# Chapter 1: The Embedded Design Life Cycle

Unlike the design of a software application on a standard platform, the design of an embedded system implies that both software and hardware are being designed in parallel. Although this isn't always the case, it is a reality for many designs today. The profound implications of this simultaneous design process heavily influence how systems are designed.

## Introduction

Figure 1.1 provides a schematic representation of the embedded design life cycle (which has been shown *ad nauseam* in marketing presentations).

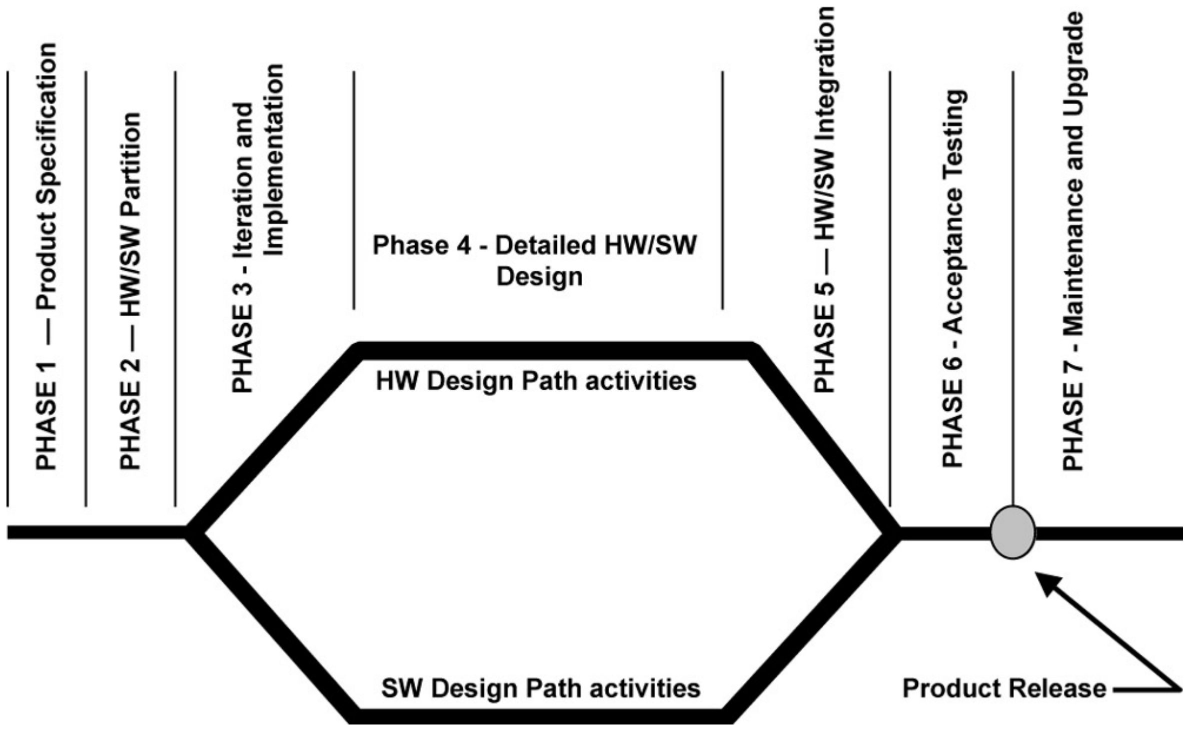
PHASE 1 — Product Specification
PHASE 2 — HW/SW Partition
PHASE 3 - Iteration and Implementation
Phase 4 - Detailed HW/SW Design
PHASE 5 — HW/SW Integration
PHASE 6 - Acceptance Testing
PHASE 7 - Maintenance and Upgrade
HW Design Path activities
SW Design Path activities
Product Release

**Figure 1.1:** Embedded design life cycle diagram.
**A phase representation of the embedded design life cycle.**

Time flows from the left and proceeds through seven phases:

- Product specification
- Partitioning of the design into its software and hardware components
- Iteration and refinement of the partitioning
- Independent hardware and software design tasks
- Integration of the hardware and software components
- Product testing and release
- On-going maintenance and upgrading

The embedded design process is not as simple as Figure 1.1 depicts. A considerable amount of iteration and optimization occurs within phases and between phases. Defects found in later stages often cause you to "go back to square 1." For example, when product testing reveals performance deficiencies that render the design non-competitive, you might have to rewrite algorithms, redesign custom hardware — such as Application-Specific Integrated Circuits (ASICs) for better performance — speed up the processor, choose a new processor, and so on.

Although this book is generally organized according to the life-cycle view in Figure 1.1, it can be helpful to look at the process from other perspectives. Dr. Daniel Mann, Advanced Micro Devices (AMD), Inc., has developed a tool-based view of the development cycle. In Mann's model, processor selection is one of the first tasks (see Figure 1.2). This is understandable, considering the selection of the *right* processor is of prime importance to AMD, a manufacturer of embedded microprocessors. However, it can be argued that including the choice of the microprocessor and some of the other key elements of a design in the specification phase is the correct approach. For example, if your existing code base is written for the 80X86 processor family, it's entirely legitimate to require that the next design also be able to leverage this code base. Similarly, if your design team is highly experienced using the Green Hills© compiler, your requirements document probably would specify that compiler as well.

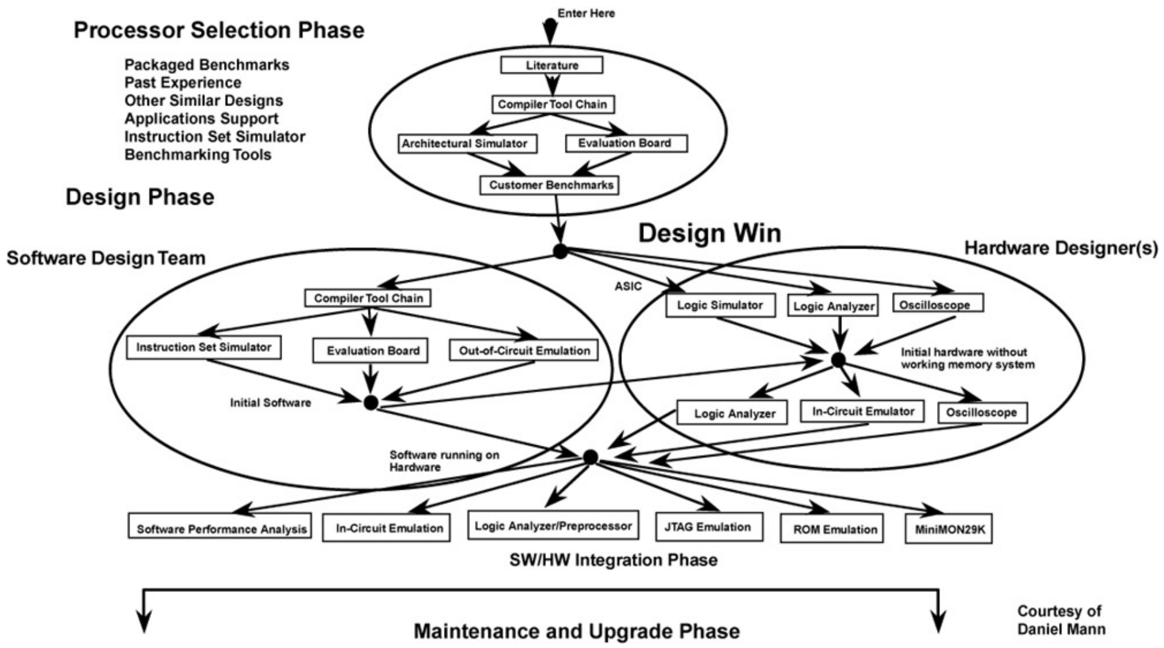

**Figure 1.2:** Tools used in the design process.

**The embedded design cycle represented in terms of the tools used in the design process (courtesy of Dr. Daniel Mann, AMD Fellow, Advanced Micro Devices, Inc., Austin, TX).**

The economics and reality of a design requirement often force decisions to be made before designers can consider the best design trade-offs for the next project. In fact, designers use the term "clean sheet of paper" when referring to a design opportunity in which the requirement constraints are minimal and can be strictly specified in terms of performance and cost goals.

Figure 1.2 shows the maintenance and upgrade phase. The engineers are responsible for maintaining and improving existing product designs until the burden of new features and requirements overwhelms the existing design. Usually, these engineers were not the same group that designed the original product. It's a miracle if the original designers are still around to answer questions about the product. Although more engineers maintain and upgrade projects than create new designs, few, if any, tools are available to help these designers reverse-engineer the product to make improvements and locate bugs. The tools used for maintenance and upgrading are the same tools designed for engineers creating new designs.

The remainder of this book is devoted to following this life cycle through the step-by-step development of embedded systems. The following sections give an overview of the steps in Figure 1.1.

## *Product Specification*

Although this book isn't intended as a marketing manual, learning how to design an embedded system should include some consideration of designing the right embedded system. For many R&D engineers, designing the right product means cramming everything possible into the product to make sure they don't miss anything. Obviously, this wastes time and resources, which is why marketing and sales departments lead (or completely execute) the product-specification process for most companies. The R&D engineers usually aren't allowed customer contact in this early stage of the design. This shortsighted policy prevents the product design engineers from acquiring a useful customer perspective about their products.

Although some methods of customer research, such as questionnaires and focus groups, clearly belong in the realm of marketing specialists, most projects benefit from including engineers in some market-research activities, especially the customer visit or customer research tour.

### The Ideal Customer Research Tour

The ideal research team is three or four people, usually a marketing or sales engineer and two or three R&D types. Each member of the team has a specific role during the visit. Often, these roles switch among the team members so each has an opportunity to try all the roles. The team prepares for the visit by developing a questionnaire to use to keep the interviews flowing smoothly. In general, the questionnaire consists of a set of open-ended questions that the team members fill in as they speak with the customers. For several customer visits, my research team spent more than two weeks preparing and refining the questionnaire.

(Considering the cost of a customer visit tour (about $1,000 per day, per person for airfare, hotels, meals, and loss of productivity), it's amazing how often little effort is put into preparing for the visit. Although it makes sense to visit your customers and get inside their heads, it makes more sense to prepare properly for the research tour.)

The lead interviewer is often the marketing person, although it doesn't have to be. The second team member takes notes and asks follow-up questions or digs down even deeper. The remaining team members are observers and technical resources. If the discussion centers on technical issues, the other team members might have to speak up, especially if the discussion concerns their area of expertise. However, their primary function is to take notes, listen carefully, and look around as much as possible.

After each visit ends, the team meets off-site for a debriefing. The debriefing step is as important as the visit itself to make sure the team members retain the following:

- What did each member hear?
- What was explicitly stated? What was implicit?
- Did they like what we had or were they being polite?
- Was someone really turned on by it?
- Did we need to refine our presentation or the form of the questionnaire?
- Were we talking to the right people?

As the debriefing continues, team members take additional notes and jot down thoughts. At the end of the day, one team member writes a summary of the visit's results.

After returning from the tour, the effort focuses on translating what the team heard from the customers into a set of product requirements to act on. These sessions are often the most difficult and the most fun. The team often is passionate in its arguments for the customers and equally passionate that the customers don't know what they want. At some point in this process, the information from the visit is distilled down to a set of requirements to guide the team through the product development phase.

Often, teams single out one or more customers for a second or third visit as the product development progresses. These visits provide a reality check and some midcourse corrections while the impact of the changes are minimal.

Participating in the customer research tour as an R&D engineer on the project has a side benefit. Not only do you have a design specification (hopefully) against which to design, you also have a picture in your mind's eye of your team's ultimate objective. A little voice in your ear now biases your endless design decisions toward the common goals of the design team. This extra insight into the product specifications can significantly impact the success of the project.

A senior engineering manager studied projects within her company that were successful not only in the marketplace but also in the execution of the product-development process. Many of these projects were embedded systems. Also, she studied projects that had failed in the market or in the development process.

## Flight Deck on the Bass Boat?

Having spent the bulk of my career as an R&D engineer and manager, I am continually fascinated by the process of turning a concept into a product. Knowing how to ask the right questions of a potential customer, understanding his needs, determining the best feature and price point, and handling all the other details of research are not easy, and certainly not straightforward to number-driven engineers.

One of the most valuable classes I ever attended was conducted by a marketing professor at Santa Clara University on how to conduct customer research. I learned that the customer wants everything yesterday and is unwilling to pay for any of it. If you ask a customer whether he wants a feature, he'll say yes every time. So, how do you avoid building an aircraft carrier when the customer really needs a fishing boat? First of all, don't ask the customer whether the product should have a flight deck. Focus your efforts on understanding what the customer wants to accomplish and then extend his requirements to your product. As a result, the product and features you define are an abstraction and a distillation of the needs of your customer.

A common factor for the successful products was that the design team shared a common vision of the product they were designing. When asked about the product, everyone involved — senior management, marketing, sales, quality assurance, and engineering — would provide the same general description. In contrast, many failed products did not produce a consistent articulation of the project goals. One engineer thought it was supposed to be a low-cost product with medium performance. Another thought it was to be a high-performance, medium-cost product, with the objective to maximize the performance-to-cost ratio. A third felt the goal was to get something together in a hurry and put it into the market as soon as possible.

Another often-overlooked part of the product-specification phase is the development tools required to design the product. Figure 1.2 shows the embedded life cycle from a different perspective. This "design tools view" of the development cycle highlights the variety of tools needed by embedded developers.

When I designed in-circuit emulators, I saw products that were late to market because the engineers did not have access to the best tools for the job. For example, only a third of the hard-core embedded developers ever used in-circuit emulators, even though they were the tools of choice for difficult debugging problems.

The development tools requirements should be part of the product specification to ensure that unreal expectations aren't being set for the product development cycle and to minimize the risk that the design team won't meet its goals.

**Tip** One of the smartest project development methods of which I'm aware is to begin each team meeting or project review meeting by showing a list of the project musts and wants. Every project stakeholder must agree that the list is still valid. If things have changed, then the project manager declares the project on hold until the differences are resolved. In most cases, this means that the project schedule and deliverables are no longer valid. When this happens, it's a big deal—comparable to an assembly line worker in an auto plant stopping the line because something is not right with the manufacturing process of the car.

In most cases, the differences are easily resolved and work continues, but not always. Sometimes a competitor may force a re-evaluation of the product features. Sometimes, technologies don't pan out, and an alternative approach must be found. Since the alternative approach is generally not as good as the primary approach, design compromises must be factored in.

## *Hardware/Software Partitioning*

Since an embedded design will involve both hardware and software components, someone must decide which portion of the problem will be solved in hardware and which in software. This choice is called the "partitioning decision."

Application developers, who normally work with pre-defined hardware resources, may have difficulty adjusting to the notion that the hardware can be enhanced to address any arbitrary portion of the problem. However, they've probably already encountered examples of such a hardware/software tradeoff. For example, in the early days of the PC (i.e., before the introduction of the 80486 processor), the 8086, 80286, and 80386 CPUs didn't have an on-chip floating-point processing unit. These processors required companion devices, the 8087, 80287, and 80387 floating-point units (FPUs), to directly execute the floating-point instructions in the application code.

If the PC did not have an FPU, the application code had to trap the floating-point instructions and execute an exception or trap routine to emulate the behavior of the hardware FPU in software. Of course, this was much slower than having the FPU on your motherboard, but at least the code ran.

As another example of hardware/software partitioning, you can purchase a modem card for your PC that plugs into an ISA slot and contains the modulation/demodulation circuitry on the board. For less money, however, you can purchase a Winmodem that plugs into a PCI slot and uses your PC's CPU to directly handle the modem functions. Finally, if you are a dedicated PC gamer, you know how important a high-performance video card is to game speed.

If you generalize the concept of the algorithm to the steps required to implement a design, you can think of the algorithm as a combination of hardware components and software components. Each of these hardware/software partitioning examples implements an algorithm. You can implement that algorithm purely in software (the CPU without the FPU example), purely in hardware (the dedicated modem chip example), or in some combination of the two (the video card example).

### Laser Printer Design Algorithm

Suppose your embedded system design task is to develop a laser printer. Figure 1.3 shows the algorithm for this project. With help from laser printer designers, you can imagine how this task might be accomplished in software. The processor places the incoming data stream — via the parallel port, RS-232C serial port, USB port, or Ethernet port — into a memory buffer.

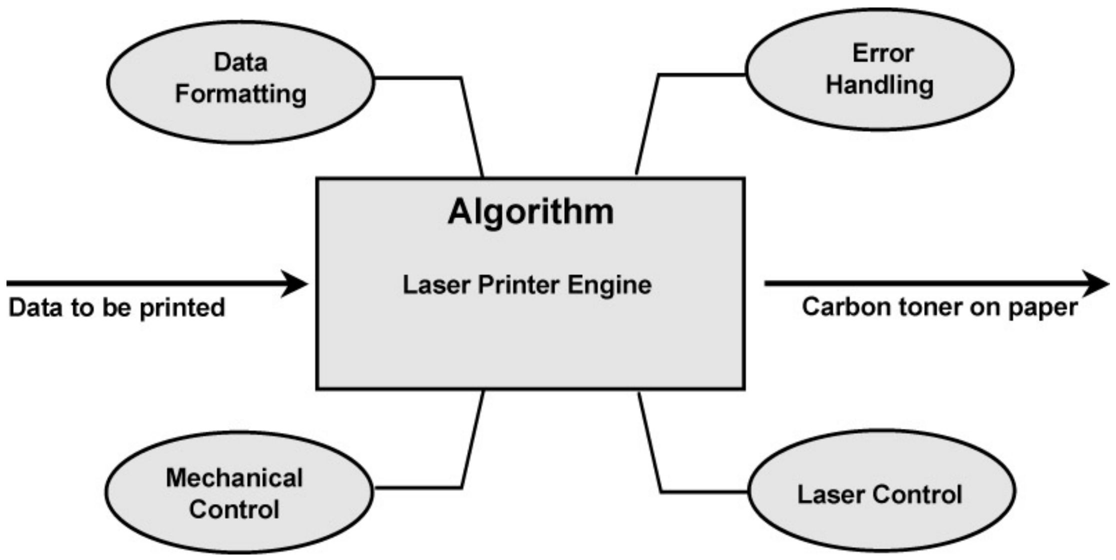

**Figure 1.3:** The laser printer design.

**A laser printer design as an algorithm. Data enters the printer and must be transformed into a legible ensemble of carbon dots fused to a piece of paper.**

Concurrently, the processor services the data port and converts the incoming data stream into a stream of modulation and control signals to a laser tube, rotating mirror, rotating drum, and assorted paper-management "stuff." You can see how this would bog down most modern microprocessors and limit the performance of the system.

You could try to improve performance by adding more processors, thus dividing the concurrent tasks among them. This would speed things up, but without more information, it's hard to determine whether that would be an optimal solution for the algorithm.

When you analyze the algorithm, however, you see that certain tasks critical to the performance of the system are also bounded and well-defined. These tasks can be easily represented by design methods that can be translated to a hardware-based solution. For this laser printer design, you could dedicate a hardware block to the process of writing the laser dots onto the photosensitive surface of the printer drum. This frees the processor to do other tasks and only requires it to initialize and service the hardware if an error is detected.

This seems like a fruitful approach until you dig a bit deeper. The requirements for hardware are more stringent than for software because it's more complicated and costly to fix a hardware defect then to fix a software bug. If the hardware is a custom application-specificc IC (ASIC), this is an even greater consideration because of the overall complexity of designing a custom integrated circuit. If this approach is deemed too risky for this project, the design team must fine-tune the software so that the hardware-assisted circuit devices are not necessary. The risk-management trade-off now becomes the time required to analyze the code and decide whether a software-only solution is possible.

The design team probably will conclude that the required acceleration is not possible unless a newer, more powerful microprocessor is used. This involves costs as well: new tools, new board layouts, wider data paths, and greater complexity. Performance improvements of several orders of magnitude are common when specialized hardware replaces software-only designs; it's hard to realize 100X or 1000X performance improvements by fine-tuning software.

These two very different design philosophies are successfully applied to the design of laser printers in two real-world companies today. One company has highly developed its ability to fine-tune the processor performance to minimize the need for specialized hardware. Conversely, the other company thinks nothing of throwing a team of ASIC designers at the problem. Both companies have competitive products but implement a different design strategy for partitioning the design into hardware and software components.

The partitioning decision is a complex optimization problem. Many embedded system designs are required to be

- Price sensitive
- Leading-edge performers
- Non-standard
- Market competitive
- Proprietary

These conflicting requirements make it difficult to create an optimal design for the embedded product. The algorithm partitioning certainly depends on which processor you use in the design and how you implement the overall design in the hardware. You can choose from several hundred microprocessors, microcontrollers, and custom ASIC cores. The choice of the CPU impacts the partitioning decision, which impacts the tools decisions, and so on.
Given this *n*-space of possible choices, the designer or design team must rely on experience to arrive at an optimal design. Also, the solution surface is generally smooth, which means an adequate solution (possibly driven by an entirely different constraint) is often not far off the best solution. Constraints usually dictate the decision path for the designers, anyway. However, when the design exercise isn't well understood, the decision process becomes much more interesting. You'll read more concerning the hardware/software partitioning problem in Chapter 3.

## Iteration and Implementation

**(Before Hardware and Software Teams Stop Communicating)**

The iteration and implementation part of the process represents a somewhat blurred area between implementation and hardware/software partitioning (refer to Figure 1.1 on page 2) in which the hardware and software paths diverge. This phase represents the early design work before the hardware and software teams build "the wall" between them.

The design is still very fluid in this phase. Even though major blocks might be partitioned between the hardware components and the software components, plenty of leeway remains to move these boundaries as more of the design constraints are understood and modeled. In Figure 1.2 earlier in this chapter, Mann represents the iteration phase as part of the selection process. The hardware designers might be using simulation tools, such as architectural simulators, to model the performance of the processor and memory systems. The software designers are probably running code benchmarks on self-contained, single-board computers that use the target micro processor. These single-board computers are often referred to as evaluation boards because they evaluate the performance of the microprocessor by running test code on it. The evaluation board also provides a convenient software design and debug environment until the real system hardware becomes available.

You'll learn more about this stage in later chapters. Just to whet your appetite, however, consider this: The technology exists today to enable the hardware and software teams to work closely together and keep the partitioning process actively engaged longer and longer into the implementation phase. The teams have a greater opportunity to get it right the first time, minimizing the risk that something might crop up late in the design phase and cause a major schedule delay as the teams scramble to fix it.

## Detailed Hardware and Software Design

This book isn't intended to teach you how to write software or design hardware. However, some aspects of embedded software and hardware design are unique to the discipline and should be discussed in detail. For example, after one of my lectures, a student asked, "Yes, but how does the code actually get into the microprocessor?" Although well-versed in C, C++, and Java, he had never faced having to initialize an environment so that the C code could run in the first place. Therefore, I have devoted separate chapters to the development environment and special software techniques.

I've given considerable thought how deeply I should describe some of the hardware design issues. This is a difficult decision to make because there is so much material that could be covered. Also, most electrical engineering students have taken courses in digital design and microprocessors, so they've had ample opportunity to be exposed to the actual hardware issues of embedded systems design. Some issues are worth mentioning, and I'll cover these as necessary.

## Hardware/Software Integration

The hardware/software integration phase of the development cycle must have special tools and methods to manage the complexity. The process of integrating embedded software and hardware is an exercise in debugging and discovery. Discovery is an especially apt term because the software team now finds out whether it really understood the hardware specification document provided by the hardware team.

### Big Endian/Little Endian Problem

One of my favorite integration discoveries is the "little endian/big endian" syndrome. The hardware designer assumes big endian organization, and the software designer assumes little endian byte order. What makes this a classic example of an interface and integration error is that both the software and hardware could be correct in isolation but fail when integrated because the "endianness" of the interface is misunderstood.

Suppose, for example that a serial port is designed for an ASIC with a 16-bit I/O bus. The port is memory mapped at address 0x400000. Eight bits of the word are the data portion of the port, and the other eight bits are the status portion of the port. Even though the hardware designer might specify what bits are status and what bits are data, the software designer could easily assign the wrong port address if writes to the port are done as byte accesses (Figure 1.5).

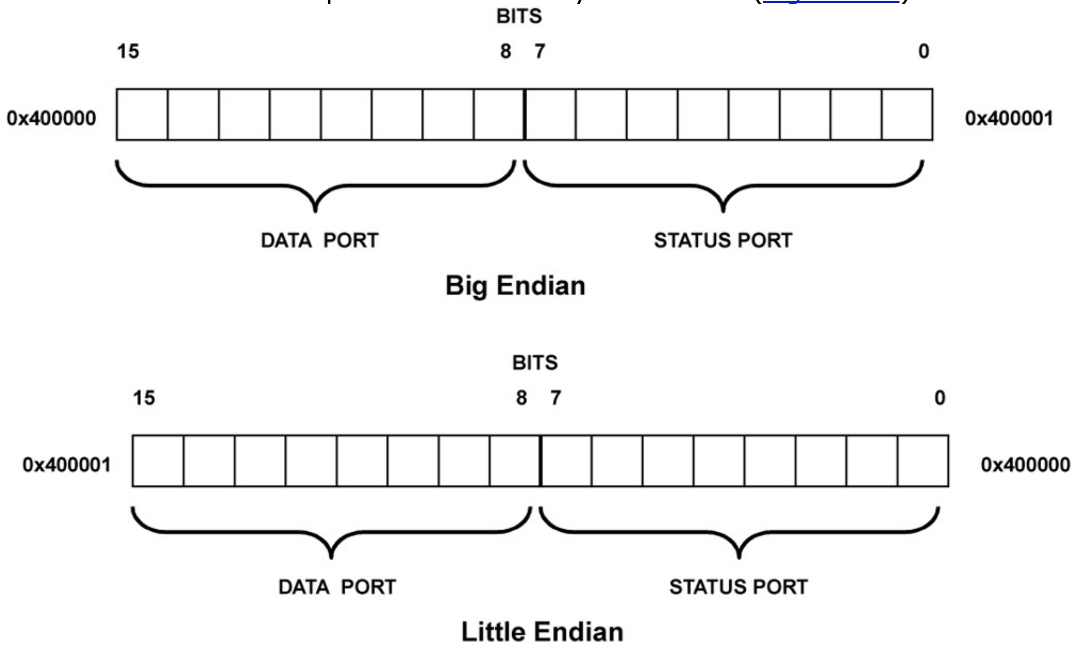

**Figure 1.5:** An example of the endianness problem in I/O addressing.

If byte addressing is used and the big endian model is assumed, then the algorithm should check the status at address 0x400001. Data should be read from and written to address 0x400000. If the little endian memory model is assumed, then the reverse is true. If 16-bit addressing is used, i.e., the port is declared as

```
unsigned short int * io_port ;
```

then the endianness ambiguity problem goes away. This means that the software might become more complex because the developer will need to do bit manipulation in order to read and write data, thus making the algorithm more complex.

The Holy Grail of embedded system design is to combine the first hardware prototype, the application software, the driver code, and the operating system software together with a pinch of optimism and to have the design work perfectly out of the chute. No green wires on the PC board, no "dead bugs," no redesigning the ASICs or Field Programmable Gate Arrays (FPGA), and no rewriting the software. Not likely, but I did say it was the Holy Grail.

**Note** Here "dead bugs" are extra ICs glued to the board with their I/O pins facing up. Green wires are then soldered to their "legs" to patch them into the rest of the circuitry.

You might wonder why this scenario is so unlikely. For one thing, the real-time nature of embedded systems leads to highly complex, nondeterministic behavior that can only be analyzed as it occurs. Attempting to accurately model or simulate the behavior can take much longer than the usable lifetime of the product being developed. This doesn't necessarily negate what I said in the previous section; in fact, it is shades of gray. As the modeling tools improve, so will the designer's ability to find bugs sooner in the process. Hopefully, the severity of the bugs that remain in the system can be easily corrected after they are uncovered. In

*Embedded Systems Programming*[1], Michael Barr discusses a software architecture that anticipates the need for code patches and makes it easy to insert them without major restructuring of the entire code image. I devote Chapters 6, , and to debugging tools and techniques.

## Debugging an Embedded System

In most ways, debugging an embedded system is similar to debugging a host-based application. If the target system contains an available communications channel to the host computer, the debugger can exist as two pieces: a debug kernel in the target system and a host application that communicates with it and manages the source database and symbol tables. (You'll learn more about this later on as well.) Remember, you can't always debug embedded systems using only the methods of the host computer, namely a good debugger and printf() statements.

Many embedded systems are impossible to debug unless they are operating at full speed. Running an embedded program under a debugger can slow the program down by one or more orders of magnitude. In most cases, scaling all the real-time dependencies back so that the debugger becomes effective is much more work than just using the correct tools to debug at full speed.

Manufacturers of embedded microprocessors also realize the difficulty of controlling these variables, so they've provided on-chip hooks to assist in the debugging of embedded systems containing their processors. Most designers won't even consider using a microprocessor in an embedded application unless the silicon manufacturer can demonstrate a complete tool chain for designing and debugging its silicon.

In general, there are three requirements for debugging an embedded or real-time system:

- Run control — The ability to start, stop, peak, and poke the processor and memory.
- Memory substitution — Replacing ROM-based memory with RAM for rapid and easy code download, debug, and repair cycles.
- Real-time analysis — Following code flow in real time with real-time trace analysis.

For many embedded systems, it is necessary also to integrate a commercial or in-house real-time operating system (RTOS) into the hardware and application software. This integration presents its own set of problems (more variables); the underlying behavior of the operating system is often hidden from the designers because it is obtained as object code from the vendor, which means these bugs are now masked by the RTOS and that another special tool must be used.

This tool is usually available from the RTOS vendor (for a price) and is indispensable for debugging the system with the RTOS present. The added complexity doesn't change the three requirements previously listed; it just makes them more complex. Add the phrase "and be RTOS aware" to each of the three listed requirements, and they would be equally valid for a system containing a RTOS.

The general methods of debugging that you've learned to use on your PC or workstation are pretty much the same as in embedded systems. The exceptions are what make it interesting. It is an exercise in futility to try to debug a software module when the source of the problem lies in the underlying hardware or the operating system. Similarly, it is nearly impossible to find a bug that can only be observed when the system is running at full speed when the only trace capability available is to single-step the processor. However, with these tools at your disposal, your approach to debugging will be remarkably similar to debugging an application designed to run on your PC or workstation.

## *Product Testing and Release*

Product testing takes on special significance when the performance of the embedded system has life or death consequences attached. You can shrug off an occasional lock-up of your PC, but you can ill-afford a software failure if the PC controls a nuclear power generating station's emergency system. Therefore, the testing and reliability requirements for an embedded system are much more stringent than the vast majority of desktop applications. Consider the embedded systems currently supporting your desktop PC: IDE disk drive, CD-ROM, scanner, printer, and other devices are all embedded systems in their own right. How many times have they failed to function so that you had to cycle power to them?

**From the Trenches**

For the longest time, my PC had a nagging problem of crashing in the middle of my word processor or graphics application. This problem persisted through Windows 95, 95 Sr-1, 98, and 98 SE. After blaming Microsoft for shoddy software, I later discovered that I had a hardware problem in my video card. After replacing the drivers and the card, the crashes went away, and my computer is behaving well. I guess hardware/software integration problems exist on the desktop as well.

However, testing is more than making sure the software doesn't crash at a critical moment, although it is by no means an insignificant consideration. Because embedded systems usually have extremely tight design margins to meet cost goals, testing must determine whether the system is performing close to its optimal capabilities. This is especially true if the code is written in a high-level language and the design team consists of many developers.

Many desktop applications have small memory leaks. Presumably, if the application ran long enough, the PC would run out of heap space, and the computer would crash. However, on a desktop machine with 64MB of RAM and virtual swap space, this is unlikely to be a problem. On the other side, in an embedded system, running continuously for weeks at a time, even a small memory leak is potentially disastrous.

### Who Does the Testing?

In many companies, the job of testing the embedded product goes to a separate team of engineers and technicians because asking a designer to test his own code or product usually results in erratic test results. It also might lead to a "circle the wagons" mentality on the part of the design team, who view the testers as a roadblock to product release, rather than equal partners trying to prevent a defective product from reaching the customer.

## Compliance Testing

Compliance testing is often overlooked. Modern embedded systems are awash in radio frequency (RF) energy. If you've traveled on a plane in the last five years, you're familiar with the requirement that all electronic devices be turned off when the plane descends below 10,000 feet. I'm not qualified to discuss the finer points of RF suppression and regulatory compliance requirements; however, I have spent many hours at open field test sites with various compliance engineering (CE) engineers trying just to get one peak down below the threshold to pass the class B test and ship the product.

I can remember one disaster when the total cost of the RF suppression hardware that had to be added came to about one-third of the cost of all the other hardware combined. Although it can be argued that this is the realm of the hardware designer and not a hardware/software design issue, most digital hardware designers have little or no training in the arcane art of RF suppression. Usually, the hotshot digital wizard has to seek out the last remaining analog designer to get clued in on how to knock down the fourth harmonic at 240MHz. Anyway, CE testing is just as crucial to a product's release as any other aspect of the test program.

CE testing had a negative impact on my hardware/software integration activities in one case. I thought we had done a great job of staying on top of the CE test requirements and had built up an early prototype especially for CE testing. The day of the tests, I proudly presented it to the CE engineer on schedule. He then asked for the test software that was supposed to exercise the hardware while the RF emissions were being monitored. Whoops, I completely forgot to write drivers to exercise the hardware. After some scrambling, we pieced together some of the turn-on code and convinced the CE engineer (after all, he had to sign all the forms) that the code was representative of the actual operational code.

Referring to Figure 1.4, notice the exponential rise in the cost to fix a defect the later you are in the design cycle. In many instances, the Test Engineering Group is the last line of defense between a smooth product release and a major financial disaster.

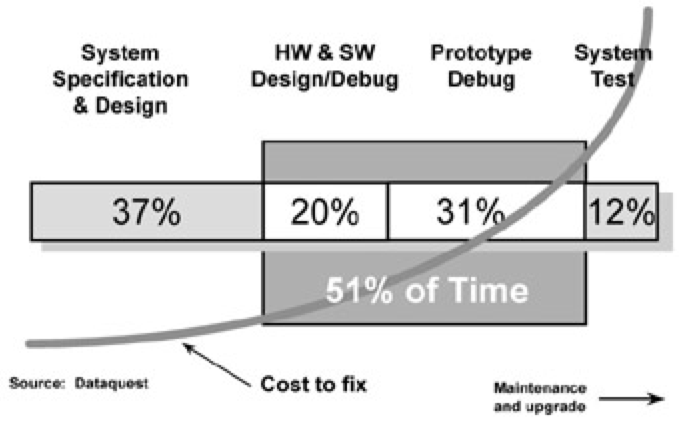

**Figure 1.4:** Where design time is spent.

**The percentage of project time spent in each phase of the embedded design life cycle. The curve shows the cost associated with fixing a defect at each stage of the process.**
Like debugging, many of the elements of reliability and performance testing map directly on the best practices for host-based software development. Much has been

written about the correct way to develop software, so I won't cover that again here. What is relevant to this subject is the best practices for testing software that has mission-critical or tight performance constraints associated with it. Just as with the particular problems associated with debugging a real-time system, testing the same system can be equally challenging. I'll address this and other testing issues in Chapter 9.

## *Maintaining and Upgrading Existing Products*

The embedded system tool community has made almost no effort to develop tools specifically targeted to products already in service. At first blush, you might not see this as a problem. Most commercially developed products are well documented, right?

The majority of embedded system designers (around 60 percent) maintain and upgrade existing products, rather than design new products. Most of these engineers were not members of the original design team for a particular product, so they must rely on only their experience, their skills, the existing documentation, and the old product to understand the original design well enough to maintain and improve it.

From the silicon vendor's point of view, this is an important gap in the tool chain because the vendor wants to keep that customer buying its silicon, instead of giving the customer the chance to do a "clean sheet of paper" redesign. Clean sheets of paper tend to have someone else's chip on them.

**From the Trenches**

One can hardly overstate the challenges facing some upgrade teams. I once visited a telecomm manufacturer that builds small office phone systems to speak to the product-support team. The team described the situation as: "They wheel the thing in on two carts. The box is on one cart, and the source listings are on the other. Then they tell us to make it better." This usually translates to improving the overall performance of the embedded system without incurring the expense of a major hardware redesign.

Another example features an engineer at a company that makes laser and ink-jet printers. His job is to study the assembly language output of their C and C++ source code and fine-tune it to improve performance by improving the code quality. Again, no hardware redesigns are allowed.

Both of these examples testify to the skill of these engineers who are able to reverse-engineer and improve upon the work of the original design teams.

This phase of a product's life cycle requires tools that are especially tailored to reverse engineering and rapidly facilitating "what if …" scenarios. For example, it's tempting to try a quick fix by speeding up the processor clock by 25 percent; however, this could cause a major ripple effect through the entire design, from memory chip access time margins to increased RF emissions. If such a possibility could be as easily explored as making measurements on a few critical code modules, however, you would have an extremely powerful tool on your hands.

Sometimes, the solutions to improved performance are embarrassingly simple. For example, a data communications manufacturer was about to completely redesign a product when the critical product review uncovered that the processor was spending most of its time in a debug module that was erroneously left in the final build of the object code. It was easy to find because the support teams had access to sophisticated tools that enabled them to observe the code as it executed in real time. Without the tools, the task might have been too time-consuming to be worthwhile.

Even with these test cases, every marketing flyer for every tool touts the tool's capability to speed "time to market." I've yet to hear any tool vendor advertise its tool as speeding "time to reverse-engineer," although one company claimed that its logic analyzer sped up the "time to insight."

Embedded systems projects aren't just "software on small machines." Unlike application development, where the hardware is a *fait accompli*, embedded projects are usually optimization exercises that strive to create both hardware and software that complement each other. This difference is the driving force that defines the three most characteristic elements of the embedded design cycle: selection, partitioning, and system integration. This difference also colors testing and debugging, which must be adapted to work with unproven, proprietary hardware.

While these characteristic differences aren't all there is to embedded system design, they are what most clearly differentiate it from application development, and thus, they are the main focus of this book. The next chapter discusses the processor selection decision. Later chapters address the other issues.

## Work Cited

1. Barr, Michael. "Architecting Embedded Systems for Add-on Software Modules." Embedded Systems Programming, September 1999, 49.