—its *utilization*. The utilization of task *i* is:

$$U_i = C_i/T_i$$

Thus, if U1 = 50 percent and U2 = 40 percent, the total requested utilization is:

$$U = U1 + U2 = 90 \text{ percent}$$

It seems logical that if utilization is less than 100 percent, there should be enough available CPU time to execute both tasks.

Let's consider a *static priority* scheduling algorithm, where task priorities are unique and cannot change at runtime. With two tasks, there are only two possibilities:

- Case 1: Priority(T1) > Priority(T2)
- Case 2: Priority(T1) < Priority(T2)

The two cases are shown in Figure 10-1. In Case 1, both tasks meet their respective deadlines. In Case 2, however, Task 1 misses a deadline, despite 10 percent idle time, because Task 2's higher priority required it to be scheduled first. This illustrates the importance of priority assignment.
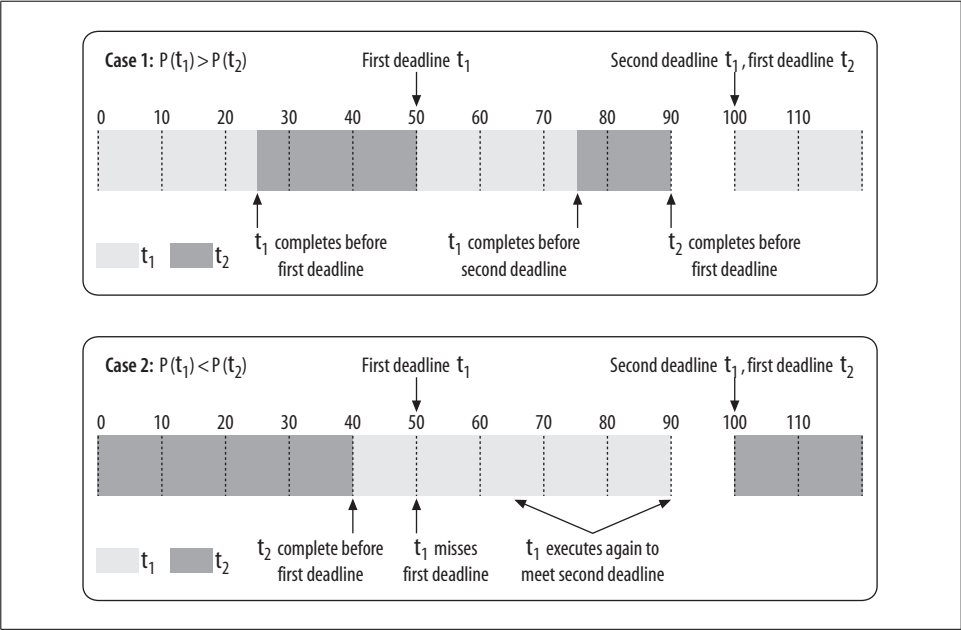
Figure 10-1. Scheduling outcome examples

Real-time systems sometimes require a way to share the processor that allows the most important tasks to grab control of the processor as soon as they need it. Therefore, most real-time operating systems utilize a priority-based scheduling algorithm that supports preemption. This is a fancy way of saying that at any given moment, the task that is currently using the processor is guaranteed to be the highest-priority task that is ready to do so. Lower-priority tasks must wait until higher-priority tasks are finished using the processor before resuming their work. The scheduler detects such conditions through interrupts or other events caused by the software; these events are called scheduling points.

Figure 10-2 shows a scenario of two tasks running at different priority levels.

There are two tasks in Figure 10-2: Task A and Task B. In this scenario, Task B has higher priority than Task A. At the start, Task A is running. When Task B is ready to run, the scheduler preempts Task A and allows Task B to run. Task B runs until it has completed its work. At this point, control of the processor returns to Task A, which continues its work from where it left off.
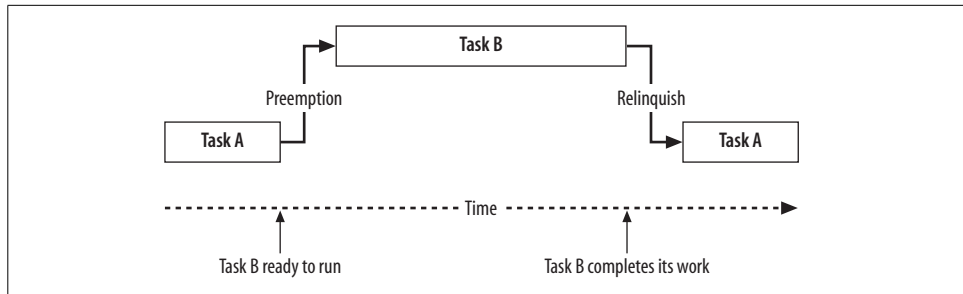
Figure 10-2. Priority scheduling of two tasks

When a priority-based scheduling algorithm is used, it is also necessary to have a backup scheduling policy for tasks with the same priority. A common scheduling algorithm for tasks with the same priority is round robin. In Figure 10-3, we show a scenario in which three tasks are running in order to demonstrate round robin scheduling with a time slice. In this example, Tasks B and C are at the same priority, which is higher than that of Task A.
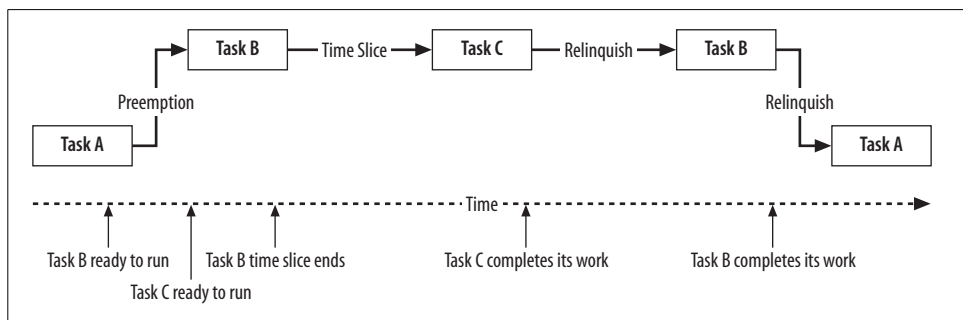
Figure 10-3. Priority scheduling of three tasks

In the scenario shown in Figure 10-3, Task A is running when Task B becomes ready to run. Task A is preempted so that Task B can run. While Task B is running, Task C also becomes ready to run. The scheduler therefore allows Task B to run for a time slice period. After this period expires, the scheduler gives Task C an opportunity to run. Then Task C completes its work. Because Task B still has work to complete and has the highest priority of all ready tasks, the scheduler allows Task B to run. Once Task B completes its work, Task A can finish.

Scheduling Points

You might be asking yourself how the scheduler—which is also a piece of software—gets an opportunity to execute and do its job. That is where *scheduling points* enter in. Simply stated, scheduling points are the set of operating system events that result in a run of the scheduler code. Following is a list of scheduling points:

Task creation

When a task is created, the scheduler is called to select the next task to be run. If the currently executing task still has the highest priority of all the ready tasks, it will be allowed to continue using the processor. Otherwise, the highest-priority ready task will be executed next.

Task deletion

As in task creation, the scheduler is called during task deletion. However, in the case of task deletion, if the currently running task is being deleted, a new task is always selected by virtue of the fact that the old one no longer exists.

Clock tick

The clock tick is the heartbeat of the system. It is a periodic event that is triggered by a timer interrupt (similar to the one we have already discussed). The clock tick provides an opportunity to awaken tasks that are waiting for a certain length of time to pass before taking their next action.

Task block

When the running task makes a system call that blocks, that task is no longer able to use the processor. Thus, the scheduler is invoked to select a new task to run.

Task unblock

A task might be waiting for some event to take place, such as for data to be received. Once the event occurs, the blocked task becomes ready to execute and thus is immediately eligible to be considered for execution.

Locking and Unlocking

Real-time operating systems can allow a task to lock the scheduler. Locking the scheduler prohibits it from executing and, in turn, keeps other tasks from running (since they cannot be scheduled). The task that locks the scheduler maintains control of the processor whether higher-priority tasks are ready to run or not. This allows the task with the scheduler lock to perform operations without worrying about thread-safe issues with other tasks. Interrupts still function and ISRs still run.
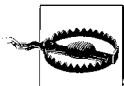
> Care must be taken when locking the scheduler, because it can hinder the responsiveness of the system. In general, locking the scheduler should be avoided whenever possible.

Every lock of the scheduler must have an unlock counterpart—otherwise, the system will stop running. The following is an example of locking and unlocking the scheduler:

```
/* Perform task operations. */

os_scheduler_lock();
```

```
/* Perform work while scheduler cannot run another task. */

os_scheduler_unlock();

/* Perform other task operations. */
```

The operating system keeps track of the scheduler lock state with a variable. This variable is incremented when calls are made to lock the scheduler and decremented when unlock calls are made. The scheduler knows it can run when the lock state variable is set to 0.

Tasks

Different types of tasks can run in an operating system. For example, a task can be periodic, where it exits after its work is complete. The task can then be restarted when there is more work to be done. Typically, however, a task runs forever, similar to the infinite loop discussed in Chapter 3. Each task also has its own stack that is typically allocated statically.

Task States

Remember how we said that only one task could actually use the processor at a given time? That task is said to be the *running* task, and no other task can be in that same state at the same time. Tasks that are ready to run—but are not currently using the processor—are in the *ready* state, and tasks that are waiting for some event external to themselves to occur before going on are in the *waiting* state. Figure 10-4 shows the relationships between these three states.
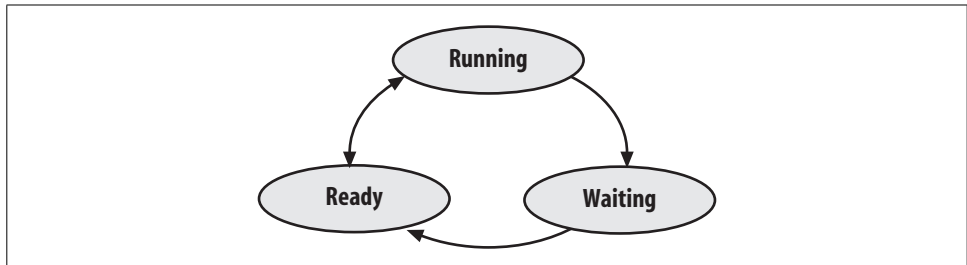
Figure 10-4. Possible states of a task

A transition between the ready and running states occurs whenever the operating system selects a new task to run during a scheduling point. The task that was previously running leaves the running state, and the new task (selected from the queue of tasks in the ready state) is promoted to running. Once it is running, a task will leave that state only if it terminates, if a higher-priority task becomes ready, or if it needs

to wait for some event external to itself to occur before continuing. In the latter case, the task is said to *block*, or wait, until that event occurs. A task can block by waiting for another task or for an I/O device, or it can block by sleeping (waiting for a specific time period to elapse).

When the task blocks, it enters the waiting state, and the operating system selects one of the ready tasks to be run. So, although there may be any number of tasks in each of the ready and waiting states, there will always be exactly one task in the running state at any time.

It is important to note that only the scheduler can promote a task to the running state. Newly created tasks and tasks that are finished waiting for their external event are placed into the ready state first. The scheduler will then include these new ready tasks in its future decision-making.

In order to keep track of the tasks, the operating system typically implements queues for each of the waiting and ready states. The ready queue is often sorted by priority so that the highest-priority task is at the head of the queue. The scheduler can then easily pick the highest-priority task to run next.

Context switch

The actual process of changing from one task to another is called a *context switch*. Task contexts are processor-specific and sometimes compiler-specific, as is the code that implements the context switch. That means it must always be written in assembly language and is very hardware-specific. Operating systems, especially for embedded systems, emphasize the goal of minimizing the time needed to switch task contexts. Figure 10-5 shows an example of a context switch operation between two tasks: A (at priority 150) and B (at a higher priority, 200).
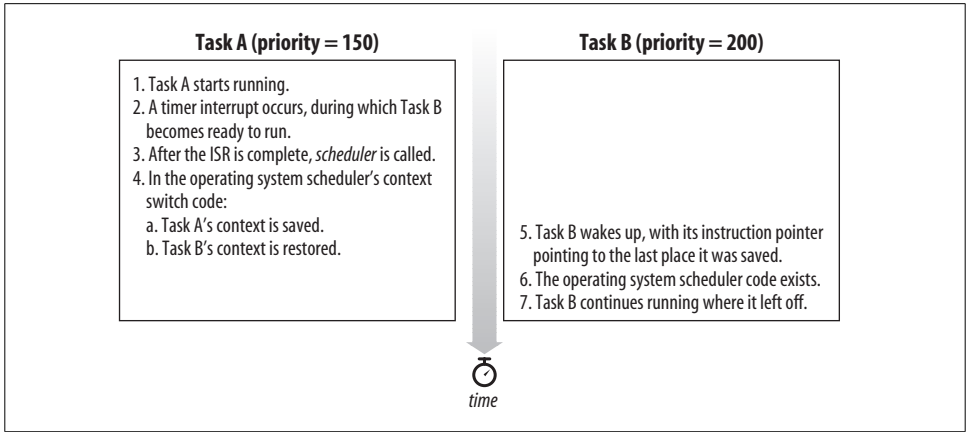
Figure 10-5. A context switch

The idle task

If there are no tasks in the ready state when the scheduler is called, the *idle task* is executed. The idle task looks the same in every operating system. It is simply an infinite loop that does nothing and never blocks. The idle task is completely hidden from the application developer. Sometimes, however, the operating system does assign it a valid task ID and priority. The idle task is always considered to be in the ready state (when it is not running), and because of its low priority, it may be found at the tail of the ready list. Other tasks are sometimes referred to as *user tasks* to distinguish them from the idle task.

Task Context

The scheduler maintains information about the state of each task. This information is called the *task context* and serves a purpose similar to a bookmark. In the earlier analogy of multiple readers, each reader of the book is presumed to have his own bookmark. The bookmark's owner must be able to recognize it (e.g., it has his name written on it), and it must indicate where he stopped reading when he last gave up control of the book. This is the reader's context.

A task's context records the state of the processor just prior to the point at which another task takes control of it. This usually consists of a pointer to the next instruction to be executed (the instruction pointer), the address of the current top of the stack (the stack pointer), and the contents of the processor's flag and general-purpose registers.

To keep tasks and their contexts organized, the operating system maintains some information about each task. Operating systems written in C often keep this information in a data structure called the *task control block*. The task control block contains a pointer to the task's context, the current state of the task, the task priority, the task entry-point function, and any task-specific data (such as parameters and task identification).

Task Priorities

Setting the priorities of the tasks in a system is important. Care needs to be taken so that lower-priority tasks get to do their work, just as the higher-priority tasks do. Otherwise, starvation of a task can occur, where a low-priority task is kept from doing any work at all.

There are several reasons that starvation may occur in a system, including the following:

- Processor overload occurs when high-priority tasks monopolize the processor and are always running or ready to run.
- Low-priority tasks are always at the end of a priority-based event queue and, therefore, may be permanently blocked from executing.
- A task may be prevented from running by a bug in another task; for example, one task fails to signal when it is supposed to.

There are solutions to these problems, such as:

- Using a faster processor.
- Using a FIFO queue of tasks rather than priority-based scheduling. This may not be possible, depending on the implementation of the operating system. In addition, it might be appropriate in some systems to starve low-priority tasks, but this must be a conscious decision.
- Fixing all bugs (this is sometimes easier said than done).

Rate monotonic scheduling

The *rate monotonic algorithm* (*RMA*) is a procedure to determine the optimal priority of each periodic task in a system. This procedure assigns fixed priorities to tasks to maximize their "schedulability." A task set is considered schedulable if all tasks meet all deadlines all the time. The algorithm is straightforward:

> Assign the priority of each task according to its period, so that the shorter the period the higher the priority.

The reasoning behind this algorithm is that the task with the shortest period has the least time to do its work once it becomes ready again. In the next example, the period of Task 1 is shorter than the period of Task 2. Following RMA's rule, Task 1 is assigned the higher priority. This corresponds to Case 1 in Figure 10-1, which is the priority assignment that succeeded in meeting both deadlines.

RMA is an example of a static priority algorithm. The alternative to a static priority algorithm is the much more complicated class of dynamic schedulers, which appear on sophisticated, commercial-grade operating systems; we don't recommend that you try to design a dynamic scheduler of your own. RMA is the optimal static priority algorithm. If a particular set of tasks cannot be scheduled using the RMA, that task set cannot be scheduled using any static priority algorithm.

One major limitation of fixed-priority scheduling is that it is not always possible to fully utilize the processor. Even though RMA is the optimal fixed-priority scheme, it has a worst-case schedulable bound of the following:

$$W_n = n \, (2^{(1/n)} - 1)$$

where n is the number of tasks in a system. As you would expect, the worst-case schedulable bound for one task is 100 percent. But as the number of tasks increases, the schedulable bound decreases, eventually approaching its limit of about 69.3 percent (ln 2, to be precise).

It is theoretically possible for a set of tasks to require only 70 percent CPU utilization in sum and still not meet all its deadlines. For example, consider the case shown in Figure 10-6. The only change from the example shown in Figure 10-1 is that both the period and execution time of Task 2 have been lowered. Based on RMA, Task 1

is assigned higher priority. Despite only 90 percent utilization, Task 2 misses its first deadline. Reversing priorities would not have improved the situation.
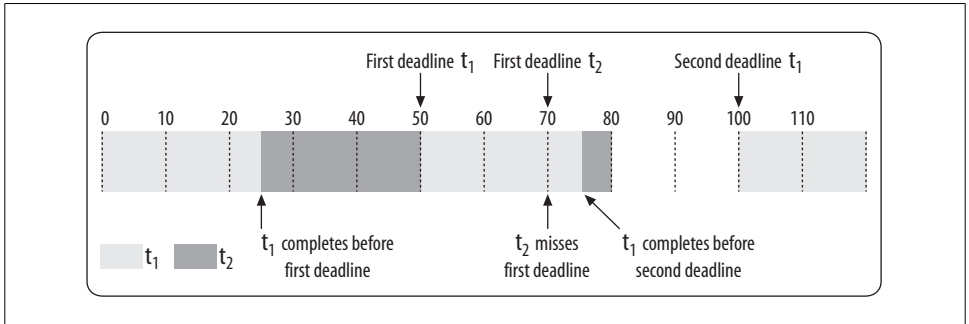
Figure 10-6. Example showing unschedulable task set

In this case, the only way to meet all deadlines is to use a dynamic scheduling algorithm, which, because it increases system complexity, is not available in many operating systems.

Sometimes a particular set of tasks will have total utilization above the worst-case schedulable bound and still be schedulable with fixed priorities. Case 1 in Figure 10-1 is a perfect example. Schedulability then depends on the specifics of the periods and execution times of each task in the set, which must be analyzed by hand. Only if the total utilization is less than W_n can you skip that manual analysis step and assume that all the tasks will meet all their deadlines.

RMA degrades gracefully in that if just one deadline will be missed, it is sure to be the one of the lowest-priority task with outstanding work. The "critical set" is the set of tasks that won't ever miss any deadlines. So, RMA works nicely for a mix of hard and soft real-time tasks, where the hard deadlines correspond to the highest-frequency tasks.

Task Mechanics

Earlier in this chapter, we discussed how the operating system makes it appear as if tasks are executing simultaneously. When writing code for tasks, it is best to keep this in mind and write tasks as if they run in parallel. The basic operation of a task is shown in Figure 10-7.

As shown in Figure 10-7, the first part initializes any task-specific variables or other resources used by the task. Then an infinite loop is used to perform the task work. First, the task waits for some type of event to occur. At this point, the task is blocked. It is in the waiting state and is put on the waiting queue. There are various types of events: a signal from another task, a signal from an ISR, and the expiration
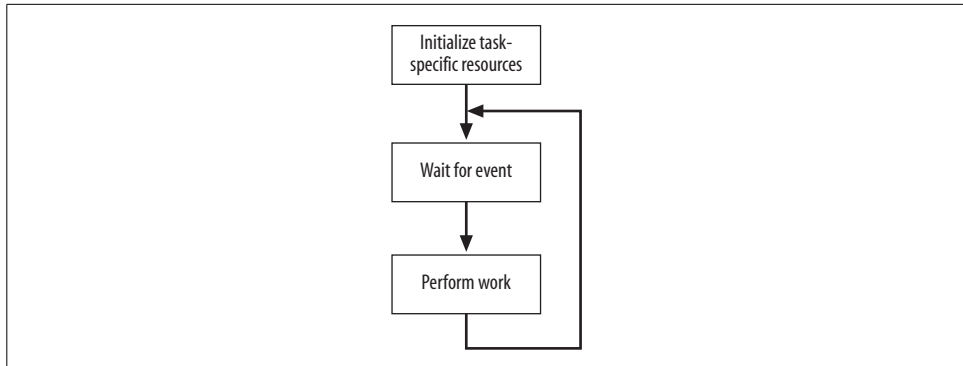
Figure 10-7. Basic task operation

of a timer set by the application. Once one of these events occurs, the task is in the ready state, and the scheduler can run the task when processor time becomes available.

The code for this basic task (using imaginary functions) would look something like this:

```
void taskBasicExample(void)
{
    /* Initalize all task-specific resources. */
    taskBasicInit();

    while (1)
    {
        /* Wait for the event to happen. */
        os_wait_for_event(event_type);

        /* Perform the work for this task. */

    }
}
```

Task Synchronization

Though we frequently talk about the tasks in a multitasking operating system as completely independent entities, that's not completely accurate. All of the tasks are working together to solve a larger problem and must occasionally communicate with one another to synchronize their activities. For example, in the print-server device, the printer task doesn't have any work to do until new data is supplied to it by one of the computer tasks. So the printer and computer tasks must communicate with one another to coordinate their access to common data buffers.

Operating systems contain various mechanisms that aid in synchronization between two tasks. Each of the mechanisms is useful for different scenarios. A particular operating system may offer additional or slightly different types of methods for synchronization, so we will cover the ones commonly found in real-time operating systems.

Application Programming Interfaces

One of the most annoying things about real-time operating systems is their lack of a common API. This is a particular problem for companies that want to share application code between products that are based on different operating systems.

The basic functionality of every real-time operating system is much the same. Each function represents a service that the operating system can perform for the application program. But there aren't that many different services possible. And it is frequently the case that the only real difference between two implementations is the name of the function.

This problem has persisted for the past several decades, and there is no end in sight. Yet during that same time, the Win32 and POSIX (pronounced "paw-zicks") APIs have taken hold on PCs and Unix workstations, respectively. *POSIX*, short for Portable Operating System Interface (the X was added to the end to make it sound like a variant of Unix), is an IEEE standard describing the API of a Unix-like process model operating system.

So why hasn't a similar standard emerged for embedded systems? It hasn't been for a lack of trying. In fact, the authors of the original POSIX standard (IEEE 1003.1) also created a standard for real-time systems (IEEE 1003.4b). And a few of the more Unix-like commercial real-time operating systems are compliant with this standard API. However, for the vast majority of application programmers, it is necessary to learn a new API for each operating system.

In Chapter 8, we introduced the concept of a critical section, which is simply a segment of code that must be executed in its entirety before the operating system is allowed to run anything else. Often a critical section consists of a single C-language statement that sets or reads a variable. We saw some examples using interrupts in Chapter 8; another example could be a status variable that is shared between a printer task and other tasks in the system.

Remember that in a multitasking environment, you generally don't know in which order the tasks will be executed at runtime. One task might be writing some data into a memory buffer when it is suddenly interrupted by a higher-priority task. If the higher-priority task were to modify that same region of memory, then bad things could happen. At the very least, some of the lower-priority task's data would be overwritten. When a round robin scheduler is in use—as it normally is on multitasking operating systems—even a task of the same priority can interrupt and access resources.

One way to ensure the instructions that make up the critical section are executed in order and without interruption is to disable interrupts. However, disabling interrupts when using an operating system may not be permitted and should be avoided; other mechanisms should be used to execute these atomic operations. We'll explore the ones you'll find in most operating systems.

Mutexes and Semaphores

One form of synchronization is a *mutex* (short for *mut*ual *ex*clusion). A mutex ensures exclusive access to shared variables or hardware. A mutex is analogous to a bathroom key in a high-traffic rest stop. Only one person (task) is allowed to use the bathroom (shared resource) at a time. That person (task) must first acquire the bathroom key (mutex), of which there's only one copy. The shopkeeper, who owns the key, is analogous to the operating system, which owns the mutex.

You can think of a mutex as being nothing more than a multitasking-aware binary flag. The meaning associated with a particular mutex must, therefore, be chosen by the software designer and understood by each of the tasks that use it. For example, the data buffer that is shared by the printer and computer task in the print server would have a mutex associated with it. When this binary flag is set, it is assumed that one of the tasks is using the shared data buffer. All other tasks must wait until that flag is cleared (and then until they set it again themselves) before reading or writing any of the data within that buffer. A task must wait for and acquire a mutex before releasing the mutex.

We say that mutexes are multitasking-aware because the processes of setting and clearing the binary flag are atomic. A task can safely change the state of the mutex without risking that a context switch will occur in the middle of the modification. If a context switch were to occur, the binary flag might be left in an unpredictable state, and a deadlock between the tasks could result. The atomicity of the mutex set and clear operations is enforced by the operating system, which disables interrupts before reading or modifying the state of the binary flag.

Mutexes are used for the protection of shared resources between tasks in an operating system. Shared resources are global variables, memory buffers, or device registers that are accessed by multiple tasks. A mutex can be used to limit access to such a resource to one task at a time.

In the operating system's code, interrupts can be disabled during the critical section. But generally, tasks should not disable interrupts. If they were allowed to do so, other tasks—even higher-priority tasks that didn't share the same resource—would not be able to execute during that interval. Mutexes provide a mechanism to protect critical sections within tasks without disabling interrupts.

Another synchronization mechanism is called a *semaphore*. A semaphore is used for intertask synchronization and is similar to the mutex. However, a semaphore's value can be any nonnegative integer value.

Let's go back to the bathroom analogy. Imagine now that there are two bathrooms (shared resources) and two keys (represented by the semaphore). We want to allow two people (tasks), but no more than two, to simultaneously use the bathrooms. If a key is available (the semaphore's value is not zero), the person can acquire the key and use the bathroom. If all keys are being used (the semphore's value is zero), the

next arriving person (task) must wait until a key becomes available. Each time a semaphore is signaled, its value is incremented. When a semaphore is acquired (after the task has waited for it), its value is decremented.

Semaphores are generally used as synchronization mechanisms, with one task signaling and another waiting. In our example of the print server, a semaphore can be used to signal to a task that incoming data is present. As data comes into the print server, it is buffered. Once a certain amount of data is accumulated, a semaphore can be used to signal the computer task that it is time to process that data. This example is similar to a relay race. In this case, the baton is the semaphore. Only the runner (task) with the baton is able to run. The computer task waiting for the incoming data is like the runner waiting for the baton. Once the baton is passed, the next runner (computer task) can proceed.

Mutexes should exclusively be used for controlling access to shared resources. While semaphores are typically used as signalling devices. A semaphore can be used to signal a task from another task or from an ISR—for example, to synchronize activities.

Deadlock and priority inversion

Mutexes are powerful tools for synchronizing access to shared resources. However, they are not without their own dangers. Two of the most important problems to watch out for are deadlock and priority inversion.

Deadlock can occur whenever there is a circular dependency between tasks and resources. The simplest example is that of two tasks: 1 and 2. Each task requires two mutexes: A and B. If Task 1 takes mutex A and waits for mutex B while Task 2 takes mutex B and waits for mutex A, then each task is waiting for the other to release the mutex. Here is an example of code demonstrating this deadlock problem:

```
void task1(void)
{
    os_wait_for_mutex(mutexA);
    os_wait_for_mutex(mutexB);

    /* Other task1 work. */
}

void task2(void)
{
    os_wait_for_mutex(mutexB);
    os_wait_for_mutex(mutexA);

    /* Other task2 work. */
}
```

These tasks may run without problems for a long time, but eventually one task may be preempted in between the wait calls, and the other task will run. In this case, Task 1 needs mutex B to be released by Task 2, while Task 2 needs mutex A to be released by Task 1. Neither of these events will ever happen.

When a deadlock occurs, it essentially brings both tasks to a halt, though other tasks might continue to run. The only way to end the deadlock is to reboot the entire system, and even that won't prevent it from happening again.

Priority inversion occurs whenever a higher-priority task is blocked, waiting for access to a shared resource that is currently not being used. This might not sound like too big of a deal—after all, the mutex is just doing its job of arbitrating access to the shared resource—because the higher-priority task is written with the knowledge that at times the lower-priority task will be using the resource they share. However, consider what happens when there is a third task with a priority level somewhere between those two. This situation is illustrated in Figure 10-8.
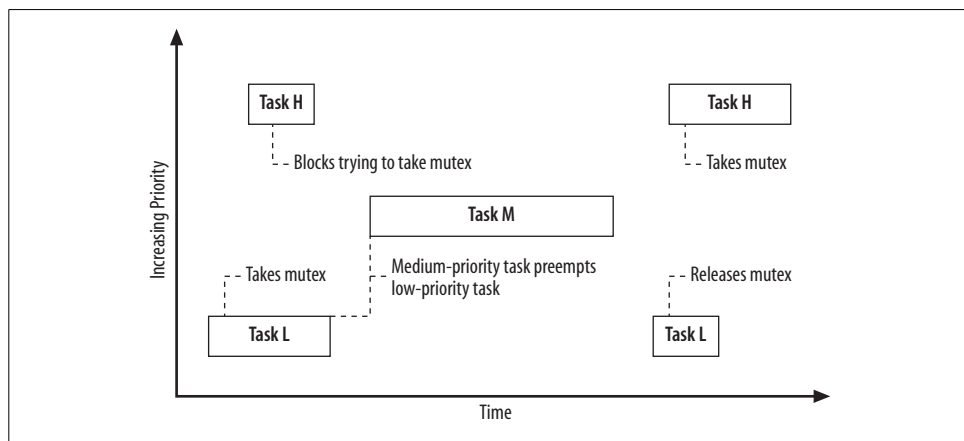
Figure 10-8. An example of priority inversion

In Figure 10-8, there are three tasks: Task H (high priority), Task M (medium priority), and Task L (low priority). Task L becomes ready first and, shortly thereafter, takes the mutex. Now, when Task H becomes ready, it must block until Task L is done with their shared resource. The problem is that Task M, which does not even require access to that resource, gets to preempt Task L and run, though it will delay Task H from using the processor. Once Task M completes, Task L runs until it releases the semaphore. Finally, at this point Task H gets its chance to run. This example shows how the task priorities can be violated because of the mutex sharing between Task H and Task L.

Several solutions to this problem have been developed. One of the most widely used solutions is called *priority inheritance*. This technique mandates that a lower-priority task inherit the priority of any higher-priority task that is waiting on a resource they share. This priority change should take place as soon as the higher-priority task begins to wait; it should end when the resource is released. This requires help from the operating system. If we apply this to the preceding example, the priority of Task L is increased to that of Task H as soon as Task H begins waiting for the mutex.

Once Task L releases the mutex, its priority is set to what it was before. Task L cannot be preempted by Task M until it releases the mutex, and Task H cannot be delayed unnecessarily.

Another solution is called *priority ceilings*. In this case, a priority value is associated with each resource; the scheduler then transfers that priority to any task that accesses the resource. The priority assigned to the resource is the priority of its highest-priority user, plus one. Once a task finishes with the resource, its priority returns to normal. One disadvantage of using priority ceilings is that the priority level for tasks using the mutex must be known ahead of time so the proper ceiling value can be set. This means the operating system cannot do the job automatically for you. Another disadvantage is that if the ceiling value is set too high, other unrelated tasks with priority levels below the ceiling can be locked out from executing. The priority ceiling is used even when priority inversion is not occurring, as a prophylactic measure.

The need for synchronization mechanisms needs to be thought out during the design of the software. Once designed into the software, everyone must use the mutex properly in order to access the shared resource. If someone breaks the rule, the software will not always operate as designed. In large, multiprogrammer projects, it can be hard for different programmers to realize that a resource is shared.

Message Passing

While semaphores can be used to signal from one task to another, there may be times when data needs to be passed in addition to the signal. For this purpose, operating systems frequently provide another mechanism called a *message queue* (or *mailbox*).

The operating system handles the buffering and buffer management for message passing as well as the safe communication of data between tasks. Message passing is therefore an alternative to the simple expedient of storing data in a global variable, and offers a much cleaner and more bug-free method of data exchange. Real-time operating systems typically use pointers for accessing the message data for reasons of speed and memory conservation.

Many applications that use message queues consist of a producer task that sends the data and a consumer task that receives it. There can be multiple producer and/or consumer tasks. The message content is typically understood between the sender and receiver ahead of time.

If the message queue is full, the operating system can block the sending task until space is available for the message. Similarly, if no message is present when the receiver attempts to read the message, the operating system blocks the receiving task. The size of the queue may vary depending on the message traffic.

Other Functionality

We have now covered the basic mechanisms that are commonly found in most real-time operating systems. Several other features may be included, depending on the operating system, to perform various other useful operations. Some of the other mechanisms commonly found in real-time operating systems are:

Event flags
> These allow a task to wait for multiple events to occur before unblocking. Either all of the events must occur (the events are ANDed together) or any of the events may occur (the events are ORed together).

Condition variables
> These are similar to a counting semaphore where a task signals another task to wake up; however, unlike a semaphore, if no task is currently waiting on the condition variable when it is signaled, the signal is lost.

Spinlocks
> These are similar to a mutex and are typically used in symmetric multiprocessing (SMP) systems. Like a mutex, a spinlock is a binary flag that a task attempts to claim. If the flag is not set, the task is able to obtain the spinlock. If the flag is set, the task will spin in a loop, constantly checking to see when the flag is not set. This might seem wasteful (and it can be); however, it is assumed that the spinlock is only held for a very short period of time, and this CPU must wait for software runnning on the other CPU to progress first anyway.

Counters and alarms
> A counter keeps track of the number of times a specific event has occurred. An alarm is used in conjunction with a counter to wake up a task (so that it will take action) when a particular number of events have occurred.

Interrupt Handling

There are several issues you need to be aware of when handling interrupts in embedded systems that use an operating system, including:

Interrupt priority
> Interrupts have the highest priority in a system—even higher than the highest operating system task. Interrupts are not scheduled; the ISR executes outside of the operating system's scheduler.

Disabling interrupts
> Because the operating system code must guarantee its data structures' integrity when they are accessed, the operating system disables interrupts during operations that alter internal operating system data, such as the ready list. This increases the interrupt latency. The responsiveness of the operating system comes at the price of longer interrupt latency.

When a task disables interrupts, it prevents the scheduler from doing its job. Tasks should not disable interrupts on their own.

Interrupt stack

Some operating systems have a separate stack space for the execution of ISRs. This is important because, if interrupts are stored on the same stack as regular tasks, each task's stack must accommodate the worst-case interrupt nesting scenario. Such large stacks increase RAM requirements across all *n* tasks.

Signaling tasks

Because ISRs execute outside of the scheduler, they are not allowed to make any operating system calls that can block. For example, an ISR cannot wait for a semaphore, though it can signal one.

Some operating systems use a split interrupt handling scheme, where the interrupt processing is divided into two parts. The first part is an ISR that handles the bare minimum processing of the interrupt. The idea is to keep the ISR short and quick.

The second part is handled by a DSR. The DSR handles the more extensive processing of the interrupt event. It runs when task scheduling is allowed; however, the DSR still has a higher priority than any task in the system. The DSR is able to signal a task to perform work triggered by the interrupt event.

For example, in the print server device, an interrupt might be used to handle incoming data from the computers on the Ethernet network. The Ethernet controller would interrupt the processor when a packet is received. Using the split interrupt handling scheme, the ISR would handle the minimal initial work: determining the interrupt event, masking further Ethernet interrupts, and acknowledging the interrupt. The ISR would then tell the operating system to run the DSR, which would then handle the low-level data packet processing before passing the data on to a task for further processing.

Real-Time Characteristics

Engineers often use the term *real-time* to describe computing problems for which a late answer is as bad as a wrong one. These problems are said to have *deadlines*, and embedded systems frequently operate under such constraints. For example, if the embedded software that controls your antilock brakes misses one of its deadlines, you might find yourself in an accident. So it is extremely important that the designers of real-time embedded systems know everything they can about the behavior and performance of their hardware and software. In this section, we will discuss the performance characteristics of real-time operating systems.

The designers of real-time systems spend a large amount of their time worrying about worst-case performance. They must constantly ask themselves questions such as: what is the worst-case amount of time between the moment a human operator presses the brake pedal and the moment an interrupt signal arrives at the processor?

What is the worst-case interrupt latency? And what is the worst-case amount of time for the software to respond by triggering the braking mechanism? Average or expected-case analysis simply will not suffice in such systems.

Most of the real-time operating systems available today are designed for possible inclusion in real-time systems. Ideally, their worst-case performance is well understood and documented. To earn the distinctive title of "real-time operating system," an operating system should be deterministic and have guaranteed worst-case interrupt latency and context switch times. Given these characteristics and the relative priorities of the tasks and interrupts in your system, it is possible to analyze the worst-case performance of the software.

An operating system is said to be *deterministic* if the worst-case execution time of each of the system calls is calculable. Operating system designers who take real-time behavior seriously usually publish a data sheet that provides the minimum, average, and maximum number of clock cycles required by each system call. These numbers are usually different for different processors. Therefore, it is important when comparing these numbers that equivalent (or better yet, the same) hardware is used. But it is reasonable to expect that if the algorithm is deterministic on one processor, it will be so on any other. The actual times can differ.

Interrupt latency is a key in determining the responsiveness of an RTOS. An RTOS adds to the interrupt latency because it must do some processing once an interrupt occurs. The amount of time this processing takes is an important characteristic of the RTOS for a real-time system. When an interrupt occurs, the processor must take several steps before executing the ISR. First, the processor must finish executing the current instruction. That probably takes less than one clock cycle, but some complex instructions require more time than that. Next, the interrupt type must be recognized. This is done by the processor hardware and does not slow or suspend the running task. Then, the RTOS must process the interrupt and determine which ISR is called. Finally, and only if interrupts are enabled, the ISR that is associated with the interrupt is started.

Of course, if interrupts are ever disabled within the operating system, the worst-case interrupt latency increases by the maximum amount of time that they are turned off. But operating systems have certain places where interrupts must be disabled. These are the internal critical sections—relating to operating system structures—described earlier; there are no alternative methods for the operating system to protect them. Each operating system will disable interrupts for a different length of time, so it is important that you know what your system's requirements are. One real-time project might require a guaranteed interrupt response time as short as 10 μs, while another requires only 100 ms.

The third real-time characteristic of an operating system is the amount of time required to perform a context switch. This is important because it represents overhead across your entire system. For example, imagine that the average execution

time of any task before it blocks is 100 ms but that the context switch time is also 100 ms. In that case, fully one-half of the processor's time is spent within the context switch routine! Again, the actual times are usually processor-specific because they are dependent on the number of registers that must be saved. Be sure to get these numbers for any operating system you are thinking of using. That way, there won't be any last-minute surprises.

To Use or Not to Use an RTOS

The answer is…it depends. In many cases, there is no clear-cut answer to this question. Many embedded systems can (and do) operate exactly as they need to by using an infinite loop, as we discussed in Chapter 3. These embedded systems do not need to be complicated by adding additional software, such as an RTOS. There's no prize for making an embedded system more complicated.

Each project should be evaluated on its own. Start with the notion that you do not need an RTOS. Then take a look at the overall system requirements. Make a list of the different software modules the system will need in order to meet these requirements.

Let's go back to the example of the print server device. The data-flow diagram in Chapter 2 is a good starting point. Some of the modules needed for the print server are an interrupt subsystem to handle timer and peripheral interrupts, a handler for the parallel port to communicate and send data to the printer, a networking stack for communication with computers via the Ethernet controller, a debug module that uses a serial port for output (this is not required, but it is helpful), and possibly a monitor and control command-line interface.

Now you can ask some questions about these modules in the system to find out the responsibilities of each. It might help to draw this out in your project notebook. How will these modules interact with each other? Are the modules independent and standalone or do they have interdependencies? Will they need to share memory or other hardware resources?

In the print server example, a networking stack is required. This might not be something you would want to create from scratch. Several networking stacks are available and many RTOSes include them as well.

There may not be easy answers to some of these questions. They are not solely based on technical issues. In the absence of a clear-cut winner, it's probably best to err on the side of what makes the software easier to read and implement. Making a list of pros and cons might aid in the decision process.

Now, if your decision is to use an RTOS, move on to the next section for a discussion of some criteria for determining which RTOS is best for the project.

RTOS Selection Process

The previous edition of this book showed how to build your own RTOS. Despite this, we strongly recommend using an existing operating system. Let us say that again: we highly recommend using an off-the-shelf operating system rather than writing your own. A wide variety of operating systems are available to suit nearly every project and pocketbook. Using an off-the-shelf operating system allows your software team to focus on the development of the application for the product. Granted, there will be a learning curve to get up to speed on using the operating system.

In this section, we will discuss the process of selecting the operating system that best fits the needs of your project. Selecting an RTOS can be tricky. There are plenty of criteria to consider when making this decision. Of course, the criteria are typically weighted differently from project to project and company to company.

Let's take a look at some of the important criteria used in making an RTOS selection:

Processor support
> The processor is typically the first choice in the hardware design on a project. Most RTOSes support the popular processors (or at least processor families) used in embedded systems. If the processor used on your project is not supported, you need to determine whether porting the RTOS to that processor is an option or if it is necessary to choose a different RTOS. Porting an RTOS is not always trivial.

Real-time characteristics
> We have already covered the real-time characteristics of an RTOS, which include interrupt latency, context switch time, and the execution time of each system call. These are technical criteria that are inherent to the system and cannot be changed.

Budget constraints
> RTOSes span the cost spectrum from open source and royalty free to tens of thousands of dollars per developer seat plus royalties for each unit shipped. You need to understand what your costs are in both cases. Open source might mean no upfront costs, but there might be costs associated with getting support when needed. You also need to understand the licensing details of the RTOS you choose.

Memory usage
> Clearly, in an embedded environment, memory constraints are a frequent concern. A few RTOSes can be scaled to fit the smallest of embedded systems—for example, by removing features to create a smaller footprint. Others require a minimum set of resources comparable to a low-end PC. It is important to keep in mind the potential need to change an RTOS in the future, when memory is not as plentiful or costs need to be reduced.

Device drivers and software components

The device drivers included with an RTOS can aid in keeping the development on schedule. This reduces the amount of code you need to develop for particular peripherals. Many RTOSes support the common devices found in embedded systems.

If additional features are needed, such as networking support, graphics libraries, web interfaces, and filesystems, an RTOS might include these and have the code already integrated and tested. Some RTOSes might require more fees for using these added features. If the necessary features are not included, you will need to identify third parties that provide them so that these components can be integrated into the system.

Technical support

This may include a number of incidents or a period of phone support. Some RTOSes require you to pay an annual fee to maintain a service contract. For open source RTOSes, an open forum or mailing list might be provided. If more specialized support is needed, you'll have to search around to see what is available. Popular open source RTOSes have companies dedicated to providing support.

Tool compatibility

Make sure the RTOS works with the assembler, compiler, linker, and debugger you have already obtained. If the RTOS does not support tools that you or your team are familiar with, the learning curve will take more time.

No matter which RTOS you choose, our advice is to get the source code if you can. The reason for this is that if you can't get support when you need it (say, at 1 A.M. for a deadline coming at 8 A.M., or if the operating system vendor stops supporting the product), you'll be glad to be able to find and fix the problem yourself. Some proprietary RTOSes provide only object code. Find out what is provided before you make your final decision.

With such a wide variety of operating systems and features to choose from, it can be difficult to decide which is the best for your project. Try putting your processor, real-time performance, and budgetary requirements first. These are criteria that you probably cannot change, so you can use them to narrow the possible choices to a dozen or fewer products. Then you can focus on the more detailed technical information.

At this point, many developers make their decision based on compatibility with existing cross-compilers, debuggers, and other development tools. But it's really up to you to decide what additional features are most important for your project. No matter what you decide, the basic kernel and task mechanisms will be pretty much the same as those described in this chapter. The differences will most likely be measured in processors supported, minimum and maximum memory requirements, availability of add-on software modules (networking protocol stacks, device drivers, and filesystems are common examples), and compatibility with third-party development tools.

Additional Resources

Another good set of criteria for RTOS selection can be found in the March 1999 *Embedded Systems Programming* article "Selecting a Real-Time Operating System," which can be found online at *http://www.embedded.com*. The list of vendors might be a bit outdated, but the information is still very useful.

If you would like to dig deeper into the inner workings of real-time operating systems, here are two resources we suggest: *MicroC/OS-II: The Real-Time Kernel*, by Jean J. Labrosse (CMP Books) and *Real-Time Concepts for Embedded Systems*, by Qing Li and Caroline Yao (CMP Books).

eCos Examples

Henry Hill: *You're a pistol, you're really funny.*
You're really funny.
Tommy DeVito: *What do you mean I'm funny?*
Henry Hill: *It's funny, you know. It's a good story, it's*
funny, you're a funny guy.
Tommy DeVito: *What do you mean, you mean the*
way I talk? What?
Henry Hill: *It's just, you know. You're just funny,*
it's…funny, the way you tell the story and everything.
Tommy DeVito: *Funny how? What's funny about it?*
—the movie *Goodfellas*

In this chapter, we will go through some examples of embedded system code that use the operating system mechanisms we covered in the previous chapter, under the principle that seeing real implementations contributes to a better understanding of the mechanisms. This chapter uses the real-time operating system eCos for the examples. The concepts and techniques in this chapter apply to other RTOSes as well, but different operating systems use different APIs to carry out the techniques. In the next chapter, we will run through the same examples using the popular Linux operating system.

Introduction

We have decided to use two operating systems for the examples of operating system use—eCos and Linux. Why did we choose these two? Both are open source, royalty-free, feature-rich, and growing in popularity. Learning how to program using them will probably enhance your ability to be productive with embedded systems. In addition, both operating systems are compatible with the free GNU software development tools. And both are up and running on the Arcom board.

eCos was developed specifically for use in real-time embedded systems, whereas Linux was developed for use on PCs and then subsequently ported to various processors used in embedded systems. Some embedded Linux distributions can require a

significant amount of resources (mainly memory and processing power), which are not found in most embedded systems. Linux was originally not real-time, but extensions are now available to add real-time features.

In short, eCos is more suited in general for embedded systems work, but the weight of widespread knowledge and support tools surrounding Linux make it attractive to embedded developers, too.

We try to keep function names, and what these functions do, consistent between the different examples in this and the following chapter. This way, you can concentrate on the details related to the operating systems. eCos includes a POSIX API, which supports a subset of POSIX functions.* However, in the following eCos examples, the native eCos API is used.

The instructions for setting up the eCos build environment and building the example eCos applications are covered in Appendix D. Additional information about eCos can be found online at *http://ecos.sourceware.org* as well as in the book *Embedded Software Development with eCos*, by Anthony Massa (Prentice Hall PTR).

> In order to keep the examples in this chapter shorter and easier to read, we don't bother to check the return values from operating system function calls (although many eCos system calls do not return a value). In general, it is a good idea to validate all return codes. This provides feedback about potential problems and allows you, as the developer, to make decisions in the software based on failed calls to the operating system. Basically, it makes your code more robust and, hopefully, less buggy.

Task Mechanics

In this first eCos example, we reuse the Blinking LED program that was covered previously. The first thing to learn is how to create a task. This example creates a task to handle the toggling of the LED at a constant rate. First, we declare the task-specific variables such as the stack and its size, the task priority, and OS-specific variables.

> eCos uses the term *thread* instead of *task* in its API and variable types. These terms mean the same thing in the embedded systems context.

The example program provides two variables to eCos to allow it to track the task. The variable ledTaskObj stores information about the task, such as its current state; ledTaskHdl is a unique value assigned to the task.

---

* The eCos POSIX API is currently compatible with the 1003.1-1996 version of the standard and includes elements from the 1004.1-2001 version.

The stack for our task is statically declared as the array `ledTaskStack`, which is 4,096 bytes in size. An arbitrary priority of 12 is assigned for this task. The code also defines the number of clock ticks per second, a value specific to the Arcom board's eCos setup. This makes it easy to change the tick interval to tune system performance.

```
#define TICKS_PER_SECOND           (100)

#define LED_TASK_STACK_SIZE        (4096)
#define LED_TASK_PRIORITY          (12)

/* Declare the task variables. */
unsigned char ledTaskStack[LED_TASK_STACK_SIZE];
cyg_thread ledTaskObj;
cyg_handle_t ledTaskHdl;
```

Next we show the code for performing the toggle. We have attempted to reuse code from the original Blinking LED example. The `ledInit` and `ledToggle` LED driver functions remain unchanged from the code described in Chapter 3.

The task `blinkLedTask` immediately enters an infinite loop. The infinite loop is used to keep the task continually running and blinking the LED. The first routine called in the infinite loop is `cyg_thread_delay`. This is an eCos function that suspends a task until a specified number of clock ticks have elapsed. The parameter passed into the delay routine determines how long to suspend the task and is based on the system clock used in eCos. At this point, the `blinkLedTask` is blocked and put in the waiting state by the eCos scheduler.

Once the timer expires, the eCos scheduler puts the `blinkLedTask` into the ready queue. If no other higher-priority tasks are ready to execute (which is the situation in this case), the scheduler runs the `blinkLedTask`; the task continues executing from the point at which it was blocked.

Next, `ledToggle` is called in order to change the state of the LED. When `ledToggle` completes and returns, `cyg_thread_delay` is called to delay for another 500 ms. The `blinkLedTask` is placed back in the waiting state until the time elapses again.

```
#include <cyg/kernel/kapi.h>
#include "led.h"

/**********************************************************************
 *
 * Function:    blinkLedTask
 *
 * Description: This task handles toggling the green LED at a
 *              constant interval.
 *
 * Notes:
 *
 * Returns:     None.
 *
 **********************************************************************/
```

```c
    void blinkLedTask(cyg_addrword_t data)
    {
        while (1)
        {
            /* Delay for 500 milliseconds. */
            cyg_thread_delay(TICKS_PER_SECOND / 2);

            ledToggle( );
        }
    }
```

Following is the code to create and start the LED task. The first thing to notice is that
instead of the function main, eCos programs use a function called cyg_user_start.

The first job of cyg_user_start is to initialize the LED by calling the function
ledInit. Next, the blinkLedTask task is created. In eCos, tasks created during initial-
ization (when the scheduler is not running) are initially suspended. To allow the
scheduler to run the task, cyg_thread_resume is called. Additionally, the scheduler
does not run until cyg_user_start exits; then the eCos scheduler takes over.

```c
    /************************************************************************
     *
     * Function:     cyg_user_start
     *
     * Description: Main routine for the eCos Blinking LED program. This
     *              function creates the LED task.
     *
     * Notes:       This routine invokes the scheduler upon exit.
     *
     * Returns:     None.
     *
     ************************************************************************/
    void cyg_user_start(void)
    {
        /* Configure the green LED control pin. */
        ledInit( );

        /* Create the LED task. */
        cyg_thread_create(LED_TASK_PRIORITY,
                          blinkLedTask,
                          (cyg_addrword_t)0,
                          "LED Task",
                          (void *)ledTaskStack,
                          LED_TASK_STACK_SIZE,
                          &ledTaskHdl,
                          &ledTaskObj);

        /* Notify the scheduler to start running the task. */
        cyg_thread_resume(ledTaskHdl);
    }
```

The previous example demonstrates how to create, resume, and delay a task in eCos. Other task operations include deleting tasks (which can sometimes occur by returning from the task function), yielding to other tasks in the system, and other mechanisms for suspending/resuming tasks.

Mutex Task Synchronization

Now we attack the next big job in task management, which is to synchronize tasks. As we saw in Chapter 10, a mutex is a common mechanism for getting two independent tasks to cooperate. For our eCos mutex example, two tasks share a common variable—the first task incrementing it and the second decrementing it at set intervals. The mutex is used to protect the shared variable.

Before accessing a shared resource, a task takes the mutex; once finished, the task releases the mutex for other tasks to use. Each operating system defines these two operations in its own way. For example, eCos offers lock (taking the mutex) and unlock (releasing the mutex) functions.

To keep track of the multiple tasks waiting on the same mutex, the mutex structure contains a queue of tasks that are waiting on that particular mutex. This queue is typically sorted by priority so the highest-priority task waiting for the mutex executes first. One possible result of releasing the mutex could be to wake a task of higher priority. In that case, the releasing task would immediately be forced (by the scheduler) to give up control of the processor, in favor of the higher-priority task.

The main function, cyg_user_start, which is shown next, calls the cyg_mutex_init function to initialize mutexes, such as the one we've named sharedVariableMutex. This mutex is used to protect the variable gSharedVariable. After returning from the mutex initialization call, the mutex is available to the first task that takes it. Once the mutex is initialized, the two tasks are created and then started in a manner like that shown earlier.
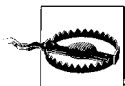
It is important to note that the mutex is initialized prior to the creation of any tasks that use this mutex. Any synchronization mechanism used by a task must be initialized prior to its use by the task. If a task were to try to take an uninitialized mutex, undefined behavior or a system crash could result.

You may notice that the function debug_printf is called at the end of cyg_user_start. This is eCos's lightweight version of printf.

```
#include <cyg/kernel/kapi.h>
#include <cyg/infra/diag.h>

cyg_mutex_t sharedVariableMutex;
int32_t gSharedVariable = 0;
```

```
    /**********************************************************************
     *
     * Function:    cyg_user_start
     *
     * Description: Main routine for the eCos mutex example. This function
     *              creates the mutex and then the increment and decrement
     *              tasks.
     *
     * Notes:       This routine invokes the scheduler upon exit.
     *
     * Returns:     None.
     *
     **********************************************************************/
    void cyg_user_start(void)
    {
        /* Create the mutex for accessing the shared variable. */
        cyg_mutex_init(&sharedVariableMutex);

        /* Create the increment and decrement tasks. */
        cyg_thread_create(INCREMENT_TASK_PRIORITY,
                          incrementTask,
                          (cyg_addrword_t)0,
                          "Increment Task",
                          (void *)incrementTaskStack,
                          INCREMENT_TASK_STACK_SIZE,
                          &incrementTaskHdl,
                          &incrementTaskObj);

        cyg_thread_create(DECREMENT_TASK_PRIORITY,
                          decrementTask,
                          (cyg_addrword_t)0,
                          "Decrement Task",
                          (void *)decrementTaskStack,
                          DECREMENT_TASK_STACK_SIZE,
                          &decrementTaskHdl,
                          &decrementTaskObj);

        /* Notify the scheduler to start running the tasks. */
        cyg_thread_resume(incrementTaskHdl);
        cyg_thread_resume(decrementTaskHdl);

        diag_printf("eCos mutex example.\n");
    }
```

The incrementTask function first delays for three seconds. After the delay, the function tries to take the mutex by calling cyg_mutex_lock, passing in the mutex it wishes to acquire. If the mutex is available, cyg_mutex_lock returns and the task can proceed. If the mutex is not available, the task blocks at this point (and is placed in the waiting state by the scheduler) and waits for the mutex to be released.

Once the incrementTask task obtains the mutex, the shared variable gSharedVariable is incremented and its value is output. The mutex is then released by calling cyg_mutex_unlock, again passing in the mutex to release as a parameter. Unlike the

cyg_mutex_lock function, the unlock function never blocks, although it may cause a reschedule.

```
/**********************************************************************
 *
 * Function:    incrementTask
 *
 * Description: This task increments a shared variable.
 *
 * Notes:
 *
 * Returns:     None.
 *
 **********************************************************************/
void incrementTask(cyg_addrword_t data)
{
    while (1)
    {
        /* Delay for 3 seconds. */
        cyg_thread_delay(TICKS_PER_SECOND * 3);

        /* Wait for the mutex to become available. */
        cyg_mutex_lock(&sharedVariableMutex);

        gSharedVariable++;

        diag_printf("Increment Task: shared variable value is %d\n",
                    gSharedVariable);

        /* Release the mutex. */
        cyg_mutex_unlock(&sharedVariableMutex);
    }
}
```

The decrementTask function is similar to the previous increment task. First, the task delays for seven seconds. Then the task waits to acquire the sharedVariableMutex. Once the task gets the mutex, it decrements the gSharedVariable value and outputs its value. Finally, the task releases the mutex.

```
/**********************************************************************
 *
 * Function:    decrementTask
 *
 * Description: This task decrements a shared variable.
 *
 * Notes:
 *
 * Returns:     None.
 *
 **********************************************************************/
void decrementTask(cyg_addrword_t data)
{
    while (1)
    {
```

```
        /* Delay for 7 seconds. */
        cyg_thread_delay(TICKS_PER_SECOND * 7);

        /* Wait for the mutex to become available. */
        cyg_mutex_lock(&sharedVariableMutex);

        gSharedVariable--;

        diag_printf("Decrement Task: shared variable value is %d\n",
                    gSharedVariable);

        /* Release the mutex. */
        cyg_mutex_unlock(&sharedVariableMutex);
    }
}
```

Semaphore Task Synchronization

The eCos semaphore example is similar to a push button light switch using an LED for the light. The following example has two tasks: a producer and a consumer. The producer task, producerTask, monitors the button labeled SW0 on the Arcom board's add-on module. Figure 11-1 shows the button used in this example.

When the SW0 button is pressed, the producer task signals the consumer task using a semaphore. The consumer, consumerTask, waits for the semaphore signal from the producer task. Upon receiving the signal, the consumer task outputs a message and toggles the green LED.

The main function, cyg_user_start, starts by initializing the LED by calling ledInit. Next, the semaphore is initialized with a call to cyg_semaphore_init. The initial value of the semaphore, semButton, is set to zero so that the consumer task that is waiting does not execute until the semaphore is signaled by the producer task. Lastly, the two tasks are created and resumed, as in the prior example, and then a message is output signifying the start of the program.

```
#include <cyg/kernel/kapi.h>
#include <cyg/infra/diag.h>
#include "led.h"

cyg_sem_t semButton;

/***********************************************************************
 *
 * Function:    cyg_user_start
 *
 * Description: Main routine for the eCos semaphore program. This
 *              function creates the semaphore and the producer and
 *              consumer tasks.
 *
 * Notes:       This routine invokes the scheduler upon exit.
 *
```
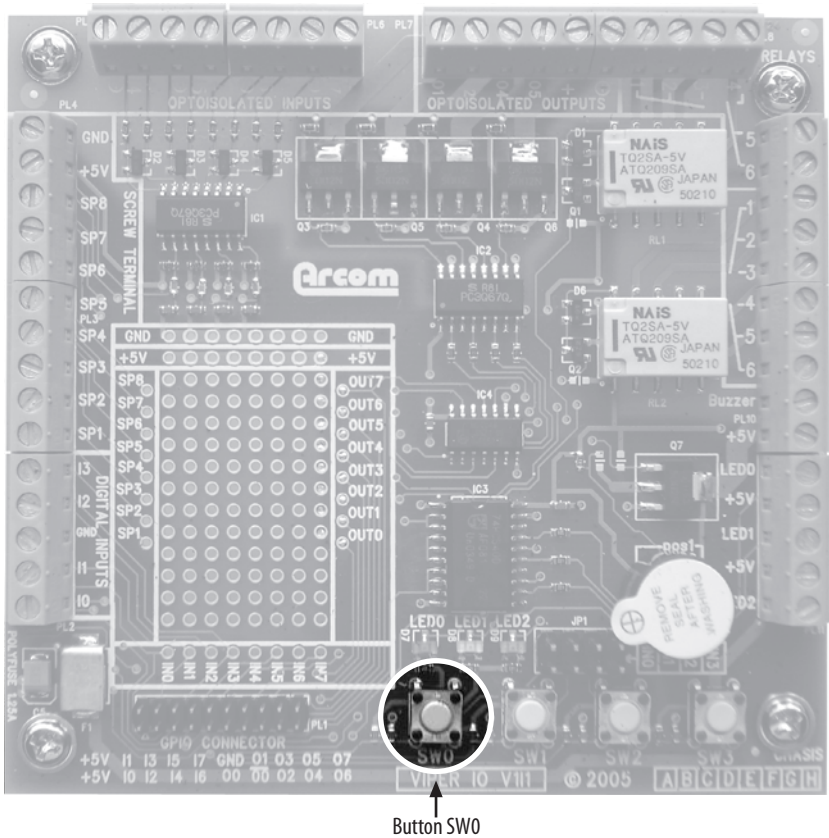
Figure 11-1. Arcom board add-on module's SW0 button

```
 * Returns:        None.
 *
 ************************************************************************/
void cyg_user_start(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    /* Create the semaphore with an initial value of zero. */
    cyg_semaphore_init(&semButton, 0);

    /* Create the producer and consumer tasks. */
    cyg_thread_create(PRODUCER_TASK_PRIORITY,
                      producerTask,
                      (cyg_addrword_t)0,
                      "Producer Task",
                      (void *)producerTaskStack,
                      PRODUCER_TASK_STACK_SIZE,
```

```
                        &producerTaskHdl,
                        &producerTaskObj);

    cyg_thread_create(CONSUMER_TASK_PRIORITY,
                        consumerTask,
                        (cyg_addrword_t)0,
                        "Consumer Task",
                        (void *)consumerTaskStack,
                        CONSUMER_TASK_STACK_SIZE,
                        &consumerTaskHdl,
                        &consumerTaskObj);

    /* Notify the scheduler to start running the tasks. */
    cyg_thread_resume(producerTaskHdl);
    cyg_thread_resume(consumerTaskHdl);

    diag_printf("eCos semaphore example - press button SW0.\n");
}
```

The producerTask contains an infinite loop that first delays for 10 ms and then checks to see whether the SW0 button has been pressed, by calling the function buttonDebounce (we will take a closer look at this function in a moment). The delay interval was selected in order to ensure the task is responsive when the button is pressed. For additional information about selecting sampling intervals, read the sidebar later in this chapter titled "Switch Debouncing."

When the SW0 button is pressed, the semaphore, semButton, is signaled by calling cyg_semaphore_post, which increments the semaphore value and wakes the consumer task waiting on this semaphore. The producer task then returns to monitoring the SW0 button.

```
#include "button.h"

/************************************************************************
 *
 * Function:     producerTask
 *
 * Description: This task monitors button SW0. Once pressed, the button
 *              is debounced and the semaphore is signaled, waking the
 *              waiting consumer task.
 *
 * Notes:
 *
 * Returns:      None.
 *
 ************************************************************************/
void producerTask(cyg_addrword_t data)
{
    int buttonOn;

    while (1)
    {
        /* Delay for 10 milliseconds. */
```

```
        cyg_thread_delay(TICKS_PER_SECOND / 100);

        /* Check whether the SW0 button has been pressed. */
        buttonOn = buttonDebounce();

        /* If button SW0 was pressed, signal the consumer task. */
        if (buttonOn)
            cyg_semaphore_post(&semButton);
    }
}
```

Now let's take a look at the function buttonDebounce. The debounce code is from the June 2004 *Embedded Systems Programming* article "My Favorite Software Debouncers," which can be found online at *http://www.embedded.com*.

The debounce function is called in the producer task every 10 ms to determine whether the SW0 button has been pressed. As shown in Figure 11-1, button SW0 is located on the add-on module. The Arcom board's *VIPER-Lite Technical Manual* and the *VIPER-I/O Technical Manual* describe how the add-on module's buttons are connected to the processor. The add-on module schematics, which are found in the *VIPER-I/O Technical Manual*, can be used to trace the connection from the button back to the processor.

The button SW0 is read from the signal IN0, as shown in the switches section in the *VIPER-I/O Technical Manual*. According to the *VIPER-Lite Technical Manual*, the IN0 signal value is retrieved by reading address 0x14500000. The least significant bit at this address contains the current state of the button SW0. The Arcom board's documentation states that the default voltage level on the button switch is high and that when the button is pressed, the value changes to low.

The debounce function first calls buttonRead, which returns the current state of the SW0 button. The current state of the SW0 button is shifted into the variable buttonState. When the leading edge of the switch closure is debounced and detected, TRUE is returned.

For additional information about debouncing buttons and switches, take a look at the sidebar "Switch Debouncing" later in this chapter.

```
/************************************************************************
 *
 * Function:    buttonDebounce
 *
 * Description: This function debounces buttons.
 *
 * Notes:
 *
 * Returns:     TRUE if the button edge is detected, otherwise
 *              FALSE is returned.
 *
 ************************************************************************/
int buttonDebounce(void)
```

```
    {
        static uint16_t buttonState = 0;
        uint8_t pinState;

        pinState = buttonRead();

        /* Store the current debounce status. */
        buttonState = ((buttonState << 1) | pinState | 0xE000);

        if (buttonState == 0xF000)
            return TRUE;

        return FALSE;
    }
```

The consumerTask contains a simple infinite loop: wait for the semaphore to be signaled, then print a message once the signal is received. The consumer task waits for the semaphore signal by calling cyg_semaphore_wait, which blocks the task if the value of the semaphore is equal to 0.

Once the semaphore signal is received, the consumer task outputs a message that the button was pressed and toggles the green LED by calling ledToggle. After the LED is toggled, the consumer task reverts to waiting for another semaphore signal.

```
    /***********************************************************************
     *
     * Function:    consumerTask
     *
     * Description: This task waits for the semaphore signal from the
     *              producer task. Once the signal is received, the task
     *              outputs a message.
     *
     * Notes:
     *
     * Returns:     None.
     *
     ***********************************************************************/
    void consumerTask(cyg_addrword_t data)
    {
        while (1)
        {
            /* Wait for the signal. */
            cyg_semaphore_wait(&semButton);

            diag_printf("Button SW0 was pressed.\n");

            ledToggle();
        }
    }
```

We cover another example using semaphores in the section "eCos Interrupt Handling" later in this chapter.

Message Passing

In this section, we'll discuss a programming technique that's useful for certain situations where you divide up tasks—in particular, when you can specify a producer task that generates data, and a consumer task that processes the producer's output. The use of message passing prevents tasks from stepping on each other's data and also simplifies coding.

The message passing example is similar to the light switch-semaphore example shown earlier in this chapter, where the producer task waits for the SW0 button to be pressed, and the consumer task outputs a message. The difference this time is that a message that contains the number of times the button has been pressed is passed from the producer to the consumer. The consumer, consumerTask, waits for the message from the producer task, producerTask. Once the message is received by the consumer task, it outputs a message and toggles the green LED.

Message queues in eCos are called mailboxes. The following main function, cyg_user_start, starts by initializing the LED by calling ledInit. Next, the mailbox is initialized with a call to cyg_mbox_create. The mailbox create function is passed mailboxHdl (a handle used for subsequent calls to perform operations with that specific mailbox) and mailbox (an area of memory for the kernel's mailbox structure). Lastly, the two tasks are created and resumed, as we saw in the prior example, and then a message is output, signifying the start of the program.

```
#include <cyg/kernel/kapi.h>
#include <cyg/infra/diag.h>
#include "led.h"

cyg_handle_t mailboxHdl;
cyg_mbox mailbox;

/***********************************************************************
 *
 * Function:    cyg_user_start
 *
 * Description: Main routine for the eCos mailbox program. This
 *              function creates the mailbox and the producer and
 *              consumer tasks.
 *
 * Notes:       This routine invokes the scheduler upon exit.
 *
 * Returns:     None.
 *
 ***********************************************************************/
void cyg_user_start(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    /* Create the mailbox for sending messages between tasks. */
```

```
            cyg_mbox_create(&mailboxHdl, &mailbox);

            /* Create the producer and consumer tasks. */
            cyg_thread_create(PRODUCER_TASK_PRIORITY,
                              producerTask,
                              (cyg_addrword_t)0,
                              "Producer Task",
                              (void *)producerTaskStack,
                              PRODUCER_TASK_STACK_SIZE,
                              &producerTaskHdl,
                              &producerTaskObj);

            cyg_thread_create(CONSUMER_TASK_PRIORITY,
                              consumerTask,
                              (cyg_addrword_t)0,
                              "Consumer Task",
                              (void *)consumerTaskStack,
                              CONSUMER_TASK_STACK_SIZE,
                              &consumerTaskHdl,
                              &consumerTaskObj);

            /* Notify the scheduler to start running the tasks. */
            cyg_thread_resume(producerTaskHdl);
            cyg_thread_resume(consumerTaskHdl);

            diag_printf("eCos mailbox example - press button SW0.\n");
        }
```

The producerTask shown next begins by initializing the button press count variable,
buttonPressCount, to zero, and then enters an infinite loop. In the loop, the task
delays for 10 ms and then checks to see whether the SW0 button has been pressed
with a call to the function buttonDebounce.

When the SW0 button is pressed, buttonPressCount is incremented and the value is
sent to the waiting consumer task via the mailbox. In eCos, messages are sent in
mailboxes using the function cyg_mbox_put, which takes two arguments: the mailbox
handle, in this case mailboxHdl, and the message to send. If there is room in the mail-
box, the message is placed there immediately; otherwise, the task blocks until room
is available.

```
    #include "button.h"

    /**********************************************************************
     *
     * Function:    producerTask
     *
     * Description: This task monitors button SW0. Once pressed, the button
     *              is debounced and a message is sent to the waiting
     *              consumer task.
     *
     * Notes:       This function is specific to the Arcom board.
     *
     * Returns:     None.
```

```
     *
     ********************************************************************/
    void producerTask(cyg_addrword_t data)
    {
        uint32_t buttonPressCount = 0;
        int buttonOn;

        while (1)
        {
            /* Delay for 10 milliseconds. */
            cyg_thread_delay(TICKS_PER_SECOND / 100);

            /* Check if the SW0 button has been pressed. */
            buttonOn = buttonDebounce();

            /* If button SW0 was pressed, send a message to the consumer task. */
            if (buttonOn)
            {
                buttonPressCount++;
                cyg_mbox_put(mailboxHdl, (void *)buttonPressCount);
            }
        }
    }
```

The consumerTask, shown in the code that follows, contains an infinite loop that
waits for an incoming message and then prints the message once it's received. The
consumer task waits for an incoming message by calling cyg_mbox_get and passing in
the mailbox handle, mailboxHdl. The mailbox get function call blocks until the pro-
ducer task sends a message. The message, which is the button press count, is stored
in the local variable rcvMsg. After the message is output and the green LED is tog-
gled, the consumer task returns to waiting for another message.

```
    /********************************************************************
     *
     * Function:    consumerTask
     *
     * Description: This task waits for a message from the producer task.
     *              Once the message is received via the mailbox, it
     *              outputs a message and toggles the green LED.
     *
     * Notes:
     *
     * Returns:     None.
     *
     ********************************************************************/
    void consumerTask(cyg_addrword_t data)
    {
        uint32_t rcvMsg;

        while (1)
        {
            /* Wait for a new message. */
            rcvMsg = (uint32_t)cyg_mbox_get(&mailboxHdl);
```

```
        diag_printf("Button SWO pressed %d times.\n", rcvMsg);

        ledToggle();
    }
}
```

Most operating systems, including eCos, have additional API functions for various synchronization and message passing operations. For example, eCos includes the functions cyg_mbox_tryput and cyg_mbox_tryget that return false if they are unsuccessful at putting or getting a message in the mailbox, instead of blocking the task. These functions can be used to do polling (that is, to check whether a mailbox is available, go off and do other tasks if it is not, and then return and try again).

There are also functions for which you pass in a timeout value that blocks for a set amount of time while the function attempts to perform the specified operation. Additional information about the eCos RTOS APIs can be found in the *eCos Reference* online at *http://ecos.sourceware.org/docs-latest*.

Switch Debouncing

When pressed or released, any mechanical input, such as a switch or button, will bounce open and closed briefly before settling. Processors are so fast that they can detect this rapid succession of opens and closes; thus, it's hard to know whether any individual read of a switch's position is accurate. This is a case where the processor sees the trees but not the forest. *Debouncing* is a technique used to smooth out the samples and make sure the processor doesn't get confused by the bouncing.

You can debounce inputs by inserting extra hardware or software to filter out this noise. Software debounce routines function more or less by testing the input regularly at a predetermined interval and making decisions only after a succession of reads that are the same. Writing debounce code is straightforward; selecting the sampling interval and deciding when the input has settled is more difficult, and specific to each input device and application.

For additional information on sampling intervals, refer to the July 2002 *Embedded Systems Programming* article "How to Choose A Sensible Sampling Rate," which can be found online at *http://www.embedded.com*.

---

eCos Interrupt Handling

As explained in Chapter 8, it is important to keep interrupt processing down to a minimum so that other (potentially higher-priority) interrupts in the system can be serviced, and so high-priority tasks can run. Thus, programmers typically divide interrupt handling into two categories: a short ISR, and a more leisurely DSR.

The question of how to make the split—what to put in the ISR and what to put in the DSR—depends on the application and the processor. Basically, you should defer whatever you can. OS functions that might block cannot be called from an ISR or DSR. And unlike many other operating systems, eCos in particular does not allow an ISR even to signal a semaphore via a nonblocking call. In eCos, semaphore signaling must be done via a call from the DSR.

In order to get a better understanding of the split interrupt handling scheme, take a look at the following eCos example, which shows the use of an interrupt to handle the timing for the Blinking LED program. In this example, a semaphore is used to signal a task from the interrupt when it is time to toggle the LED.

The initialization sequence between the hardware and software in the cyg_user_start function is important. For example, you wouldn't want to call timerInit to start the timer interrupt before you have created and installed an interrupt handler for the timer.

First, the LED is initialized. Next, the semaphore, ledToggleSemaphore, which signals the blinkLedTask when it is time to toggle the LED, is initialized. Then the blinkLedTask is created as shown previously.

Next, the ISR and DSR are created for the timer interrupt with a call to cyg_interrupt_create, which fills in the kernel interrupt structure. The interrupt vector (27 for Timer 1) is defined by the macro TIMER1_INT and passed in as the first parameter to cyg_interrupt_create. The next two parameters are the interrupt priority and interrupt-private data, which are set to zero. The ISR function, timerIsr, and DSR function, timerDsr, are passed in next. When the cyg_interrupt_create function returns, it sets the final two arguments: the interrupt handle, timerInterruptHdl, and interrupt object, timerInterruptObj. The interrupt handle is used in subsequent operations for this interrupt. The interrupt object provides the kernel with an area of memory containing the interrupt handler and its associated data (in case any is necessary).

Next, the interrupt ISR and DSR are attached to the interrupt source by calling cyg_interrupt_attach and passing in the handle from the interrupt create function. As a precaution, cyg_interrupt_acknowledge is called in case there is a pending timer interrupt. Lastly, the interrupt is unmasked by calling cyg_interrupt_unmask.

The final step in the initialization sequence is to configure the timer registers and enable the interrupt, which is done in the function timerInit:

```
#include <cyg/kernel/kapi.h>
#include "timer.h"
#include "led.h"

/* Declare the ISR variables. */
cyg_handle_t timerInterruptHdl;
cyg_interrupt timerInterruptObj;
cyg_vector_t timerInterruptVector = TIMER1_INT;

cyg_sem_t ledToggleSemaphore;
```

```
/*********************************************************************
 *
 * Function:    cyg_user_start
 *
 * Description: Main routine for eCos interrupt Blinking LED program.
 *              This function creates the LED toggle semaphore, the
 *              LED task, and the timer interrupt handler.
 *
 * Notes:       This routine invokes the scheduler upon exit.
 *
 * Returns:     None.
 *
 *********************************************************************/
void cyg_user_start(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    /* Create the semaphore for the task signaling. This semaphore
     * is initialized with a value of 0 so the toggling task must wait
     * for the first time period to elapse. */
    cyg_semaphore_init(&ledToggleSemaphore, 0);

    /* Create the LED task. */
    cyg_thread_create(LED_TASK_PRIORITY,
                      blinkLedTask,
                      (cyg_addrword_t)0,
                      "LED Task",
                      (void *)ledTaskStack,
                      LED_TASK_STACK_SIZE,
                      &ledTaskHdl,
                      &ledTaskObj);

    /* Notify the scheduler to start running the task. */
    cyg_thread_resume(ledTaskHdl);

    /* Initialize the interrupt for the timer. */
    cyg_interrupt_create(timerInterruptVector,
                         0,
                         0,
                         timerIsr,
                         timerDsr,
                         &timerInterruptHdl,
                         &timerInterruptObj);

    cyg_interrupt_attach(timerInterruptHdl);
    cyg_interrupt_acknowledge(timerInterruptVector);
    cyg_interrupt_unmask(timerInterruptVector);

    /* Initialize the timer registers. */
    timerInit();
}
```

eCos provides the functionality of saving and restoring the processor's context when an interrupt occurs so that the ISR does not need to perform these operations. Thus we reduce the workload of the timer interrupt handler, timerIsr, to three critical operations that must be carried out before interrupts are enabled again.

The first operation masks the timer interrupt, with a call to cyg_interrupt_mask passing in the interrupt vector timerInterruptVector, until the DSR is run. This blocks the ISR from being called again until the current interrupt has been processed.

The second operation performed in the ISR is to acknowledge the interrupt. The interrupt must be acknowledged in the processor's interrupt controller and timer peripheral. The interrupt is acknowledged in the interrupt controller using the eCos function cyg_interrupt_acknowledge and passing in the interrupt vector timerInterruptVector. The interrupt is acknowledged in the timer peripheral by writing the TIMER_1_MATCH (0x02) bit to the timer status register.

The third operation, performed when returning, is to inform the operating system that the timer interrupt has been handled (with the macro CYG_ISR_HANDLED) and that the DSR needs to be run (with the macro CYG_ISR_CALL_DSR). Notifying eCos that the interrupt has been handled prevents it from calling any other ISRs to handle the same interrupt.

The following timerIsr function shows these operations:

```
#include <cyg/hal/hal_intr.h>

/************************************************************************
 *
 * Function:    timerIsr
 *
 * Description: Interrupt service routine for the timer interrupt.
 *
 * Notes:
 *
 * Returns:     Bitmask to inform operating system that the
 *              interrupt has been handled and to schedule the
 *              deferred service routine.
 *
 ***********************************************************************/
uint32_t timerIsr(cyg_vector_t vector, cyg_addrword_t data)
{
    /* Block the timer interrupt from occurring until the DSR runs. */
    cyg_interrupt_mask(timerInterruptVector);

    /* Acknowledge the interrupt in the interrupt controller and the
     * timer peripheral. */
    cyg_interrupt_acknowledge(timerInterruptVector);
    TIMER_STATUS_REG = TIMER_1_MATCH;
```

```
          /* Inform the operating system that the interrupt is handled by this
           * ISR and that the DSR needs to run. */
          return (CYG_ISR_HANDLED | CYG_ISR_CALL_DSR);
    }
```

The DSR function, timerDsr, which is shown next, is scheduled to be run by eCos once the ISR completes. The DSR signals the LED task using the semaphore ledToggleSemaphore with the function call cyg_semaphore_post.

Next, the new timer interval is programmed into the timer match register, which is set to expire 500 ms from the current timer count. Finally, before exiting, the DSR unmasks the timer interrupt in the operating system, by calling cyg_interrupt_unmask and passing in the interrupt vector, which reenables the handling of incoming timer interrupts.

```
    /**********************************************************************
     *
     * Function:    timerDsr
     *
     * Description: Deferred service routine for the timer interrupt.
     *
     * Notes:
     *
     * Returns:     None.
     *
     **********************************************************************/
    void timerDsr(cyg_vector_t vector, cyg_ucount32 count, cyg_addrword_t data)
    {
        /* Signal the task to toggle the LED. */
        cyg_semaphore_post(&ledToggleSemaphore);

        /* Set the new timer interval. */
        TIMER_1_MATCH_REG = (TIMER_COUNT_REG + TIMER_INTERVAL_500MS);

        /* Enable processing of incoming timer interrupts. */
        cyg_interrupt_unmask(timerInterruptVector);
    }
```

The blinkLedTask contains an infinite loop that waits for the semaphore ledToggle-Semaphore to be signaled by calling cyg_semaphore_wait. When the semaphore is signaled by the timer DSR, the task calls ledToggle to change the state of the LED.

```
    /**********************************************************************
     *
     * Function:    blinkLedTask
     *
     * Description: This task handles toggling the LED when it is
     *              signaled from the timer interrupt handler.
     *
```

```
 * Notes:
 *
 * Returns:      None.
 *
 ********************************************************************/
void blinkLedTask(cyg_addrword_t data)
{
    while (1)
    {
        /* Wait for the signal that it is time to toggle the LED. */
        cyg_semaphore_wait(&ledToggleSemaphore);

        /* Change the state of the green LED. */
        ledToggle();
    }
}
```

This concludes our brief introduction to the eCos operating system and its API. Hopefully, these few examples have clarified some of the points made elsewhere in the book. These are valuable programming techniques used frequently in embedded systems.

Embedded Linux Examples

> **Ty Webb:** *Don't be obsessed with your desires,*
> *Danny. The Zen philosopher, Basho, once wrote: "A*
> *flute with no holes is not a flute…and a doughnut with*
> *no hole is a Danish." He was a funny guy.*
> —the movie *Caddyshack*

In this chapter, we will explore some examples using embedded Linux. The examples in this chapter are similar to (or in some cases the same as) the eCos examples we covered in the previous chapter. The idea here is to get an introduction to embedded Linux and understand some basic operating system functionality.

Introduction

The embedded Linux examples demonstrate certain basic APIs for various operations. Additional APIs exist that offer other functionality. You should research the additional APIs on your own to determine whether there are other, better ways to perform the operations necessary for your particular embedded system.

One aspect of Linux you need to be familiar with is its thread model. The Linux API conforms to the key POSIX standard in the space, POSIX 1003.1c, commonly called the *pthreads* standard. POSIX leaves many of the implementation details up to the operating system implementer. A good source of information on pthreads is the book *Pthreads Programming*, by Bradford Nichols, Dick Buttlar, and Jacqueline Farrell (O'Reilly).

The version of embedded Linux used on the Arcom board is a standard kernel tree (version 2.6) with additional ARM and XScale support from the ARM Linux Project at *http://www.arm.linux.org.uk*.

A plethora of books about Linux and embedded Linux are available. Some good resources include *Understanding the Linux Kernel*, by Daniel P. Bovet and Marco Cesati (O'Reilly), *Linux Device Drivers*, by Alessandro Rubini and Jonathan Corbet (O'Reilly), and *Building Embedded Linux Systems*, by Karim Yaghmour (O'Reilly).

The instructions for configuring the embedded Linux build environment and building the example Linux applications are detailed in Appendix E. Additional information about using embedded Linux on the Arcom board can be found in the *Arcom Embedded Linux Technical Manual* and the *VIPER-Lite Technical Manual*.

> In order to keep the examples in this chapter shorter and easier to read, we don't check the return values from function calls. In general, it is a good idea to validate all return codes. This provides feedback about potential problems and allows you, as the developer, to make decisions in the software based on failed calls. Basically, it makes your code more robust and, hopefully, less buggy.

Accessing Hardware in Linux

Before proceeding with the Linux examples, it is important to have a basic understanding of hardware access in Linux. Linux, like most desktop operating systems, partitions its memory management into *user space* and *kernel space*.

User space is where applications run (including the Linux examples that follow). User space applications are allowed to access the hardware only through kernel-supported functions. Kernel space is typically where device drivers exist. This allows the device drivers to have direct access to the hardware.

The Linux examples use a function called mmap in order to access a particular address range. The mmap function asks the kernel to provide access to a physical address range contained in the hardware. For details on how we use mmap, refer to the book's source code.

Task Mechanics

For the first Linux example, we reuse the Blinking LED program. This example shows how to create a task to toggle the LED at a constant rate.

The task blinkLedTask delays for 500 ms and then toggles the green LED. The task uses an infinite loop that calls the function usleep to suspend for the proper amount of time. The usleep function is passed the number of microseconds to suspend, which in this case is 50,000 μs.

After the delay function call, the blinkLedTask is blocked and put in the waiting state. The task is put in the ready state once the time has elapsed and then has been run by the scheduler. After the delay, the ledToggle function toggles the green LED.

```
#include <unistd.h>
#include "led.h"

/**********************************************************************
 *
```

```
 * Function:     blinkLedTask
 *
 * Description: This task handles toggling the green LED at a
 *              constant interval.
 *
 * Notes:
 *
 * Returns:      None.
 *
 **********************************************************************/
void blinkLedTask(void *param)
{
    while (1)
    {
        /* Delay for 500 milliseconds. */
        usleep(50000);

        ledToggle();
    }
}
```

The main function's job in this example is to create the LED task. The task is created by calling the function pthread_create. For this example, the default task attributes are used, and no parameters are passed to the task blinkLedTask.

The function pthread_join is used to suspend the main function until the blinkLedTask task terminates. In this case, the blinkLedTask task runs forever, so neither function exits.

```
#include <pthread.h>

/* Declare the task variables. */
pthread_t ledTaskObj;

/**********************************************************************
 *
 * Function:     main
 *
 * Description: Main routine for Linux Blinking LED program. This
 *              function creates the LED task.
 *
 * Notes:
 *
 * Returns:      0.
 *
 **********************************************************************/
int main(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    /* Create the LED task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&ledTaskObj, NULL, (void *)blinkLedTask, NULL);
```

```
        /* Allow the LED task to run. */
        pthread_join(ledTaskObj, NULL);

        return 0;
    }
```

Additional task operations are supported in Linux. These operations include termi-nating tasks, modifying task attributes, yielding to other tasks in the system, and suspending/resuming tasks.

Mutex Task Synchronization

In the Linux mutex example (just as in the eCos example), two tasks share a com-mon variable called gSharedVariable. One task increments the global variable at a set interval, and the other task decrements the variable at a set interval. The mutex pro-tects the shared variable.

The function main starts by creating the mutex, sharedVariableMutex, by calling the function pthread_mutex_init. Because the default attributes are used in the mutex creation, NULL is passed in as the second parameter. In Linux, mutexes have attributes that you can set using the second parameter, but we won't use them in this book, so we'll just pass NULL.

Lastly, the two tasks incrementTask and decrementTask are created. It is important to create the mutex before creating the tasks that use it, because otherwise the tasks could crash the program.

```
#include <pthread.h>

pthread_mutex_t sharedVariableMutex;
int32_t gSharedVariable = 0;

/***********************************************************************
 *
 * Function:    main
 *
 * Description: Main routine for the Linux mutex example. This
 *              function creates the mutex and then the increment and
 *              decrement tasks.
 *
 * Notes:
 *
 * Returns:     0.
 *
 ***********************************************************************/
int main(void)
{
    /* Create the mutex for accessing the shared variable using the
     * default attributes. */
    pthread_mutex_init(&sharedVariableMutex, NULL);
```

```
        /* Create the increment and decrement tasks using the default task
         * attributes. Do not pass in any parameters to the tasks. */
        pthread_create(&incrementTaskObj, NULL, (void *)incrementTask, NULL);
        pthread_create(&decrementTaskObj, NULL, (void *)decrementTask, NULL);

        /* Allow the tasks to run. */
        pthread_join(incrementTaskObj, NULL);
        pthread_join(decrementTaskObj, NULL);

        return 0;
    }
```

The task incrementTask, shown following, includes an infinite loop that starts by delaying for three seconds by calling sleep, which suspends the task for a specified number of seconds.

Once the delay time elapses, the increment task resumes from where it left off. The task then calls pthread_mutex_lock and passes in the sharedVariableMutex in order to take the mutex and access the shared variable. If the mutex is available, it is locked, and the increment task proceeds to increment gSharedVariable. If the mutex is not available, the increment task blocks until it can acquire the mutex.

After incrementing the shared variable and outputting a message, the mutex is released with a call to pthread_mutex_unlock. The mutex unlock function never blocks.

```c
    #include <stdio.h>
    #include <unistd.h>

    /***********************************************************************
     *
     * Function:    incrementTask
     *
     * Description: This task increments a shared variable.
     *
     * Notes:
     *
     * Returns:     None.
     *
     ***********************************************************************/
    void incrementTask(void *param)
    {
        while (1)
        {
            /* Delay for 3 seconds. */
            sleep(3);

            /* Wait for the mutex before accessing the GPIO registers. */
            pthread_mutex_lock(&sharedVariableMutex);

            gSharedVariable++;

            printf("Increment Task: shared variable value is %d\n",
                    gSharedVariable);
```

```
            /* Release the mutex for other task to use. */
            pthread_mutex_unlock(&sharedVariableMutex);
        }
    }
```

The task decrementTask is similar to the increment task. In its infinite loop, the task first suspends for seven seconds, then waits to acquire the sharedVariableMutex. After taking the mutex, the task decrements the value of gSharedVariable, outputs a message, and then releases the mutex, as shown here:

```
/************************************************************************
 *
 * Function:    decrementTask
 *
 * Description: This task decrements a shared variable.
 *
 * Notes:
 *
 * Returns:     None.
 *
 ************************************************************************/
void decrementTask(void *param)
{
    while (1)
    {
        /* Delay for 7 seconds. */
        sleep(7);

        /* Wait for the mutex to become available. */
        pthread_mutex_lock(&sharedVariableMutex);

        gSharedVariable--;

        printf("Decrement Task: shared variable value is %d\n",
               gSharedVariable);

        /* Release the mutex. */
        pthread_mutex_unlock(&sharedVariableMutex);
    }
}
```

The Linux pthread API supports additional mutex functions that provide other functionality. For example, the function pthread_mutex_trylock can be used to attempt to get a mutex. If the mutex is available, the task acquires the mutex; if the mutex is not available, the task can proceed with other work without waiting for the mutex to be freed up.

Semaphore Task Synchronization

The Linux semaphore example is similar to a push button light switch, using an LED for the light, as was the case in the eCos semaphore example. There are two tasks in this example: a producer and a consumer. The producer task, producerTask,

monitors the button labeled SW0 on the Arcom board's add-on module. Figure 11-1 in Chapter 11 shows the button used in this example.

When the SW0 button is pressed, the producer task signals the consumer task using a semaphore. The consumer, consumerTask, waits for the semaphore signal from the producer task; once received, the consumer task outputs a message and toggles the green LED.

The main function first initializes the LED by calling ledInit. Next, the semaphore is initialized with a call to sem_init. The initial value of the semaphore, semButton, is set to zero by the last parameter so that the consumer task that is waiting does not execute until the semaphore is signaled by the producer task. The second parameter notifies the operating system that this semaphore may be used by this process only. Lastly, the two tasks are created and a message is output signifying the start of the program.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include "led.h"

sem_t semButton;

/***********************************************************************
 *
 * Function:    main
 *
 * Description: Main routine for the Linux semaphore example. This
 *              function creates the semaphore and then the increment
 *              and decrement tasks.
 *
 * Notes:
 *
 * Returns:     0.
 *
 ***********************************************************************/
int main(void)
{
    /* Configure the green LED control pin. */
    ledInit();

    /* Create the semaphore for this process only and with an initial
     * value of zero. */
    sem_init(&semButton, 0, 0);

    /* Create the producer and consumer tasks using the default task
     * attributes. Do not pass in any parameters to the tasks. */
    pthread_create(&producerTaskObj, NULL, (void *)producerTask, NULL);
    pthread_create(&consumerTaskObj, NULL, (void *)consumerTask, NULL);

    printf("Linux semaphore example - press button SW0.\n");
```

```
        /* Allow the tasks to run. */
        pthread_join(producerTaskObj, NULL);
        pthread_join(consumerTaskObj, NULL);

        return 0;
    }
```

The producerTask function shown next contains an infinite loop that first delays for 10 ms and then checks to see whether the SW0 button has been pressed, by calling the function buttonDebounce. For additional information about selecting sampling intervals and button debouncing, read the sidebar "Switch Debouncing" in Chapter 11.

When the SW0 button is pressed, the producerTask function signals the semButton semaphore by calling sem_post. This increments the semaphore value and wakes the consumer task. The producer task then returns to monitoring the SW0 button.

```
    #include <unistd.h>
    #include "button.h"

    /*********************************************************************
     *
     * Function:    producerTask
     *
     * Description: This task monitors button SW0. Once pressed, the button
     *              is debounced and the semaphore is signaled, waking the
     *              waiting consumer task.
     *
     * Notes:
     *
     * Returns:     None.
     *
     *********************************************************************/
    void producerTask(void *param)
    {
        int buttonOn;

        while (1)
        {
            /* Delay for 10 milliseconds. */
            usleep(10000);

            /* Check whether the SW0 button has been pressed. */
            buttonOn = buttonDebounce();

            /* If button SW0 was pressed, signal the consumer task. */
            if (buttonOn)
                sem_post(&semButton);
        }
    }
```

The following consumerTask function contains an infinite loop that waits for the semaphore to be signaled by calling sem_wait. The wait function blocks the task if the value of the semaphore is 0.

Once the semaphore signal is received, the consumer task outputs a message and toggles the green LED by calling ledToggle. After toggling the LED, the consumer task returns to waiting for another semaphore signal.

```
/*********************************************************************
 *
 * Function:    consumerTask
 *
 * Description: This task waits for the semaphore signal from the
 *              producer task. Once the signal is received, the task
 *              outputs a message and toggles the green LED.
 *
 * Notes:
 *
 * Returns:     None.
 *
 *********************************************************************/
void consumerTask(void *param)
{
    while (1)
    {
        /* Wait for the signal. */
        sem_wait(&semButton);

        printf("Button SW0 was pressed.\n");

        ledToggle();
    }
}
```

Message Passing

The Linux message passing example is a light switch, as is the eCos example, with a producer and a consumer task. The producer task monitors button SW0. Once the button is pressed, the producer sends the button press count to the consumer task. The consumer task waits for this message, outputs the button count, and then toggles the green LED. This example uses a POSIX message queue for passing the message from the producer to the consumer task.

> The source code for the message queue example is not included on the book's web site because it does not work on the Arcom board as shipped. In order to get the message queue code running on the Arcom board, the Linux kernel needs to be rebuilt with message queue support included.

The main routine demonstrates how to create the message queue. First, the LED is initialized by calling ledInit. Then the message queue is created by calling mq_open. The first parameter specifies the name of the queue as message queue. The second

parameter, which is the OR of the file status flags and access modes, specifies the following:

O_CREAT

Create the message queue if it does not exist.

O_EXCL

Used with O_CREAT to create and open a message queue if a queue of the same name does not already exist. If a queue does exist with the same name, the message queue is not opened.

O_RDWR

Open for read and write access.

After the message queue is created successfully, the tasks are created as shown previously. The techniques we've just discussed are shown in the following main function:

```c
#include <pthread.h>
#include <mqueue.h>
#include "led.h"

int8_t messageQueuePath[] = "message queue";

/***********************************************************************
 *
 * Function:    main
 *
 * Description: Main routine for the Linux message queue program. This
 *              function creates the message queue and the producer
 *              and consumer tasks.
 *
 * Notes:
 *
 * Returns:     0.
 *
 ***********************************************************************/
int main(void)
{
    mqd_t messageQueueDescr;

    /* Configure the green LED control pin. */
    ledInit();

    /* Create the message queue for sending information between tasks. */
    messageQueueDescr = mq_open(messageQueuePath, (O_CREAT | O_EXCL | O_RDWR));

    /* Create the producer task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&producerTaskObj, NULL, (void *)producerTask, NULL);

    /* Create the consumer task using the default task attributes. Do not
     * pass in any parameters to the task. */
    pthread_create(&consumerTaskObj, NULL, (void *)consumerTask, NULL);
```

```
        /* Allow the tasks to run. */
        pthread_join(producerTaskObj, NULL);
        pthread_join(consumerTaskObj, NULL);

        return 0;
    }
```

Prior to entering its infinite loop, the producerTask starts by initializing the variable that keeps track of the number of button presses, buttonPressCount, to zero. Then the producer task opens the message queue, whose name is specified by messageQueuePath, which was created in the main function. The queue is opened with write-only permission, specified by the second parameter flag O_WRONLY, because the producer sends only messages. Because the flag O_NONBLOCK is not specified in the second parameter to mq_open, if a message cannot be inserted into the queue, the producer task blocks. The function mq_open returns a message queue descriptor that is used in subsequent accesses to the message queue.

In the infinite loop, the producer task first delays for 10 ms by calling usleep. The delay interval selected ensures the task is responsive to button presses. Next, the function buttonDebounce is called to determine if the SW0 button has been pressed.

Each time the SW0 button is pressed, buttonPressCount is incremented and the value is sent to the waiting consumer task, using the message queue. To accommodate the message queue send function, the union msgbuf_t is used to contain the message. This union consists of a 32-bit count and a 4-byte buffer array. The message is sent using the function mq_send, with the message queue descriptor, messageQueueDescr, in the first parameter, the button press count in the second parameter, the size of the message in the third parameter, and the priority of the message in the last parameter.

Unlike the eCos message queue implementation, messages under Linux have a priority (which is specified in the last parameter passed to mq_send). You can use this parameter to insert higher-priority messages at the front of the message queue, to be read by the receiving task.

After the message is successfully sent, the loop returns to calling the delay function. Here is the producerTask function:

```
    #include <unistd.h>
    #include "button.h"

    typedef union
    {
        uint32_t count;
        uint8_t  buf[4];
    } msgbuf_t;

    /***********************************************************************
     *
     * Function:    producerTask
     *
```

```
    * Description: This task monitors button SW0. Once pressed, the button
    *              is debounced and a message is sent to the waiting
    *              consumer task.
    *
    * Notes:
    *
    * Returns:      None.
    *
    ***********************************************************************/
    void producerTask(void *param)
    {
        uint32_t buttonPressCount = 0;
        mqd_t messageQueueDescr;
        uint8_t button;
        msgbuf_t msg;

        /* Open the existing message queue using the write-only flag because
         * this task only sends messages. Set the queue to block if
         * a send cannot take place immediately. */
        messageQueueDescr = mq_open(messageQueuePath, O_WRONLY);

        while (1)
        {
            /* Delay for 10 milliseconds. */
            usleep(10000);

            /* Check whether the SW0 button has been pressed. */
            button = buttonDebounce( );

            /* If button SW0 was pressed, send a message to the consumer task. */
            if (button & BUTTON_SW0)
            {
                buttonPressCount++;
                msg.count = buttonPressCount;

                mq_send(messageQueueDescr, &msg.buf[0], sizeof(buttonPressCount), 0);
            }
        }
    }
```

The task consumerTask in the following function begins like the producer task by opening the message queue with a call to mq_open. But the queue is opened with read-only permission by the second parameter flag, O_RDONLY, because the consumer task receives only messages. Since the flag O_NONBLOCK isn't specified in the second parameter to mq_open, if no message is available in the queue when the consumer calls the message queue receive function, it blocks until a message is present. The function mq_open returns a message queue descriptor used in subsequent accesses to the message queue.

The consumer task then enters an infinite loop where it waits for a message by calling mq_receive. Once a message is available, it is copied* (by Linux) into the rcvMsg

---

* There may be an implementation of the message queue routine that does not use a copy.

variable. The consumerTask then outputs the message and toggles the green LED. The task then returns to waiting for the next message.

```c
#include <stdio.h>

/************************************************************************
 *
 * Function:    consumerTask
 *
 * Description: This task waits for a message from the producer task.
 *              Once the message is received via the message queue,
 *              the task outputs a message.
 *
 * Notes:
 *
 * Returns:     None.
 *
 ************************************************************************/
void consumerTask(void *param)
{
    mqd_t messageQueueDescr;
    msgbuf_t rcvMsg;

    /* Open the existing message queue using the read-only flag because
     * this task only receives messages. Set the queue to block if
     * a message is not available. */
    messageQueueDescr = mq_open(messageQueuePath, O_RDONLY);

    while (1)
    {
        /* Wait for a new message. */
        mq_receive(messageQueueDescr, &rcvMsg.buf[0], 4, NULL);

        printf("Button SW0 pressed %d times.\n", rcvMsg.count);

        ledToggle( );
    }
}
```

Linux includes numerous other API functions that offer additional functionality for the mechanisms covered previously, as well as other types of synchronization mechanisms. Other mechanisms include condition variables and reader-writer locks. Condition variables allow multiple tasks to wait until a specific event occurs or until the variable reaches a specific value. Reader-writer locks allow multiple tasks to read data concurrently, whereas any task writing data has exclusive access.

Interrupt handling in Linux is more complex than that found in RTOSes. For this reason, we have omitted an interrupt example using Linux. For a better understanding of Linux interrupt handling, take a look at *Linux Device Drivers*, by Alessandro Rubini and Jonathan Corbet (O'Reilly).

Extending Functionality

Kramer: *It's just a write-off for them.*
Jerry: *How is it a write-off?*
Kramer: *They just write it off.*
Jerry: *Write it off what?*
Kramer: *Jerry, all these big companies, they write-off everything.*
Jerry: *You don't even know what a write-off is.*
Kramer: *Do you?*
Jerry: *No. I don't.*
Kramer: *But they do and they are the ones writing it off.*
—the television series "Seinfeld," episode "The Package"

In this chapter, we introduce additional hardware and software technologies that you may encounter in embedded systems. We begin with a look at a pair of chip interconnection buses called I²C and SPI. Next we introduce programmable logic, including FPGAs. And finally, we take a look at adding a TCP/IP network.

Common Peripherals

As you work on more and more embedded systems, you will come across different peripherals that you will have to use. In this section, we take a look at some of the common embedded peripherals that you will likely encounter. Sometimes it is necessary to implement these protocols entirely in software (see the sidebar "Serial Bit Banging" later in this chapter).

Serial buses can be either *asynchronous* or *synchronous*. In an asynchronous serial connection, the data is sent without using a common timing clock signal. To align the receiver with the sender, there is some sort of start condition to signify when the transmission begins, and a stop condition to indicate the end of transmission. A synchronous serial connection typically uses a separate clock signal to synchronize the receiver with the sender. Synchronous connections may also use a start and stop condition to synchronize the receiver and sender, after which the sender must send characters one right after the other.

A serial interface that can send and receive data at the same time is called *full-duplex*. A serial interface that must alternate between sending and receiving data is called *half-duplex*.

Inter-Integrated Circuit Bus

One of the common serial buses used in embedded systems is the *Inter-Integrated Circuit bus*, commonly referred to as the I^2C (pronounced "eye squared see") bus. This bus is common in embedded systems because it doesn't require much in the way of hardware resources and is ideal for low-speed, short-distance communications. The I^2C bus, created by Philips, is a two-wire (data and clock) communication system. I^2C includes the addressing of individual devices (up to 127 in Standard-mode, 1024 in extended mode) to allow multiple devices on the same bus.

Various maximum data rates are supported by different revisions to the I^2C specification. These data rates are up to 100 Kbps for Standard-mode, 400 Kbps for Fast-mode, and 3.4 Mbps for High-speed mode. Devices of different data rates can be mixed on the same bus. The I^2C specification can be found online at *http://www. semiconductors.philips.com*.

The PXA255 processor includes an I^2C bus interface unit. Additional information about this can be found in the *PXA255 Processor Developer's Manual*.

Some devices that typically contain I^2C bus interfaces are EEPROMs and real-time clock chips. Figure 13-1 shows an example I^2C bus with multiple devices.
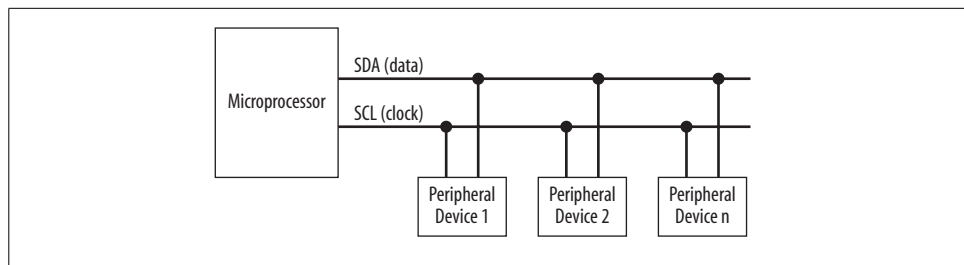
Figure 13-1. Example I^2C bus structure

The two I^2C bus signals are serial data (SDA) and serial clock (SCL). The master on the bus initiates all transfers; other devices on the bus are called slaves. In Figure 13-1, the microprocessor is the master and the other devices are the slaves. Both master and slaves can receive and transmit data on the bus.

The master initiates transactions on the bus and controls the clock signal. Because of this, a slave device needs a way of holding off the master during a transaction. When a slave holds off the master device to perform flow control on the incoming data, it is called *clock stretching*. During this time the slave keeps the clock line pulled low until it is ready to continue with the transaction. It is important that all master devices

support this feature. Figure 13-2 is the format of an I²C bus transaction. All data on the bus is communicated most significant bit first (MSB).

MSB | ADDRESS ADDRESS ADDRESS ADDRESS ADDRESS ADDRESS | LSB R/W | ACK | MSB DATA DATA DATA DATA DATA DATA DATA | LSB DATA | ACK | MSB DATA DATA DATA DATA DATA DATA DATA | LSB DATA | ACK | STOP

START | ADDRESS ADDRESS ADDRESS ADDRESS ADDRESS ADDRESS ADDRESS | R/W | ACK | DATA DATA DATA DATA DATA DATA DATA DATA | ACK | DATA DATA DATA DATA DATA DATA DATA DATA | ACK | STOP

Figure 13-2. Format of a transaction on an I²C bus

An I²C bus data transaction begins by the master initiating a start condition. A start condition occurs when the master causes a high-to-low transition on the data line while the clock line is held high.

Next, the 7-bit unique address of the device is sent out by the master device. Each device on the bus checks this address with its own to determine whether the master is communicating with it. I²C slave devices come with a predefined device address. The lower bits of this address are sometimes configurable in hardware.

Then the master outputs the read or write bit. If the bit is high, the transaction is a read, where data goes from the slave to the master device. If the bit is low, the transaction is a write from the master to the slave device.

The slave device then sends an acknowledge bit. For the acknowledge bit, the data line is kept low while the clock signal is high. Acknowledge bits always are sent by the slave device.

Now, depending on the type of transaction (read or write), the transmitter (which can be slave or master) begins sending a byte of data, starting with the MSB. At the end of the data byte, the receiver (either slave or master) issues an acknowledge bit. This pattern is continued until all of the data has been transferred.

The transaction ends with the master device causing a *stop condition*. A stop condition occurs when the master causes a low-to-high transition on the data line while holding the clock line high. Note that the I²C protocol supports multiple masters.

Serial Peripheral Interface

Another serial bus that is commonly used in embedded systems is the Motorola *serial peripheral interface* (SPI, pronounced "spy"). Another similar serial interface is *Microwire*, trademarked by National Semiconductor, which is a restricted subset of SPI.

SPI can operate at data rates up to 1 Mbps. It can additionally operate in full-duplex mode, making it better suited than I²C for applications where data is constantly flowing. I²C uses fewer signals than SPI, can communicate over several feet (a meter or

Serial Bit Banging

Some embedded systems don't have hardware dedicated to performing all of the interface functions of a serial interface. In this case, general-purpose I/O signals are connected to external devices, and it is up to the software to implement the communication protocol. *Bit banging* is a slang term for the process of transferring serial data under software control.

Bit banging can be used for any serial interface, including I²C, SPI, and even UARTs. When implementing a serial interface via bit banging, the software controls all of the signals to operate the interface. For example, the following function, serialSendData, demonstrates sending a byte of data, dataByte, on a serial interface.

The function starts off with the bitMask set to the most significant bit. The if statement determines whether the bit is high or low and sets the GPIO data signal accordingly. For each bit that is transmitted on the serial interface, the GPIO clock signal is toggled. The while loop continues until all eight bits of the data byte have been sent.

```
void serialSendData(uint8_t dataByte)
{
    uint8_t bitMask;

    /* Loop through each bit in the byte of data. */
    for (bitMask = 0x80; bitMask != 0x00; bitMask >>= 1)
    {
        /* See if the next data bit is high or low
         * and set the GPIO data line accordingly. */
        if (dataByte & bitMask)
            gpioDataSignal = 1;
        else
            gpioDataSignal = 0;

        /* Toggle the clock GPIO line. */
        gpioClkSignal = 1;

        /* Delay for the proper amount of time. */

        gpioClkSignal = 0;
    }
}
```

more), and has a well defined specification. SPI, on the other hand, has a limited communication length of a few inches. SPI does not support multiple masters or specify a device addressing scheme; therefore, additional hardware signals are needed in order to select specific slaves. This lack of addressing can be a benefit because it reduces the overhead in single-master, single-slave SPI interfaces.

Figure 13-3 shows an example of an SPI bus structure. In this figure, there is a single master and two slave devices connected to the SPI bus.
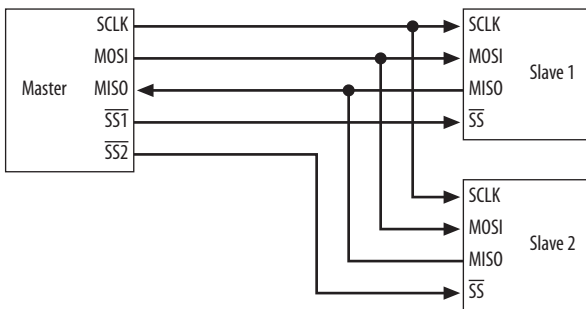
Figure 13-3. Example SPI bus structure

The SPI bus includes 3 + N signals, where N is the number of slaves on the bus. In Figure 13-3, there are two slaves, so the SPI bus requires five signals. These signals are serial clock (SCLK), data signal Master Out Slave In (MOSI), data signal Master In Slave Out (MISO), and Slave Select (SS1 and SS2). The slave select signal is used to select which slave the master wants to communicate with. In this example, because there are two slave devices, two slave select signals are needed. This shows how additional hardware resources are needed in the SPI interface to accommodate the lack of addressing in the protocol.

SPI operates in full-duplex mode. During communications, the master device initiates a transaction by generating a clock and selecting a device using the slave select signal. Data is then transferred in both directions on the MOSI and MISO lines. Because data is transferred in both directions, it is up to each device to know whether the incoming data is meaningful—that is, whether the transaction was a read, write, or both.

Another difference between SPI and I²C is that SPI does not include any type of acknowledgment mechanism. The transmitter has no way of knowing data has been received at the destination. There is also no mechanism for flow control included in SPI. If flow control is needed, it must be implemented outside of SPI.

Programmable Logic

Programmable logic chips are widely used in embedded systems. These devices allow hardware engineers to perform various tasks (such as chip select logic) in hardware. As a programmer, you might not design the logic within the programmable device, but you may need to write a driver to download the program into an FPGA.

This section will give you a better basis for communicating with hardware designers to determine which functions should be implemented in dedicated logic, programmable logic, and/or software. We've found that there are valid reasons for choosing each of these three implementation techniques. You must pay close attention to the requirements of the particular application to make the correct decision.

Many types of programmable logic are available. The current range of offerings includes everything from small devices capable of implementing only a handful of logic equations to huge devices that can hold an entire processor core (plus peripherals!). In addition to this incredible difference in size, architectures also vary greatly. We'll introduce you to the most common types of programmable logic and highlight the most important features of each type.

Programmable Logic Device

At the low end of the spectrum is the original *Programmable Logic Device* (PLD). PLDs were the first chips that could be used to implement a flexible digital logic design in hardware. In the early days, you could remove a couple of the 7400-series Transistor-Transistor-Logic (TTL) parts (ANDs, ORs, and NOTs) from your board and replace them with a single PLD. Other names you might encounter for this class of device are *Programmable Logic Array* (PLA), *Programmable Array Logic* (PAL), and *Generic Array Logic* (GAL).

PLDs are often used for address decoding, where they have several clear advantages over the 7400-series TTL parts that they replaced. Firstly, one chip typically requires less board area and wiring. Another advantage is that the design inside the chip is flexible, so a change in the logic doesn't require any rewiring of the board. Rather, the decoding logic can be altered by simply replacing the single, previously installed PLD with another part that has been programmed with the new Boolean logic.

Inside each PLD is a set of connected *macrocells*. These macrocells are typically comprised of some amount of combinatorial logic (AND and OR gates, for example) and a flip-flop. In other words, a small Boolean logic equation can be built within each macrocell. This equation combines the state of some number of binary inputs into a binary output and, if necessary, stores that output in the flip-flop until the next clock edge. Of course, the particulars of the available logic gates and flip-flops are specific to each manufacturer and product family. But the general idea is always the same.

Hardware designs for these simple PLDs are generally written in languages such as ABEL or PALASM (the hardware equivalents of assembly language) or drawn with the help of a schematic capture tool.

Complex Programmable Logic Device

As chip densities increased, it was natural for the PLD manufacturers to evolve their products into larger parts (logically, but not necessarily physically) called *Complex Programmable Logic Devices* (CPLDs). For most practical purposes, CPLDs can be thought of as multiple PLDs (plus some programmable interconnect) in a single chip. The larger capacity of a CPLD allows you to implement either more logic equations or a more complicated design. In fact, these chips are large enough to replace dozens of those pesky 7400-series parts.

Figure 13-4 contains a block diagram of a hypothetical CPLD. Each of the four logic blocks shown is equivalent to one PLD. However, in an actual CPLD there may be more (or fewer) than four logic blocks. Figure 13-6 is a simplified version. The switch matrix allows signal routing and communication between the logic blocks. Note also that these logic blocks are themselves comprised of macrocells and interconnect wiring, just like an ordinary PLD.
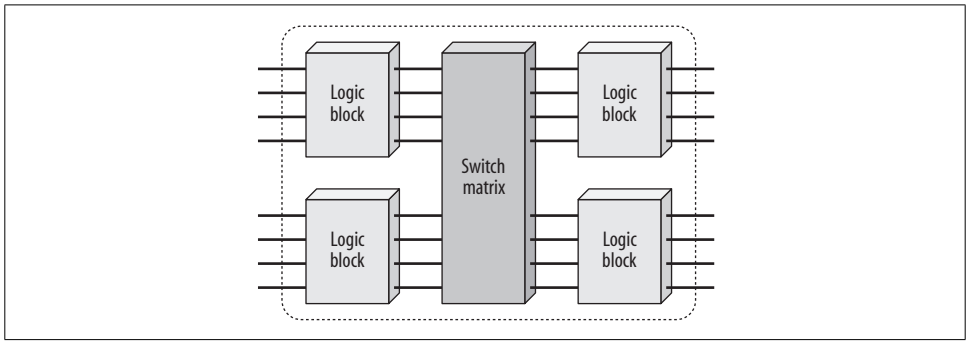
Figure 13-4. CPLD internal structure

Because CPLDs can hold larger designs than PLDs, their potential uses are more varied. They are still sometimes used for simple applications such as address decoding, but more often contain high-performance control logic or finite state machines. At the high end (in terms of numbers of gates), there is also a lot of overlap in potential applications with FPGAs. Because of its less flexible internal architecture, the delay through a CPLD (measured in nanoseconds) is more predictable and usually shorter.

Field Programmable Gate Array

A *Field Programmable Gate Array* (FPGA) can be used to implement just about any hardware design. One use is to prototype a lump of hardware that will eventually find its way into an ASIC. However, there is nothing to say that the FPGA can't remain in the final product, and it quite often does. Whether it does will depend on the relative weights of the development cost and production cost for a particular project, as well as the need to upgrade the hardware design after the product ships. (It costs significantly more to develop an ASIC, but the cost per chip will be lower if you produce them in sufficient quantities. The cost tradeoff involves the expected number of chips to be produced and the expected likelihood of hardware bugs and/ or changes. This makes for a rather complicated cost analysis.)

The historical development of the technology in an FPGA was distinct from the PLD/ CPLD evolution just described. This is apparent when you look at the structures inside. Figure 13-5 illustrates a typical FPGA architecture. There are three key parts to its structure: logic blocks, interconnect, and I/O blocks. The I/O blocks form a ring around the outer edge of the part. Each of these provides individually selectable

input, output, or bi-directional access to one of the GPIO pins on the exterior of the FPGA package. Inside the ring of I/O blocks lies a rectangular array of logic blocks. And finally, connecting logic blocks to logic blocks and I/O blocks to logic blocks, the programmable interconnect wiring runs through the array.
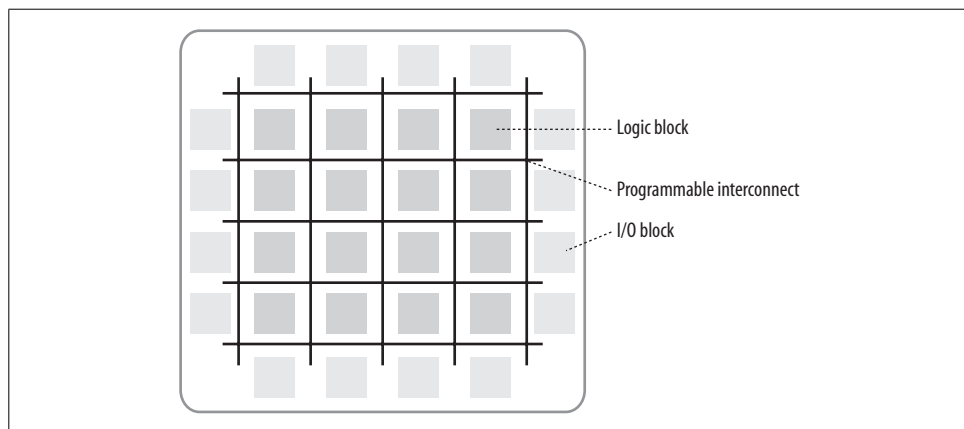
Figure 13-5. FPGA internal structure

The logic blocks within an FPGA can be as small and simple as the macrocells in a PLD (a so-called fine-grained architecture) or larger and more complex (a coarse-grained architecture). However, the logic blocks in an FPGA are never as large as an entire PLD, as are the logic blocks of a CPLD. Remember that the logic blocks of a CPLD contain multiple macrocells. But the logic blocks in an FPGA are generally nothing more than a couple of logic gates or a look-up table and a flip-flop.

Because of all the extra flip-flops, the architecture of an FPGA is much more flexible than that of a CPLD. This makes FPGAs better in register-heavy and pipelined applications. They are also often used in place of a processor-plus-software solution, particularly where the processing of input data streams must be performed at a very fast pace. In addition, FPGAs are usually denser (more gates in a given area) than their CPLD cousins, so they are the de facto choice for larger logic designs.

Pulse Width Modulation

Pulse width modulation (PWM) is a powerful technique for controlling analog circuits with a processor's digital outputs. PWM is employed in a wide variety of applications, ranging from measurement and communications to power control and conversion.

Analog circuits

An analog signal has a continuously varying value, with effectively infinite resolution in both time and magnitude. A 9 V battery is an example of an analog device, in that

its output voltage is not precisely 9 V, but changes over time and can take any real value from 0.0 V to about 9.5 V. Similarly, the amount of current drawn from a battery is not limited to a finite set of possible values. Analog signals are distinguishable from digital signals because the latter always take values only from a finite set of predetermined possibilities, such as the set of two values (0 V, 5 V).

Analog voltages and currents can be used to control things directly, such as the volume of a car radio. In a simple analog radio, a knob is connected to a variable resistor. As you turn the knob, the resistance goes up or down. As that happens, the current flowing through the resistor increases or decreases. This might directly change the amount of voltage driving the speakers, thus increasing or decreasing the volume.

As intuitive and simple as analog control may seem, it is not always economically attractive or otherwise practical. For one thing, analog circuits tend to drift over time and can, therefore, be very difficult to tune. Precision analog circuits, which solve that problem, can be very large, heavy (just think of old home stereo equipment), and expensive. Analog circuits can also get very hot; the power dissipated is proportional to the voltage across the active elements multiplied by the current through them. Analog circuitry can also be sensitive to noise. Because of its infinite resolution, any perturbation or noise on an analog signal necessarily changes the current value.

Digital control

Controlling analog circuits digitally can drastically reduce system costs and power consumption. What's more, many microcontrollers and DSPs already include on-chip PWM controllers, making implementation easy.

In a nutshell, PWM is a way of digitally encoding analog signal levels. Through the use of high-resolution counters, the *duty cycle* (the percentage of time that a signal is asserted) of a square wave is modulated to encode a specific analog signal level. The PWM signal is still digital because, at any given instant, the full DC supply is either fully on or fully off. The voltage or current source is supplied to the analog load by means of a repeating series of on and off pulses. The *on-time* is the time during which the DC supply is applied to the load, and the *off-time* is the period during which that supply is switched off. Given a sufficiently small period of the PWM signal, any analog value can be encoded with PWM.

To help explain the relation between digital encoding and analog values, we show three different PWM signals in Figure 13-6. Figure 13-6(a) shows a PWM output at a 10 percent duty cycle. That is, the signal is on for 10 percent of the period and off for the other 90 percent. Figures 13-6(b) and 13-6(c) show PWM outputs at 50 percent and 90 percent duty cycles, respectively. These three PWM outputs encode three different analog signal values, at 10 percent, 50 percent, and 90 percent of the full strength. If, for example, the supply is 9 V and the duty cycle is 10 percent, a 0.9 V analog signal results.
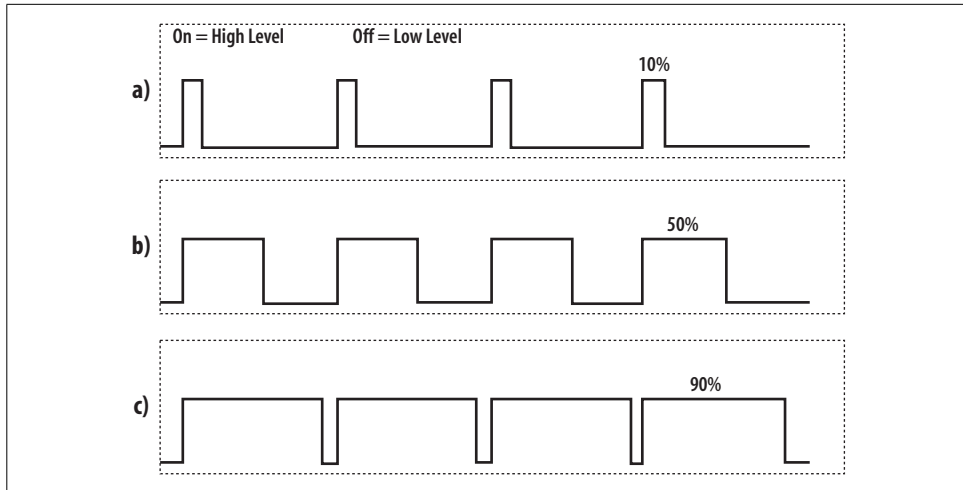
Figure 13-6. PWM signals with varying duty cycles

In Figure 13-7, we show a simple circuit that could be driven using PWM. In this figure, a 9 V battery powers an incandescent lightbulb. If the switch connecting the battery and lamp is closed for 50 ms, the bulb receives the full 9 V during that interval. If we then open the switch for the next 50 ms, the bulb receives 0 V. If we repeat this cycle 10 times a second, the bulb will be lit as though it were connected to a 4.5 V battery (50 percent of 9 V). We say that the duty cycle is 50 percent and the modulating frequency is 10 Hz. (Note that we're not advocating you actually power a lightbulb this way; we just think this an easy-to-understand example.)
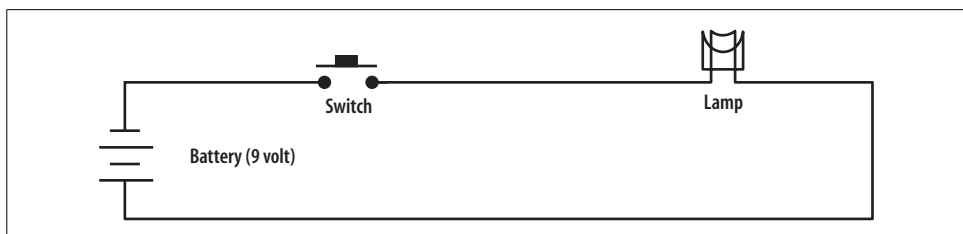
Figure 13-7. A simple PWM circuit

Most loads require a much higher modulating frequency than 10 Hz. Imagine that our lamp was switched on for five seconds, then off for five seconds, then on again. The duty cycle would still be 50 percent, but the bulb would appear brightly lit for the first five seconds and not lit at all for the next. In order for the bulb to see a voltage of 4.5 V, the cycle period must be short relative to the load's response time to a change in the switch state. To achieve the desired effect of a dimmer (but always lit) lamp, it is necessary to increase the modulating frequency. The same is true in other applications of PWM. Common modulating frequencies range from 1 to 200 kHz.

One of the advantages of PWM is that the signal remains digital all the way from the processor to the controlled system; no digital-to-analog conversion is necessary. Keeping the signal digital minimizes noise effects. Noise can affect a digital signal only if the noise is strong enough to change a logical 1 to a logical 0, or vice versa.

This increased noise immunity is another benefit of choosing PWM over analog control and is the principal reason PWM is sometimes used for communications. Switching from an analog signal to PWM can increase the length of a communications channel dramatically. At the receiving end, a suitable resistor-capacitor (RC) or inductor-capacitor (LC) network can remove the modulating high-frequency square wave and return the signal to analog form.

PWM finds application in a variety of systems. As a concrete example, consider a PWM-controlled brake. To put it simply, a brake is a device that clamps down hard on something. In many brakes, the amount of clamping pressure (or stopping power) is controlled with an analog input signal. The more voltage or current that's applied to the brake, the more pressure the brake will exert.

The output of a PWM controller could be connected to a switch between the supply and the brake. To produce more stopping power, the software need only increase the duty cycle of the PWM output. If a specific amount of braking pressure is desired, measurements would need to be taken to determine the mathematical relationship between duty cycle and pressure. (And the resulting formulae or lookup tables would be tweaked for operating temperature, surface wear, and so on.)

To set the pressure on the brake to, say, 100 psi, the software would do a reverse lookup to determine the duty cycle that should produce that amount of force. It would then set the PWM duty cycle to the new value and the brake would respond accordingly. If a sensor is available in the system, the duty cycle can be tweaked, under closed-loop control, until the desired pressure is precisely achieved.

Networking for All Devices Great and Small

Incorporating networking support in an embedded device might seem like a daunting task at first glance. However, even an older embedded system can be updated with a software network stack to extend its feature set and incorporate modern conveniences such as emailing an administrator when alarms occur, and a web server to provide a remote user interface accessible from any web browser.

Certainly, there are costs to including a network stack. The network interface (such as Ethernet) can quickly become expensive and complicated. You may need extra hardware (with additional costs for chips and connectors, board space, and power consumption) and software (with new drivers). However, this does not have to be the case. You could run a simple Serial Line Interface Protocol (SLIP) or Point to Point Protocol (PPP) over a UART port for the network interface. Most embedded processors include at least one UART, and SLIP and PPP are very basic protocols to

implement—no more difficult than the Monitor and Control program we looked at in Chapter 9.

The demands of a network stack may cause you to worry that too many system resources are required. The processing power and memory needed to accommodate network support can be greatly reduced by choosing the proper network stack. Several software network stacks that are targeted at embedded systems, where processor cycles and memory are limited, are currently available in the open source community.

The next section is intended to give you an overview of some of the benefits of adding networking support and includes some options of resource-conscious networking stacks that are ideal for incorporation into embedded systems.

Benefits of Network Support

Adding networking support, Transmission Control Protocol/Internet Protocol (TCP/IP) and the other supporting protocols grouped in this suite enable standardized access to a device. TCP/IP enables a device to communicate using the native protocol of most networking infrastructures, which allows the device to be accessed from a PC, PDA, or web-enabled cellular phone. The network-enabled device can communicate over a Local Area Network (LAN) or through the global Internet.

For additional information about networking protocols, take a look at the three-volume series *TCP/IP Illustrated*, by Richard Stevens (Addison-Wesley). Another resource is *TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*, by Charles Kozierok (No Starch Press).

Each device can be tailored to use the ideal protocol to transmit information over the network. The very low-overhead User Datagram Protocol (UDP) can be used for data not requiring acknowledgment, whereas TCP is available for data that needs confirmation of its receipt from the destination.

Networking support can improve the basic feature set that an embedded device is capable of supporting. For example, if an alarm condition occurs in the system, the device can generate an email and send it off to notify the network operator of the error. The network operator can then quickly perform the necessary maintenance to correct the problem and get the system back to normal—keeping the system downtime to a minimum.

Another benefit of networking is that *web-based management* can be easily incorporated into the device. Web-based management allows configuration and control of a system using TCP/IP protocols and a web browser. A technician can connect directly to a device for configuration and monitoring using a standard PDA equipped with a standard web browser. The device's web server and HyperText Markup Language (HTML) pages are the new user interface for the device. Many devices today, such as cable modem routers and firewalls, include a web interface for configuration.

Figure 13-8 shows the interface to a network of sensor devices presented by a web server that enables web-based management.
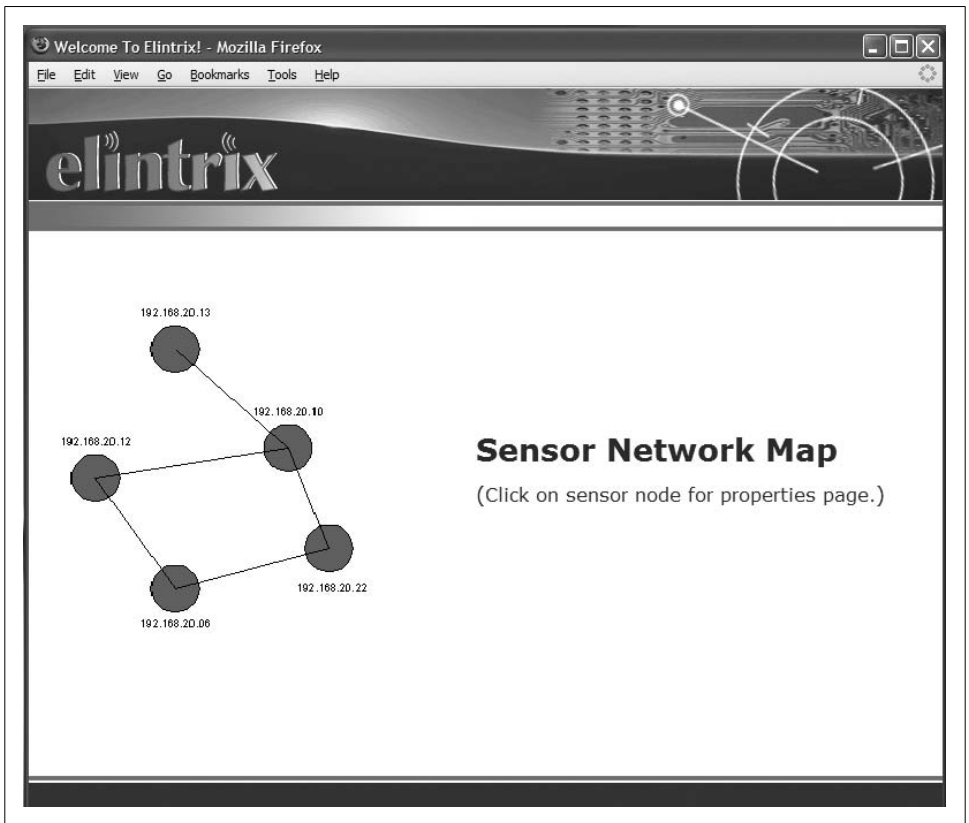
Figure 13-8. Example of web-based management with a network stack

Networking Solutions for Embedded Systems

There are several commercial and open source network stack solutions available today. Most stacks offer standard protocol suites, and some include example applications to help you extend your device's basic feature set. Figure 13-9 shows some of the common network protocol components included with most networking stacks. We'll list even more protocols later in this chapter as we describe particular networking solutions.

One of the keys in deciding which stack will best fit your device is to determine the resource requirements the software needs in order to operate. The amount of data the device transmits and receives during communication sequences should also dictate which solution is right for your design. For example, if your device is a sensor node that wakes up every hour to transmit a few bytes of data, a compact network

Advantages of Web-based Management

Web-based management uses TCP to send packets across the network, which provides a reliable method for transferring data using an acknowledgment and retransmission scheme.

Web-based management relies on a standard browser for the client-side interface. This gives users a standard interface that they are familiar with and comfortable using. A web-enabled device contains a server that simply sends the web pages to the user when the device is accessed. The browser handles the task of rendering the images and presenting the graphical images to the user.

The Simple Network Management Protocol (SNMP) has been the standard for monitoring and controlling networked devices. Integrating a network stack allows web-based management to be utilized. SNMP can be implemented on the smallest embedded systems; however, it has several shortcomings.

One deficiency of SNMP is that it uses UDP for the transmission of packets across a network. UDP is a connectionless, unreliable protocol and has no mechanism for the retransmission of packets that are lost. There can be disastrous consequences if crucial information about the health of the system is lost and the sender has no way of knowing it was not received. SNMP also often requires the use of costly and complex network management software on the client side.
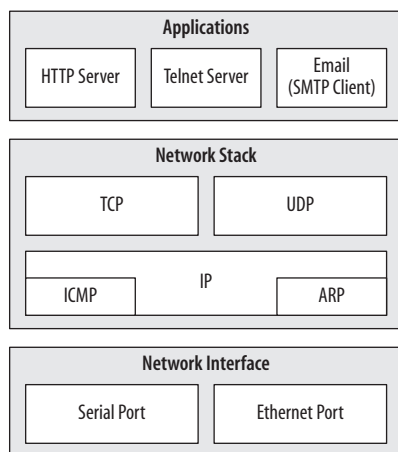
Figure 13-9. Common network protocol components

stack implementation is ideal. In contrast, a video monitoring system that is constantly transmitting large amounts of data might need a stack implementation that offers better packet buffer management.

Implementations focused on small embedded devices allow networking to be integrated into even the most resource-constrained system. It is important that the "lightweight" network stack implementation you choose allows communication with standard, full-scale TCP/IP devices. An implementation that is specialized for a particular device and network might cause problems by limiting your ability to extend the device's network capabilities in other, generic networks.

It is always possible to go off and roll your own network stack. However, given the wide range of solutions available in the open source community, leveraging existing technology is usually the better choice and enables you to quickly move your development forward.

In the following list of software network stacks, we focus on open source solutions that are ideal for resource-constrained embedded devices. This list is not intended to be comprehensive, but rather a starting point for further investigation. All of the networking stacks listed include TCP/IP protocol support. A brief description of each network stack is included; for more detailed information, refer to the specific web pages listed. (BSD networking code is something of an industry standard and therefore appears as the basis of many of the projects listed.)

lwIP (http://savannah.nongnu.org/projects/lwip)
    This "lightweight IP" stack is a simplified but full-scale TCP/IP implementation. lwIP was designed to be run in a multithreaded system with applications executing in concurrent threads, but it can also be implemented on a system with no operating system. In addition to the standard TCP/IP protocol support, lwIP also includes Internet Control Message Protocol (ICMP), Dynamic Host Configuration Protocol (DHCP), Address Resolution Protocol (ARP), and UDP. It supports multiple local network interfaces. lwIP has a flexible configuration that allows it to be easily used in a wide variety of devices and scaled to fit different resource requirements.

OpenTCP (http://www.opentcp.org)
    Tailored to 8- and 16-bit microcontrollers, OpenTCP incorporates the ICMP, DHCP, Bootstrap Protocol (BOOTP), ARP, and UDP. This package also includes several applications, such as a Trivial File Transfer Protocol (TFTP) server, a Post Office Protocol Version 3 (POP3) client to retrieve email, Simple Mail Transfer Protocol (SMTP) support to send email, and a Hypertext Transfer Protocol (HTTP) server for web-based device management.

TinyTCP (http://www.unusualresearch.com/tinytcp/tinytcp.htm)
    This network stack is designed to be very modular and to include only the software required by the system. For example, different protocols can be included based on your configuration. TinyTCP provides a BSD-compatible socket library and includes the ARP, ICMP, UDP, DHCP, BOOTP, and Internet Group Management Protocol (IGMP).

uC/IP (http://ucip.sourceforge.net)
> uC/IP (pronounced mew-kip)[*] is designed for microcontrollers and based on BSD network software. Protocol support includes ICMP and Point-to-Point Protocol (PPP).

uIP (http://www.sics.se/~adam/uip)
> This "micro IP" stack is designed to incorporate only the minimal set of components necessary for a full TCP/IP stack solution. There is support for only a single network interface. Application examples included with uIP are SMTP for sending email, a Telnet server and client, an HTTP server and web client, and Domain Name System (DNS) resolution.

Each network stack has been ported to various processors and microcontrollers. The device driver support for the network interface varies from stack to stack. It is a good idea to review the license for the network stack you decide to use, to make sure it does not place undesirable limitations or requirements on your product.

In addition, some operating systems include or have network stacks ported to them. The operating systems covered earlier in this book, eCos and embedded Linux (see Chapters 11 and 12), both offer networking support modules. eCos includes the OpenBSD, FreeBSD, and lwIP network stacks as well as application-layer support for many of the extended features discussed previously. Embedded Linux, having been developed for a desktop PC environment, offers extensive network support.

If a network stack is included or already exists in your device, several embedded web servers are available to incorporate web-based control. One such open source solution is the GoAhead WebServer (*http://www.goahead.com*).

Embedding a networking stack is no longer a daunting task that requires an enormous amount of resources. The solutions listed previously can quickly be leveraged and integrated to bring networking features to any embedded system. Tailoring one of the network stack solutions to the specific characteristics of a device ensures that the system will operate at its optimal level and best utilize system resources.

---

[*] The *u* at the front of uC/IP and uIP is a crude but common way to represent the Greek letter *mu*.

Optimization Techniques

*Things should be made as simple as possible, but not
any simpler.*

—Albert Einstein

This chapter offers some tips to optimize code to reduce resource utilization. These techniques can be roughly divided into strategies for reducing memory usage, increasing code efficiency, and lowering power requirements. The need for low-cost versions of our products drives hardware designers to provide just barely enough memory and processing power to get the job done.

Most of the optimizations performed on code involve a tradeoff between execution speed and code size. Your program can be made either faster or smaller, but not both. In fact, an improvement in one of these areas can have a negative impact on the other. It is up to the programmer to decide which of these improvements is most important. Given that single piece of information, the compiler's optimization phase can make the appropriate choice whenever a speed versus size tradeoff is encountered.

The first step in optimization is to determine which problems you have. You might have size issues, speed issues, or both. If you have one type of issue, you can have the compiler help you out with the optimization. If you have both size and speed issues, we recommend letting the compiler do what it can to reduce the size of your program. Then you can find the time-critical code or bottlenecks (where the program is spending most of its time) and manually optimize that code for speed. (In battery-powered devices, every unnecessary processor cycle results in reduced runtime; therefore, the thing to do is optimize for speed across the entire application.)

Execution speed is usually important only within certain of those few portions of the code that have short deadlines and those most frequently executed. There are many things you can do to improve the efficiency of those sections by hand. However, code size is a difficult thing to influence manually, and the compiler is in a much better position to make this change across all of your software modules.

Increasing Code Efficiency

By the time your program is working, you might already know, or have a pretty good idea, which functions and modules are the most critical for overall code efficiency. ISRs, high-priority tasks, calculations with real-time deadlines, and functions that are either compute-intensive or frequently called are all likely candidates.

A tool called a *profiler*, included with some software development suites, can be used to narrow your focus to those routines in which the program spends most (or too much) of its time. A profiler collects and reports execution statistics for a program. These execution statistics include the number of calls to and the total time spent in each routine.

Once you've identified the routines that require greater code efficiency, you can use the following techniques to reduce their execution time. Note that the techniques described here are very compiler-dependent. In most cases, there aren't general rules that can be applied in all situations. The best way to determine if a technique will provide improvement is to look at the compiler's assembly output.

Inline functions

> In C99, the keyword `inline` can be added to any function declaration. This keyword asks the compiler to replace all calls to the indicated function with copies of the code that is inside. This eliminates the runtime overhead associated with the function call and is most effective when the function is used frequently but contains only a few lines of code.
>
> Inline functions provide a perfect example of how execution speed and code size are sometimes inversely linked. The repetitive addition of the inline code will increase the size of your program in direct proportion to the number of times the function is called. And, obviously, the larger the function, the more significant the size increase will be. However, you will lose the overhead of setting up the stack frame if parameters are passed into the function. The resulting program runs faster but requires more code memory.

Table lookups

> A `switch` statement is one common programming technique that should be used with care. Each test and jump that makes up the machine language implementation uses up valuable processor time simply deciding what work should be done next. To speed things up, try to put the individual cases in order by their relative frequency of occurrence. In other words, put the most likely cases first and the least likely cases last. This will reduce the average execution time, though it will not improve at all upon the worst-case time.
>
> If there is a lot of work to be done within each case, it might be more efficient to replace the entire `switch` statement with a table of pointers to functions. For example, the following block of code is a candidate for this improvement:
>
> ```
> enum NodeType {NODE_A, NODE_B, NODE_C};
>
> switch (getNodeType())
> ```

```
    {
        case NODE_A:
            .
            .
        case NODE_B:
            .
            .
        case NODE_C:
            .
            .
    }
```

To speed things up, replace this switch statement with the following alternative. The first part of this is the setup: the creation of an array of function pointers. The second part is a one-line replacement for the switch statement that executes more efficiently.

```
int processNodeA(void);
int processNodeB(void);
int processNodeC(void);

/* Establishment of a table of pointers to functions. */
int (* nodeFunctions[])() = {processNodeA, processNodeB, processNodeC};

.
.

/* The entire switch statement is replaced by the next line. */
status = nodeFunctions[getNodeType()]();
```

Hand-coded assembly

Some software modules are best written in assembly language. This gives the programmer an opportunity to make them as efficient as possible. Though most C compilers produce much better machine code than the average programmer, a skilled and experienced assembly programmer might do better work than the compiler for a given function.

For example, on one of our past projects, a digital filtering algorithm was implemented in C and targeted to a TI TMS320C30 DSP. The compiler was unable to take advantage of a special instruction that performed exactly the mathematical operations needed. By manually replacing one for loop of the C program with inline assembly instructions that did the same thing, overall computation time decreased by more than a factor of 10.

Register variables

The keyword register can be used when declaring local variables. This asks the compiler to place the variable into a general-purpose register rather than on the stack. Used judiciously, this technique provides hints to the compiler about the most frequently accessed variables and will somewhat enhance the performance of the function. The more frequently the function is called, the more likely it is that such a change will improve the code's performance. But some compilers ignore the register keyword.

Global variables

It is sometimes more efficient to use a global variable than to pass a parameter to a function. This eliminates the need to push the parameter onto the stack before the function call and pop it back off once the function is completed. In fact, the most efficient implementation of any subroutine would have no parameters at all. However, the decision to use a global variable can also have some negative effects on the program. The software engineering community generally discourages the use of global variables in an effort to promote the goals of modularity and reentrancy, which are also important considerations.

Polling

ISRs are often used to improve a program's responsiveness. However, there are some rare cases in which the overhead associated with the interrupts actually causes inefficiency. These are cases in which the average time between interrupts is of the same order of magnitude as the interrupt latency. In such cases, it might be better to use polling to communicate with the hardware device. But this too can lead to a less modular software design.

Fixed-point arithmetic

Unless your target platform features a floating-point processor, you'll pay a very large penalty for manipulating float data in your program. The compiler-supplied floating-point library contains a set of software subroutines that emulate the floating-point instructions. Many of these functions take a long time to execute relative to their integer counterparts and also might not be reentrant.

If you are using floating-point for only a few calculations, it might be better to implement the calculations themselves using fixed-point arithmetic. For example, two fractional bits representing a value of 0.00, 0.25, 0.50, or 0.75 are easily stored in any integer by merely multiplying the real value by 4 (e.g., << 2). Addition and subtraction can be accomplished via the integer instruction set, as long as both values have the same imaginary binary point. Multiplication and division can be accomplished similarly, if the other number is a whole integer.

It is theoretically possible to perform any floating-point calculation with fixed-point arithmetic. (After all, that's how the floating-point software library does it, right?) Your biggest advantage is that you probably don't need to implement the entire IEEE 754 standard just to perform one or two calculations. If you do need that kind of complete functionality, stick with the compiler's floating-point library and look for other ways to speed up your program.

Variable size

It is typically best to use the processor's native register width for variables whenever possible (whether it is 8, 16, or 32 bits). This allows the compiler to produce code that takes advantage of the fast registers built into the processor's machine opcodes. Obviously, you need to ensure that the variable size accommodates the number range that the variable represents. For example, if you need a count that goes from 0 to 512, you can't use an 8-bit variable.

A variable size tailored to the processor can also speed up processing by limiting the number of external memory accesses. If a processor has a 16-bit data bus and it needs to access a 32-bit variable in external RAM, two data fetches must be performed for the processor to get the variable.

C99 defines integer types `int_fastN_t` and `uint_fastN_t` (where `N` represents the integer length) in *stdint.h*. These types are meant to be used when you need at least "X bits" (e.g., X = 16) to store your data but don't care if the field is larger than X in width, to make access as fast as possible. These "fast" integer types are thus no good for use with peripheral registers, which always have a fixed width that cannot be larger or smaller than X.

Loop unrolling

In some cases, repetitive loop code can be optimized by performing *loop unrolling*. In loop unrolling, the loop overhead at the start and end of a loop is eliminated. Here's an example of a for loop:

```
for (idx = 0; idx < 5; idx++)
{
    value[idx] = incomingData[idx];
}
```

Here's the unrolled version without the loop overhead:

```
value[0] = incomingData[0];
value[1] = incomingData[1];
value[2] = incomingData[2];
value[3] = incomingData[3];
value[4] = incomingData[4];
```

Some compilers offer loop unrolling as an optimization; in other cases, it might be better for the developer to code it. It is helpful to check the assembly output from the compiler to see whether efficiency has actually been improved.

The amount of rolling that you—or the compiler—choose to do must balance the gain in speed versus the increased size of the code. Loop unrolling increases code size—another situation where you must trade code size for speed. Also, loop unrolling can be used only when the number of iterations through the loop are fixed. One example of an optimized implementation of loop unrolling is the coding technique known as Duff's device (*http://en.wikipedia.org/wiki/Duff's_device*).

Decreasing Code Size

As stated earlier, when it comes to reducing code size, your best bet is to let the compiler do the work for you. However, if the resulting program is still too large for your available ROM, there are several programming techniques you can use to further reduce the size of your program.

Once you've got the automatic optimizations working, take a look at these tips for further reducing the size of your code by hand:

Avoid standard library routines

One of the best things you can do to reduce the size of your program is to avoid using large standard library routines. Many of the largest routines are costly in terms of size because they try to handle all possible cases. For example, the strupr function might be small, but a call to it might drag other functions such as strlower, strcmp, strcpy, and others into your program whether they are used or not.

It might be possible to implement a subset of the functionality yourself with significantly less code. For example, the standard C library's sprintf routine is notoriously large. Much of this bulk is located within the floating-point manipulation routines on which it depends. But if you don't need to format and display floating-point values (%a, %e, %f, or %g), you could write your own integer-only version of sprintf and save several kilobytes of code space. In fact, a few implementations of the standard C library (Cygnus's *newlib* comes to mind) include just such a function, called siprintf.

Use goto *statements*

As with global variables, good software engineering practice dictates against the use of this technique. But in a pinch, goto statements can be used to remove complicated control structures or to share a block of oft-repeated code.

For example, many programmers use the goto statement to bail out of a routine in case of error. In this way, the programmer can group together any things that must be done before exiting the routine, as shown here:

```
int functionWork(void)
{
    /* Do some work here. */
    ...

    /* If there was an error doing the work, exit. */
    goto CLEANUP;

    /* Do some more work here. */
    ...

    /* If there was an error doing the work, exit. */
    goto CLEANUP;

    ...

    /* Otherwise, everything succeeded. */
    return SUCCESS;
```

```
CLEANUP:
    /* Clean up code here. */

    return FAILURE;
}
```

In addition to these techniques for reducing code size, several of the ones described in the prior section could be helpful, specifically table lookups, hand-coded assembly, register variables, and global variables. Of these techniques, the use of hand-coded assembly usually yields the largest decrease in code size.

Problems with Optimizing Compilers

The GNU C compiler has several optimization command-line options, all of which are variants of *–O*. Specifying *–O3* turns on all available *gcc* optimizations, regardless of their effects on the speed-versus-size tradeoff. The command-line option *–Os* also optimizes the code for size. For a detailed explanation of the different *gcc* optimization levels, refer to the *gcc* online manual at *http://gcc.gnu.org/onlinedocs*.

Murphy's Law dictates that the first time you enable the compiler's optimization feature, your previously working program will suddenly fail. Perhaps the most notorious of the automatic optimizations is "dead code elimination." This optimization eliminates code that the compiler believes to be either redundant or irrelevant. For example, adding zero to a variable requires no runtime calculation whatsoever. But you might still want the compiler to generate those "irrelevant" instructions if they perform some function that the compiler doesn't know about.

For example, given the following block of code, most optimizing compilers would remove the first statement because the value of *pControl is not used before it is overwritten (on the third line):

```
*pControl = DISABLE;
*pData    = 'a';
*pControl = ENABLE;
```

But what if pControl and pData are actually pointers to memory-mapped device registers? In that case, the peripheral device would not receive the DISABLE command before the byte of data was written. This could potentially wreak havoc on all future interactions between the processor and this peripheral. To protect yourself from such problems, you must declare all pointers to memory-mapped registers and global variables that are shared between tasks (or a task and an ISR) with the keyword volatile. And if you miss just one of them, Murphy's Law will come back to haunt you in the final days of your project—guaranteed.
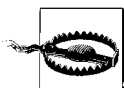
Never make the mistake of assuming that the optimized program will behave the same way as the unoptimized one. You must completely retest your software at each new optimization level to be sure its behavior hasn't changed.

To make matters worse, debugging an optimized program is challenging, to say the least. With the compiler's optimization enabled, the correlation between a line of source code and the set of processor instructions that implements that line is much weaker. Those particular instructions might have moved or been split up, or two similar code blocks might now share a common implementation. In fact, some lines of the high-level language program might have been removed from the program altogether (as they were in the previous example)! As a result, you might be unable to set a breakpoint on a particular line of the program or examine the value of a variable of interest.

Reducing Memory Usage

In some cases, RAM rather than ROM is the limiting factor for your application. In these cases, you'll want to reduce your dependence on global data, the stack, and the heap. These are all optimizations better made by the programmer than by the compiler.

Because ROM is usually cheaper than RAM (on a per-byte basis), one acceptable strategy for reducing the amount of global data might be to move constant data into ROM. This can be done automatically by the compiler if you declare all of your constant data with the keyword const. Most C compilers place all of the constant global data they encounter into a special data segment that is recognizable to the locator as ROM-able. This technique is most valuable if there are lots of strings or table-oriented data that will not change at runtime.

If some of the data is fixed once the program is running but not necessarily constant, the constant data segment could be placed in a hybrid memory device such as flash or EEPROM. This memory device could then be updated over a network or by a technician assigned to make the change. An example of such data is the sales tax rate for each locale in which your product will be deployed. If a tax rate changes, the memory device can be updated, but additional RAM can be saved in the meantime.

Stack size reductions can also lower your program's RAM requirement. One way to figure out approximately how much stack you need is to fill the entire memory area reserved for the stack with a special data pattern, such as 0xAAAA. Then, after the software has been running for a while—under both normal and stressful conditions—use a debugger to examine the modified stack. The part of the stack memory area that still contains your special data pattern has never been overwritten, so you can reduce the size of the stack area by that amount.[*]

Be especially conscious of stack space if you are using a real-time operating system. Preemptive operating systems create a separate stack for each task. These stacks are

---

[*] Of course, you probably want to leave a little extra space on the stack, in case your testing didn't last long enough or did not accurately reflect all possible runtime scenarios. Never forget that a stack overflow is a potentially fatal event for your software and should be avoided at all costs.

used for function calls and ISRs that occur within the context of a task. You can determine the amount of memory required for each task stack in the manner previously described. You might also try to reduce the number of tasks or switch to an operating system that has a distinct "interrupt stack" for execution of all ISRs. The latter method can significantly reduce the stack size requirement of each task.

The size of the heap is limited to the amount of RAM left over after all of the global data and stack space has been allocated. If the heap is too small, your program will not be able to allocate dynamic memory when it is needed, so always be sure to compare the result of `malloc` with `NULL` before dereferencing the memory you tried to allocate. If you've tried all of these suggestions and your program is still requiring too much memory, you might have no choice but to eliminate the heap altogether. This isn't entirely bad in the case of embedded systems, which frequently allocate all memory needed by the system at initialization time.

Note that many embedded programmers avoid the use of `malloc`, and thus the need for a heap, altogether. But the key benefit of dynamic memory allocation is that you don't need to spend RAM to keep variables around that are only used briefly in the program. This is a way to reduce total memory utilization.

Power-Saving Techniques

A major concern in battery-powered embedded systems design is power consumption. In this section, we take a brief look at areas where embedded software can assist in conserving the system's vital energy source.

Power consumption is a major concern for portable or battery-operated devices. Power issues, such as how long the device needs to run and whether the batteries can be recharged, need to be thought out ahead of time. In some systems, replacing a battery in a device can be a big expense. This means the system must be conscious of the amount of power it uses and take appropriate steps to conserve battery life.

There are several methods to conserve power in an embedded system, including clock control, power-sensitive processors, low-voltage ICs, and circuit shutdown. Some of these techniques must be addressed by the hardware designer in his selection of the different system ICs. There may be lower-power versions of certain peripherals. Some power-saving techniques are under software control.

It might seem ideal to select the fastest and most powerful processor available for a particular embedded system. However, one of the tasks of the hardware designer is to use just enough processing power to enable the device to get its job done. This helps reduce the power consumed by the device. The processor selected plays a key role in determining the amount of power an embedded system will consume. In addition, some processors can automatically shut down different execution units when they are not in use.

Processor Modes

One software technique offered by many embedded processors to conserve power is different operating modes. These modes allow the software to scale processor power consumption to match the moment-by-moment needs of the application. For example, the Arcom board's PXA255 processor has four operating modes:

Turbo mode

> The processing core runs at the peak frequency. Minimizing external memory accesses would be worthwhile in this mode, because the processor would have to wait for the external memory.

Run mode

> The processor core runs at its normal frequency. This is the normal or default operating mode.

Idle mode

> The processor core is not clocked, but the other peripheral components operate as normal.

Sleep mode

> This is the lowest power state for the processor.

Understanding the details of these modes and how to get into and out of them is key. For example, the PXA255 can conserve power by entering and exiting idle mode multiple times in a second, because the processor is quickly reactivated in the prior state. However, in sleep mode, the processor state is not maintained and may require a complete system reboot when exiting this mode.

Operating the processor in different modes can save quite a bit of power. The power consumption for the PXA255 processor (running at 200 MHz) in normal run mode is typically 178 mW. While in idle mode, the PXA255 typically consumes 63 mW.

There are several issues to consider when planning the power management software design. First, you must ensure that each task is able to get enough cycles to perform its assigned work. If a system doubles battery life by entering idle modes often but is thus unable to perform its work, the product fails to meet its design goals.

You also need to determine when the system is not doing anything and how to wake up the processor when it needs to operate, and you need to know what events will wake up the system. For example, in an embedded system that sends some data across a network every few minutes, it makes sense to be able to shut down the device to conserve power until it is time to send the data. The device must still be able to wake up in case an error condition arises. Therefore, you must understand how a peripheral circuit wakes up the processor when the processor needs to operate (including how long it takes the circuit to wake up and whether any reinitialization needs to be done).

The optimization techniques presented earlier in this chapter can be used to conserve power as well. By reducing the amount of execution time for the main tasks in a system, you allow the system to spend more time in its low-power state.

Even though the processor is in idle mode, various peripherals still operate and can be programmed to wake up the processor. Typically, interrupts can be used to wake up the processor to perform some task. This is why power management must be considered when designing the software. For example, some behaviors can be achieved by polling. But when the events are less frequent and power management is an issue, it makes sense to use interrupts rather than polling because this allows the processor to sleep for the maximum amount of time before waking up to handle an event. When you choose to use polling, the processor must constantly perform the polling operation, which typically happens at a set interval. This wastes power when the polling operation executes and no events have occurred.

You can also take advantage of peripherals that operate while the processor is in idle mode. For example, if you are transferring data from an external peripheral into RAM and need to process the data once a certain amount of data is received, you can use the DMA controller. This way, instead of the processor handling each byte received, it sleeps while the data is transferred. You can configure the DMA controller to interrupt the processor once this data has been received.

Clock Frequency

Another power-saving technique that can be controlled by software is to vary processor clock speeds. Some processors accept a fixed-input clock frequency but feature the ability to reduce internal clock speeds by programming clock configuration registers. Software can reduce the clock speed to save power during the execution of noncritical tasks and increase the clock speed when processing demands are high.

The PXA255 datasheet shows the power consumption while the processor core operates at different frequencies. Table 14-1 shows a comparison for three different PXA255 core clock frequencies and the associated power consumption at each frequency.

Table 14-1. PXA255 power consumption comparison

Processor core clock speed (MHz)	Power consumption (mW)
400	411
300	283
200	178

As the software designer, you need to understand what happens during the frequency change sequence—what to do if an interrupt occurs during the frequency change and what needs to be reconfigured (such as DRAM refresh cycles) for the new frequency.

You will need comprehensive knowledge of all software operation in the system if you decide to alter the processor frequency on the fly. For example, it can be tricky to know when to lower the clock speed when a multitasking RTOS is used.

In other cases, particular peripherals can be completely disabled when they are not in use. For example, if a particular peripheral module is not used in the PXA255 processor, the clock to that unit can be disabled using the Clock Enable Register (CKEN).

External Memory Access

There are several things that can be done to reduce external memory accesses. If a *cache* is available, you can enable it to avoid having the processor fetch data or instructions from external memory. A cache is very high-speed, on-chip memory that supplies the most recently used instructions and/or data to the processor with no or few wait states.

Similarly, internal processor memory can be used, if available. In some cases, the internal memory can be used for both data and code. It might not be feasible to incorporate all the system software in internal memory. If there is not enough memory available for the entire system software, you must determine what data and code should be included. It's best to include the stack and frequently used variables or functions for limiting external memory accesses. In an embedded system where power consumption is the top priority, it might make sense to switch to a processor with more on-chip memory in order to reduce off-chip accesses.

Optimization techniques can help here as well. For power optimization, instead of focusing solely on speed or code size, you need to focus on analyzing code to determine how to reduce external bus transactions.

These are just a few considerations to keep in mind when working on power management software. Future products may include technology with self-renewing sources of energy. There are ways to harvest energy for circuits that are self-powered using sources such as vibration, light, and thermal sources. These techniques are discussed in the June 2005 *Embedded Systems Design* article "Energy-Harvesting Chips: The Quest for Everlasting Life," which can be found online at *http://www.embedded.com*.

Limiting the Impact of C++

One of the issues we faced upon deciding to write this book was whether or not to include C++ in the discussion and examples. For almost all of the projects we have worked on throughout our respective careers, the embedded software was written in C and assembly language. In addition, there has been much debate within the embedded software community about the appropriateness of C++. It is widely believed that C++ programs produce larger executables that run more slowly than programs written entirely in C. However, C++ has many benefits for programmers.

We believe that many readers will face the choice of using C++ in their embedded programming. This section covers some of the C++ features that are useful for embedded system software and warns you about some of the more expensive features in the language.

Embedded C++

You might be wondering why the creators of the C++ language included so many features that are expensive in terms of execution time and code size. You are not alone; people around the world have wondered the same thing—especially the users of C++ for embedded programming. Many of these expensive features are recent additions that are neither strictly necessary nor part of the original C++ specification. These features have been added one by one as part of the ongoing "standardization" process.

In 1996, a group of Japanese processor vendors joined together to define a subset of the C++ language and libraries that is better suited for embedded software development. They call their industry standard Embedded C++ (EC++). EC++ generated a great deal of initial interest and excitement within the embedded community.

A proper subset of the draft C++ standard, EC++ omits pretty much anything that can be left out without limiting the expressiveness of the underlying language. This includes not only expensive features such as multiple inheritance, virtual base classes, runtime type identification, and exception handling, but also some of the newest additions such as templates, namespaces, and new-style casts. What's left is a simpler version of C++ that is still object-oriented and a superset of C, but has significantly less runtime overhead and smaller runtime libraries.

A number of commercial C++ compilers support the EC++ standard as an option. Several others allow you to manually disable individual language features, thus enabling you to emulate EC++ (or create your very own flavor of the C++ language).

Of course, not every feature of C++ is expensive. In fact, the earliest C++ compilers used a technology called C-front to turn C++ programs into C, which was then fed into a standard C compiler. That this is even possible demonstrates that many of the syntactical differences between the languages have little or no runtime cost.[*] For example, the definition of a `class` is completely benign. The list of public and private member data and functions is not much different from a `struct` and a list of function prototypes. However, the C++ compiler is able to use the `public` and `private` keywords to determine which method calls and data accesses are allowed and prohibited. Because this determination is made at compile time, there is no penalty paid at runtime. Thus, the use of classes alone affects neither the code size nor efficiency of your programs.

---

[*] Moreover, we want to make clear that there is no penalty for compiling an orginal C program with a C++ compiler.

Default parameter values are also penalty-free. The compiler simply inserts code to pass the default value whenever the function is called without an argument in that position. Similarly, function name overloading involves only a compile-time code modification. Functions with the same names but different parameters are each assigned unique names during the compilation process. The compiler alters the function name each time it appears in your program, and the linker matches them up appropriately.

Operator overloading is another feature that might be used in embedded systems. Whenever the compiler sees such an operator, it simply replaces it with the appropriate function call. So in the C++ code listing that follows, the last two lines are equivalent, and the performance penalty is easily understood:

```
Complex  a, b, c;

c = operator+(a, b);         // The traditional way: Function Call
c = a + b;                   // The C++ way: Operator Overloading
```

Constructors and destructors have a slight penalty. These special methods are guaranteed to be called each time an object of the type is created or goes out of scope, respectively. However, this small amount of overhead is a reasonable price to pay for fewer bugs. Constructors eliminate an entire class of C programming errors having to do with uninitialized data structures. This feature has also proved useful for hiding the awkward initialization sequences associated with some classes.

Virtual functions also have a reasonable cost/benefit ratio. Without going into too much detail about what virtual functions are, let's just say that polymorphism would be impossible without them. And without polymorphism, C++ would not be a true object-oriented language. The only significant cost of virtual functions is one additional memory lookup before a virtual function can be called. Ordinary function and method calls are not affected.

The features of C++ that are typically too expensive for embedded systems are templates, exceptions, and runtime type identification. All three of these negatively impact code size, and exceptions and runtime type identification also increase execution time. Before deciding whether to use these features, you might want to do some experiments to see how they will affect the size and speed of your own application.

The Arcom VIPER-Lite Development Kit

All of the examples in this book have been written for and tested on an embedded platform called the VIPER-Lite. This board is a high-speed embedded controller that is designed, manufactured, and sold by Arcom. The following paragraphs contain information about the hardware, software development tools, and instructions for ordering a board for yourself.

The VIPER-Lite hardware includes the following:

- Processor: PXA255 XScale (based on the ARM v.5TE architecture) (200 MHz)
- RAM: 64 MB of SDRAM
- ROM: 16 MB of flash and 1 MB boot ROM
- Three RS232-compatible serial ports (with external DB9 connectors)
- 10/100baseTx Ethernet port
- USB v1.1 client port
- CompactFlash slot
- Four programmable timer/counters
- Sixteen-channel DMA controller
- Watchdog timer
- Real-time clock
- Eight buffered digital inputs
- Eight buffered digital outputs
- RedBoot debug monitor program resident in boot ROM
- Embedded Linux (based on kernel version 2.6) resident in flash[*]
- JTAG port for system debugging

---

[*] The Windows CE operating system can be specified instead when ordering the VIPER-Lite board. There is an additional cost for the VIPER-Lite with the Windows CE operating system. Examples in the book target the embedded Linux operating system version of the board.

Arcom has also built an add-on module ideal for learning embedded software. The add-on board was designed specifically for the examples shown in this book and is included with a book-specific version of the VIPER-Lite development kit. The VIPER-Lite add-on module includes the following:

- Three LEDs
- Four buttons and jumpers
- Four opto-isolated inputs
- Four opto-isolated outputs
- Two relay outputs
- A buzzer
- A small prototyping area

The VIPER-Lite development kit includes all of the necessary cables for interfacing to the board and a power supply. A photograph of the complete VIPER-Lite development kit in its blue case is shown in Figure A-1.

Figure A-1. Arcom VIPER-Lite development system

The CD-ROM that comes with the Arcom development kit includes all VIPER-Lite manuals and reference documents, datasheets for all components on the board, source code for RedBoot, embedded Linux packages with source code, and binary images for RedBoot and embedded Linux.

The software development tools for the Arcom board are located on the book's web site. We built these tools ourselves for the ARM processor by following the instructions shown in Appendix C. The software tools include the GNU C compiler (*gcc*), assembler (*as*), linker (*ld*), and debugger (*gdb*). We encourage you to investigate the other GNU tools included in the development kit. All programs in this book were built using the tools contained on the book's web site.

For readers of this book, the VIPER-Lite development kit is available at a special discount price of $295 (plus shipping). Use one of the following order codes when contacting Arcom, depending on the operating system you want:

- VIPER-Lite Embedded Linux Development Kit
- VIPER-Lite Windows CE Development Kit

Make sure you mention the book so that you receive the add-on module. Here is Arcom's contact information:

Arcom
7500 West 161st Street
Overland Park, KS 66085
Web: *http://www.arcom.com*
America and Asia: +1 913-549-1000 or *us-sales@arcom.com*
EMEA: +44 (0)1223-403410 or *sales@arcom.co.uk*

Setting Up Your Software Development Environment

This appendix shows the procedure for setting up the GNU software development tools and example source code. The GNU software development tools setup procedure is broken down into two sections: one for Windows and one for Linux. The GNU software tools we use for the example code include the *gcc* version 3.4.4, *as* version 2.15, *ld* version 2.15, *gdb* version 6.3, and *binutils* version 2.15.

This book's web site contains several compressed archive files that expand to include the various source code and tools used in this book. These files are:

windowshost.zip
> Contains the Cygwin setup files and the Windows-based GNU software development tools

linuxhost.tar.gz
> Contains the Linux-based GNU software development tools

ecos.tar.gz
> Contains the eCos source code repository and the eCos development tools (see Appendix D for additional information on setting up the eCos host environment)

examples.zip and examples.tar.gz
> Contain the book's example code

> The Linux example code in Chapter 12 has not been built and tested using a host computer running Windows. It is common to use a Linux host system for developing embedded Linux applications.
>
> Building applications for Linux using a Windows host is beyond the scope of this book. It involves the use of the Cygwin free software toolset, a somewhat more involved procedure than the one described in this chapter.

Windows Host Installation

The GNU software development tools were run on a Pentium 4 computer with Windows XP (Service Pack 2). The first phase of the Windows setup is to install *Cygwin*, a Unix environment for Windows. Additional information about Cygwin can be found online at *http://www.cygwin.com*.

The first step in the Windows host installation procedure is to download the *windowshost.zip* file and unzip it to temporary directory.

Cygwin Installation

Cygwin is used for building all of the examples in this book under Windows. The following instructions assume that *C:* is your hard disk drive where the files are installed. The Cygwin environment is installed under the *C:\cygwin* directory.

1. Run the Cygwin installation program *setup.exe*. The Cygwin install files are located under the *cygwin* directory in the *windowshost.zip* file.

2. The first dialog box is titled Cygwin Net Release Setup Program. This gives the details about the setup program version information. Click Next to continue.

> Looking at the commands and directories here might get a bit confusing, because Windows and Unix environments differ in how they separate directories: Windows uses the backslash (\) and Unix uses the forward slash (/).

3. Now select the directory from which to install the Cygwin tools. In this case, we select "Install from Local Directory" and then click Next.

4. In the next dialog box, select the location on your hard drive where you want the Cygwin tools to be installed. Leave the default as *C:\cygwin*. (If you want to choose an alternate destination, change the drive and directory location accordingly.) In the Install For selection box, select All Users, and for the Default Text File Type, select DOS. Then click Next.

5. Tell the Cygwin setup where the local files that you want to install reside. Browse to the *cygwin* temporary directory, where you unzipped the Cygwin install files, and then click Next. This will cause the Cygwin setup program to inventory the available tools and display the available list.

6. Select the tools to install. Make sure all tools that you want to install are selected for installation. In order to add tools, click on the View button to get the full list of packages available for installation. Then, click on the circular arrow in the New column in order to select the appropriate package for installation. This should change the text next to the circular arrow from "Skip" to the version of the particular package that will be installed. After all packages have been selected for install, click Next to start the installation.

7. After the installation is complete, click Finish. This will add Cygwin icons to the desktop and start menu. Click Ok on the Installation Complete dialog box.

At this point, you should verify that you have a *C:\cygwin* directory with the Cygwin directories and files installed.

GNU Software Tools Installation

The next phase of the Windows host tools setup is to set up the GNU software development tools, as shown here. The GNU software tools are installed under the *\cygwin\opt* directory.

1. The Windows version of the ARM-based GNU tools is located under the *\gnutools* directory in the *windowshost.zip* file. Unzip the file *gnutools.zip* to the *\cygwin\opt* directory on your hard drive.

2. Set the path to the GNU tools location in the Cygwin *bash* shell. To ensure the path is set correctly each time the Cygwin *bash* shell is started, edit the *bash* profile file. The file is named *.bash_profile* and is located under the *$HOME* directory (which is specific to your environment). Add the following to the last line in this file:

```
PATH=/opt/gnutools/arm-elf/bin:$PATH ; export PATH
```

You should notice that the GNU development tools are installed under *C:\cygwin\opt\gnutools*. If you look under the *arm-elf\bin* directory, you should see the GNU tools (such as *gcc*, *as*, *gdb*, and *ld*) executable files, prepended with the name *arm-elf*. This describes the processor for which the tools are built, *arm*, and the object file format, *elf* (which stands for "executable and linkable format").

To test that you installed the tools correctly and set up the path properly, open a Cygwin *bash* shell and enter the command:

```
# arm-elf-gcc -v
```

You should see this response:

```
Reading specs from /opt/gnutools/arm-elf/lib/gcc/arm-elf/3.4.4/specs
Configured with: /src/gcc-3.4.4/configure --target=arm-elf --prefix=/opt/gnutool
s/arm-elf --enable-languages=c,c++ --with-gnu-as --with-gnu-ld --with-newlib --w
ith-gxx-include-dir=/opt/gnutools/arm-elf/arm-elf/include -v
Thread model: single
gcc version 3.4.4
```

Linux Host Installation

The GNU software development tools were tested on a Celeron computer running Linux Fedora Core 5.

The GNU software tools for Linux that are set up in this procedure enable you to build the examples for all chapters except Chapter 12. The examples in Chapter 12 are intended to run on the Arcom board's embedded Linux operating system. The GNU tools installation for building the Chapter 12 example code is covered in Appendix E.

GNU Software Tools Installation

The GNU software tools are installed under the */opt* directory. To install them and make them usable, follow these steps. (You will need to ensure that you have permission to become superuser (root) in order to perform the Linux setup successfully).

1. Open a terminal window and change to the */opt* directory with the command:

   ```
   # cd /opt
   ```

2. The Linux version of the ARM-based GNU tools is located in the file *linuxhost. tar.gz*. Copy this file to the */opt* directory. Next, decompress the file on your hard drive using the command:

   ```
   # tar xvzf linuxhost.tar.gz
   ```

3. Finally, set the path to the GNU tools location in your *bash* shell profile. This ensures the path is set correctly each time the *bash* shell is started. Edit the *bash* profile file named *$HOME/.bash_profile* (where *$HOME* is specific to your environment). Add the following to the last line in this file:

   ```
   PATH=/opt/gnutools/arm-elf/bin:$PATH ; export PATH
   ```

You should notice that the GNU development tools are installed under */opt/gnutools*. The executable files, such as *arm-elf-gcc*, are contained under the */opt/gnutools/bin* directory. The prepended name *arm-elf* describes the processor for which the tools are built, *arm*, and the object file format, *elf*.

To test that you installed the tools correctly and set up the path properly, close the existing terminal window. Open a new terminal window (to ensure the path is set properly) and enter the command:

```
# arm-elf-gcc -v
```

You should see this response:

```
Reading specs from /opt/gnutools/arm-elf/lib/gcc/arm-elf/3.4.4/specs
Configured with: /src/gcc-3.4.4/configure --target=arm-elf --prefix=/opt/gnutool
s/arm-elf --enable-languages=c,c++ --with-gnu-as --with-gnu-ld --with-newlib --w
ith-gxx-include-dir=/opt/gnutools/arm-elf/arm-elf/include -v
Thread model: single
gcc version 3.4.4
```

Example Code Installation

The example files for the book can be extracted into any directory. We recommend installing the example source code files in the */opt/ProgEmbSys* directory. The book's example code is contained in the files *examples.zip* and *examples.tar.gz*.

Each chapter's directory contains all the example source code needed to build the chapter's executables. Makefiles are included in order to simplify the build process. The details of the build procedure are covered in Chapter 4. The procedure for downloading and debugging the examples is covered in Chapter 5.

Building the GNU Software Tools

This appendix shows how to build the ARM-targeted (arm-elf) GNU software tools (*gcc*, *binutils*, *gdb*, *newlib*) used in this book. The instructions are for both Windows (running Cygwin) and Linux host operating systems.
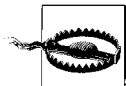
> Unless there is some specific reason for building a new set of GNU tools, we recommend you use the prebuilt GNU binaries provided on the book's web site and included on the Arcom board's CD-ROM. The installation of the prebuilt tools is covered in Appendix B and Appendix E, respectively.

Approximately 1.2 GB of disk space is needed for the GNU tools source code and build output directories. These instructions are adapted from the web page "Building a toolchain for use with eCos," found online at *http://ecos.sourceware.org*.

Another popular site for building cross development tools is found online at *http://kegel.com/crosstool*. This site contains instructions and numerous scripts for building various GNU cross development tool chains for various processors and host systems. You might find that the tools on this site suit your needs better.

The following steps should be followed at a *bash* shell prompt in Cygwin or a terminal in Linux. We built the GNU tools under Windows XP (SP2) and Linux Fedora Core 5. These instructions locate the GNU tools' source code under the */src* directory, the build output under */tmp/build*, and the resulting GNU tools under */opt/gnutools*. In Windows, all of these subdirectories are contained under the main *cygwin* directory.

Extracting the Source Files

After downloading the *gnutoolssrc.tar.gz* file from the book's web site, become superuser (*root*) and proceed as follows:

1. Create a directory to contain the tool sources (avoid directory names containing spaces, as these can confuse the build system). Under Windows, this directory is contained under the *cygwin* directory. Under Linux, this directory is expected to be located under the root (/) directory.

   ```
   # mkdir -p /src
   ```

2. Change to the newly created directory:

   ```
   # cd /src
   ```

3. Copy the *gnutoolssrc.tar.gz* file to the *src* directory.

4. Extract the sources from the compressed file, as shown here. This should create the following directories under *src*: *binutils-2.15*, *gcc-3.4.4*, *gdb-6.3*, and *newlib-1.13.0*.

   ```
   # tar xvzf gnutoolssrc.tar.gz
   ```

   Normally, you would apply any patches needed at this point; however, we have already applied the necessary patches to the source files. There is a *patch* utility to aid in applying patches.

Building the Toolchain

Now you need to compile the tools for your hardware and operating system:

1. Before attempting to build the tools, ensure that the GNU native compiler tools directory is on the PATH and precedes the current directory.

   ```
   # PATH=/bin:$PATH ; export PATH
   ```

2. Configure the GNU binary utilities (*binutils*):

   ```
   # mkdir -p /tmp/build/binutils
   # cd /tmp/build/binutils
   # /src/binutils-2.15/configure --target=arm-elf \
   --prefix=/opt/gnutools/arm-elf -v 2>&1 | tee configure.out
   ```

   The resulting output is contained in the file *configure.out*. If there are any problems configuring the tools, refer to this file.

3. Build and install the GNU *binutils* (this step may take an especially long time):

   ```
   # make -w all install 2>&1 | tee make.out
   ```

   The resulting output is contained in the file *make.out*. If there are any problems building the tools, refer to this file.

4. Ensure that the *binutils* are at the head of the PATH:

   ```
   # PATH=/opt/gnutools/arm-elf/bin:$PATH ; export PATH
   ```

5. Configure *gcc*:

   ```
   # mkdir -p /tmp/build/gcc
   # cd /tmp/build/gcc
   # /src/gcc-3.4.4/configure --target=arm-elf \
   --prefix=/opt/gnutools/arm-elf --enable-languages=c,c++ \
   --with-gnu-as --with-gnu-ld --with-newlib \
   ```

```
--with-gxx-include-dir=/opt/gnutools/arm-elf/arm-elf/include \
-v 2>&1 | tee configure.out
```

6. Build and install *gcc* (this step may take an especially long time):

```
# make -w all install 2>&1 | tee make.out
```

7. Configure *gdb*:

```
# mkdir -p /tmp/build/gdb
# cd /tmp/build/gdb
# /src/gdb-6.3/configure --target=arm-elf \
--prefix=/opt/gnutools/arm-elf --disable-nls \
-v 2>&1 | tee configure.out
```

8. Build and install *gdb* (this step may take an especially long time):

```
# make -w all install 2>&1 | tee make.out
```

Following the successful building and installation of the GNU tools, the associated build tree (located under */tmp/build*) may be deleted to save space if necessary. The toolchain executable files directory (*/opt/gnutools/arm-elf/bin*) should be added to the head of your PATH.

Setting Up the eCos Development Environment

To build an eCos application, you must link the application with the eCos library. This appendix describes how to build a new eCos library for linking with application code.

A prebuilt eCos library is already included in the book's compressed file, but you may need to know the eCos library build procedure if you move to a new development board or require additional functionality in the eCos library.

The eCos examples, covered in Chapter 11, are built in the same manner as the other examples in the book. The build procedure is covered in Chapter 4, and the download-and-debug procedure in Chapter 5. The eCos examples are built using the GNU tools set up in Appendix B, along with the prebuilt eCos library.

Additional details about building eCos applications can be found in the book *Embedded Software Development with eCos*, by Anthony Massa (Prentice Hall PTR).

The eCos Build Environment

Enter the following commands in a Cygwin *bash* shell under Windows or in a terminal window under Linux. The instructions extract the eCos source code to the */opt/ ecos* directory. Other directories can be used, but the instructions need to be adjusted accordingly.
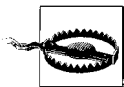
Using Linux, you will need permission to become superuser (*root*) to perform the setup successfully.

eCos Source Code Installation

The eCos source code is contained in the file *ecos.tar.gz*. If you use a different version of the eCos source code, change the following instructions accordingly:

1. Make a directory where the eCos source code is extracted, with the following command:

   ```
   # mkdir -p /opt/ecos
   # cd /opt/ecos
   ```

2. Copy the eCos source file *ecos.tar.gz* to the */opt/ecos* directory.

3. Extract the eCos source code files using the following:

   ```
   # tar xvzf ecos.tar.gz
   ```

   The directory that contains the eCos source code should now be available under */opt/ecos/ecos-redboot-viper-v3i7*.

4. Set up the environment variables. Edit the *$HOME/.bash_profile* file (where *$HOME* is specific to your environment) and add the following lines:

   ```
   PATH=/opt/ecos/ecos-redboot-viper-v3i7/tools:$PATH ; export PATH
   ECOS_REPOSITORY=/opt/ecos/ecos-redboot-viper-v3i7/packages ; export ECOS_REPOSITORY
   ```

   Close the current *bash* environment and open a new one. This allows the changes just made to the environment to become effective.

Building the eCos Library

For this build procedure, we use the eCos command-line configuration tool *ecosconfig*, which is included under the */opt/ecos/ecos-redboot-viper-v3i7/tools* directory. Therefore, these commands are also executed either at a Cygwin *bash* shell prompt on Windows platforms or on a command line on Linux. There is also a graphical configuration tool that could be used to accomplish the same outcome.

The resulting eCos files are located under the */opt/ProgEmbSys/chapter11/ecos* directory. If you have installed the book's source code, there should already be an eCos library present at this location. You need to rename or move the existing eCos library directory before proceeding.

1. Make a directory where the eCos files are going to be built, using the following commands:

   ```
   # mkdir -p /opt/ProgEmbSys/chapter11/ecos
   # cd /opt/ProgEmbSys/chapter11/ecos
   ```

2. Create a new configuration for the Arcom board using the eCos default template by entering the following command:

   ```
   # ecosconfig new arcom-viper default
   ```

3. Create the eCos build tree using the command:

   ```
   # ecosconfig tree
   ```

4. Finally, build the eCos library by entering the command:

   ```
   # make
   ```

If you encounter an error, make sure the path is set up correctly, as previously shown. After successfully building an eCos library, you should see the following message:

```
build finished
```

You should have various directories under */opt/ProgEmbSys/chapter11/ecos*, including the directory *install/lib*. The *lib* directory contains the eCos operating system archive files that get linked with eCos applications.

The eCos makefiles contain a variable, ECOS_INSTALL_DIR, which is set to the location of the eCos install directory—*/opt/ProgEmbSys/chapter11/ecos/install* in this case. If the eCos install directory location changes, this variable must also be changed.

Setting Up the Embedded Linux Development Environment

In this appendix, we present the procedure for setting up a Linux host development environment to build the example applications from Chapter 12 for embedded Linux residing on the Arcom board. These GNU tools are different from the GNU tools covered in Appendix B.

Depending on the version of the Arcom board you ordered, embedded Linux comes preinstalled in flash memory. When the Arcom board is powered up, RedBoot executes a boot script to execute the Linux kernel residing in flash.

> The procedure in this appendix is based on the Arcom board instructions found in the *Arcom Embedded Linux Technical Manual*, which is included in the development kit CD-ROM. Refer to the section titled "Installing the AEL Host Environment," where AEL stands for Arcom Embedded Linux.
>
> The install procedure has been verified on a Celeron computer running Linux Fedora Core 5. If you encounter problems installing the host build environment, we suggest you contact Arcom directly. For technical support in Europe, send email to *euro-support@arcom.com*. For technical support in the United States, send email to *us-support@arcom.com*.

Linux Build Environment Setup

To ensure that applications you develop run properly on the Arcom board, you must compile and link your source code using the libraries present on the target board's version of Linux.
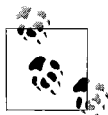
> Building applications for Linux using a Windows (and Cygwin) host is beyond the scope of this book. The Internet is replete with how-to instructions for configuring Cygwin build environments.

The Linux distribution running on the host system must be compatible with the Linux Standard Base (LSB) version 1.3 (see *http://www.linuxbase.org* for more information). We used Fedora Core 5 (*http://fedora.redhat.com*) on our host development system.
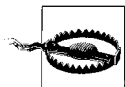
> You will need permission to become superuser (*root*) to perform the installation procedure successfully.

The following commands are executed from a terminal window with the Arcom board development CD-ROM inserted in the drive.

1. Mount the CD-ROM where the Arcom board development CD-ROM is located using the following command:

   # **mount /dev/cdrom /mnt**

2. Run the install script:

   # **perl /mnt/install**

   At this point, the installation program outputs a message similar to the following:

   ```
   Host Distribution Type: Red Hat Linux
   Host Environment: /mnt/host
   Packages: /mnt/packages
   Temporary Storage: /tmp

   Checking for Linux Standard Base version 1
     lsb_release -- /usr/bin/lsb_release
     lsb_release reports version `:core-3.0-ia32:core-3.0-noarch:graphics-3.0-ia32:
   graphics-3.0-noarch' -- LSB version >= 2 detected

   Use of uninitialized value in pattern match (m//) at /mnt/install line 700.
       core-3.0-ia32
       core-3.0-noarch
       graphics-3.0-ia32
       graphics-3.0-noarch

   Several LSB components were found, however none of them appear to provide
   version 1 as required.

   If possible it is recommended that a host distribution that natively
   supports LSB version 1 is used, there may be an additional package for LSB
   version 1 support availble for the host distribution. However if your
   distribution only supports a different LSB version it may be possible to
   install using these LSB components however this is not guaranteed.

   Shall I attempt to install using this version of LSB? [y/N]y


   Checking for the LSB dynamic linker
     /lib/ld-lsb.so.1 -- ok
   ```

```
Checking for required binaries in $PATH...
  rpm -- /bin/rpm

Checking for required packages...
  lsb version 1.3 -- 3.0-9.2 ok (did not check version)
  rpm -- 4.4.2-15.2 ok
  rpm-build -- 4.4.2-15.2 ok
  wget -- 1.10.2-3.2.1 ok

Checking available disk space
  /opt/arcom -- 4095 megabytes -- ok
  /tmp -- 4095 megabytes -- ok
  /var/tmp -- 4095 megabytes -- ok

Installation of the Arcom Embedded Linux Host Environment will now begin.
This may take some time to complete.

Installing Base Arcom Embedded Linux Host Environment

Checking that a simple LSB application can be executed
  ok

Preparing Arcom Embedded Linux Host Environment

Installing Arcom Embedded Linux Host Environment
```

3. Once the installation is successfully completed, the following message will be output:

```
Installation complete.

You should add `/opt/arcom/bin'' to your PATH. You can do this for the
current login session with the following command:
        export PATH=/opt/arcom/bin:$PATH
or you can modify the path for all login sessions for this user by adding
the same statement to `$HOME/.bash_profile'' or for all users by adding it
to `/etc/profile''.
```

4. In order to ensure the path is set correctly each time you open a terminal shell, edit the *$HOME/.bash_profile* file (where *$HOME* is specific to your environment) by adding the following line:

   PATH=/opt/arcom/bin:$PATH ; export PATH

The directory */opt/arcom* should be populated with various files. The executable files, such as *arm-linux-gcc*, are contained under the */opt/arcom/bin* directory.

To test that you installed the tools correctly and set up the path properly, close the existing terminal window. Open a new terminal window (to ensure the path is set properly) and enter the command:

```
# arm-linux-gcc -v
```

You should see a response similar to the following:

```
Reading specs from /opt/arcom/lib/gcc/arm-linux/3.4.4/specs
Configured with: ../gcc-3.4.4/configure --with-gxx-include-dir=/opt/arcom/arm-linux/
include/c++/3.4.4 --target=arm-linux --host=i386-pc-linux-gnu --enable-cross-
toolchain --enable-languages=c,c++ --with-gnu-as --with-gnu-ld --prefix=/opt/arcom --
mandir=/opt/arcom/share/man --with-slibdir=/opt/arcom/arm-linux/lib --enable-
symvers=gnu --enable-c99 --enable-long-long --program-prefix=arm-linux- --program-
suffix=-3.4 --with-headers=/opt/arcom/arm-linux/include --without-newlib --enable-
threads --enable-shared --enable-__cxa_atexit --with-arch=armv4t --with-float=hard
Thread model: posix
gcc version 3.4.4
```

Embedded Linux Examples

The Linux examples are contained in the book's compressed file under the
ProgEmbSys/chapter12 directory. If you followed the installation instructions in
Appendix B, the files should be located in the proper directory. Otherwise, these files
can be extracted into any directory, but we recommend installing the example source
code files in the */opt/ProgEmbSys/chapter12* directory.

> We do not cover building the Linux kernel that runs on the Arcom
> board. For additional information about building the Linux kernel,
> refer to the *Arcom Embedded Linux Technical Manual*.

Building the Linux Examples

To build any of the Chapter 12 Linux examples, proceed as follows.

1. Open a terminal window and change to the directory of the examples you would
   like to build. For instance, to build the *blink* example, enter the command:

       # **cd /opt/ProgEmbSys/chapter12/blink**

2. Then build the example code using the *makefile* with the following command:

       # **make**

   This should produce two executable files named *blink* and *blinkdbg*.

Downloading and Running the Linux Examples

To download the examples and run them on the Arcom board, first boot embedded
Linux. Make sure you connect the Arcom board's COM1 port to your PC's serial
port and are running a console program, such as minicom.

Ensure you have the Arcom board's Ethernet board connected to the main board. Then connect an Ethernet cable between the Arcom board and your computer (either directly or via a hub). The instructions for connecting the Ethernet board are shown in the *Arcom VIPER Technical Manual* and the *VIPER-I/O Technical Manual*.

The following instructions also assume that a Dynamic Host Configuration Protocol (DHCP) server is present on your network. This allows the Arcom board to obtain a dynamic Internet Protocol (IP) address. If you do not have a DHCP server on your network, refer to the *Arcom Embedded Linux Technical Manual* section on statically configuring a network interface.

Power up the Arcom board and allow RedBoot to run the Linux boot script. You should see output similar to the following once Linux begins its boot process:

```
RedBoot> clock -l 27 -m 2 -n 10
mem:99.532MHz run:199.065MHz turbo:199.065MHz. cccr 0x141 (L=27 M=1 N=1.0)
RedBoot> mount -t jffs2 -f filesystem
RedBoot> load -r -b %{FREEMEMLO} %{kernel}
Using default protocol (file)
Raw file loaded 0x00400000-0x004d4c3f, assumed entry at 0x00400000
RedBoot> exec -c %{cmdline}
Using base address 0x00400000 and length 0x000d4c40
Uncompressing Linux............
```

After Linux has successfully booted, you should see the Arcom board's login prompt:

```
viper login:
```

You can then download and run the Linux examples by following these steps:

1. At the board's login prompt, enter root for the login name and arcom for the password. The Arcom board should then output a command prompt:

   ```
   root@viper root#
   ```

2. Next, download the program from the host computer to the Arcom board. For instance, to download the *blink* example, open a terminal window on the host Linux system and enter the following commands:

   ```
   # cd /opt/ProgEmbSys/chapter12/blink
   # scp blink 192.168.0.4:/tmp/blink
   ```

   You may need to replace the IP address (192.168.0.4) with the IP address appropriate for your Arcom board.

   You are then prompted to enter the password for the board as shown here:

   ```
   root@192.168.0.4's password:
   ```

Enter the password arcom. The download will take place and the terminal should show output similar to the following:

```
blink                          100% 3620     3.5KB/s   00:00
```

3. To execute the downloaded program, enter the following at the Arcom board's prompt:

```
root@viper root# /tmp/blink
```

If the program downloaded properly, the green LED should be toggling.

The Linux examples take control over the Arcom board and are intended to run forever. In order to terminate a specific example, press Ctrl-C at the console; the Arcom board should abort the program and return to the VIPER-Lite prompt.

Debugging Embedded Linux Examples

This section gets you started with remote debugging; additional instructions for debugging embedded Linux applications are contained in the *Arcom Embedded Linux Technical Manual*. The embedded Linux debug procedure takes place over the Ethernet connection rather than via a serial connection.

After you have downloaded a program, as previously shown, you can start a *gdb* debug session. The following example shows how to debug the *blink* example.

1. Start the debug session by launching the *gdb* server process on the Arcom board using the command:

```
root@viper root# gdbserver :9000 /tmp/blink "blink"
```

The Arcom board will then wait for you to connect your host to the target, indicating that it is waiting by outputting the following message:

```
Process /tmp/blink created; pid = 706
Listening on port 9000
```

2. Begin a *gdb* session on the host by entering the following in a terminal window:

```
# arm-linux-gdb blinkdbg
```

You should then see the familiar *gdb* prompt (as we covered in Chapter 5), similar to the following:

```
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i386-redhat-linux-gnu --target=arm-linux"...
(gdb)
```

3. Connect the host to the target by entering the following command at the host's *gdb* prompt:

```
(gdb) target remote 192.168.0.4:9000
```

You need to change the IP address (192.168.0.4) to the IP address appropriate for your Arcom board.

Upon successful connection, the Arcom board outputs the following message (the following IP address may be different for your host):

```
Remote debugging from host 192.168.0.3
```

You are now ready to start debugging! For additional information about debugging with *gdb*, refer to Chapter 5.

Index

Symbols

Numbers

A