Chapter 1

Embedded Computing

- Fundamental problems in embedded computing
- Applications that make use of embedded computing
- Design methodologies and system modeling for embedded systems
- Models of computation
- Reliability and security
- Consumer electronics

1.1 The Landscape of High-Performance Embedded Computing

The overarching theme of this book is that many embedded computing systems are high-performance computing systems that must be carefully designed so that they meet stringent requirements. Not only do they require lots of computation, but they also must meet quantifiable goals: real-time performance, not just average performance; power/energy consumption; and cost. The fact that it has quantifiable goals makes the design of embedded computing systems a very different experience than the design of general-purpose computing systems for which their users are unpredictable.

When trying to design computer systems to meet various sorts of quantifiable goals, we quickly come to the conclusion that no one system is best for all applications. Different requirements lead to making different trade-offs between performance and power, hardware and software, and so on. We must create different implementations to meet the needs of a family of applications. Solutions should be programmable enough to make the design flexible and long-lived, but
need not provide unnecessary flexibility that would detract from meeting system requirements.

General-purpose computing systems separate the design of hardware and software, but in embedded computing systems we can simultaneously design the hardware and software. Often, a problem can be solved by hardware means, software means, or a combination of the two. Various solutions can have different trade-offs; the larger design space afforded by joint hardware/software design allows us to find better solutions to design problems.

As illustrated in Figure 1-1 the study of embedded system design properly takes into account three aspects of the field: architectures, applications, and methodologies. Compared to the design of general-purpose computers, embedded computer designers rely much more heavily on both methodologies and basic knowledge of applications. Let us consider these aspects one at a time.

Because embedded system designers work with both hardware and software, they must study architectures broadly speaking, including hardware, software, and the relationships between the two. Hardware architecture problems can range from special-purpose hardware units as created by hardware/software co-design, microarchitectures for processors, multiprocessors, or networks of distributed processors. Software architectures determine how we can take advantage of parallelism and nondeterminism to improve performance and lower cost.

Understanding your application is key to getting the most out of an embedded computing system. We can use the characteristics of the application to optimize the design. This can be an advantage that enables us to perform many powerful optimizations that would not be possible in a general-purpose system. But it also means that we must have enough understanding of the application to take advantage of its characteristics and avoid creating problems for system implementers.

Methodologies play an especially important role in embedded computing. Not only must we design many different types of embedded systems, but we
also must do so reliably and predictably. The cost of the design process itself is often a significant component of the total system cost. Methodologies, which may combine tools and manual steps, codify our knowledge of how to design systems. Methodologies help us make large and small design decisions.

The designers of general-purpose computers stick to a more narrowly defined hardware design methodology that uses standard benchmarks as inputs to tracing and simulation. The changes to the processor are generally made by hand and may be the result of invention. Embedded computing system designers need more complex methodologies because their system design encompasses both hardware and software. The varying characteristics of embedded systems—system-on-chip for communications, automotive network, and so on—also push designers to tweak methodologies for their own purposes.

Steps in a methodology may be implemented as tools. Analysis and simulation tools are widely used to evaluate cost, performance, and power consumption. Synthesis tools create optimized implementations based on specifications. Tools are particularly important in embedded computer design for two reasons. First, we are designing an application-specific system, and we can use tools to help us understand the characteristics of the application. Second, we are often pressed for time when designing an embedded system, and tools help us work faster and produce more predictable tools.

The design of embedded computing systems increasingly relies on a hierarchy of models. Models have been used for many years in computer science to provide abstractions. Abstractions for performance, energy consumption, and functionality are important. Because embedded computing systems have complex functionality built on top of sophisticated platforms, designers must use a series of models to have some chance of successfully completing their system design. Early stages of the design process need reasonably accurate simple models; later design stages need more sophisticated and accurate models.

Embedded computing makes use of several related disciplines; the two core ones are real-time computing and hardware/software co-design. The study of real-time systems predates the emergence of embedded computing as a discipline. Real-time systems take a software-oriented view of how to design computers that complete computations in a timely fashion. The scheduling techniques developed by the real-time systems community stand at the core of the body of techniques used to design embedded systems. Hardware/software co-design emerged as a field at the dawn of the modern era of embedded computing. Co-design takes a holistic view of the hardware and software used to perform deadline-oriented computations.

Figure 1-2 shows highlights in the development of embedded computing. We can see that computers were embedded early in the history of computing:

* Many of the dates in this figure were found in Wikipedia; others are from http://www.motofuture.motorola.com and http://www.mvista.com.
one of the earliest computers, the MIT Whirlwind, was designed for artillery control. As computer science and engineering solidified into a field, early research established basic techniques for real-time computing. Some techniques used today in embedded computing were developed specifically for the problems of embedded systems while others, such as those in the following list, were adapted from general-purpose computing techniques.

- Low-power design began as primarily hardware-oriented but now encompasses both software and hardware techniques.
- Programming languages and compilers have provided tools, such as Java and highly optimized code generators, for embedded system designers.
- Operating systems provide not only schedulers but also file systems and other facilities that are now commonplace in high-performance embedded systems.
1.2 Example Applications

Some knowledge of the applications that will run on an embedded system is of great help to system designers. This section looks at several basic concepts in three common applications: communications/networking, multimedia, and vehicles.

1.2.1 Radio and Networking

Modern communications systems combine wireless and networking. As illustrated in Figure 1-3 radios carry digital information and are used to connect to networks. Those networks may be specialized, as in traditional cell phones, but increasingly radios are used as the physical layer in Internet protocol systems.

The Open Systems Interconnection (OSI) model [Sta97a] of the International Standards Organization (ISO) defines the following model for network services.

1. **Physical layer**—The electrical and physical connection
2. **Data link layer**—Access and error control across a single link
3. **Network layer**—Basic end-to-end service
4. **Transport layer**—Connection-oriented services
5. **Session layer**—Control activities such as checkpointing
6. **Presentation layer**—Data exchange formats
7. **Application layer**—The interface between the application and the network

Although it may seem that embedded systems are too simple to require use of the OSI model, it is in fact quite useful. Even relatively simple embedded networks provide physical, data link, and network services. An increasing number
of embedded systems provide Internet service that requires implementing the full range of functions in the OSI model.

The Internet is one example of a network that follows the OSI model. The Internet Protocol (IP) \cite{Los97; Sta97a} is the fundamental protocol of the Internet. An IP is used to internetwork between different types of networks—the internetworking standard. The IP sits at the network layer in the OSI model. It does not provide guaranteed end-to-end service; instead, it provides best-effort routing of packets. Higher-level protocols must be used to manage the stream of packets between source and destination.

Wireless data communication is widely used. On the receiver side, digital communication must perform the following tasks.

- **Demodulate** the signal down to the baseband
- **Detect** the baseband signal to identify bits
- **Correct errors** in the raw bit stream

**Software radio**

Wireless data transmitters may be built from combinations of analog, hard-wired digital, configurable, and programmable components. A software radio is, broadly speaking, a radio that can be programmed; the term software-defined radio (SDR) is often used to mean either a purely or partly programmable radio. Given the clock rates at which today’s digital processors operate, they

![Figure 1-3 A radio and network connection.](image-url)
1.2 Example Applications

are used primarily for baseband operations. Several processors can run fast enough to be used for some radio-frequency processing.

The SDR Forum, a technical group for software radio, defines the following five tiers of SDR [SDR05].

- Tier 0—A **hardware radio** cannot be programmed.
- Tier 1—A **software-controlled radio** has some functions implemented in software, but operations like modulation and filtering cannot be altered without changing hardware.
- Tier 2—A **software-defined radio** may use multiple antennas for different bands, but the radio can cover a wide range of frequencies and use multiple modulation techniques.
- Tier 3—An **ideal software-defined radio** does not use analog amplification or heterodyne mixing before A/D conversion.
- Tier 4—An **ultimate software radio** is lightweight, consumes very little power, and requires no external antenna.

**digital demodulation**

Demodulation requires multiplying the received signal by a signal from an oscillator and filtering the result to select the signal’s lower-frequency version. The bit-detection process depends somewhat on the modulation scheme, but digital communication mechanisms often rely on phase. High-data rate systems often use multiple frequencies arranged in a **constellation**. The phases of the component frequencies of the signal can be modulated to create different symbols.

**error correction**

Traditional error-correction codes can be checked using combinational logic. For example, a convolutional coder can be used as an error-correction coder. The convolutional coder convolves the input with itself according to a chosen polynomial. Figure 1-4 shows a fragment of a trellis that represents possible states of a decoder; the label on an edge shows the input bit and the produced output bits. Any bits in the transmission may have been corrupted; the decoder must determine the most likely sequence of data bits that could have produced the received sequence.

Several more powerful codes that require iterative decoding have recently become popular. **Turbo codes** use multiple encoders. The input data is encoded by two convolutional encoders, each of which uses a different but generally simple code. One of the coders is fed the input data directly; the other is fed a permuted version of the input stream. Both coded versions of the data are sent across the channel. The decoder uses two decoding units, one for each code. The two decoders are operated iteratively. At each iteration, the two decoders swap likelihood estimates of the decoded bits; each decoder uses the other’s likelihoods as a priori estimates for its own next iteration.
Low-density parity check (LDPC) codes also require multiple iterations to determine errors and corrections. An LDPC code can be defined using a bipartite graph like the one shown in Figure 1-5; the codes are called “low density” because their graphs are sparse. The nodes on the left are called message nodes, and the ones on the right are check nodes. Each check node defines a sum of message node values. The message nodes define the coordinates for codewords; a legal codeword is a set of message node values that sets all the check nodes to 1. During decoding, an LDPC decoding algorithm passes messages between the message nodes and check nodes. One approach is to pass probabilities for the data bit values as messages. Multiple iterations should cause the algorithm to settle onto a good estimate of the data bit values.

A radio may simply act as the physical layer of a standard network stack, but many new networks are being designed that take advantage of the inherent characteristics of wireless networks. For example, traditional wired networks have only a limited number of nodes connected to a link, but radios inherently broadcast; broadcasts can be used to improve network control, error correction, and security. Wireless networks are generally ad hoc in that the members of the

![Figure 1-4](image1.png)  
**Figure 1-4** A trellis representation for a convolutional code.

![Figure 1-5](image2.png)  
**Figure 1-5** A bipartite graph that defines an LDPC code.
network are not predetermined, and nodes may enter or leave during network operation. **Ad hoc networks** require somewhat different network control than is used in fixed, wired networks.

Example 1-1 looks at a cell phone communication standard.

---

**Example 1-1**

*cdma2000*

cdma2000 [Van04] is a widely used standard for spread spectrum-based cellular telephony. It uses direct sequence spread spectrum transmission. The data appears as noise unless the receiver knows the pseudorandom sequence. Several radios can use the same frequency band without interference because the pseudorandom codes allow their signals to be separated. A simplified diagram of the system follows.

The spreader modulates the data with the pseudorandom code. The interleaver transposes coded data blocks to make the code more resistant to burst errors. The transmitter’s power is controlled so that all signals have the same strength at the receiver.

The physical layer protocol defines a set of channels that can carry data or control. A **forward channel** goes from a base station to a mobile station, while a **reverse channel** goes from a mobile station to a base station. Pilot channels are used to acquire the CDMA signal, provide phase information, and enable the mobile station to estimate the channel’s characteristics. A number of different types of channels are defined for data, control, power control, and so on.
The link layer defines medium access control (MAC) and signaling link access control (LAC). The MAC layer multiplexes logical channels onto the physical medium, provides reliable transport of user traffic, and manages quality-of-service. The LAC layer provides a variety of services: authentication, integrity, segmentation, reassembly, and so on.

Example 1-2 describes a major effort to develop software radios for data communication.

Example 1-2

*Joint Tactical Radio System*

The Joint Tactical Radio System (JTRS) [Joi05; Ree02] is an initiative of the U.S. Department of Defense to develop next-generation communication systems based on radios that perform many functions in software. JTRS radios are designed to provide secure communication. They are also designed to be compatible with a wide range of existing radios as well as to be upgradeable through software.

The reference model for the hardware architecture has two major components. The front-end subsystem performs low-level radio operations while the back-end subsystem performs higher-level network functions. The information security enforcement module that connects the front and back ends helps protect the radio and the network from each other.
1.2.2 Multimedia

Today’s dominant multimedia applications are based on compression: digital television and radio broadcast, portable music, and digital cameras all rely on compression algorithms. This section reviews some of the algorithms developed for multimedia compression.

It is important to remember that multimedia compression methods are lossy—the decompressed signal is different from the original signal before compression. Compression algorithms make use of perceptual coding techniques that try to throw away data that is less perceptible to the human eye and ear. These algorithms also combine lossless compression with perceptual coding to efficiently code the signal.

The JPEG standard [ITU92] is widely used for image compression. The two major techniques used by JPEG are the discrete cosine transform (DCT) plus quantization, which performs perceptual coding, plus Huffman coding as a form of entropy coding for lossless encoding. Figure 1-6 shows a simplified view of DCT-based image compression: blocks in the image are transformed using the DCT; the transformed data is quantized and the result is entropy coded.

The DCT is a frequency transform whose coefficients describe the spatial frequency content of an image. Because it is designed to transform images, the DCT operates on a two-dimensional set of pixels, in contrast to the Fourier transform, which operates on a one-dimensional signal. However, the advantage of the DCT over other two-dimensional transforms is that it can be decomposed into two one-dimensional transforms, making it much easier to compute. The form of the DCT of a set of values $u(i)$ is

$$ (v)(k) = \sqrt{\frac{2}{N}} C(k) \sum_{1 \leq t \leq N} u(t) \cos \left( \frac{\pi(2t+1)k}{2N} \right), \quad \text{(EQ 1-1)} $$

where

$$ C(k) = 2^{-1/2} \text{ for } k = 0, 1 \text{ otherwise.} \quad \text{(EQ 1-2)} $$

Many efficient algorithms have been developed to compute the DCT.

![Figure 1-6](image-url) Simplified view of a DCT-based image-compression system.
JPEG performs the DCT on 8 × 8 blocks of pixels. The discrete cosine transform itself does not compress the image. The DCT coefficients are quantized to add loss and change the signal in such a way that lossless compression can more efficiently compress them. Low-order coefficients of the DCT correspond to large features in the 8 × 8 block, and high-order coefficients correspond to fine features. Quantization concentrates on changing the higher-order coefficients to zero. This removes some fine features but provides long strings of zeros that can be efficiently encoded to lossless compression.

Huffman coding, which is sometimes called variable-length coding, forms the basis for the lossless compression stage. As shown in Figure 1-7, a specialized technique is used to order the quantized DCT coefficients in a way that can be easily Huffman encoded. The DCT coefficients can be arranged in an 8 × 8 matrix. The 0,0 entry at the top left is known as the DC coefficient since it describes the lowest-resolution or DC component of the image. The 7,7 entry is the highest-order AC coefficient. Quantization has changed the higher-order AC coefficients to zero. If we were to traverse the matrix in row or column order, we would intersperse nonzero lower-order coefficients with higher-order coefficients that have been zeroed. By traversing the matrix in a zigzag pattern, we move from low-order to high-order coefficients more uniformly. This creates longer strings of zeroes that can be efficiently encoded.

The JPEG 2000 standard is compatible with JPEG but adds wavelet compression. Wavelets are a hierarchical waveform representation of the image that do not rely on blocks. Wavelets can be more computationally expensive but provide higher-quality compressed images.

Figure 1-7 The zigzag pattern used to transmit DCT coefficients.
There are two major families of video compression standards. The MPEG series of standards was developed primarily for broadcast applications. Broadcast systems are asymmetric—more powerful and more expensive transmitters allows receivers to be simpler and cheaper. The H.26x series is designed for symmetric applications, such as videoconferencing, in which both sides must encode and decode. The two groups have recently completed a joint standard known as Advanced Video Codec (AVC), or H.264, designed to cover both types of applications. An issue of the Proceedings of the IEEE [Wu06] is devoted to digital television.

Video encoding standards are typically defined as being composed of several streams. A useful video system must include audio data; users may want to send text or other data as well. A compressed video stream is often represented as a system stream, which is made up of a sequence of system packets. Each system packet may include any of the following types of data:

- Video data
- Audio data
- Nonaudiovisual data
- Synchronization information

Because several streams are combined into one system stream, synchronizing the streams for decoding can be a challenge. Audio and video data must be closely synchronized to avoid annoying the viewer/listener. Text data, such as closed captioning, may also need to be synchronized with the program.

Figure 1-8 shows the block diagram of an MPEG-1 or MPEG-2 style encoder. (The MPEG-2 standard is the basis for digital television broadcasting in the United States.) The encoder makes use of the DCT and variable-length coding. It adds motion estimation and motion compensation to encode the relationships between frames.

Motion estimation allows one frame to be encoded as translational motion from another frame. Motion estimation is performed on 16 × 16 macroblocks. A macroblock from one frame is selected and a search area in the reference frame is searched to find an identical or closely matching macroblock. At each search point, a sum-of-absolute-differences (SAD) computation is used to measure the difference between the search macroblock S and the macroblock R at the selected point in the reference frame:

\[
SAD = \sum_{0 \leq x \leq 15} \left( \sum_{0 \leq y \leq 15} |S(x, y) - R(x, y)| \right)
\]  

(EQ 1-3)
The search point with the smallest SAD is chosen as the point to which $S$ has moved in the reference frame. That position describes a motion vector for the macroblock (see Figure 1-9). During decompression, motion compensation copies the block to the position specified by the motion vector, thus saving the system from transmitting the entire image.

Motion estimation does not perfectly predict a frame because elements of the block may move, the search may not provide the exact match, and so on. An error signal

Figure 1-8 Structure of an MPEG-1 and MPEG-2 style video encoder.

Figure 1-9 Motion estimation results in a motion vector.
1.2 Example Applications

The inverse DCT and picture/store predictor in the feedback are used to generate the uncompressed version of the lossily compressed signal that would be seen by the receiver; that reconstruction is used to generate the error signal.

Digital audio compression also uses combinations of lossy and lossless coding. However, the auditory centers of the brain are somewhat better understood than the visual center, allowing for more sophisticated perceptual encoding approaches.

Many audio-encoding standards have been developed. The best known name in audio compression is MP3. This is a nickname for MPEG-1 Audio Layer 3, the most sophisticated of the three levels of audio compression developed for MPEG-1. However, U.S. HDTV broadcasting, although it uses the MPEG-2 system and video streams, is based on Dolby Digital. Many open-source audio codecs have been developed, with Ogg Vorbis being one popular example.

As shown in Figure 1-10, an MPEG-1 audio encoder has four major components [ISO93]. The mapper filters and subsamples the input audio samples. The quantizer codes subbands, allocating bits to the various subbands. Its parameters are adjusted by a psychoacoustic model, which looks for phenomena that will not be heard well and so can be eliminated. The framer generates the final bit stream.

1.2.3 Vehicle Control and Operation

Real-time vehicle control is one of the major applications of embedded computing. Machines like automobiles and airplanes require control systems that are physically distributed around the vehicle. Networks have been designed specifically to meet the needs of real-time distributed control for automotive electronics and avionics.
The basic fact that drives the design of control systems for vehicles is that they are safety-critical systems. Errors of any kind—component failure, design flaws, and so on—can injure or kill people. Not only must these systems be carefully verified, but they also must be architected to guarantee certain properties.

As shown in Figure 1-11, modern automobiles use a number of electronic devices [Lee02b]. Today’s low-end cars often include 40 microprocessors while high-end cars can contain 100 microprocessors. These devices are generally organized into several networks. The critical control systems, such as engine and brake control, may be on one network while noncritical functions, such as entertainment devices, may be on a separate network.

Until the advent of digital electronics, cars generally used point-to-point wiring organized into harnesses, which are bundles of wires. Connecting devices into a shared network saves a great deal of weight—15 kilograms or more [Lee02b]. Networks require somewhat more complicated devices that include network access hardware and software, but that overhead is relatively small and is shrinking over time thanks to Moore’s Law.

But why not use general-purpose networks for real-time control? We can find reasons to build specialized automotive networks at several levels of specialization.

**Figure 1-11** Electronic devices in modern automobiles. From Lee [Lee02b] © 2002 IEEE
abstraction in the network stack. One reason is electrical—automotive networks require reliable signaling under vary harsh environments. The ignition systems of automobile engines generate huge amounts of electromagnetic interference that can render many networks useless. Automobiles must also operate under wide temperature ranges and survive large doses of moisture.

Most important, real-time control requires guaranteed behavior from the network. Many communications networks do not provide hard real-time requirements. Communications systems are also more tolerant of latency than are control systems. While data or voice communications may be useful when the network introduces transmission delays of hundreds of milliseconds or even seconds, long latencies can easily cause disastrous oscillations in real-time control systems. Automotive networks must also operate within limited power budgets that may not apply to communications networks.

Avionics

Aviation electronics systems developed in parallel to automotive electronics are now starting to converge. Avionics must be certified for use in aircraft by governmental authorities (in the U.S., aircraft are certified by the Federal Aviation Administration—FAA), which means that devices for aircraft are often designed specifically for aviation use. The fact that aviation systems are certified has made it easier to use electronics for critical operations such as the operation of flight control surfaces (e.g., ailerons, rudders, elevators). Airplane cockpits are also highly automated. Some commercial airplanes already provide Internet access to passengers; we expect to see such services become common in cars over the next decade.

X-by-wire

Control systems have traditionally relied on mechanics or hydraulics to implement feedback and reaction. Microprocessors allow us to use hardware and software not just to sense and actuate but to implement the control laws. In general, the controller may not be physically close to the device being controlled: the controller may operate several different devices, or it may be physically shielded from dangerous operating areas. Electronic control of critical functions was first performed in aircraft where the technique was known as fly-by-wire. Control operations performed over the network are called X-by-wire where X may be brake, steer, and so on.

Noncontrol uses

Powerful embedded devices—television systems, navigation systems, Internet access, and so on—are being introduced into cars. These devices do not perform real-time control, but they can eat up large amounts of bandwidth and require real-time service for streaming data. Since we can only expect the amount of data being transmitted within a car to increase, automotive networks must be designed to be future-proof and handle workloads that are even more challenging than what we see today.

In general, we can divide the uses of a network in a vehicle into several categories along the following axes.

- **Operator versus passenger**—This is the most basic distinction in vehicle networks. The passenger may want to use the network for a variety of pur-
poses: entertainment, information, and so on. But the passenger’s network must never interfere with the basic control functions required to drive or fly the vehicle.

- **Control versus instrumentation**—The operation of the vehicle relies on a wide range of devices. The basic control functions—steering, brakes, throttle, and so on in a car or the control surfaces and throttle in an airplane—must operate with very low latencies and be completely reliable. Other functions used by the operator may be less important. At least some of the instrumentation in an airplane is extremely important for monitoring in-flight meteorological conditions, but pilots generally identify a minimal set of instruments required to control the airplane. Cars are usually driven with relatively little attention paid to the instruments. While instrumentation is very important, we may separate it from fundamental controls in order to protect the operation of the control systems.

### 1.2.4 Sensor Networks

Sensor networks are distributed systems designed to capture and process data. They typically use radio links to transmit data between themselves and to servers. Sensor networks can be used to monitor buildings, equipment, and people.

A key aspect of the design of sensor networks is the use of ad hoc networks. Sensor networks can be deployed in a variety of configurations and nodes can be added or removed at any time. As a result, both the network and the applications running on the sensor nodes must be designed to dynamically determine their configuration and take the necessary steps to operate under that network configuration.

For example, when data is transmitted to a server, the nodes do not know in advance the path that data should take to arrive at the server. The nodes must provide multihop routing services to transmit data from node to node in order to arrive at the network. This problem is challenging because not all nodes are within radio range, and it may take network effort and computation to determine the topology of the network.

Examples 1-3 and 1-4 describe a sensor network node and its operating system, and Example 1-5 describes an application of sensor networks.

---

**Example 1-3**

*The Intel mote*² Sensor Node

The Intel mote², which uses a 802.15.4 radio (the ChipCon 2420 radio) as its communication link, is a third-generation sensor network node.
1.2 Example Applications

An antenna is built into the board. Each side of the board has a pair of connectors for sensor devices, one side for basic devices and another for advanced devices. Several boards can be stacked using these connectors to build a complete system.

The on-board processor is an Intel XScale. The processor can be operated at low voltages and frequencies (0.85V and 15 MHz, respectively) and can be run up to 416 MHz at the highest operating voltage. The board includes 265 MBytes of SRAM organized into four banks.

Source: Courtesy Intel.

Example 1-4

TinyOS and nesC

TinyOS (http://www.tinyos.net) is an operating system for sensor networks. It is designed to support networks and devices on a small platform using only about 200 bytes of memory.

TinyOS code is written in a new language known as nesC. This language supports the TinyOS concurrency model based on tasks and hardware event handlers. The nesC compiler detects data races at compile time. An nesC
program includes one set of functions known as events. The program may also include functions called commands to help implement the program, but another component uses the events to call the program. A set of components can be assembled into a system using interface connections known as wiring.

TinyOS executes only one program using two threads: one containing tasks and another containing hardware event handlers. The tasks are scheduled by TinyOS; tasks are run to completion and do not preempt each other. Hardware event handlers are initiated by hardware interrupts. They may preempt tasks or other handlers and run to completion.

The sensor node radio is one of the devices in the system. TinyOS provides code for packet-based communication, including multihop communication.

---

**Example 1-5**

*ZebraNet*

ZebraNet [Jua02] is designed to record the movements of zebras in the wild. Each zebra wears a collar that includes a GPS positioning system, a network radio, a processor, and a solar cell for power. The processors periodically read the GPS position and store it in on-board memory. The collar reads positions every three minutes, along with information indicating whether the zebra is in sun or shade. For three minutes every hour, the collar takes detailed readings to determine the zebra’s speed. This generates about 6 kilo (k) of data per zebra per day.

Experiments show that computation is much less expensive than radio transmissions:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Current @ 3.6V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>&lt;1 mA</td>
</tr>
<tr>
<td>GPS position sampling and CPU/storage</td>
<td>177 mA</td>
</tr>
<tr>
<td>Base discovery only</td>
<td>432 mA</td>
</tr>
<tr>
<td>Transmit data to base</td>
<td>1622 mA</td>
</tr>
</tbody>
</table>

Thus conservation of radio energy is critical. The data from the zebras is read only intermittently when biologists travel to the field. They do not want to leave behind a permanent base station, which would be difficult to maintain. Instead, they bring with them a node that reads data from the network.
Because the zebras move over a wide area, not all of them are within range of the base station, and it is impossible to predict which (if any) of the zebras will be. As a result, the ZebraNet nodes must replicate data across the network. The nodes transmit copies of their position data to each other as zebras come within range of each other. When a zebra comes within range of a base station, the base station reads all of that zebra’s data, including data it has gathered from other zebras.

The ZebraNet group experimented with two data-transfer protocols. One protocol—flooding—sent all data to every other available node. The other, history-based protocol chose one peer to send data to based on which peer had the best past history of delivering data to the base. Simulations showed that flooding delivered the most data for short-range radios, but the history-based protocol delivered the most data for long-range radio. However, flooding consumed much more energy than history-based routing.

### 1.3 Design Goals

Given an application area for which we want to design an embedded system, we must determine specific goals for the project and estimate their feasibility. The application determines the basic **functional requirements**. We must also determine the **nonfunctional requirements**, some of which are derived directly from the application and some from other factors, such as marketing. An embedded system design project may have many goals. Some of these are measurable, and others are less so.

Several key metrics of a digital system design can be accurately measured and predicted. The first is **performance**, by which we mean some aspect of speed. (Every field seems to use performance as the name for its preferred metric—image quality, packet loss, and so on.) Performance, however, can be measured in many different ways, including:

- Average performance versus worst-case or best-case
- Throughput versus latency
- Peak versus sustained

**Energy** and/or **power consumption** are critical metrics for many embedded systems. Energy consumption is particularly important for battery life. Power consumption affects heat generation.

The monetary cost of a system is clearly of interest to many people. Cost can be measured in several ways. **Manufacturing cost** is determined by the cost of components and the manufacturing processes used. **Design cost** is determined...
both by labor and by the equipment used to support the designers. (The server farm and CAD tools required to design a large chip cost several million dollars.) **Lifetime cost** takes into account software and hardware maintenance and upgrades.

**design time**

The time required to design a system may be important. If the design program takes too long to finish, the product may miss its intended market. Calculators, for example, must be ready for the back-to-school market each fall.

**reliability**

Different markets place different values on reliability. In some consumer markets, customers do not expect to keep the product for a long period. Automobiles, in contrast, must be designed to be safe.

**quality**

Quality is important but may be difficult to define and measure. It may be related to reliability in some markets. In other markets—for instance, consumer devices—factors such as user interface design may be associated with quality.

### 1.4 Design Methodologies

**design repeatability**

Design methodology has traditionally been important to VLSI design but not to general-purpose computer architecture. Many different chips are designed; methodologies codify design practices. However, computer architecture has traditionally been treated as a field of invention for which methodologies are irrelevant.

However, the situation is different in embedded computing. While invention is useful and important when designing novel embedded systems, so are repeatability and design time. Although high-end embedded platforms are heterogeneous multiprocessors, many more of them are designed per year than are general-purpose processors. As a result, we need to understand and document design practices. Not only does this save time in the long run, but it also makes system design more repeatable. When embarking on a new project, we need to be able to predict the time and resources required to complete the project. The better we understand our methodology, the better able we are to predict design costs.

**synthesis and simulation**

A design methodology is not simply an abstraction—it must be defined in terms of available tools and resources. The designers of high-performance embedded systems face many challenges, some of which include the following.

- The design space is large and irregular. We do not have adequate synthesis tools for many important steps in the design process. As a result, designers must rely on analysis and simulation for many design phases.

- We cannot afford to simulate everything in extreme detail. Not only do simulations take time, but also the cost of the server farm required to run large
1.4 Design Methodologies

Simulations is a significant element of overall design cost. In particular, we cannot perform a cycle-accurate simulation of the entire design for the large data sets required to validate large applications.

- We need to be able to develop simulators quickly. Simulators must reflect the structure of application-specific designs. System architects need tools to help them construct application-specific simulators.

- Software developers for systems-on-chips need to be able to write and evaluate software before the hardware is completed. They need to be able to evaluate not just functionality but performance and power as well.

System designers need tools to help them quickly and reliably build heterogeneous architectures. They need tools to help them integrate several different types of processors, and they need tools to help them build multiprocessors from networks, memories, and processing elements.

Figure 1-12 shows the growth of design complexity and designer productivity over time, as estimated by the Sematech in the mid-1990s. Design complexity is fundamentally estimated by Moore’s Law, which predicts a 58% annual increase in the number of transistors per chip. Sematech estimates that designer productivity has grown and will continue to grow by only 21% per year. The result is a wide and growing gap between the chips we can manufacture and the chips we can design. Embedded computing is one partial answer to the designer productivity problem, since we move some of the design tasks to software. But we also need improved methodologies for embedded computing systems to ensure we can continue to design platforms and load them with useful software.

![Figure 1-12 Design complexity and designer productivity trends.](image)
1.4.1 Basic Design Methodologies

Much of the early writings on design methodologies for computer systems cover software, but the methodologies for hardware tend to use a wider variety of tools since hardware design makes more use of synthesis and simulation tools. An ideal embedded systems methodology makes use of the best of both hardware and software traditions.

One of the earliest models for software development was the **waterfall model** illustrated in Figure 1-13. The waterfall model is divided into five major stages: requirements, specification, architecture, coding, and maintenance. The software is successively refined through these stages, with maintenance including software delivery and follow-on updates and fixes. Most of the information in this methodology flows from the top down—that is, from more abstract stages to more concrete stages—although some information could flow back from one stage to the preceding stage to improve the design. The general flow of design information down the levels of abstraction gives the waterfall model its name. The waterfall model was important for codifying the basic steps of software development, but researchers soon realized that the limited flow of information from detailed design back to improve the more abstract phases was both an unrealistic picture of software design practices and an undesirable feature of an ideal methodology. In practice, designers can and should use experience from design steps to go back, rethink earlier decisions, and redo some work.

The **spiral model**, also shown in Figure 1-13, was a reaction to and a refinement of the waterfall model. This model envisions software design as an iterative process in which several versions of the system, each better than the last, are created. At each phase, designers go through a requirements/architecture/coding...
cycle. The results of one cycle are used to guide the decisions in the next round of development. Experience from one stage should both help produce a better design at the next stage and allow the design team to create that improved design more quickly.

Figure 1-14 shows a simplified version of the hardware design flows used in many VLSI designs. Modern hardware design makes extensive use of several techniques not as frequently seen in software design: search-based synthesis algorithms and models and estimation algorithms. Hardware designs also have more quantifiable design metrics than traditional software designs. Hardware designs must meet strict cycle-time requirements, power budgets, and area budgets. Although we have not shown backward design flow from lower to higher levels of abstraction, most design flows allow such iterative design.

Modern hardware synthesis uses many types of models. In Figure 1-14, the cell library describes the cells used for logic gates and registers, both concretely in terms of layout primitives and more abstractly in terms of delay, area, and so on. The technology database captures data not directly associated with cells, such as wire characteristics. These databases generally carry static data in the form of tables. Algorithms are also used to evaluate models. For example,
several types of wirability models are used to estimate the properties of the wiring in the layout before that wiring is complete. Timing and power models evaluate the performance and power consumption of designs before all details of the design are known; for example, although both timing and power depend on the exact wiring, wire length estimates can be used to help estimate timing and power before the delay is complete. Good estimators help keep design iterations local. The tools can search the design space to find a good design, but within a given level of abstraction and based on models at that level. Good models combined with effective heuristic search can minimize the need for backtracking and throwing out design results.

1.4.2 Embedded Systems Design Flows

Embedded computing systems combine hardware and software components that must work closely together. Embedded system designers have evolved design methodologies that play into our ability to embody part of the functionality of the system in software.

Co-design flows

Early researchers in hardware/software co-design emphasized the importance of concurrent design. Once the system architecture has been defined, the hardware and software components can be designed relatively separately. The goal of co-design is to make appropriate architectural decisions that allow later implementation phases to be carried out separately. Good architectural decisions, because they must satisfy hard metrics such as real-time performance and power consumption, require appropriate analysis methods.

Figure 1-15 shows a generic co-design methodology. Given an executable specification, most methodologies perform some initial system analysis to determine parallelism opportunities and perhaps break the specification into processes. Hardware/software partitioning chooses an architecture in which some operations are performed directly by hardware and others are performed by software running on programmable platforms. Hardware/software partitioning produces module designs that can be implemented separately. Those modules are then combined, tested for performance or power consumption, and debugged to create the final system.

Platform-based design

Platform-based design is a common approach to using systems-on-chips. Platforms allow several customers to customize the same basic platform into different products. Platforms are particularly useful in standards-based markets where some basic features must be supported but other features must be customized to differentiate products.

Two-stage process

As shown in Figure 1-16, platform-based design is a two-stage process. First, the platform must be designed based on the overall system requirements (the standard, for example) and how the platform should be customizable. Once the platform has been designed, it can be used to design a product. The product makes use of the platform features and adds its own features.
1.4 Design Methodologies

Figure 1-15 A design flow for hardware/software co-design

Figure 1-16 Platform-based design.
Platform design requires several design phases:

- Profiling and analysis turn system requirements and software models into more specific requirements on the platform hardware architecture.
- Design space exploration evaluates hardware alternatives.
- Architectural simulation helps evaluate and optimize the details of the architecture.
- Base software—hardware abstraction layers, operating system ports, communication, application libraries, debugging—must be developed for the platform.

Platform use is challenging in part because the platform requires a custom programming environment. Programmers are accustomed to rich development environments for standard platforms. Those environments provide a number of tools—compilers, editors, debuggers, simulators—in a single graphical user interface. However, rich programming environments are typically available for uniprocessors. Multiprocessors are more difficult to program, and heterogeneous multiprocessors are even more difficult than homogeneous multiprocessors. The platform developers must provide tools that allow software developers to use the platform. Some of these tools come from the component CPUs, but other tools must be developed from scratch. Debugging is particularly important and difficult, since debugging access is hardware-dependent. Interprocess communication is also challenging but is a critical tool for application developers.

### 1.4.3 Standards-Based Design Methodologies

Many high-performance embedded computing systems implement standards. Multimedia, communications, and networking all provide standards for various capabilities. One product may even implement several different standards. This section considers the effects of the standards on embedded systems design methodologies [Wol04].

On the one hand, standards enable products and in particular systems-on-chips. Standards create large markets for particular types of functions: they allow devices to interoperate, and they reassure customers that the devices provide the required functions. Large markets help justify any system design project, but they are particularly important in system-on-chip (SoC) design. To cover the costs of SoC design and manufacturing, several million of the chips must be sold in many cases. Such large markets are generally created by standards.

On the other hand, the fact that the standard exists means that the chip designers have much less control over the specification of what they need to design. Standards define complex behavior that must be adhered to. As a result, some features of the architecture will be dictated by the standard.
Most standards do provide for improvements. Many standards define that certain operations must be performed, but they do not specify how they are to be performed. The implementer can choose a method based on performance, power, cost, quality, or ease of implementation. For example, video compression standards define basic parameters of motion estimation but not which motion estimation algorithm should be performed.

The intellectual property and effort required to implement a standard goes into different parts of the system than would be the case for a blank-sheet design. Algorithm design effort goes into unspecified parts of the standard and parts of the system that lie beyond the standard. For example, cell phones must adhere to communication standards but are free to be designed for many aspects of their user interfaces.

Standards are often complex, and standards in a given field tend to become more complex over time. As a field evolves, practitioners learn more about how to do a better job and strive to build that knowledge into the standard. While these improvements may lead to higher-quality systems, they also make system implementation more extensive.

Standards bodies typically provide a reference implementation. This is an executable program that conforms to the standard. It is often written in C, but may be written in Java or some other language. The reference implementation is first used to aid standard developers. It is then distributed to implementers of the specification. (The reference implementation may be available free of charge, but in many cases, an implementer must pay a license fee to the standards body to build a system that conforms to the specification. The license fee goes primarily to patent holders whose inventions are used within the standard.) There may be several reference implementations if multiple groups experiment with the standard and each releases results.

The reference implementation is something of a mixed blessing for system designers. On the one hand, the reference implementation saves designers a great deal of time; on the other hand, it comes with some liabilities. Of course, learning someone else’s code is always time-consuming. Furthermore, the code generally cannot be used as-is. Reference implementations are typically written to run on large workstations with infinite memory; they are generally not designed to operate in real time. The code must often be restructured in many ways: eliminating features that will not be implemented; replacing heap allocation with custom memory management; and improving cache utilization, function, inlining, and many other tasks.

The implementer of a standard must perform several design tasks:

- The unspecified parts of the implementation must be designed.
- Parts of the system not specified by the standard (user interface, for example) must be designed.
An initial round of platform-independent optimization must be used to improve the chosen reference implementation.

- The reference implementation and other code must be profiled and analyzed.
- The hardware platform must be designed based on initial characterization.
- The system software must be further optimized to better match the platform.
- The platform itself must be further optimized based on additional profiling.
- The platform and software must be verified for conformance to the standard as well as nonfunctional parameters such as performance and energy consumption.

The next example introduces the Advanced Video Coding standard.

---

**Example 1-6**

**AVC/H.264**

The latest generation of video compression standards is known by several names. It is officially part 10 of the MPEG-4 standard, known as Advanced Video Coding (AVC). However, the MPEG group joined forces with the H.26x group, so it is also known as H.264.

The MPEG family of standards is primarily oriented toward broadcast, in which the transmitter is more complex in favor of cheaper receivers. The H.26x family of standards, in contrast, has traditionally targeted videoconferencing, in which systems must both transmit and receive, giving little incentive to trade transmitter complexity for receiver complexity.

The H.264 standard provides many features that give improved picture quality and compression ratio. H.264 codecs typically generate encoded streams that are half the size of MPEG-2 encodings. For example, the H.264 standard allows multiple reference frames so that motion estimation can use pixels from several frames to handle occlusion. This is an example of a feature that improves quality at the cost of increased receiver complexity.

The reference implementation for H.264 is more than 120,000 lines of C code; it uses a fairly simple algorithm for some unspecified parts of the standard, such as motion estimation. However, it implements both video coding and decoding, and reference implementation does so for the full range of display sizes supported by the standard, ranging from $176 \times 120$ resolution of NTSC quarter CIF (QCIF) to high-definition resolutions of $1280 \times 720$ or more.
1.4.4 Design Verification and Validation

Making sure that the implementation is correct is a critical part of any design. A variety of techniques are used in practice to ensure that the final system operates correctly.

We can distinguish between several types of activities:

- **Testing** exercises an implementation by providing stimuli and evaluating the implementation’s outputs.
- **Validation** generally refers to comparing the implementation to the initial requirements or specification.
- **Verification** may be performed at any stage of the design process and compares the design at one level of abstraction to another.

A number of different techniques are used to verify designs.

- Simulation accepts stimulus and computes the expected outputs. Simulation may directly interpret a design model, or the simulator may be compiled from the model.
- Formal methods perform some sort of proof; they require some sort of description of the property to be tested but not particular input stimuli. Formal methods may, for example, search the state space of the system to determine whether a property holds.
- Manual methods can catch many errors. Design walkthroughs, for example, are often used to identify problems during the implementation.

Verification and validation should not be performed as a final step to check the complete implementation. The design should be repeatedly verified at each level of abstraction. Design errors become more expensive to fix as they propagate through the design—allowing a bug to be carried to a more detailed level of implementation requires more engineering effort to fix the bug.

1.4.5 A Methodology of Methodologies

The design of high-performance embedded systems is not described well by simple methodologies. Given that these systems implement specifications that are millions of lines long, it should not be surprising that we have to use many different types of design processes to build complex embedded systems.

We discuss throughout this book many tools and techniques that can be built into methodologies. Quite a few of these tools are complex and require
specialized knowledge of how to use them. Methodologies that we use in embedded system design include:

- **Software performance analysis**—Executable specifications must be analyzed to determine how much computing horsepower is needed and which types of operations must be performed. We will discuss performance analysis in Section 3.4.

- **Architectural optimization**—Single processor architectures can be tuned and optimized for the application. We will discuss such methods in Chapter 3. We can also tune multiprocessor architectures, as will we discuss in Chapter 5.

- **Hardware/software co-design**—Co-design helps create efficient heterogeneous architectures. We will look at co-design algorithms and methodologies in detail in Chapter 7.

- **Network design**—Whether in distributed embedded systems or systems-on-chips, networks must provide the necessary bandwidth at reasonable energy levels. We will look at on-chip networks in Section 5.6 and multichip networks, such as those used in automobiles, in Section 5.8.

- **Software verification**—Software must be evaluated for functional correctness. We will look at software-verification techniques for concurrent systems in Section 4.5.

- **Software tool generation**—Tools to program the system must be generated from the hardware and software architectures. We will discuss compiler generation for configurable processors in Section 2.9. We will look at software generation for multiprocessors in Section 6.3.

### 1.4.6 Joint Algorithm and Architecture Development

It is important to keep in mind that algorithmic design occurs at least in part before embedded system design. Because algorithms are eventually implemented in software to be used, it is easy to confuse algorithmic design and software design. But, in fact, the design of algorithms for signal processing, networking, and so on is a very different skill than that of designing software. This book is primarily about embedded software and hardware, not algorithms. One of the goals here is to demonstrate the skills required to design efficient, compact software and to show that those skills are applicable to a broad range of algorithms.

However, it is also true that algorithm and embedded system designers need to talk more often. Algorithm designers need to understand the characteristics of their platforms in order to design implementable algorithms. Embedded system designers need to understand which types of features are needed for algorithms
in different application spaces to ensure that the systems they design are optimized for the proper characteristics.

Embedded systems architectures may be designed along with the algorithms they will execute. This is true even in standards-based systems, since standards generally allow for algorithmic enhancements. Joint algorithm/architecture development creates some special challenges for system designers.

Algorithm designers need estimates and models to help them tailor the algorithm to the architecture. Even though the architecture is not complete, the hardware architects should be able to supply estimates of performance and power consumption. These should be useful for simulators that take models of the underlying architecture.

Algorithm designers also need to be able to develop software. This requires functional simulators that run as fast as possible. If hardware were available, algorithm designers could run code at native speeds. Functional simulators can provide adequate levels of performance for many applications even if they do not run at hardware speeds. Fast turnaround of compilation and simulation is very important to successful software development.

1.5 Models of Computation

A model of computation defines the basic capabilities of an abstract computer. In the early days of computer science, models of computation helped researchers understand the basic capabilities of computers. In embedded computing, models of computation help us understand how to correctly and easily program complex systems. This section considers several models of computation and the relationships between them. The study of models of computation have influenced the way real embedded systems are designed; we balance the theory in this section with mentions of how some of these theoretical techniques have influenced embedded software design.

1.5.1 Why Study Models of Computation?

Models of computation help us understand the expressiveness of various programming languages. Expressiveness has several different aspects. On the one hand, we can prove that some models are more expressive than others—that some styles of computing can do some things that other styles cannot. But expressiveness also has implications for programming style that are at least as important for embedded system designers. Two languages that are both equally expressive, formally, may be good at different types of applications. For example, control and data are often programmed in different ways; a language can express one only with difficulty but the other easily.
Experienced programmers can think of several types of expressiveness that can be useful when writing programs.

- **Finite versus infinite state**—Some models assume that an infinite number of states can exist; other models are finite-state.

- **Control versus data**—This is one of the most basic dichotomies in programming. Although control and data are equivalent formally, we tend to think about them very differently. Many programming languages have been developed for control-intense applications such as protocol design. Similarly, many other programming languages have been designed for data-intense applications such as signal processing.

- **Sequential versus parallel**—This is another basic theme in computer programming. Many languages have been developed to make it easy to describe parallel programs in a way that is both intuitive and formally verifiable. However, programmers still feel comfortable with sequential programming when they can get away with it.

The astute reader will note that we are not concerned here with some traditional programming language issues such as modularity. While modularity and maintainability are important, they are not unique to embedded computing. Some of the other aspects of languages that we mention are more central to embedded systems that must implement several different styles of computation so that they can work together smoothly.

Expressiveness may lead to the use of more than one programming language to build a system—we call these systems **heterogeneously programmed**. When programming languages are mixed, we must satisfy the extra burden of correctly designing the communication between modules of different programming languages. Within a given language, the language system often helps verify certain basic operations, and it is much easier to think about how the program works. When we mix and match multiple languages, it is much more difficult for us to convince ourselves that the programs will work together properly. Understanding the model under which each programming language works, and the conditions under which they can reliably communicate, is a critical step in the design of heterogeneously programmed systems.

### 1.5.2 Finite versus Infinite State

The amount of state that can be described by a model is one of the most fundamental aspects of any model of computation. Early work on computability emphasized the capabilities of finite-state versus infinite-state machines; infinite state was generally considered to be good because it showed that the machine was more capable. However, finite-state models are much easier to verify in
both theory and practice. As a result, finite-state programming models have an important place in embedded computing.

Finite-state machines (FSMs) are well understood by both software and hardware designers. An example is shown in Figure 1-17. An FSM is typically defined as

$$ M = \{ I, O, S, \Delta, T \} $$

where \( I \) and \( O \) are the inputs and outputs of the machine, \( S \) is its current state, and \( \Delta \) and \( T \) are the states and transitions, respectively, of the state-transition graph. In a Moore machine, the output is a function only of \( S \), while in a Mealy machine the output is a function of both the present state and the current input.

Although there are models for asynchronous FSMs, a key feature in the development of the finite-state machine model is the notion of synchronous operation: inputs are accepted only at certain moments. Finite-state machines view time as integer-valued, not real-valued. At each input, the FSM evaluates its state and determines its next state based on the input received as well as the present state.

In addition to the machine itself, we need to model its inputs and outputs. A stream is widely used as a model of terminal behavior because it describes sequential behavior—time as ordinals, not in real values. The elements of a stream are symbols in an alphabet. The alphabet may be binary, may be in some other base, or may consist of other types of values, but the stream itself does not impose any semantics on the alphabet. A stream is a totally ordered set of symbols \( s_0, s_1, \ldots \). A stream may be finite or infinite. Informally, the time at which a symbol appears in a stream is given by its ordinality in the stream. In this equation

$$ S(t) = s_t $$

the symbol \( s_t \) is the \( t \)th element of the stream \( S \).
We can use streams to describe the input/output or terminal behavior of a finite-state machine. If we view the FSM as having several binary-valued inputs, the alphabet for the input stream will be binary numbers; in some cases, it is useful to think of the inputs as forming a group whose values are determined by a single symbol that defines the states of all the inputs. Similar thinking can be applied to the outputs. The behavior of the inputs is then described as one or more streams, depending on the alphabet used. Similarly, the output behavior is described as one or more streams. At time $i$, the FSM consumes a symbol on each of its input streams and produces a symbol on each of its output streams. The mapping from inputs to outputs is determined by the state-transition graph and the machine’s internal state. From the terminal view, the FSM is synchronous because the consumption of inputs and generation of outputs is coordinated.

Although synchronous finite-state machines may be the most familiar to hardware designers, synchronous behavior is a growing trend in the design of languages for embedded computing. Finite-state machines make interesting models for software because they can be more easily verified than infinite-state machines. Because an FSM has a finite number of states, we can visit all the states and exercise all the transitions in a finite amount of time. If a system has infinite state, we cannot visit all its states in finite time. Although it may seem impractical to walk through all the states of an FSM in practice, research over the past 20 years has led us to efficient algorithms for exploring large state spaces.

The ordered Boolean decision diagram (OBDD) [Bry86] can be used to describe combinational Boolean functions. Techniques have been developed to describe state spaces in terms of OBDDs such that, in many cases, properties of those state spaces can be efficiently checked. OBDDs do not take away the basic NP-completeness of combinational and state space search problems; in some cases the OBDDs can become very large and slow to evaluate. But in many cases they run very fast and even in pathological cases can be faster than competing methods. OBDDs allow us to perform many checks that are useful tests of the correctness of practical systems.

- **Product machines**—It is often easier to express complex functions as systems of communicating machines. However, hidden bugs may lurk in the communication between those components. Building the product of the communicating machines is the first step in many correctness checks.

- **Reachability**—Many bugs manifest themselves as inabilities to reach certain states in the machine. In some cases, unreachable states may simply describe useless but unimportant behavior. In other cases, unreachable states may signal a missing feature in the system.

**Nondeterministic FSMs**, also known as nondeterministic finite automata (NFAs), are used to describe some types of systems. An example is shown in
1.5 Models of Computation

Figure 1-18: two transitions out of state $s_1$ have the same input label. One way to think about this model is that the machine nondeterministically chooses a transition such that future inputs will cause the machine to be in the proper state; another way to think about execution is that the machine follows all possible transitions simultaneously until future inputs cause it to prune some paths. It is important to remember that nondeterministic automata are not formally more expressive than deterministic FSMs. An algorithm can transform any NFA into an equivalent deterministic machine. But NFAs can be exponentially smaller than its equivalent deterministic machine. This is a simple but clear example of the stylistic aspect of expressiveness.

**control-oriented languages**

A number of languages have been developed to describe control. Statecharts is a well-known example. We will discuss Statecharts in more detail in Section 3.5.3.

**Turing machine**

The Turing machine is the most well-known infinite-state model for computation. (Church developed his lambda calculus first, but the Turing machine more closely models the operation of practical computing machines.) As illustrated in Figure 1-19, the Turing machine itself consists of a program, a read head, and a state. The machine reads and writes a tape that has been divided into

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1-19: A Turing machine.
Chapter 1  Embedded Computing

cells, each of which contains a symbol. The tape can move back and forth underneath the head; the head can both read and write symbols on the tape. Because the tape may be of infinite length, it can describe infinite-state computations. An operating cycle of a Turing machine consists of the following steps.

1. The machine uses the head to read the symbol in the tape cell underneath the head.
2. The machine erases the symbol on the cell underneath the head.
3. The machine consults its program to determine what to do next. Based on the current state and the symbol that was read, the machine may write a new symbol and/or move the tape.
4. The machine changes its state as described by the program.

The Turing machine is a powerful model that allows us to demonstrate the capabilities and limits of computability. However, as we noted previously, finite state enables verification of many important aspects of real programs even though the basic programming model is more limited. For example, one of the key results of theoretical computer science is the halting problem—the Turing model allows us to confirm that we cannot, in general, show that an arbitrary program will halt in a finite amount of time. The failure to ensure that programs will halt makes it impossible to verify many important problems of programs on infinite-state systems. In contrast, because we can visit all the states of a finite-state system in finite time, important properties become more tractable.

1.5.3 Control Flow and Data Flow Models

Control and data are basic units of programming. Although control and data are fundamentally equivalent, we tend to think of data operations, such as arithmetic, as more regular, and control as less regular and more likely to involve state.

A basic model of control is the control flow graph (CFG), as shown in Figure 1-20. The nodes in the graph are either unconditionally executed operations (the rectangles) or conditions (the diamond). The control flow graph has a single thread of control, which can be thought of as a program counter moving through the program. The CFG is a finite-state model of computation. Many compilers model a program using a control data flow graph (CDFG), which adds data flow models that we will use to describe the operations of the unconditional nodes and the decisions in the conditional nodes.

A basic model of data is the data flow graph (DFG), an example of which is shown in Figure 1-21. Like the task graph, the data flow graph consists of nodes and directed edges, where the directed edges represent data dependencies. The nodes in the DFG represent the data operations, such as arithmetic...
1.5 Models of Computation

operations. Some edges in the DFG terminate at a node but do not start at one; these sources provide inputs. Similarly, sinks start at a node but do not terminate at one. (An alternative formulation is to provide three types of nodes: operator, input, and output.)

We require that DFGs be trees—they cannot have cycles. This makes the graphs easier to analyze but does limit their uses. Basic DFGs are commonly used in compilers.

The DFG is finite-state. It describes parallelism in that it defines only a partial order on the operations in the graph. Whether we execute those operations one at a time or several at once, any order of operations that satisfies the data dependencies is acceptable.

We can use streams to model the behavior of the DFG. Each source in the data flow graph has its own stream, and each sink of the graph is a stream as well. The nodes in the DFG use firing rules to determine their behavior. The simplest firing rule is similar to the operation of finite-state machines: firing consumes a token on each of the node’s input streams and generates one token on its output; we will call this the “standard data flow firing rule.” One way to introduce conditions into the DFG is with a conditional node with $n + 1$ terminals: data inputs $d_0, d_1, \ldots$ and control input $k$. When $k = 0$, data input $d_0$ is consumed and sent to the output; when $k = 1$, $d_1$ is consumed and transferred to the

Figure 1-20 A control flow graph.

Figure 1-21 A data flow graph.
output, and so on. In this firing rule, not all of the inputs to the node consume a token at once.

A slightly more sophisticated version of data flow is the signal flow graph (SFG), commonly used in signal processing. As shown in Figure 1-22, the signal flow graph adds a new type of node generally called a delay node. As signified by the Δ symbol, the delay node delays a stream by \( n \) (by default, one) time steps. Given a stream \( S \), the result of a delay operator is \( \Delta(t) = S(t-1) \). Edges in the SFG may be given weights that indicate that the value given to a node is to be multiplied by the weight. We also allow SFGs to have cycles. SFGs are commonly used to describe digital filters.

A more sophisticated data flow model is the synchronous data flow (SDF) model introduced by Lee and Messerschmitt [Lee87]. Synchronous data flow graphs allow feedback and provide methods for determining when a system with feedback is in fact legal. A simple SDF graph is shown in Figure 1-23. As with basic data flow graphs, nodes define operations and directed edges define the flow of data. The data flowing along the edges can be modeled as streams. Each edge has two labels: \( r_o \) describes the number of tokens produced per invocation while \( r_i \) gives the number of tokens consumed per invocation. Each edge may also be labeled with a delay \( \delta \) that describes the amount of time between when a token is produced at the source and when it is consumed at the edge; by convention the default delay is 0.

Lee and Parks [Lee95] identified the networks of Kahn processes as important models for systems of communicating processes. Kahn processes can be used to build networks of communicating processes. Based on certain properties of the Kahn process, we can determine some important properties of the network. We can prove these properties without making use of the concept of time in the wall clock sense. The inputs and outputs of a Kahn process are modeled as streams, which define sequential patterns but do not define a time base. A Kahn process is shown in Figure 1-24. The process itself is connected to its inputs by infinite-size buffers. The process maps its input streams to output

Figure 1-22 A signal flow graph.

Figure 1-23 A simple SDF graph.
1.5 Models of Computation

Streams. A process can have one or more inputs and one or more outputs. If \( X \) is a stream, then \( F(X) \) is the output of a Kahn process when given that stream. One important property of a Kahn process is monotonicity.

\[
X \in X' \Rightarrow F(X) \in F(X') .
\]

The behavior of a monotonic process is physical in the sense that adding more inputs will not cause it to mysteriously generate fewer outputs.

A network of Kahn processes, also shown in Figure 1-24, equates the input and output streams of processes in the network; for example, the output of process \( a \) is equal to the input of process \( b \) as well as the input of process \( c \). If \( I \) is the input stream to a network and \( X \) is the set of internal streams and outputs, then the fixed point behavior of the network is

\[
X = F(X, I) .
\]

Kahn showed that a network of monotonic processes is itself monotonic. This means that we can compose monotonic processes without worrying about the network becoming nonmonotonic.

### 1.5.4 Parallelism and Communication

Parallelism is a fundamental concept in computer science and of great practical importance in embedded systems. Many embedded systems perform many tasks simultaneously. The real parallelism embodied in the hardware must be matched by apparent parallelism in the programs.
We need to capture parallelism during the early stages of design so that we can use it to optimize the design. Parallel algorithms describe time as partially ordered—the exact sequence of operations is not determined up front. As we bind operations to the architecture, we move the description toward a totally ordered description (although some operations may be left partially ordered to be managed by the operating system). Different choices for ordering require different amounts of hardware resources, affecting cost and power consumption.

A simple model of parallelism is the task graph, as shown in Figure 1-25. The nodes in the task graph represent processes or tasks while the directed edges represent data dependencies. In the example, process $P_4$ must complete before $P_5$ can start. Task graphs model concurrency because sets of tasks that are not connected by data dependencies may operate in parallel. In the example, $\tau_1$ and $\tau_2$ are separate components of the graph that can run independently. Task graphs are often used to represent multirate systems. Unless we expose the computation within the processes, a task graph is less powerful than a Turing machine. The basic task graph cannot even describe conditional behavior. Several extended task graphs have been developed that describe conditions but even these are finite-state machines.

The Petri net [Mur89] is one well-known parallel model of computation. Petri nets were originally considered to be more powerful than Turing machines, but later work showed that the two are in fact equivalent. However, Petri nets explicitly describe parallelism in a way that makes some types of systems easier to specify. An example Petri net is shown in Figure 1-26. A net is a weighted, directed bipartite graph. One type of node is the place; the other type is a transition. Arcs connect places and transitions; an arc is weighted with a non-negative integer. There are no arcs between two places or two transitions.

The state of the executing system is defined by tokens that define a marking. The tokens move around the net in accordance with firing rules. In general, a place can hold zero, one, or more than one tokens.
The Petri net’s behavior is defined by a sequence of markings, each defining a state of the net. The firing rule or transition rule determines the next state given the current state. A transition is enabled if each input place of the transition is marked with at least as many tokens as are specified by the weight of each incoming arc. Enabled transitions may fire but are not required to do so. Firing removes tokens from the places that feed the transition equal to the weight of the arc from the place to the transition and adds the same number of tokens to each output place equal to the weight of the arc from the transition to the place.

Petri nets have been used to study many problems in parallel programming. They are sometimes used to write parallel programs but are not often used directly as programs. However, the notion of multiple tokens is a powerful one that serves us well in many types of programs.

Useful parallelism necessarily involves communication between the parallel components of the system. Different types of parallel models use different styles of communication. These styles can have profound implications on the efficiency of the implementation of communication. We can distinguish two basic styles of communication: buffered and unbuffered. A buffered communication assumes that memory is available to store a value if the receiving process is temporarily not ready to receive it. An unbuffered model assumes no memory in between the sender and the receiver.

Even a simple model like the FSM can exhibit parallelism and communication. Figure 1-27 shows two communicating FSMs. Each machine, $M_1$ and $M_2$, has an input from the outside world and an output to the outside world. But each has one output connected to the input of the other machine. The behavior of each machine therefore depends on the behavior of the other machine. As we
noted before, the first step in analyzing the behavior of such networks of FSMs is often to form the equivalent product machine.

Communicating FSM languages such as Esterel [Ben91] have been used for software as well as hardware. As we will see in Chapter 3, each process in an Esterel program is considered as finite-state machine, and the behavior of the system of process is determined by building the product of the component machines. Esterel has been widely used to program avionics and other critical applications.

The communicating FSMs of Figure 1-27 communicate without buffers. A buffer would correspond to a register (in hardware) or variable (in software) in between an output on one machine and the corresponding input on the other machine. However, we can implement both synchronous and asynchronous behavior using this simple unbuffered mechanism as shown in Figure 1-28. Synchronous communication simply has one machine throw a value to the other machine. In the figure, the synchronously communicating $M_1$ sends $val$ to $M_2$. 

![Figure 1-27](image) **Figure 1-27** Two communicating FSMs.

![Figure 1-28](image) **Figure 1-28** Synchronous and asynchronous communication in FSMs.
without checking whether \( M_2 \) is ready. If the machines are designed properly, this is very efficient, but if \( M_1 \) and \( M_2 \) fall out of step, then \( M_2 \) will misbehave because \( \text{val} \) is either early or late. Asynchronous communication uses a handshake. On the right side of the figure, the asynchronous \( M_1 \) first sends a ready signal, then a value. \( M_2 \) waits for the ready signal before looking for \( \text{val} \). This requires extra states but does not require that the machines move in lockstep.

Another fundamental distinction between communication methods, involves blocking versus nonblocking behavior. In blocking communication, the sending process blocks, or waits until the receiving process has the data. Nonblocking communication does not require the sender to wait for the receiver to receive the data. If there are no buffers between the sender and receiver, and the receiver is not ready, nonblocking communication will drop data. Adding a buffer allows the sender to move on even if the receiver is not ready, assuming that the buffer is not already full. An infinite-size buffer allows unlimited nonblocking communication.

A natural question in the case of buffered communication concerns the size of the buffer required. In some systems, an infinite-size buffer is required to avoid losing data. In a multirate system in which the sender may always produce data faster than the consumer, the buffer size may grow indefinitely. However, it may be possible to show that the producer cannot keep more than some finite number of elements in the buffer even in the worst case. If we can prove the size of the buffer required, we can create a less-expensive implementation. Proving that the buffer is finite also tells us that it is possible to build a system in which the buffer never overflows. As with other problems, proving buffer sizes is easier in finite-state systems.

### 1.5.5 Sources and Uses of Parallelism

One of the benefits of using a programming language or model of computation suited to a problem is that it may be easier to identify and extract opportunities for parallelism from the application. Concurrent hardware enables us to perform operations that take advantage of the independence of the operations we want to perform. Parallelism may be found at many different levels of abstraction.

**Instruction-level parallelism** is exploited by high-performance microprocessors. Instruction-level parallelism in its purest form is truly at the instruction level, such as load, stores, and ALU operations. The instruction-level operations are not visible in the source code and so cannot be manipulated by the programmer.

**Data-level parallelism** can be found in C or Fortran programs by optimizing compilers. Data parallelism on a small scale can be found in a basic block of a program. Larger-scale data parallelism can be found in a nest of loops that perform array calculations.
Task-level parallelism can be found in many applications. It is particularly important in embedded systems because these systems often perform several different types of computation on data streams. In many ways, task-level parallelism is easy to exploit since tasks can be allocated to processors. However, task structure is not easily conveyed in C; programmers often resort to programming conventions combined with calls to interprocess communication routines to define tasks. More abstract programming models may help clarify the task structure of an application.

Some types of parallelism can be found statically, by examining the program. Other opportunities can be found only dynamically, by executing the program. Static parallelism is easier to implement but does not cover all important sources of parallelism (at least, not in a Turing-complete model). Dynamic discovery of parallelism can be performed at many levels of abstraction: instruction, data/control, task, and so on.

### 1.6 Reliability, Safety, and Security

This section looks at aspects of reliable system design that are particularly important to embedded system design. Reliability, safety, and security are closely related.

- **Reliable (or dependable) system design** is concerned with making systems work even in the face of internal or external problems. Reliable system design most often assumes that problems are not caused maliciously.

- **Safety-critical system design** studies methods to make sure systems operate safely, independent of the cause of the problem.

- **Security** is concerned largely with malicious attacks.

![Figure 1-29](image-url)  
*Dependability and security as described by Avizienis et al. [Avi04].*
1.6 Reliability, Safety, and Security

Avizienis et al. [Avi04] describe the relationship between dependability and security as shown in Figure 1-29. Dependability and security are composed of several attributes:

- **Availability** of correct service
- **Continuity** of correct service
- **Safety** from catastrophic consequences on users and their environment
- **Integrity** from improper system alterations
- **Maintainability** through modifications and repairs
- **Confidentiality** of information

Embedded systems are increasingly subject to malicious attack. But whatever the source of the problem, many embedded systems must operate properly in the presence of faults.

1.6.1 Why Reliable Embedded Systems?

Certainly many embedded systems do not need to be highly reliable. Some consumer electronics devices are so inexpensive as to be nearly disposable. Many markets do not require highly reliable embedded computers. But many embedded computers must be built to be highly reliable:

- automotive electronics;
- avionics;
- medical equipment;
- critical communications.

Embedded computers may also handle critical data, such as purchasing data or medical information.

The definition of reliability can vary widely with context. Certainly, computer systems that run for weeks at a time without failing are not unknown. Telephone switching systems have been designed to be down for less than 30 seconds per year.

The study of reliable digital system design goes back several decades. A variety of architectures and methodologies have been developed to allow digital systems to operate for long periods with very low failure rates. What is the difference between the designs of traditional reliable computers and reliable embedded systems?
First, reliable embedded computers are often distributed systems. Automotive electronics, avionics, and medical equipment are all examples of distributed embedded systems that must be highly reliable. Distributed computing can work to our advantage when designing reliable systems, but distributed computers can also be very unreliable if improperly designed.

Second, embedded computers are vulnerable to many new types of attacks. Reliable computers were traditionally servers or machines that were physically inaccessible—physical security has long been a key strategy for computer security. However, embedded computers generally operate in unprotected environments. This allows for new types of faults and attacks that require new methods of protection.

### 1.6.2 Fundamentals of Reliable System Design

#### sources of faults

Reliable systems are designed to recover from faults. A fault may be permanent or transient. A fault may have many sources, some of which are the following.

- **Physical faults** are caused by manufacturing defects, radiation hazards, and so on.
- **Design faults** are the result of improperly designed systems.
- **Operational faults** come from human error, security breaches, poorly designed human–computer interfaces, and so on.

While the details of how such faults occur and how they affect the system may vary, the system’s users do not really care what caused a problem, only that the system reacted properly to it. Whether a fault comes from a manufacturing defect or a security problem, the system must react in such a way as to minimize the fault’s effect on the user.

#### system reliability metrics

Users judge systems by how reliable they are, not by the problems that cause them to fail. Several metrics are used to quantify system reliability [Sie98].

**Mean time to failure (MTTF)** is one well-known metric. Given a set of perfectly functioning systems at time 0, MTTF is the expected time for the first system in that set to fail. Although it is defined for a large set of systems, it is also often used to characterize the reliability of a single system. The mean time to failure can be calculated by

\[
MTTF = \int_{0}^{\infty} R(t)\,dt
\]

where \(R(t)\) is the reliability function of the system.
The **reliability function** of a system describes the probability that the system will operate correctly in the time period \([0, t]\). \(R(0) = 1\) and \(R(t)\) monotonically decreases with time.

The **hazard function** \(z(t)\) is the failure rate of components. For a given probability function, the hazard function is defined as

\[
z(t) = \frac{pdf}{1 - CDF}.
\]  

(Figure 1-9)

Faults can be measured empirically or modeled by a probability distribution. Empirical studies are usually the basis for choosing an appropriate probability distribution. One common model for failures is exponential distribution. In this case, the hazard function is

\[
z(t) = \lambda.
\]  

(Figure 1-10)

Another function used to model failures is the Weibull distribution:

\[
z(t) = \alpha \lambda (\lambda t)^{\alpha - 1}.
\]  

(Figure 1-11)

In this formula, \(\alpha\) is known as the shape parameter and \(\lambda\) is known as the scale parameter. The Weibull distribution normally must be solved numerically.

A distribution that is observed empirically for many hardware components is the **bathtub function** shown in Figure 1-30. The bathtub curve gets its name from its similarity to the cross-section of a bathtub. Hardware components generally show infant mortality in which marginal components fail quickly, then a long period with few failures, followed by a period of increased failures due to long-term wear mechanisms.

![Figure 1-30 A bathtub function.](image-url)
Chapter 1  Embedded Computing

actions after faults

The system can do many things after a fault. Generally several of these actions are taken in order until an action gets the system back into running condition. Actions from least to most severe include the following.

- **Fail**—All too many systems fail without trying to even detect an error.
- **Detect**—An error may be detected. Even if the system stops at this point, the diagnostic information provided by the detector can be useful.
- **Correct**—An error may be corrected. Memory errors are routinely corrected. A simple correction causes no long-term disturbance to the system.
- **Recover**—A recovery may take more time than a simple correction. For example, a correction may cause a noticeable pause in system operation.
- **Contain**—The system may take steps to ensure that a failure does not corrupt a large part of the system. This is particularly true of software or hardware failures that can, for example, cause large parts of memory to change.
- **Reconfigure**—One way to contain a fault is to reconfigure the system so that different parts of the system perform some operations. For example, a faulty unit may be disabled and another unit enabled to perform its work.
- **Restart**—Restarting the system may be the best way to wipe out the effects of an error. This is particularly true of transient errors and software errors.
- **Repair**—Either hardware or software components can be modified or replaced to repair the system.

reliability methods

Many techniques have been developed to make digital systems more reliable. Some are more applicable to hardware, others to software, and some may be used in both hardware and software.

error-correction codes

Error-correction codes were developed in the 1950s, starting with Hamming, to both detect and correct errors. They are widely used throughout digital systems to identify and correct transient or permanent errors. These codes introduce redundant information in a way such that certain types of errors can be guaranteed to be detected or corrected. For example, a code that is single-error correcting/double-error detecting can both detect and correct an error in a single bit and detect, but not correct, two bit errors.

voting systems

Voting schemes are often used to check at higher levels of abstraction. One well-known voting method is **triple modular redundancy**, illustrated in Figure 1-31. The computation unit $C$ has three copies, $C_1$, $C_2$, and $C_3$. All three units receive the same input. A separate unit compares the results generated by each input. If at least two results agree, then that value is chosen as correct by the voter. If all three results differ, then no correct result can be given.

watchdog timers

The **watchdog timer** is widely used to detect system problems. As shown in Figure 1-32, the watchdog timer is connected to a system that it watches. If the
1.6 Reliability, Safety, and Security

Watchdog timer rolls over, it generates a done signal that should be attached to an error interrupt in the system. The system should be designed so that, when running properly, it always resets the timer before it has a chance to roll over. Thus, a done signal from the watchdog timer indicates that the system is somehow operating improperly. The watchdog timer can be used to guard against a wide variety of faults.

**Design diversity** is a design methodology intended to reduce the chance that certain systematic errors creep into the design. When a design calls for several instances of a given type of module, different implementations of that module are used rather than using the same type of module everywhere. For example, a system with several CPUs may use several different types of CPUs rather than use the same type of CPU everywhere. In a triple modular redundant system, the components that produce results for voting may be of different implementations to decrease the chance that all embody the same design error.
1.6.3 Novel Attacks and Countermeasures

physical access

A key reason that embedded computers are more vulnerable than general-purpose computers is that many embedded computers are physically accessible to attackers. Physical security is an important technique used to secure the information in general-purpose systems—servers are physically isolated from potential attackers. When embedded computers with secure data are physically available, the attacker can gain a great deal more information about the hardware and software. This information not only can be used to attack that particular node but also helps the attacker develop ways to interfere with other nodes of that model.

Internet attacks

Some attacks on embedded systems are made much easier by Internet access. Many embedded systems today are connected to the Internet. Viruses can be downloaded, or other sorts of attacks can be perpetrated via the Internet. Siewiorek et al. [Sie04] argue that global volume is a key trend in reliable computing systems. They point out that hundreds of millions of networked devices are sold every year, primarily to users with little or no formal training. The combination of large numbers of devices and untrained users means that many tasks formerly performed in the privacy of a machine room must now be automated and reliably delivered to the masses and that these systems must be designed to shield against both faults and malicious attacks.

attacks on automobiles

But many devastating problems can be caused even without Internet access. Consider, for example, attacks on automobiles. Most modern cars use microprocessors to control their engines, and many other microprocessors are used throughout the car. The software in the engine controller could, for example, be changed to cause the car to stall under certain circumstances. This would be annoying or occasionally dangerous when performed on a single car. If a large number of cars were programmed to all stall at the same time, the resulting traffic accidents could cause significant harm. This sort of programmed accident is arguably worse if only some of the cars on the road have been programmed to stall.

Clearly, a stalling accident could be perpetrated on a wider scale if automobiles provided Internet access to the engine controller software. Prototype cars have demonstrated Internet access to at least part of the car’s internal network. However, Internet-enabled cars are not strictly necessary. Auto enthusiasts have reprogrammed engine controllers for more than 20 years to change the characteristics of their engines. A determined attacker could spread viruses through auto repair shops; adding Internet access to the internals of automobiles would open up attacks to a wider variety of perpetrators.

battery attack

One novel category of attack is the battery attack. This attack tries to disable the node by draining its battery. If a node is operated by a battery, the node’s power management system can be subverted by network operations. For example, pinging a node over the Internet may be enough to cause it to operate more often than intended and drain its battery prematurely.
Battery attacks are clearly threats to battery-operated devices such as cell phones and PDAs. Consider, for example, a cell phone virus that causes it to repeatedly make calls. Cell phone viruses have already been reported [Jap05]. But many other devices use batteries even though they also receive energy from the power grid. The battery may be used to run a real-time clock (as is done in many PCs) or to maintain other system states. A battery attack on this sort of device could cause problems that would not be noticed for quite some time.

**QoS attacks**

Denial-of-service attacks are well known in general-purpose systems, but real-time embedded systems may be vulnerable to quality-of-service (QoS) attacks. If the network delivers real-time data, then small delays in delivery can cause the data to be useless. If that data is used for real-time control, then those small delays can cause the system to fail. We also refer to this threat as a timing attack because it changes the real-time characteristics of the system. A QoS or timing attack is powerful because its effects are not limited to just information. The dynamics of the system being controlled help to determine the response of the system. A relatively small change in timing can cause a great deal of damage if a large, heavy, fast-moving object is being controlled.

Wood and Stankovic [Woo02] identified a number of ways, which are briefly described in the following list, to perform denial-of-service attacks on sensor networks at different levels of the network hierarchy.

- **Physical layer**—Jamming, tampering
- **Link layer**—Collision, exhaustion, unfairness
- **Network and routing layer**—Neglect and greed, horning, misdirection, black holes, authorization, probing, redundancy
- **Transport layer**—Flooding, desynchronization

**Power attack**

An example of an attack that is much more easily used against embedded computers than general-purpose computers is the power attack. Kocher et al. [Koc99] showed that measurements of the power supply current of a CPU can be used to determine a great deal about the processor’s internal activity. They developed two methods of power attacks. Simple power analysis inspects a trace manually and tries to determine the location of program actions, such as branches, based on knowledge of the power consumption of various CPU operations. Based on program actions, the attacker then deduces bits of the key. Differential power analysis uses correlation to identify actions and key bits. This attack was originally aimed at smart cards, which draw their power from the external card reader, but it can be applied to many embedded systems.

**Physical security**

In some cases, it may be possible to build tamper-resistant embedded systems. Making the electronic devices difficult to detect and analyze slows down attackers. Limiting information within a chip also helps deter attackers from revealing data.
1.7 Consumer Electronics Architectures

Consumer electronics devices are increasingly complex and rely on embedded computers to provide services that are not necessarily tied to the core function of the device. Music players may, for example, store files or perform cryptography. Consumer electronics devices may be connected into networks to make them easier to use and to provide access to audio and video data across the home. This section looks at networks used in consumer devices, then considers the challenges of integrating networks into consumer devices.

1.7.1 Bluetooth

Bluetooth is a personal area network designed to connect devices in close proximity to a person. The Bluetooth radio operates in the 2.5 GHz spectrum. Its wireless links typically operate within 2 meters, although advanced antennas can extend that range to 30 meters. A Bluetooth network can have one master and up to seven active slaves; more slaves can be parked for a total of 255. Although its low-level communication mechanisms do require master–slave synchronization, the higher levels of Bluetooth protocols generally operate as a peer-to-peer network, without masters or slaves.

Figure 1-33 shows the transport group protocols, which belong to layers 1 and 2 of the OSI model.

- The physical layer provides basic radio functions.
- The baseband layer defines master–slave relationships and frequency hopping.
- The link manager provides mechanisms for negotiating link properties such as bandwidth and quality-of-service.
- The logical link control and adaptation protocol (L2CAP) hides baseband-layer operations such as frequency hopping.

The Bluetooth radio transmits using frequency-hopping spread spectrum, which allows several radios to operate in the same frequency band without interference. The band is divided into 79 channels that are 1 MHz wide; the Bluetooth radio hops between these frequencies at a rate of 1,600 hops/per second. The radio’s transmission power can also be controlled.

The baseband layer chooses frequency hops according to a pseudorandom sequence that is agreed on by the radios; it also controls the radio signal strength to ensure that the receiver can properly decode the spread spectrum signal. The baseband layer also provides medium access control, determining packet types
1.7 Consumer Electronics Architectures

The link manager builds on the baseband layer to provide several functions. It schedules transmissions, choosing which data packets to send next. Transmission scheduling takes QoS contracts into account. It manages overall power consumption. The link manager also manages security and encrypts transmissions as specified.

The L2CAP layer serves as an interface between general-purpose protocols to the lower levels of the Bluetooth stack. Primarily, it provides asynchronous transmissions. It also allows higher layers to exchange QoS information.

The middleware group protocol provides several widely used protocols. It provides a serial port abstraction for general-purpose communication and protocols to interoperate with IrDA infrared networks. It provides a service discovery protocol. In addition, because Bluetooth is widely used for telephone headsets, it provides a telephony control protocol.

The Bluetooth serial interface is known as RFCOMM. It multiplexes several logical serial communications onto the radio channel. Its signals are compatible with the traditional RS-232 serial standard. It provides several enhancements, including remote status and configuration, specialized status and configuration, and connection establishment and termination. RFCOMM can emulate a serial port within a single device to provide seamless data service whether the ends of the link are on the same processor or not.

The service discovery protocol allows a Bluetooth client device to determine whether a server device in the network provides a particular service. Services are defined by service records, which consists of a set of <ID,value> attributes. All service records include a few basic attributes such as class and protocol stack information. A service may define its own attributes, such as capabilities. To discover a service, the client asks a server for a type of service. The server then responds with a service record.

**Figure 1-33** Bluetooth transport protocols.

<table>
<thead>
<tr>
<th>Logical link control and adaptation (L2CAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Manager</td>
</tr>
<tr>
<td>Baseband</td>
</tr>
<tr>
<td>Physical</td>
</tr>
</tbody>
</table>

and processing. It controls the radio power, provides a real-time clock, and provides basic security algorithms.
Chapter 1  Embedded Computing

1.7.2 WiFi

The WiFi family of standards (http://grouper.ieee.org/groups/802/11, http://www.wi-fi.org) provides wireless data communication for computers and other devices. WiFi is a family of standards known as 802.11 from the IEEE 802 committee. The original 802.11 specification was approved in 1997. An improved version of it, known as 802.11b, was presented in 1999. This standard used improved encoding methods to increase the standard’s bandwidth. Later standards include 802.11a, which provided substantially wider bandwidth, and 802.11g, which extended 802.11b. Table 1-1 compares the properties of these networks.

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Band</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.11b</td>
<td>11 Mbps</td>
</tr>
<tr>
<td>802.11a</td>
<td>54 Mbps</td>
</tr>
<tr>
<td>802.11g</td>
<td>802.11g</td>
</tr>
</tbody>
</table>

Table 1-1 802.11 specifications

Full-duplex communication requires two radios, one for each direction. Some devices use only one radio, which means that a device cannot transmit and receive simultaneously.

1.7.3 Networked Consumer Devices

Networked consumer devices have been proposed for a variety of functions, particularly for home entertainment. These systems have not yet entirely fulfilled their potential. A brief survey helps us understand the challenges in building such systems.

Figure 1-34 shows a typical organization of an entertainment-oriented home network.

- The PC acts as a server for file storage of music, images, movies, and so on. Today’s disk drives are large enough to hold substantial amounts of music or images.
- Some devices may be permanently attached to the PC. For example, the USB port can be used to send audio to an audio receiver for high-quality amplification.
Mobile devices, such as portable music players, can dock with the PC through a base. The PC is used to manage the device.

Other devices can connect via wireless links. They may connect to the server or to each other. For example, digital video recorders may have their own storage and stream video to other devices.

Several companies have proposed home server devices for audio and video. Built from commodity hardware, these servers provide specialized interfaces for detailing with media. They may also include capture subsystems, such as DVD drives, for reading movies from DVDs.

A key challenge in the design of home entertainment networks is configurability. Consumers do not want to spend time configuring their components for operation on their networks; in many cases, the devices do not have keyboards so configuration is very difficult. Many levels of the network hierarchy must be configured. Clearly, physical and link-level parameters must be configured. But another important aspect of configuration is service discovery and configuration. Each device added to the network must be able to determine which other devices it can work with, which services it can get from other devices, and which services it needs to provide to the rest of the network. Service discovery protocols help devices adapt to the network.

Java has been used to develop middleware for consumer networks. Java can be efficiently executed on a number of different platforms, which not only simplifies software development but also enables devices to trade Java code to provide services.
1.7.4 High-Level Services

Today’s consumer devices must provide many advanced services. Some of the services were originally developed for the Internet and are now being pushed down into small devices. Other services are unique to consumer applications.

**Service discovery** mechanisms have been used on the Internet for quite some time and are now used to simplify the configuration of networks of consumer devices. A service discovery mechanism is a protocol and a data schema. The data schema is used by devices on the network to describe the services they provide. The protocol allows devices to search for nodes on the network that provide the services they desire.

The next example describes the Jini service discovery protocol.

---

**Example 1-7**

*Jini*

Jini ([http://www.jini.org](http://www.jini.org)) [Shi05] is a Java-based network service discovery system. Services are defined by a Java interface type. As such, it need not define an explicit communication protocol.

Jini **lookup services** hold **service proxies** that describe services available on the network. Clients can download these service proxies. The Jini discovery protocol is the method by which Jini services and clients communicate. A service uses the **join protocol** to add itself to the lookup service. It may use either multicast or unicast, depending on the network environment. Generally multicast is used in local area networks and unicast is used in wide area networks.

The service proxy is an implementation of the Java interface type for a particular service. It provides the graphical user interface, the communication protocol necessary to talk to the device, and the device driver.

A client obtains a **lease** for a given service. A lease may be exclusive or shared. The lease is a loose contract that provides the service to the client for a given time period. The timeout implicit in a lease allows the network to recover from a variety of problems.

---

**Digital rights management** (DRM) is a newer type of service that has evolved for multimedia applications. DRM may be used on PCs or consumer devices; in some cases, a PC is used to manage rights for a handheld device. **DRM** is a protocol that enforces the terms of a license agreement. Because digital media can be copied, copyright owners and media companies have demanded that certain steps be used to limit the ways in which music, movies, and other
types of copyrighted material are used in computer systems. A copyrighted work may be sold with a number of restrictions: how many times it can be played, how many machines it can be played on, expiration date, and so on. DRM determines the rights associated with a copyrighted work and enforces those rights on the device.

Digital rights management makes use of cryptography but is itself much more than cryptography. Once an encrypted work has been decrypted, it can be used and modified freely. The DRM system may encrypt both the content and its associated rights information, but it enforces those rights throughout the life of the content and the device on which it is used.

Figure 1-35 illustrates the architecture of a DRM system. The content provider maintains a license server as well as the infrastructure necessary to distribute the copyrighted material itself. A digital rights management module is installed in each media device. The DRM module communicates with the license server to obtain the rights associated with a particular piece of content. The DRM module also communicates with the subsystems in the media device that use the content to ensure that the available rights are enforced.

The next example describes the Windows Media Rights Manager.

---

Example 1-8

*Windows Media Rights Manager*

Windows Media Rights Manager is used in Windows Media Player and other multimedia software to enforce digital rights management.

A copyrighted work to be used with Windows Media Rights Manager is packaged by encrypting the content. The key is contained in a license object that is
encrypted and separately distributed. A license server authenticates requests for licenses, perhaps by checking for payment or other permissions. The license request may be triggered automatically when the content is first requested to be played, or the license may be requested manually.

Licenses provided by the Windows Media Rights Manager can specify a number of conditions on use, such as expiration date, number of plays, whether the content can be played on a Secure Digital Music Initiative-compliant player, and whether the license can be backed up and restored.

### 1.8 Summary and a Look Ahead

The designers of high-performance embedded computing systems are required to master several skills. They must be expert at system specification, not only in the informal sense but also in creating executable models. They must understand the basic architectural techniques for both hardware and software. They must be able to analyze performance and energy consumption for both hardware and software. And they must be able to make trade-offs between hardware and software at all stages in the design process.

The rest of this book will proceed roughly bottom-up from simpler components to complex systems. Chapters 2 through 4 concentrate on single processor systems:

- Chapter 2 will cover CPUs (including the range of microarchitectures available to embedded system designers), processor performance, and power consumption.
- Chapter 3 will look at programs, including languages and design and how to compile efficient executable versions of programs.
- Chapter 4 will study real-time scheduling and operating systems.

Chapters 5 through 7 concentrate on problems specific to multiprocessors:

- In Chapter 5 we will introduce a taxonomy of multiprocessor hardware architectures and the sorts of multiprocessor structures that are useful in optimizing embedded system designs.
- In Chapter 6 we will look at software for multiprocessors.
- Chapter 7 will describe methods for hardware/software co-design, which produces application-specific multiprocessors.
What We Have Learned

- Many embedded computing systems are based on standards. The form in which the standard is expressed affects the methodology used to design the embedded system.
- Embedded systems are vulnerable to new types of security and reliability threats. Embedded computers that perform real-time control pose particular concerns.

Further Reading


Questions

Q1-1 What are the essential characteristics of embedded computing systems?
Q1-2 Using commercially available products, give two examples of the following.
   a. Embedded hardware architectures.
   b. Embedded software architectures.
Q1-3 Which part of a digital radio provides the most exploitable parallelism: the demodulator or the error corrector? Explain.
Q1-4 Can Turbo codes be decoded on a hard real-time deadline? Explain.
Q1-5 Draw a data flow diagram for a 4-input, 4-output DCT computation.
Q1-6 A video compressor performs motion estimation on 16 × 16 macroblocks; the search field is 31 pixels vertically and 41 pixels horizontally.
   a. If we search every point in the search area, how many SAD operations must we perform to find the motion vector for one macroblock?
   b. If we search 16 points in the search area, how many SAD operations must we perform to find the motion vector for one macroblock?
Q1-7 Which is more important in an embedded system: throughput or latency? Explain.
Q1-8  Are hardware designer productivity and software designer productivity related in an embedded system design? Explain.

Q1-9  Why was the spiral development model considered an improvement on the waterfall model?

Q1-10 What are the important characteristics of:
   a. A software design methodology for embedded computing systems.
   b. A hardware design methodology.
   c. A complete hardware/software methodology.

Q1-11 If we use a reference implementation of a standard as the starting point for the implementation of our embedded system for that standard, can we also use the reference implementation to verify the behavior of our design? Explain.

Q1-12 What are the essential properties of a data flow graph?

Q1-13 Is it possible to describe the behavior of a DAG-style data flow graph as a finite-state machine? Explain.

Q1-14 What are the essential properties of a Petri net?

Q1-15 A pair of processes communicate via a fixed-size buffer. How would you verify that the programs never overflow the buffer? Explain.

Q1-16 Describe how each of the dependability and security attributes given in Figure 1-29 apply to:
   a. An automotive electronics system.
   b. A cell phone.

Q1-17 Plot computation versus communication energy in a wireless network. State your assumptions.
   a. Determine the computation and communication required for a node to receive two 16-bit integers and multiply them together.
   b. Plot total system energy as a function of computation versus communication energy.

Q1-18 Modify (EQ 1-8) to compute the MTTF of an embedded computing network with \( n \) nodes and a communication bus.

Q1-19 Which of the actions after faults described in Section 1.6.2 apply to a CD player that has failed to properly read a section of a CD?

Q1-20 What sorts of embedded systems are vulnerable to battery attacks?

Q1-21 You are designing a portable media player that must include a digital rights management module. Should you implement the DRM functions on a general-purpose processor or on a specialized cryptoprocessor? Explain.
Lab Exercises

**L1-1** Select a device of your choice and determine whether it uses embedded computers. Determine, to the extent possible, the internal hardware architecture of the device.

**L1-2** Estimate the amount of code included in a device with an embedded computer.

**L1-3** How much computation must be done to demodulate a cdma2000 signal?

**L1-4** How much computation must be done to decompress an MPEG-4 video signal at QCIF (176 × 120) resolution?

**L1-5** Identify an example embedded computing system that could be described using two different communicating models of computation.

**L1-6** Develop a system architecture for an Internet-enabled home heating and air conditioning system that is designed to prevent attacks. Consider both Internet attacks and physical attacks.