Chapter 1. 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.

1.1. 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.

1.1.1. 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.

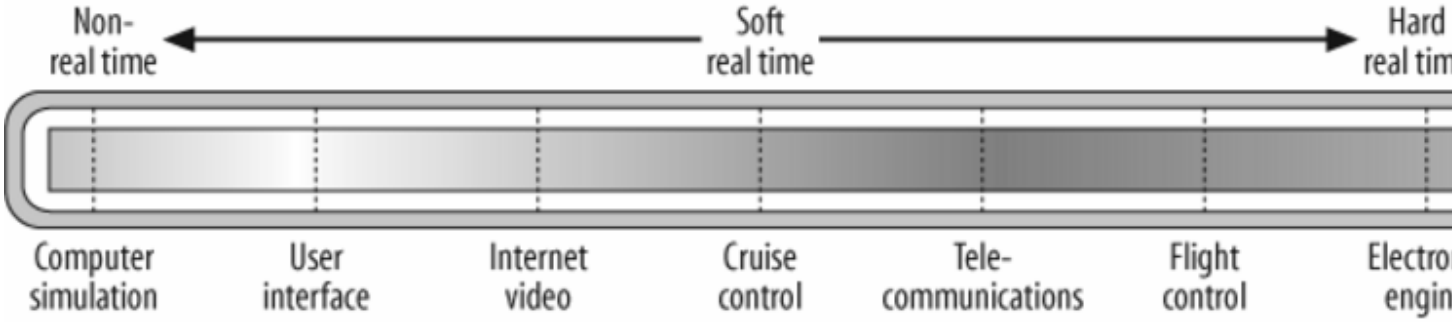
It seems inevitable that the number of embedded systems will continue to increase rapidly. Already there are promising new embedded devices that have enormous market potential: light switches and thermostats that are networked together and can be controlled wirelessly by a central computer, intelligent 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.

1.1.2. 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.

Figure 1-1. A range of example 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.

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.

1.2. 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.

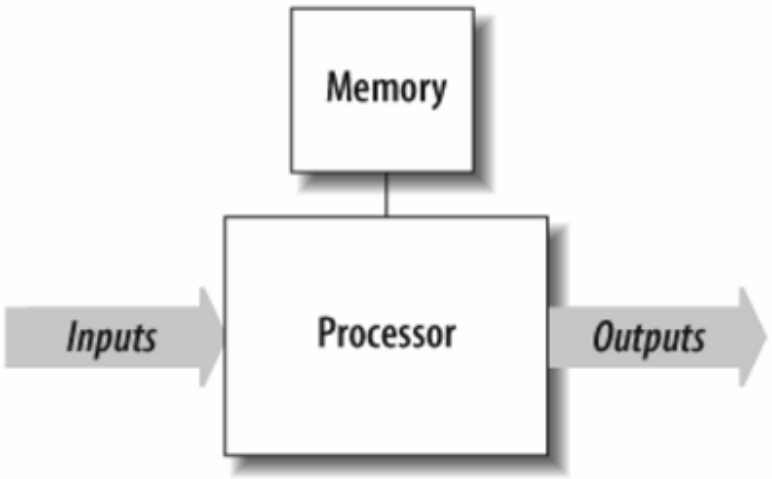
In this section, you will learn which hardware features are common across all embedded systems and why there is so much variation with respect to just about everything else. Later in the book, we will look at some techniques that can be used to minimize the impact of software changes so they are not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

1.2.1. 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.

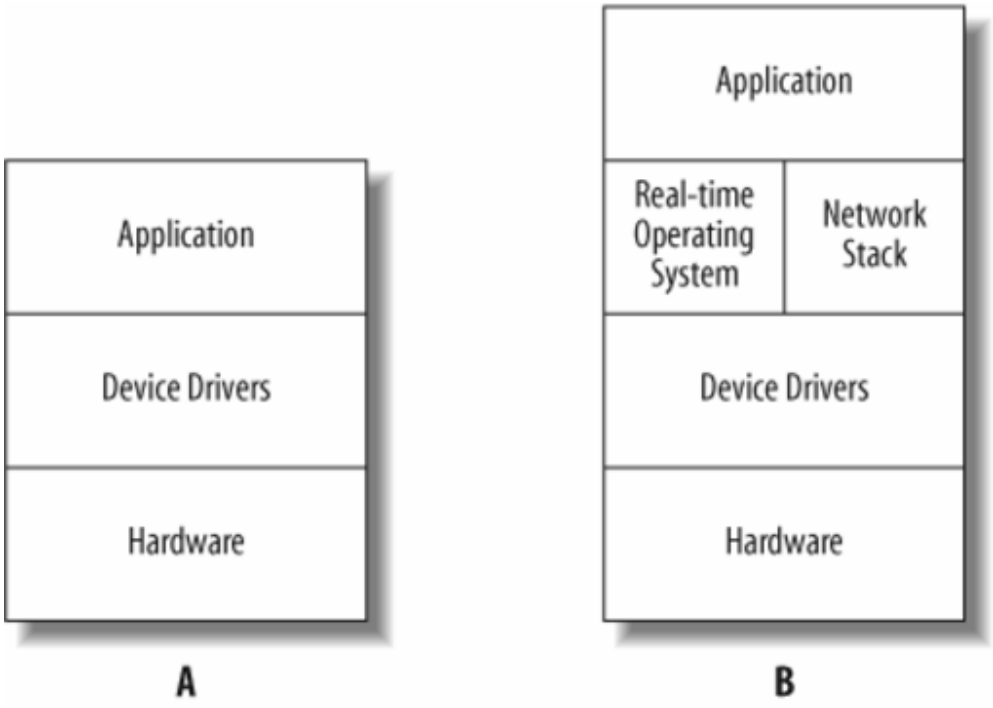
Figure 1-2. A generic embedded system



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.

Figure 1-3. (a) Basic embedded software diagram and (b) a more complex embedded software diagram



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, 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.

1.2.2. 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). [*]

[*] 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.

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).

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

1.3. 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.

1.3.1. 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.

1.3.2. 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.

[†] For example, Atari and Nintendo have designed some of their systems this way.

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.

1.3.3. 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 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.

1.4. 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.

1.5. 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.

1.5.1. 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++.

1.5.2. 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.

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.

1.5.3. 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.

1.6. 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.

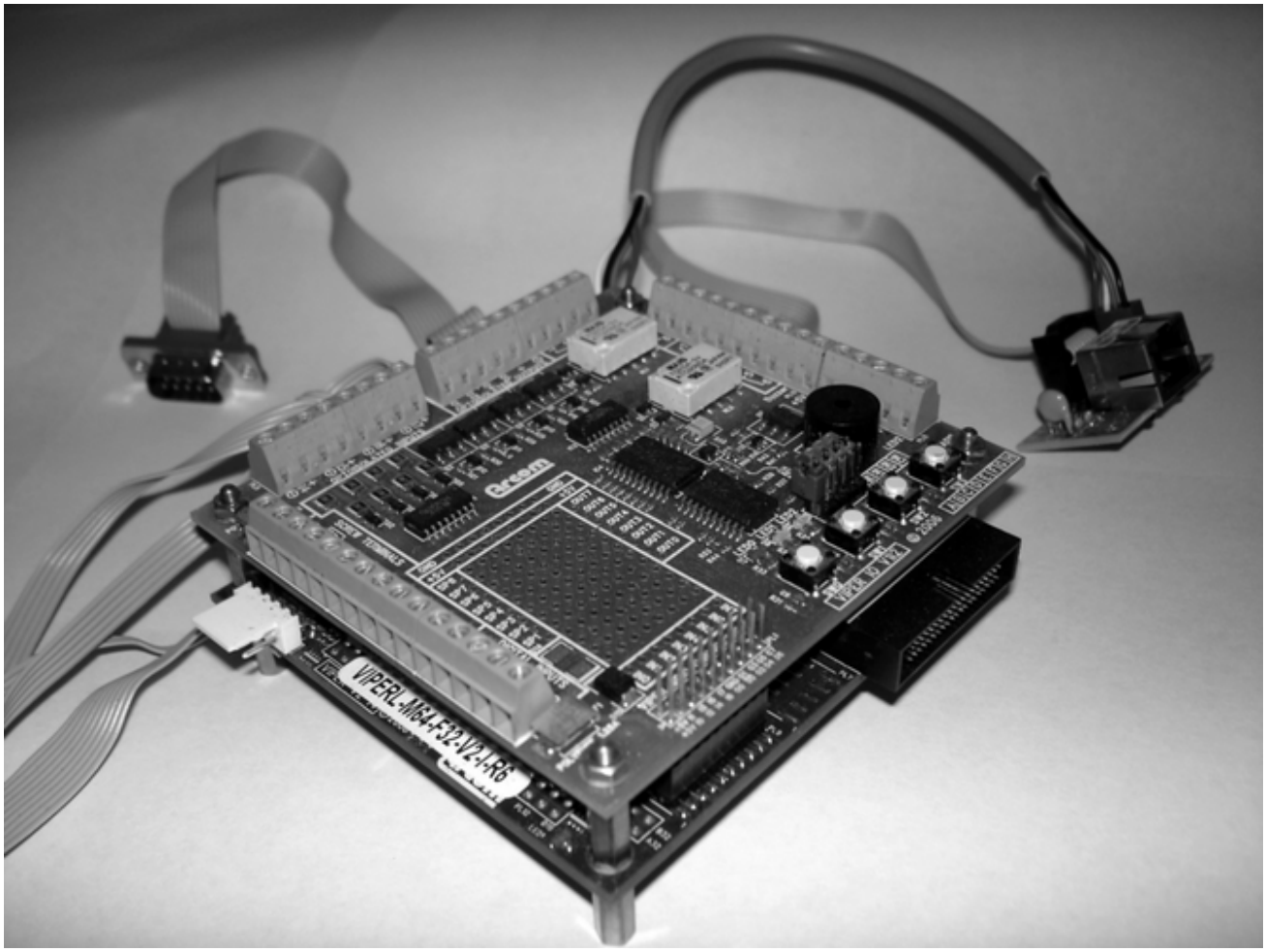
Even selecting a standard embedded platform is difficult. As you have already learned, there is no such thing as a "typical" embedded system. Whatever hardware is selected, the majority of readers will not have access to it. But despite this rather significant problem, we do feel it is important to select 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),[1] 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.

[1] 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.

Chapter 2. Getting to Know the Hardware