SECTION

I

Introduction to Embedded Systems

This Page Intentionally Left Blank

Introduction to Embedded Systems

The field of embedded systems is wide and varied, and it is difficult to pin down exact definitions or descriptions. However, Chapter 1 introduces a useful model that can be applied to any embedded system. This model is introduced as a means for the reader to understand the major components that make up different types of electronic devices, regardless of their complexity or differences. Chapter 2 introduces and defines the common standards adhered to when building an embedded system. Because this book is an overview of embedded systems architecture, covering every possible standards-based component that could be implemented is beyond its scope. Therefore, significant examples of current standards-based components were selected, such as networking and Java, to demonstrate how standards define major components in an embedded system. The intention is for the reader to be able to use the methodology behind the model, standards, and real-world examples to understand any embedded system, and to be able to apply any other standard to an embedded system's design.

This Page Intentionally Left Blank

CHAPTER 1

A Systems Engineering Approach to Embedded Systems Design

In This Chapter

- *Define embedded system*
- *Introduce the design process*
- *Define an embedded systems architecture*
- *Discuss the impact of architecture*
- *Summarize the remaining sections of the book*

1.1 What Is an Embedded System?

An embedded system is an applied computer system, as distinguished from other types of computer systems such as personal computers (PCs) or supercomputers. However, you will find that the definition of "embedded system" is fluid and difficult to pin down, as it constantly evolves with advances in technology and dramatic decreases in the cost of implementing various hardware and software components. In recent years, the field has outgrown many of its traditional descriptions. Because the reader will likely encounter some of these descriptions and definitions, it is important to understand the reasoning behind them and why they may or may not be accurate today, and to be able to discuss them knowledgeably. Following are a few of the more common descriptions of an embedded system:

- *Embedded systems are more limited in hardware and/or software functionality than a personal computer (PC).* This holds true for a significant subset of the embedded systems family of computer systems. In terms of hardware limitations, this can mean limitations in processing performance, power consumption, memory, hardware functionality, and so forth. In software, this typically means limitations relative to a PC—fewer applications, scaled-down applications, no operating system (OS) or a limited OS, or less abstraction-level code. However, this definition is only partially true today as boards and software typically found in PCs of past and present have been repackaged into more complex embedded system designs.
- *An embedded system is designed to perform a dedicated function.* Most embedded devices are primarily designed for one specific function. However, we now see devices such as personal data assistant (PDA)/cell phone hybrids, which are embedded systems designed to be able to do a variety of primary functions. Also, the latest digital TVs include interactive applications that perform a wide variety of general

functions unrelated to the "TV" function but just as important, such as e-mail, web browsing, and games.

- *An embedded system is a computer system with higher quality and reliability requirements than other types of computer systems.* Some families of embedded devices have a very high threshold of quality and reliability requirements. For example, if a car's engine controller crashes while driving on a busy freeway or a critical medical device malfunctions during surgery, very serious problems result. However, there are also embedded devices, such as TVs, games, and cell phones, in which a malfunction is an inconvenience but not usually a life-threatening situation.
- *Some devices that are called embedded systems, such as PDAs or web pads, are not really embedded systems.* There is some discussion as to whether or not computer systems that meet some, but not all of the traditional embedded system definitions are actually embedded systems or something else. Some feel that the designation of these more complex designs, such as PDAs, as embedded systems is driven by nontechnical marketing and sales professionals, rather than engineers. In reality, embedded engineers are divided as to whether these designs are or are not embedded systems, even though currently these systems are often discussed as such among these same designers. Whether or not the traditional embedded definitions should continue to evolve, or a new field of computer systems be designated to include these more complex systems will ultimately be determined by others in the industry. For now, since there is no new industry-supported field of computer systems designated for designs that fall in between the traditional embedded system and the general-purpose PC systems, this book supports the evolutionary view of embedded systems that encompasses these types of computer system designs.

Electronic devices in just about every engineering market segment are classified as embedded systems (see Table 1-1). In short, outside of being "types of computer systems," the only specific characterization that continues to hold true for the wide spectrum of embedded system devices is that *there is no single definition reflecting them all*.

Table 1-1: Examples of embedded systems and their markets [1-1]

Market	Embedded Device
Automotive	Ignition System
	Engine Control
	Brake System (i.e., Antilock Braking System)
Consumer Electronics	Digital and Analog Televisions
	Set-Top Boxes (DVDs, VCRs, Cable Boxes, etc.)
	Personal Data Assistants (PDAs)
	Kitchen Appliances (Refrigerators, Toasters, Microwave Ovens)
	Automobiles
	Toys/Games
	Telephones/Cell Phones/Pagers
	Cameras
	Global Positioning Systems (GPS)

Table 1-1: Examples of embedded systems and their markets [1-1] (continued)

Market	Embedded Device
Industrial Control	Robotics and Control Systems (Manufacturing)
Medical	Infusion Pumps
	Dialysis Machines
	Prosthetic Devices
	Cardiac Monitors
Networking	Routers
	Hubs
	Gateways
Office Automation	Fax Machine
	Photocopier
	Printers
	Monitors
	Scanners

1.2 Embedded Systems Design

When approaching embedded systems architecture design from a systems engineering point of view, several models can be applied to describe the cycle of embedded system design. Most of these models are based upon one or some combination of the following development models:[1-5]

- The *big-bang* model, in which there is essentially no planning or processes in place before and during the development of a system.
- The *code-and-fix* model, in which product requirements are defined but no formal processes are in place before the start of development.
- The *waterfall* model, in which there is a process for developing a system in steps, where results of one step flow into the next step.
- The *spiral* model, in which there is a process for developing a system in steps, and throughout the various steps, feedback is obtained and incorporated back into the process.

This book supports the model shown in Figure 1-1, which I refer to as the Embedded Systems Design and Development Lifecycle Model. This model is based on a combination of the popular waterfall and spiral industry models.[1-2] When I investigated and analyzed the many successful embedded projects that I have been a part of or had detailed knowledge about over the years, and analyzed the failed projects or those that ran into many difficulties meeting technical and/or business requirements, I concluded that the successful projects contained at least one common factor that the problem projects lacked. This factor is the process shown in Figure 1-1, and this is why I introduce this model as an important tool in understanding an embedded system's design process.

As shown in Figure 1-1, the embedded system design and development process is divided into four phases: creating the architecture, implementing the architecture, testing the system, and

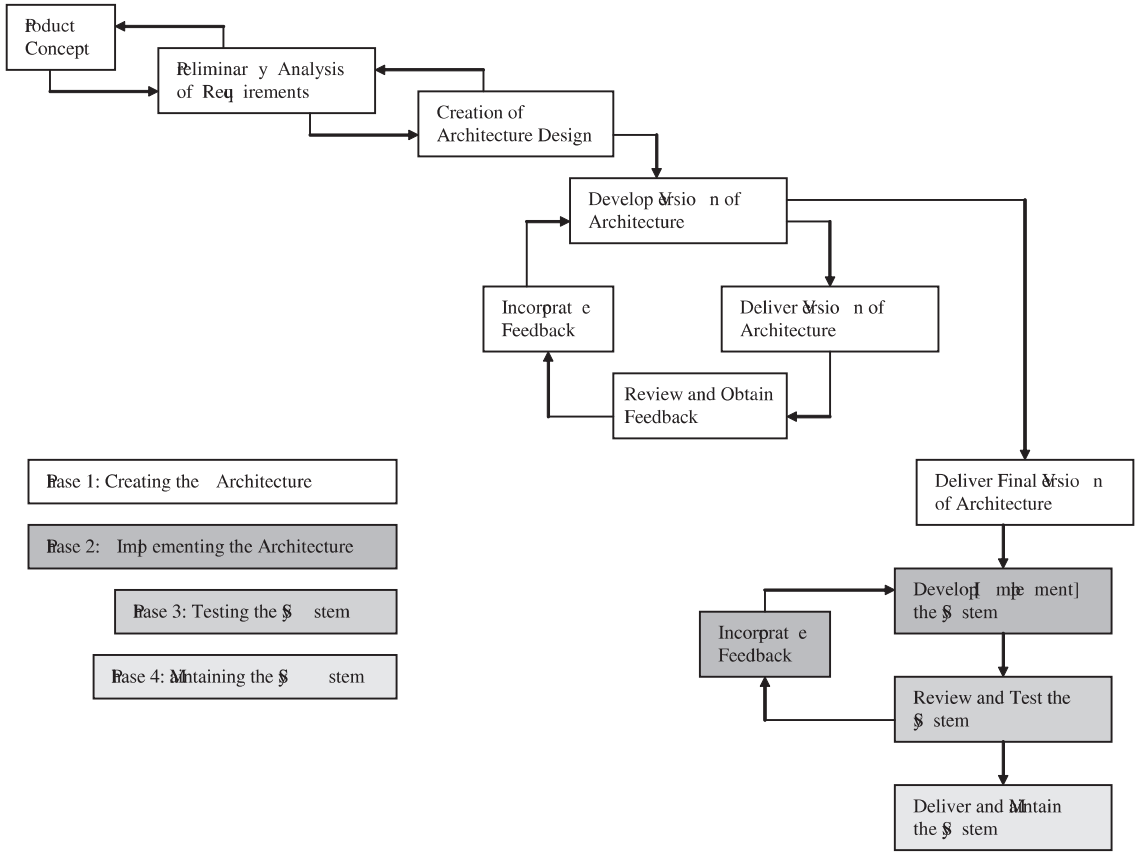


Figure 1-1: Embedded Systems Design and Development Lifecycle Model [1-2]

maintaining the system. Most of this book is dedicated to discussing phase 1, and the rest of *this* chapter is dedicated to discussing why so much of this book has been devoted to creating an embedded system's architecture.

Within this text, phase 1 is defined as being made up of six stages: having a strong technical foundation (stage 1), understanding the Architectural Business Cycle (stage 2), defining the architectural patterns and models (stage 3), defining the architectural structures (stage 4), documenting the architecture (stage 5), and analyzing and reviewing the architecture (stage 6)[1-3]. Chapters 2–10 focus on providing a strong technical foundation for understanding the major components of an embedded system design. Chapter 11 discusses the remaining stages of phase 1, and Chapter 12 introduces the last three phases.

1.3 An Introduction to Embedded Systems Architecture

The ***architecture*** of an embedded system is an *abstraction* of the embedded device, meaning that it is a generalization of the system that typically doesn't show detailed implementation information such as software source code or hardware circuit design. At the architectural level, the hardware and software components in an embedded system are instead represented as some composition of interacting *elements*. Elements are representations of hardware and/or software whose implementation details have been abstracted out, leaving only behavioral and inter-relationship information. Architectural elements can be internally integrated within the embedded device, or exist externally to the embedded system and interact with internal elements. In short, an embedded architecture includes elements of the embedded system, elements interacting with an embedded system, the properties of each of the individual elements, and the interactive relationships between the elements.

Architecture-level information is physically represented in the form of *structures*. A structure is one possible representation of the architecture, containing its own set of represented elements, properties, and inter-relationship information. A structure is therefore a "snapshot" of the system's hardware and software at design time and/or at run-time, given a particular environment and a given set of elements. Since it is very difficult for one "snapshot" to capture all the complexities of a system, an architecture is typically made up of more than one structure. All structures within an architecture are inherently related to each other, and it is the *sum* of all these *structures* that is the embedded *architecture* of a device. Table 1-2 summarizes some of the most common structures that can make up embedded architectures, and shows generally what the elements of a particular structure represent and how these elements interrelate. While Table 1-2 introduces concepts to be defined and discussed later, it also demonstrates the wide variety of architectural structures available to represent an embedded system. Architectures and their structures—how they interrelate, how to create an architecture, and so on—will be discussed in more detail in Chapter 11.

Table 1-2: Examples of architectural structures [1-4]

Structure Types*			Definition
Module			Elements (referred to as modules) are defined as the different functional components (the essential hardware and/or software that the system needs to function correctly) within an embedded device. Marketing and sales architectural diagrams are typically represented as modular structures, since software or hardware is typically packaged for sale as modules (i.e., an operating system, a processor, a JVM, and so on).
	Uses (also referred to as subsystem and component)		A type of modular structure representing system at runtime in which modules are inter-related by their usages (what module uses what other module, for example).
		Layers	A type of Uses structure in which modules are organized in layers (i.e., hierarchical) in which modules in higher layers use (require) modules of lower layers.
		Kernel	Structure presents modules that use modules (services) of an operating system kernel or are manipulated by the kernel.
		Channel Architecture	Structure presents modules sequentially, showing the module transformations through their usages.
		Virtual Machine	Structure presents modules that use modules of a virtual machine.
	Decomposition		A type of modular structure in which some modules are actually subunits (decomposed units) of other modules, and inter-relations are indicated as such. Typically used to determine resource allocation, project management (planning), data management (encapsulation, privitization, etc.).
	Class (also referred to as generalization)		This is a type of modular structure representing software and in which modules are referred to as classes, and inter-relationships are defined according to the object-oriented approach in which classes are inheriting from other classes, or are actual instances of a parent class (for example). Useful in designing systems with similar foundations.
Component and Connector			These structures are composed of elements that are either components (main hw/sw processing units, such as processors, a Java Virtual Machine, etc.) or connectors (communication mechanism that inter-connects components, such as a hw bus, or sw OS messages, etc.).
	Client/Server (also referred to as distribution)		Structure of system at runtime where components are clients or servers (or objects), and connectors are the mechanisms used (protocols, messages, packets, etc.) used to intercommunicate between clients and servers (or objects).
	Process (also referred to as communicating processes)		This structure is a SW structure of a system containing an operating system. Components are processes and/or threads (see Chapter 9 on OSes), and their connecters are the inter-process communication mechanisms (shared data, pipes, etc.) Useful for analyzing scheduling and performance.
	Concurrency and Resource		This structure is a runtime snap shot of a system containing an OS, and in which components are connected via threads running in parallel (see Chapter 9, Operating Systems). Essentially, this structure is used for resource management and to determine if there are any problems with shared resources, as well as to determine what sw can be executed in parallel.
		Interrupt	Structure represents the interrupt handling mechanisms in system.
		Scheduling (EDF, priority, round-robin)	Structure represents the task scheduling mechanism of threads demonstrating the fairness of the OS scheduler.
	Memory		This runtime representation is of memory and data components with the memory allocation and deallocation (connector) schemes—essentially the memory management scheme of the system.
		Garbage Collection	This structure represents the garbage allocation scheme (more in Chapter 2).
		Allocation	This structure represents the memory allocation scheme of the system (static or dynamic, size, and so on).
	Safety and Reliability		This structure is of the system at runtime in which redundant components (hw and sw elements) and their intercommunication mechanisms demonstrate the reliability and safety of a system in the event of problems (its ability to recover from a variety of problems).
Allocation			A structure representing relationships between sw and/or hw elements, and external elements in various environments.
	Work Assignment		This structure assigns module responsibility to various development and design teams. Typically used in project management.
	Implementation		This is a sw structure indicating where the sw is located on the development system's file system.
	Deployment		This structure is of the system at runtime where elements in this structure are hw and sw, and the relationship between elements are where the sw maps to in the hardware (resides, migrates to, etc).

** Note that in many cases the terms "architecture" and "structure" (one snapshot) are sometimes used interchangeably, and this will be the case in this book.*

1.4 Why Is the Architecture of an Embedded System Important?

This book uses an architectural systems engineering approach to embedded systems because it is one of the most powerful tools that can be used to understand an embedded systems design or to resolve challenges faced when designing a new system. The most common of these challenges include:

- defining and capturing the design of a system
- cost limitations
- determining a system's integrity, such as reliability and safety
- working within the confines of available elemental functionality (i.e., processing power, memory, battery life, etc.)
- marketability and sellability
- deterministic requirements

In short, an embedded systems architecture can be used to resolve these challenges early in a project. Without defining or knowing any of the internal implementation details, the architecture of an embedded device can be the first tool to be analyzed and used as a high-level blueprint defining the infrastructure of a design, possible design options, and design constraints. What makes the architectural approach so powerful is its ability to informally and quickly communicate a design to a variety of people with or without technical backgrounds, even acting as a foundation in planning the project or actually designing a device. Because it clearly outlines the requirements of the system, an architecture can act as a solid basis for analyzing and testing the quality of a device and its performance under various circumstances. Furthermore, if understood, created, and leveraged correctly, an architecture can be used to accurately estimate and reduce costs through its demonstration of the risks involved in implementing the various elements, allowing for the mitigation of these risks. Finally, the various structures of an architecture can then be leveraged for designing future products with similar characteristics, thus allowing design knowledge to be reused, and leading to a decrease of future design and development costs.

By using the architectural approach in this book, I hope to relay to the reader that **defining and understanding the architecture of an embedded system is an essential component of good system design**. This is because, in addition to the benefits listed above:

1. *Every embedded system has an architecture, whether it is or is not documented, because every embedded system is composed of interacting elements (whether hardware or software). An architecture by definition is a set of representations of those elements and their relationships. Rather than having a faulty and costly architecture forced on you by **not** taking the time to define an architecture before starting development, take control of the design by defining the architecture first.*
2. *Because an embedded architecture captures various views, which are representations of the system, it is a useful tool in understanding **all** of the major elements, why each component is there, and why the elements behave the way they do. None of the*

elements within an embedded system works in a vacuum. Every element within a device interacts with some other element in some fashion. Furthermore, externally visible characteristics of elements may differ given a different set of other elements to work with. Without understanding the "whys" behind an element's provided functionality, performance, and so on, it would be difficult to determine how the system would behave under a variety of circumstances in the real world.

Even if the architectural structures are rough and informal, *it is still better than nothing*. As long as the architecture conveys in some way the critical components of a design and their relationships to each other, it can provide project members with key information about whether the device can meet its requirements, and how such a system can be constructed successfully.

1.5 The Embedded Systems Model

Within the scope of this book, a variety of architectural structures are used to introduce technical concepts and fundamentals of an embedded system. I also introduce emerging architectural tools (i.e., reference models) used as the foundation for these architectural structures. At the highest level, the primary architectural tool used to introduce the major elements located within an embedded system design is what I will refer to as the Embedded Systems Model, shown in Figure 1-2.

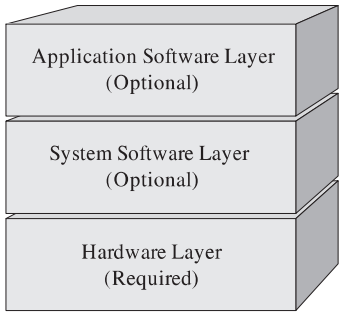


Figure 1-2: Embedded Systems Model

What the Embedded Systems Model indicates is that all embedded systems share one similarity at the highest level; that is, they all have at least one layer (hardware) or all layers (hardware, system software and application software) into which all components fall. The hardware layer contains all the major physical components located on an embedded board, whereas the system and application software layers contain all of the software located on and being processed by the embedded system.

This reference model is essentially a layered (modular) representation of an embedded systems architecture from which a modular architectural structure can be derived. Regardless of the differences between the devices shown in Table 1-1, it is possible to understand the architecture of all of these systems by visualizing and grouping the components within these devices as ***layers***. While the concept of layering isn't unique to embedded system design (architectures are relevant to all computer systems, and an embedded system is a type

of computer system), it is a useful tool in visualizing the possible combinations of hundreds, if not thousands, of hardware and software components that can be used in designing an embedded system. In general, I selected this modular representation of embedded systems architecture as the primary structure for this book for two main reasons:

1. *The visual representation of the main elements and their associated functions.* The layered approach allows readers to visualize the various components of an embedded system and their interrelationship.
2. *Modular architectural representations are typically the structures leveraged to structure the entire embedded project.* This is mainly because the various modules (elements) within this type of structure are usually functionally independent. These elements also have a higher degree of interaction, thus separating these types of elements into layers improves the structural organization of the system without the risk of oversimplifying complex interactions or overlooking required functionality.

Sections 2 and 3 of this book define the major modules that fall into the layers of the Embedded Systems Model, essentially outlining the major components that can be found in most embedded systems. Section 4 then puts these layers together from a design and development viewpoint, demonstrating to the reader how to apply the technical concepts covered in previous chapters along with the architectural process introduced in this chapter. Throughout this book, real-world suggestions and examples are provided to present a pragmatic view of the technical theories, and as the key teaching tool of embedded concepts. As you read these various examples, in order to gain the maximum benefits from this text and to be able to apply the information provided to future embedded projects, I recommend that the reader note:

- *the patterns that all these various examples follow,* by mapping them not only to the technical concepts introduced in the section, but ultimately to the higher-level architectural representations. These patterns are what can be universally applied to understand or design any embedded system, regardless of the embedded system design being analyzed.
- *where the information came from*. This is because valuable information on embedded systems design can be gathered from a variety of sources, including the internet, articles from embedded magazines, the Embedded Systems Conference, data sheets, user manuals, programming manuals, and schematics—to name just a few.

1.6 Summary

This chapter began by defining what an embedded system is, including in the definition the most complex and recent innovations in the market. It then defined what an embedded systems architecture is in terms of the sum of the various representations (structures) of a system. This chapter also introduced why the architectural approach is used as the approach to introducing embedded concepts in this book, because it presents a clear visual of what the system is, or could be, composed of and how these elements function. In addition, this approach can provide early indicators into what may and may not work in a system, and possibly improve the integrity of a system and lower costs via reusability.

The next chapter contains the first real-world examples of the book in reference to how industry standards play into an embedded design. Its purpose is to show the importance of knowing and understanding the standards associated with a particular device, and leveraging these standards to understand or create an architecture.

Chapter 1 Problems

1. Name three traditional or not-so-traditional definitions of embedded systems.

2. In what ways do traditional assumptions apply and not apply to more recent complex embedded designs? Give four examples.

3. [T/F] Embedded systems are all:
   A. medical devices.
   B. computer systems.
   C. very reliable.
   D. All of the above.
   E. None of the above.

4. [a] Name and describe five different markets under which embedded systems commonly fall.
   [b] Provide examples of four devices in each market.

5. Name and describe the four development models which most embedded projects are based upon.

6. [a] What is the Embedded Systems Design and Development Lifecycle Model [draw it]?
   [b] What development models is this model based upon?
   [c] How many phases are in this model?
   [d] Name and describe each of its phases.

7. Which of the stages below is not part of creating an architecture, phase 1 of the Embedded Systems Design and Development Lifecycle Model?
   A. Understanding the architecture business cycle.
   B. Documenting the architecture.
   C. Maintaining the embedded system.
   D. Having a strong technical foundation.
   E. None of the above.

8. Name five challenges commonly faced when designing an embedded system.

9. What is the architecture of an embedded system?

10. [T/F] Every embedded system has an architecture.

11. [a] What is an element of the embedded system architecture?
    [b] Give four examples of architectural elements.

12. What is an architectural structure?

13. Name and define five types of structures.

14. [a] Name at least three challenges in designing embedded systems.
    [b] How can an architecture resolve these challenges?

15. [a] What is the Embedded Systems Model?
    [b] What structural approach does the Embedded Systems Model take?
    [c] Draw and define the layers of this model.
    [d] Why is this model introduced?

16. Why is a modular architectural representation useful?

17. All of the major elements within an embedded system fall under:
    A. The Hardware Layer.
    B. The System Software Layer.
    C. The Application Software Layer.
    D. The Hardware, System Software, and Application Software Layers.
    E. A or D, depending on the device.

18. Name six sources that can be used to gather embedded systems design information.

CHAPTER 2

Know Your Standards

In This Chapter

- *Defining the meaning of standards*
- *Listing examples of different types of standards*
- *Discussing the impact of programming language standards on the architecture*
- *Discussing the OSI model and examples of networking protocols*
- *Using digital TV as an example that implements many standards*

Some of the most important components within an embedded system are derived from specific methodologies, commonly referred to as ***standards***. Standards dictate how these components should be designed, and what additional components are required in the system to allow for their successful integration and function. As shown in Figure 2-1, standards can define functionality that is specific to each of the layers of the embedded systems model, and can be classified as *market-specific* standards, *general-purpose* standards, or standards that are applicable to *both* categories.

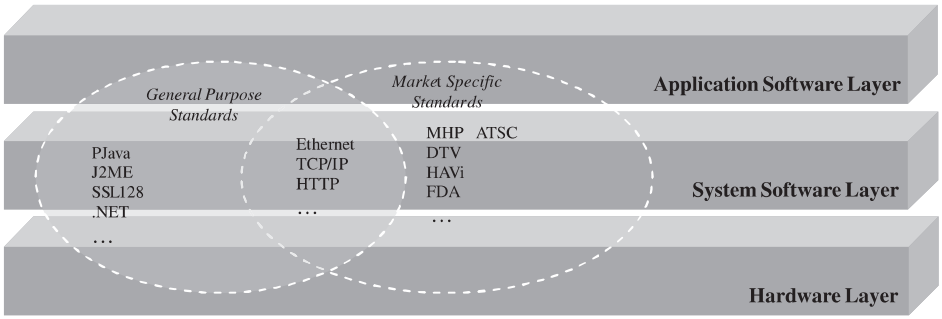


Figure 2-1: Standards diagram

Standards that are strictly market-specific define functionality that is relative to a particular group of related embedded systems that share similar technical or end user characteristics, including:

- *Consumer Electronics.* Typically includes devices used by consumers in their personal lives, such as PDAs (personal data assistants), TVs (analog and digital), games, toys, home appliances (i.e., microwave ovens, dishwashers, washing machines), and internet appliances.[2-1]
- *Medical.* Defined as "...any instrument, apparatus, appliance, material or other article, whether used alone or in combination, including the software necessary for its proper application intended by the manufacturer to be used for human beings for the purpose of:
  - diagnosis, prevention, monitoring, treatment or alleviation of disease,
  - diagnosis, monitoring, treatment, alleviation of or compensation for an injury or handicap,
  - investigation, replacement or modification of the anatomy or of a physiological process,
  - control of conception,

  and which does not achieve its principal intended action in or on the human body by pharmacological, immunological or metabolic means, but which may be assisted in its function by such means..."

  —*European Medical Device Directive (93/42/EEC)* [2-14]

  This includes dialysis machines, infusion pumps, cardiac monitors, drug delivery, prosthetics, and so forth.[2-1]
- *Industrial Automation and Control.* "Smart" robotic devices (smart sensors, motion controllers, man/machine interface devices, industrial switches, etc.) used mainly in manufacturing industries to execute a cyclic automated process.[2-1]
- *Networking and Communications.* Intermediary devices connecting networked end systems, devices such as hubs, gateways, routers and switches. This market segment also includes devices used for audio/video communication, such as cell phones (includes cell phone/PDA hybrids), pagers, video phones and ATM machines.[2-1]
- *Automotive.* Subsystems implemented within automobiles, such as entertainment centers, engine controls, security, antilock brake controls and instrumentation.[2-1]
- *Aerospace and Defense.* Systems implemented within aircraft or used by the military, such as flight management, "smart" weaponry and jet engine control.[2-1]
- *Commercial Office/Home Office Automation.* Devices used in an office setting, such as printers, scanners, monitors, fax machines, photocopiers, printers and barcode readers/writers.[2-1]

Practical Tip

Embedded system market segments and their associated devices are always changing as new devices emerge and other devices are phased out. The market definitions can also vary from company to company semantically as well as how the devices are grouped by market segment. When I want a (very) quick overview of the current terms used to describe embedded markets and how devices are being vertically grouped, I go to three or four websites of leading embedded system software vendors. (In my engineering work, I have adopted the journalistic rule of checking with three or more independent sources to verify information.) Alternately, I simply use a search engine with keywords "embedded market segments" and take a look at the latest developments in device grouping.

Most market-specific standards, excluding networking and some TV standards, are only implemented into embedded systems, because by definition they are intended for specific groups of embedded devices. General-purpose standards, on the other hand, are typically not intended for just one specific market of embedded devices; some are adopted (and in some cases originated) in nonembedded devices as well. Programming language-based standards are examples of general-purpose standards that can be implemented in a variety of embedded systems as well as nonembedded systems. Standards that can be considered both market-specific as well as general purpose include networking standards and some television standards. Networking functionality can be implemented in devices that fall under the networking market space, such as hubs and routers; in devices across various markets, such as wireless communication in networking devices, consumer electronics, etc.; and also in nonembedded devices. Television standards have been implemented in PCs, as well as in traditional TVs and set-top boxes.

Table 2-1 lists some current real-world standards, and some of the purposes behind their implementations.

Table 2-1: Examples of standards implemented in embedded systems

Standard Type		Standard	Purpose
Market Specific	*Consumer Electronics*	JavaTV	The Java TV Application Programming Interface (API) is an extension of the Java platform that provides access to functionality unique to a digital television receiver, such as: audio video streaming, conditional access, access to in-band and out-of-band data channels, access to service information data, tuner control for channel changing, on-screen graphics control, media-synchronization (allows interactive television content to be synchronized with the underlying video and background audio of a television program) and application lifecycle control. (Enables content to gracefully coexist with television programming content such as commercials).[2-3] (See java.sun.com)
		DVB (Digital Video Broadcasting) – MHP (Multimedia Home Platform)	Java-based standard used in digital TV designs. Introduces components in the system software layer, as well as provides recommendations for hardware and the types of applications that would be compatible with MHP. Basically, defines a generic interface between interactive digital applications and the terminals ranging from low-end to high-end set top boxes, integrated digital TV sets and multimedia PCs on which those applications execute. This interface decouples different provider's applications from the specific hardware and software details of different MHP terminal implementations enabling digital content providers to address all types of terminals. The MHP extends the existing DVB open standards for broadcast and interactive services in all transmission networks including satellite, cable, terrestrial and microwave systems.[2-2] (See www.mhp.org)
		ISO/IEC 16500 DAVIC (Digital Audio Visual Council)	DAVIC is an industry standard for end-to-end interoperability of broadcast and interactive digital audio-visual information, and of multimedia communication.[2-4] (See www.davic.org or www.iso.ch)
		ATSC (Advanced Television Standards Committee) – DASE (Digital TV Applications Software Environment)	The DASE standard defines a system software layer that allows programming content and applications to run on a "common receiver." Interactive and enhanced applications need access to common receiver features in a platform-independent manner. This environment provides enhanced and interactive content creators with the specifications necessary to ensure that their applications and data will run uniformly on all brands and models of receivers. Manufacturers will thus be able to choose hardware platforms and operating systems for receivers, but provide the commonality necessary to support applications made by many content creators.[2-5] (See www.atsc.org)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Consumer Electronics (cont.)*	ATVEF (Advanced Television Enhancement Forum) – SMPTE (Society of Motion Picture and Television Engineers) DDE-1	The ATVEF Enhanced Content Specification defines fundamentals necessary to enable creation of HTML-enhanced television content that can be reliably broadcast across any network to any compliant receiver. ATVEF is a standard for creating enhanced, interactive television content and delivering that content to a range of television, set-top, and PC-based receivers. ATVEF [SMPTE DDE-1] defines the standards used to create enhanced content that can be delivered over a variety of mediums—including analog (NTSC) and digital (ATSC) television broadcasts—and a variety of networks, including terrestrial broadcast, cable, and satellite.[2-6] (See www.smpte.org/ or www.atvef.com)
		DTVIA (Digital Television Industrial Alliance of China)	DTVIA is an organization made up of leading TV manufacturers, research institutes and broadcasting academies working on the key technologies and specifications for the China TV industry to transfer from analog to digital. DTVIA and Sun are working together to define the standard for next-generation interactive digital television leveraging Sun's Java TV application programming interface (API) specification.[2-7] (See http://java.sun.com/pr/2000/05/pr000508-02.html, http://netvision.qianlong.com/8737/2003-6-4/39@878954.htm, or contact Guo Ke Digital TV Industry Alliance (DTVIA), +86-10-64383425, email: guo-ke@btamail.net.cn)
		ARIB-BML (Association of Radio Industries and Business of Japan)	ARIB in 1999 established their standard titled "Data Coding and Transmission Specification for Digital Broadcasting" in Japan, an XML-based specification. The ARIB B24 specification derives BML (broadcast markup language) from an early working draft of the XHTML 1.0 Strict document type, which it extends and alters.[2-7] (See www.arib.or.jp)
		OCAP (OpenCable Application Forum)	The OpenCable Application Platform (OCAP) is a system software layer that provides an interface enabling application portability (applications written for OpenCable must be capable of running on any network and on any hardware platform, without recompilation). The OCAP specification is built on the DVB MHP specification with modifications for the North American Cable environment that includes a full time return channel. A major modification to the MHP is the addition of a Presentation Engine (PE), that supports HTML, XML, ECMAScript. A bridge between the PE and the Java Execution Engine (EE), enables PE applications to obtain privileges and directly manipulate privileged operations.[2-8] (See www.opencable.com)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Consumer Electronics (cont.)*	OSGi (Open Services Gateway Initiative)	The OSGi specification is designed to enhance all residential networking standards, such as Bluetooth™, CAL, CEBus, Convergence, emNET, HAVi™, HomePNA™, HomePlug™, HomeRF™, Jini™ technology, LonWorks, UPnP, 802.11B and VESA. The OSGi Framework and Specifications facilitate the installation and operation of multiple services on a single Open Services Gateway (set-top box, cable or DSL modem, PC, Web phone, automotive, multimedia gateway or dedicated residential gateway).[2-9] (See www.osgi.org)
		OpenTV	OpenTV has a proprietary DVB-compliant system software layer, called EN2, for interactive television digital set-top boxes. It complements MHP functionality. and provides functionality that is beyond the scope of the current MHP specification, such as HTML rendering and web browsing.[2-10] (See www.opentv.com)
		MicrosoftTV	MicrosoftTV is a proprietary interactive TV system software layer that combines both analog and digital TV technologies with Internet functionality. MicrosoftTV Technologies support current broadcast formats and standards, including NTSC, PAL, SECAM, ATSC, OpenCable, DVB, and SMPTE 363M (ATVEF specification) as well as Internet standards such as HTML, XML, and so on.[2-11] (See www.microsoft.com)
		HAVi (Home Audio Video Initiative)	HAVi provides a home networking standard for seamless interoperability between digital audio and video consumer devices, allowing all audio and video appliances within the network to interact with each other and allow functions on one or more appliances to be controlled from another appliance, regardless of the network configuration and appliance manufacturer.[2-12] (See www.havi.org)
		CEA (Consumer Electronics Association)	Attempts to foster consumer electronics industry growth by developing industry standards and technical specifications that enable new products to come to market and encourage interoperability with existing devices. Standards include ANSI-EIA-639 Consumer Camcorder or Video Camera Low Light Performance, CEA-CEB4 Recommended Practice for VCR Specifications, and so on.[2-17] (See www.ce.org)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Medical Devices*	FDA (USA)	U.S. government standards for medical devices relating to the aspects of safety and/or effectiveness of the device. **Class I devices** are defined as non-life sustaining. These products are the least complicated and their failure poses little risk. **Class II devices** are more complicated and present more risk than Class I, though are also non-life sustaining. They are also subject to any specific performance standards. **Class III devices** sustain or support life, so that their failure is life threatening. Standards include areas of anesthesia (i.e., Standard Specification for Minimum Performance and Safety Requirements for Resuscitators Intended for Use with Humans, Standard Specification for Ventilators Intended for Use in Critical Care, etc.), cardiovascular/neurology (i.e., Intracranial pressure monitoring devices, etc.), dental/ENT (i.e., Medical Electrical Equipment – Part 2: Particular Requirements for the Safety of Endoscope Equipment, etc.), plastic surgery (i.e., Standard Performance and Safety Specification for Cryosurgical Medical Instrumentation, etc.) ObGyn/Gastroenterology (i.e., Medical electrical equipment – Part 2: Particular requirements for the safety of haemodialysis, haemodiafiltration and haemofiltration equipment, etc.), and so on.[2-13] (See www.fda.gov/)
		Medical Devices Directive (EU)	European Medical Device Directive are standards for medical devices for EU member states relating to the aspects of safety and/or effectiveness of these devices. The lowest risk devices fall into Class I (Internal Control of Production and compilation of a Technical File compliance), whereas devices which exchange energy with the patient in a therapeutic manner or are used to diagnose or monitor medical conditions, are in Class IIa (i.e., ISO 9002 + EN 46002 compliance). If this is done in manner which could be hazardous for the patient, then the device falls into Class Iib (i.e., ISO 9001 + EN 46001). A device that connects directly with the Central Circulatory System or the Central Nervous System or contains a medicinal product, then the device falls into Class III (i.e., ISO 9001 + EN 46001 compliance, compilation of a Design Dossier).[2-14] (See europa.eu.int)
		IEEE1073 Medical Device Communications	IEEE 1073 standards for medical device communication provide plug-and-play interoperability at the point-of-care, optimized for the acute care environment. The IEEE 1073 General Committee is chartered under the IEEE Engineering in Medicine and Biology Society, and works closely with other national and international organizations, including HL7, NCCLS, ISO TC215, CEN TC251, and ANSI HISB.[2-15] (See www.ieee1073.org/)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Industrial Control*	DICOM (Digital Imaging and Communications in Medicine)	The American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA) formed a joint committee in 1983 to develop the DICOM standard for transferring images and associated information between devices manufactured by various vendors, specifically to: • Promote communication of digital image information, regardless of device manufacturer • Facilitate the development and expansion of picture archiving and communication systems (PACS) that can also interface with other systems of hospital information • Allow the creation of diagnostic information data bases that can be interrogated by a wide variety of devices distributed geographically.[2-16] (See http://medical.nema.org/)
		Department of Commerce (USA) **– Office of Microelectronics, Medical Equipment and Instrumentation**	Maintains a website that contains the global medical device regulatory requirements on a per country basis. (See www.ita.doc.gov/td/mdequip/regulations.html)
		(EU) The Machinery Directive 98/37/EC	EU directive for all machinery, moving machines, machine installations, and machines for lifting and transporting people, as well as safety components. In general, machinery being sold or used in the EU must comply with applicable mandatory Essential Health and Safety Requirements (EHSRs) from a long list given in the directive, and must undertake the correct conformity assessment procedure. Most machinery, considered less dangerous can be self-assessed by the supplier, and being able to assemble a Technical File. The 98/37/EC applies to an assembly of linked parts or components with at least one movable part—actuators, controls, and power circuits, processing, treating, moving, or packaging a material—several machines acting in combination, and so on.[2-18] (See www.europa.eu.int)
		IEC (International Electrotechnical Commission 60204-1)	Applies to the electrical and electronic equipment of industrial machines. Promotes the safety of persons who come into contact with industrial machines, not only from hazards associated with electricity (such as electrical shock and fire), but also resulting from the malfunction of the electrical equipment itself. Addresses hazards associated with the machine and its environment. Replaces the second edition of IEC 60204-1 as well as parts of IEC 60550 and ISO 4336.[2-19] (See www.iec.ch)
		ISO (International Standards Organization) Standards	Many standards in the manufacturing engineering segment, such as ISO/TR 10450—Industrial automation systems and integration—Operating conditions for discrete part manufacturing; Equipment in industrial environments, ISO/TR 13283 Industrial automation; Time-critical communications architectures; User requirements and network management for time-critical communications systems, and so on.[2-20] (See www.iso.ch)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Networking and Communications*	TCP (Transmission Control Protocol)/IP (Internet Protocol)	Protocol stack based on RFCs (Request for Comments) 791(IP) & 793 (TCP) that define system software components (more information in Chapter 10). (See www.faqs.org/rfcs/)
		PPP (Point-to-Point Protocol)	System software component based on RFCs 1661, 1332, and 1334 (more information in Chapter 10). (See www.faqs.org/rfcs/)
		IEEE (Institute of Electronics and Electrical Engineers) 802.3 Ethernet	Networking protocol that defines hardware and system software components for local area networks (LANs) (more information in Chapters 6 and 8). (See www.ieee.org)
		Cellular	Networking protocols implemented within cellular phones, such as CDMA (Code Division Multiple Access) and TDMA (Time Division Multiple Access) typically used in the US. TDMA is the basis of GSM (Global System for Mobile telecommunications) European international standard, UMTS (Universal Mobile Telecommunications System) broadband digital standard (3rd generation) (See http://www.cdg.org/ for the CDMA Development Group, http://www.tiaonline.org/ for TDMA and GSM)
	Automotive	GM Global	GM standards are used in the design, manufacture, quality control, and assembly of automotive components and materials related to General Motors, specifically: adhesives, electrical, fuels and lubricants, general, paints, plastics, procedures, textiles, metals, metric and design.[2-27] Standards can be purchased from HIS Global at: http://www.ihs.com/standards/index.html
		Ford Standards	The Ford standards are from the Engineering Material Specifications and Laboratory Test Methods volumes, the Approved Source List Collection, Global Manufacturing Standards, Non-Production Material Specifications, and the Engineering Material Specs & Lab Test Methods Handbook.[2-27] Standards can be purchased from IHS Global at http://www.ihs.com/standards/index.html
		FMVSS (Federal Motor Vehicle Safety Standards)	The Code of Federal Regulations (CFR) contains the text of public regulations issued by the agencies of the U.S. Federal government. The CFR is divided into several titles which represent broad areas subject to Federal Regulation.[2-27] see http://www.nhtsa.dot.gov/cars/rules/standards/safstan2.htm USA National Highway Traffic Safety Administration

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Automotive (cont.)*	OPEL Engineering Material Specifications	OPEL's standards are available in sections, such as: Metals, Miscellaneous, Plastics and Elastomers, Materials of Body—Equipment, Systems and Component Test Specifications, Test Methods, Laboratory Test Procedures (GME/GMI), body and electric, chassis, powertrain, road test procedures (GME/GMI), body and electric, chassis, powertrain, process, paint & environmental engineering materials, and so on.[2-27] Standards can be purchased from HIS Global at: http://www.ihs.com/standards/index.html
		Jaguar Procedures and Standards Collection	The Jaguar standards are available as a complete collection or as individual standards collections such as: Jaguar-Test Procedures Collection, Jaguar-Engine & Fastener Standards Collection, Jaguar-Non-Metallic/Metallic Material Standards Collection, Jaguar-Laboratory Test Standards Collection, etc.[2-27] Standards can be purchased from IHS Global at http://www.ihs.com/standards/index.html
		ISO/TS 16949 – The Harmonized Standard for the Automotive Supply Chain	Jointly developed by the IATF (International Automotive Task Force) members and forms the requirements for automotive production and relevant service part organizations. Based on ISO 9001:2000, AVSQ (Italian), EAQF (French), QS-9000 (U.S.) and VDA6.1 (German) automotive catalogs.[2-30] (See http://www.iaob.org/)
	Aerospace and Defense	SAE (Society of Automotive Engineers) – The Engineering Society For Advancing Mobility in Land Sea Air and Space	SAE Aerospace Material Specifications, SAE Aerospace Standards (includes Aerospace Standards (AS), Aerospace Information Reports (AIR), and Aerospace Recommended Practices (ARP)). [2-27] (See www.sae.org)
		AIA/NAS – Aerospace Industries Association of America, Inc.	This standards service includes National Aerospace Standards (NAS) and metric standards (NA Series). It is an extensive collection that provides standards for components, design and process specifications for aircraft, spacecraft, major weapons systems and all types of ground and airborne electronic systems. It also contains procurement documents for parts and components of high technology systems, including fasteners, high pressure hoses, fittings, high-density electrical connectors, bearings and more.[2-27] (See http://www.aia-aerospace.org/)
		Department of Defense (DOD) –JTA (Joint Technical Architecture)	DOD initiatives, such as the Joint Technical Architecture (JTA) permits the smooth flow of information necessary to achieve interoperability, resulting in optimal readiness. The JTA was established by the U.S. Department of Defense to specify a minimal set of information technology standards, including Web standards, to achieve military interoperability. [2-27] (See http://www.disa.mil/main/jta.html)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
Market Specific *(cont.)*	*Office Automation*	IEEE Std 1284.1-1997 IEEE Standard for Information TechnologyTransport Independent Printer/System Interface (TIP/SI)	A protocol and methodology for software developers, computer vendors, and printer manufacturers to facilitate the orderly exchange of information between printers and host computers are defined in this standard. A minimum set of functions that permit meaningful data exchange is provided. Thus a foundation is established upon which compatible applications, computers, and printers can be developed, without compromising an individual organization's desire for design innovation.[2-28] (See www.ieee.org)
		Postscript	A programming language from Adobe that describes the appearance of a printed page that is an industry standard for printing and imaging. All major printer manufacturers make printers that contain or can be loaded with Postscript software (.ps file extensions). (See www.adobe.com)
		ANSI/AIM BC2-1995, Uniform Symbology Specification for Bar Codes	For encoding general purpose all-numeric data. Reference symbology for UCC/EAN Shipping Container Symbol. Character encoding, reference decode algorithm and optional check character calculation are included in this document. This specification is intended to be significantly identical to corresponding Commission for European Normalization (CEN) specification.[2-29] (See http://www.aimglobal.org/standards/aimpubs.htm)
General Purpose	*Networking*	HTTP (Hypertext Transfer Protocol)	A World Wide Web (WWW) protocol defined by a number of different RFCs, including RFC2616, 2016, 2069, 2109, and so on—Application Layer networking protocol implemented within browsers on any device, for example. (See http://www.w3c.org/Protocols/Specs.html)
		TCP (Transmission Control Protocol)/IP (Internet Protocol)	Protocol stack based on RFCs (Request for Comments) 791(IP) & 793 (TCP) that define system software components (more information in Chapter 10). (See http://www.faqs.org/rfcs/)
		IEEE (Institute of Electronics and Electrical Engineers) 802.3 Ethernet	Networking protocol that defines hardware and system software components for local area networks (LANs) (more information in Chapters 6 and 8). (See www.ieee.org)
		Bluetooth	Bluetooth Specifications are developed by the Bluetooth Special Interest Group (SIG), which allows for developing interactive services and applications over interoperable radio modules and data communication protocols (more information on Bluetooth in Chapter 10).[2-21] (See www.bluetooth.org)

Table 2-1: Examples of standards implemented in embedded systems (continued)

Standard Type		Standard	Purpose
General Purpose *(cont.)*	*Programming Languages*	pJava (Personal Java)	Embedded Java standard from Sun Microsystems targeted and larger embedded systems (more information in Section 2.1). (See java.sun.com)
		J2ME (Java 2 Micro Edition)	Set of embedded standards from Sun Microsystems targeting the entire range of embedded systems, both in size and vertical markets (more information in Section 2.1). (See java.sun.com)
		.NET Compact Framework	Microsoft-based system that allows an embedded system to support applications written in several different languages, including C# and Visual Basic (more information in Section 2.1). (See www.microsoft.com)
		HTML (Hyper Text Markup Language)	Scripting language whose interpreter typically is implemented in a browser, WWW protocol (more information in Section 2.1). (See www.w3c.org)
	Security	Netscape IETF (Internet Engineering Task Force) SSL (Secure Socket Layer) 128-bit Encryption	The SSL is a security protocol that provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection, and is typically integrated into browsers and web servers. There are different versions of SSL (40-bit, 128-bit, etc.), with "128-bit" referring to the length of the "session key" generated by every encrypted transaction (the longer the key, the more difficult it is to break the encryption code). SSL relies on session keys, as well as digital certificates (digital identification cards) for the authentication algorithm. (See http://wp.netscape.com/eng/ssl3/ for version 3 (latest version at the time this book was written) of Netscape's SSL specification.
		IEEE 802.10 Standards for Interoperable LAN/MAN Security (SILS)	Provides a group of specifications at the hardware and system software layer to implement security in networks. (See http://standards.ieee.org/getieee802/index.html)
	Quality Assurance	ISO 9000 Standards	A set of quality management process standards when developing products (not product standards in themselves) or providing a service, including ISO 9000:2000, ISO 9001:2000, ISO 9004:2000, and so on. ISO 9001:2000 presents requirements, while ISO 9000:2000 and ISO 9004:2000 present guidelines. (See www.iso.ch)

Warning!

While Table 2-1 lists market-specific standards in the context of a single market, some market-specific standards listed in this table have been implemented or adopted by other device market segments. This table simply shows "some" real-world examples. Furthermore, different countries, and even different regions of one country, may have unique standards for particular families of devices (i.e., DTV or cell phone standards; see Table 2-1). Also, in most industries, competing standards can exist for the same device, supported by competing interests. Find out who has adopted which standards and how these competing standards differ by using the Internet to look up published data sheets or manuals of the particular device, the documentation provided by the vendors of the components integrated within the device, or by attending the various tradeshows, seminars, and conferences associated with that particular industry or vendor, such as the Embedded Systems Conference (ESC), Java One, Real-Time Embedded and Computing Conference, Embedded Processor Forum, etc.

This warning note is especially important for hardware engineers, who may have come from an environment where certain standards bodies, such as IEEE, have a strong influence on what is adopted. In the embedded software field there is currently ***no single standards body*** *that has the level of influence that IEEE has in the hardware arena.*

The next three sections of this chapter contain real-world examples showing how specific standards define some of the most critical components of an embedded system. Section 2.1 presents general-purpose programming language standards that can affect the architecture of an embedded system. Section 2.2 presents networking protocols that can be implemented in a specific family of devices, across markets, and in stand-alone applications. Finally, Section 2.3 presents an example of a consumer appliance that implements functionality from a number of different standards. These examples demonstrate that a good starting point in demystifying the design of an embedded system is to simply derive from industry standards what the system requirements are and then determine where in the overall system these derived components belong.

2.1 An Overview of Programming Languages and Examples of Their Standards

Why Use Programming Languages as a Standards Example?

In embedded systems design, there is no single language that is the perfect solution for every system. Programming language standards, and what they introduce into an embedded systems architecture, are used as an example in this section, because a programming language can introduce an additional component into an embedded architecture. In addition, embedded systems software is inherently based on one or some combination of multiple languages. The examples discussed in-depth in this section, such as Java and the .NET Compact Framework, are based upon specifications that add additional elements to an embedded architecture. Other languages which can be based upon a variety of standards, such as ANSI C vs. Kernighan and Ritchie C, aren't discussed in-depth because using these languages in an embedded design does not usually require introducing an additional component into the architecture.

Note: Details of when to use what programming language and the pros and cons of such usage are covered in Chapter 11. It is important for the reader to first understand the various components of an embedded system before trying to understand the reasoning behind using certain components over others at the design and development level. Language choice decisions aren't based on the features of the language alone, and are often dependent on the other components within the system.

The hardware components within an embedded system can only directly transmit, store, and execute ***machine code***, a basic language consisting of ones and zeros. Machine code was used in earlier days to program computer systems, which made creating any complex application a long and tedious ordeal. In order to make programming more efficient, machine code was made visible to programmers through the creation of a hardware-specific set of instructions, where each instruction corresponded to one or more machine code operations. These hardware-specific sets of instructions were referred to as ***assembly language***. Over time, other programming languages, such as C, C++, Java, etc., evolved with instruction sets that were (among other things) more hardware-independent. These are commonly referred to as ***high-level*** languages because they are semantically further away from machine code, they more resemble human languages, and are typically independent of the hardware. This is in contrast to a ***low-level*** language, such as assembly language, which more closely resembles machine code. Unlike high-level languages, low-level languages are hardware dependent, meaning there is a unique instruction set for processors with different architectures. Table 2-2 outlines this evolution of programming languages.

Table 2-2: Evolution of programming languages [2-22]

	Language	Details
1st Generation	Machine code	Binary (0,1) and hardware dependent.
2nd Generation	Assembly language	Hardware-dependent representing corresponding binary machine code.
3rd Generation	HOL (high-order languages)/procedural languages	High-level languages with more English-like phrases and more transportable, such as C, Pascal, etc.
4th Generation	VHLL (very high level languages)/non-procedural languages	"Very" high-level languages: object-oriented languages (C++, Java,…), database query languages (SQL), etc.
5th Generation	Natural languages	Programming similar to conversational languages, typically used in artificial intelligence (AI). Still in the research and development phases in most cases—not yet applicable in mainstream embedded systems.

Note: Even in systems that implement some higher-level languages, some portions of embedded systems software are implemented in assembly language for architecture-specific or optimized-performance code.

Because machine code is the only language the hardware can directly execute, all other languages need some type of mechanism to generate the corresponding machine code. This mechanism usually includes one or some combination of *preprocessing*, *translation*, and *interpretation*. Depending on the language, these mechanisms exist on the programmer's ***host*** system (typically a nonembedded development system, such as a PC or Sparc station), or the ***target*** system (the embedded system being developed). See Figure 2-2.

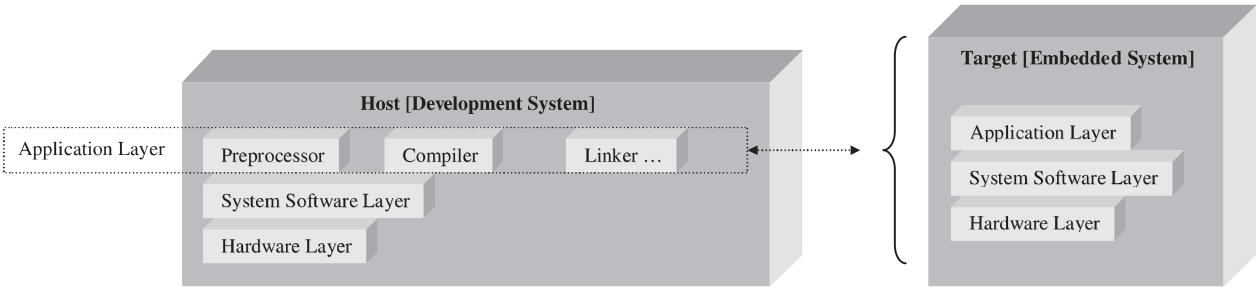


Figure 2-2: Host and target system diagram

Preprocessing is an optional step that occurs before either the translation or interpretation of source code, and whose functionality is commonly implemented by a ***preprocessor***. The preprocessor's role is to organize and restructure the source code to make translation or interpretation of this code easier. As an example, in languages like C and C++, it is a preprocessor that allows the use of named code fragments, such as *macros*, that simplify code development by allowing the use of the macro's name in the code to replace fragments of code. The preprocessor then replaces the macro name with the contents of the macro during preprocessing. The preprocessor can exist as a separate entity, or can be integrated within the translation or interpretation unit.

Many languages convert source code, either directly or after having been preprocessed through use of a ***compiler***, a program that generates a particular target language—such as machine code and Java byte code—from the source language (see Figure 2-3).

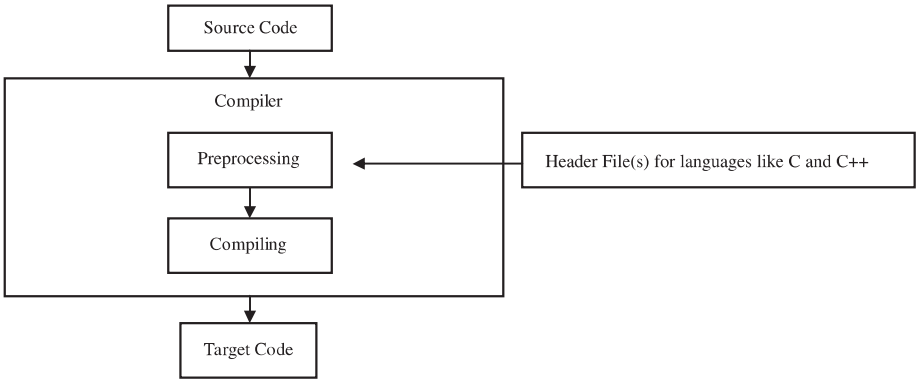


Figure 2-3: Compilation diagram

A compiler typically "translates" all of the source code to some target code at one time. As is usually the case in embedded systems, compilers are located on the programmer's host machine and generate target code for hardware platforms that differ from the platform the compiler is actually running on. These compilers are commonly referred to as ***cross-compilers***. In the case of assembly language, the compiler is simply a specialized cross-compiler referred to as an ***assembler***, and it always generates machine code. Other high-level language compilers are commonly referred to by the language name plus the term "compiler," such as "Java compiler" and "C compiler." High-level language compilers vary widely in terms of what is generated. Some generate machine code, while others generate other high-level code, which then requires what is produced to be run through at least one more compiler or interpreter, as discussed later in this section. Other compilers generate assembly code, which then must be run through an assembler.

After all the compilation on the programmer's host machine is completed, the remaining target code file is commonly referred to as an ***object file***, and can contain anything from machine code to Java byte code (discussed later in this section), depending on the programming language used. As shown in Figure 2-4, after linking this object file to any system libraries required, the object file, commonly referred to as an ***executable***, is then ready to be transferred to the target embedded system's memory.

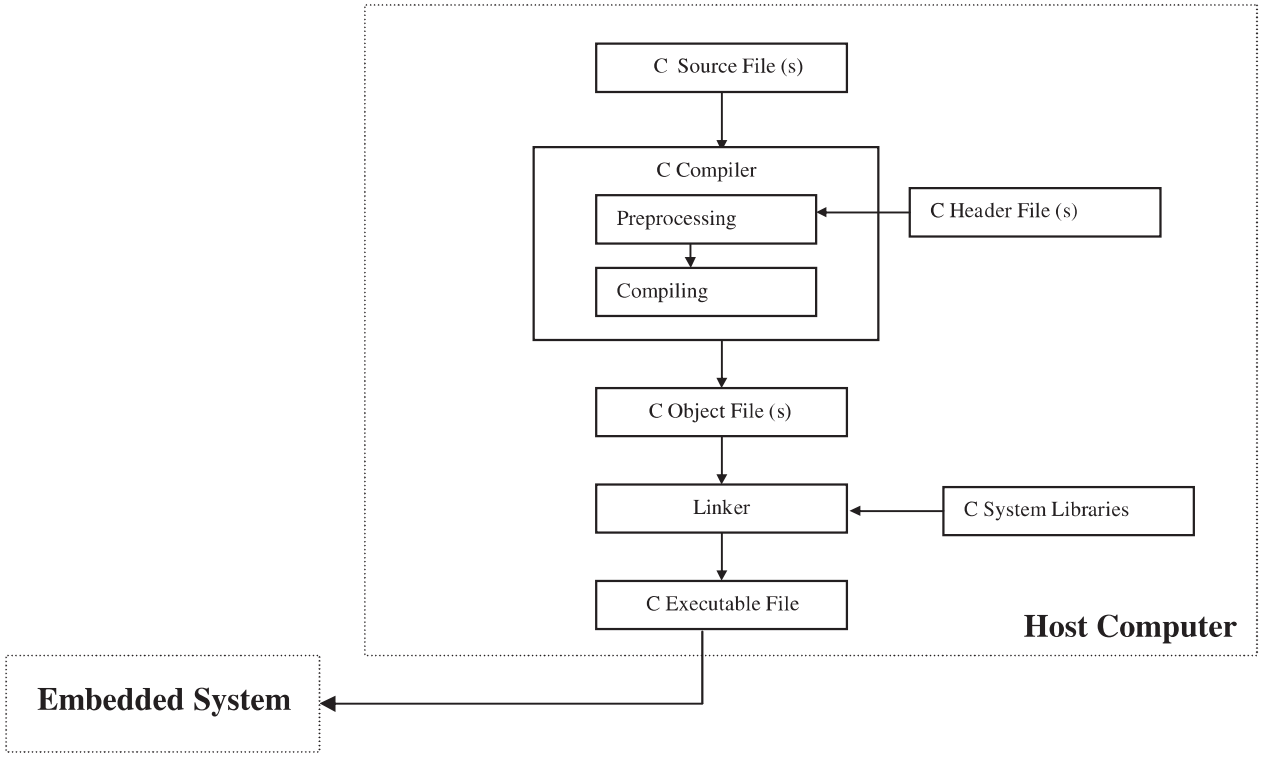


Figure 2-4: C Example compilation/linking steps and object file results

How Does the Executable Get from the Host to the Target?

A combination of mechanisms are used to accomplish this. Details on memory and how files are executed from it will be discussed in more detail in Section II, while the different transmission mediums available for transmitting the executable file from a host system to an embedded system will be discussed in more detail in the next section of this chapter (Section 2.2). Finally, the common development tools used will be discussed in Chapter 12.

Examples of Programming Languages that Affect an Embedded Architecture: Scripting Languages, Java, and .NET

Where a compiler usually translates all of the given source code at one time, an ***interpreter*** generates (interprets) machine code one source code line at a time (see Figure 2-5).

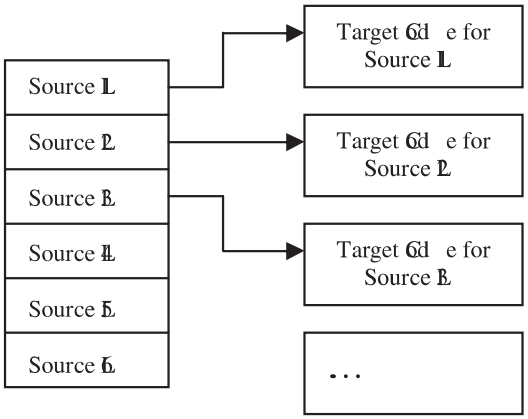


Figure 2-5: Interpretation diagram

One of the most common subclasses of interpreted programming languages are ***scripting languages,*** which include PERL, JavaScript, and HTML. Scripting languages are high-level programming languages with enhanced features, including:

- More platform independence than their compiled high-level language counterparts.[2-23]
- Late binding, which is the resolution of data types on-the-fly (rather than at compile time) to allow for greater flexibility in their resolution.[2-23]
- Importation and generation of source code at runtime, which is then executed immediately.[2-23]
- Optimizations for efficient programming and rapid prototyping of certain types of applications, such as internet applications and graphical user interfaces (GUIs).[2-23]

With embedded platforms that support programs written in a scripting language, an additional component—an interpreter—must be included in the embedded system's architecture to allow for "on-the-fly" processing of code. Such is the case with the embedded system architectural software stack shown in Figure 2-6, where an internet browser can contain both an HTML and JavaScript interpreter to process downloaded web pages.

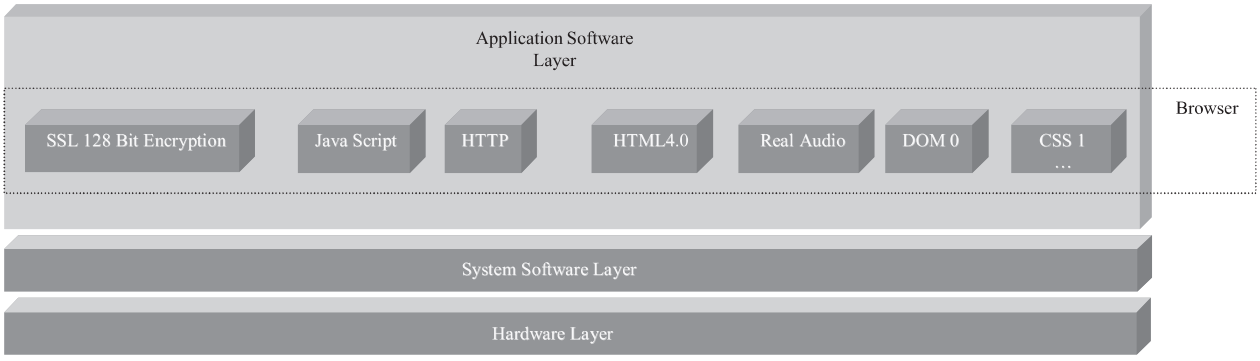


Figure 2-6: HTML and Javascript in the application layer

While all scripting languages are interpreted, not all interpreted languages are scripting languages. For example, one popular embedded programming language that incorporates both compiling and interpreting machine code generation methods is ***Java***. On the programmer's host machine, Java must go through a compilation procedure that generates Java byte code from Java source code (see Figure 2-7).

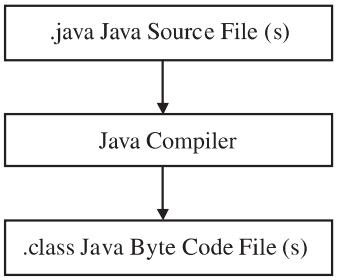


Figure 2-7: Embedded Java compilation and linking diagram

Java byte code is target code intended to be platform independent. In order for the Java byte code to run on an embedded system, a ***Java Virtual Machine*** (JVM) must exist on that system. Real-world JVMs are currently implemented in an embedded system in one of three ways: in the hardware, in the system software layer, or in the application layer (see Figure 2-8).

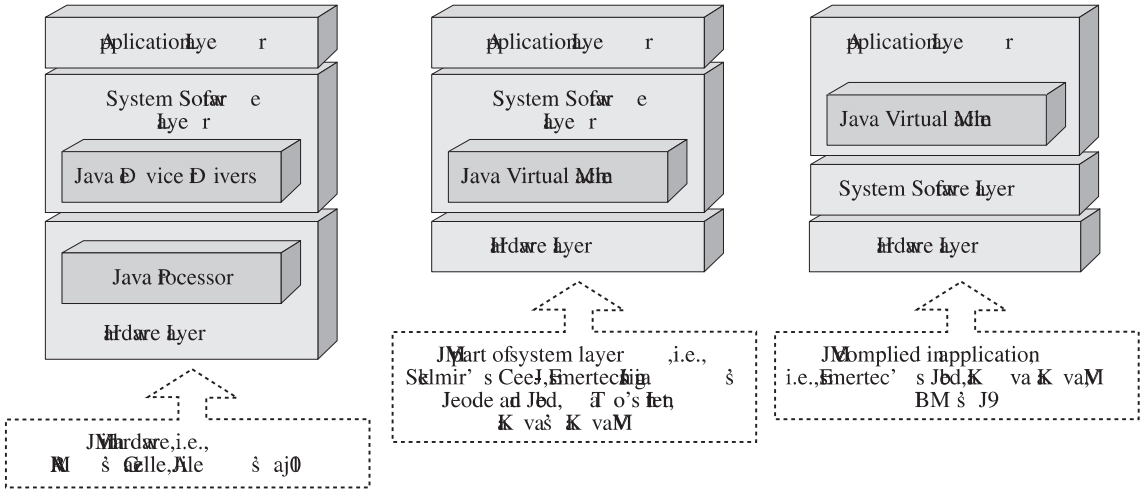


Figure 2-8: JVMs and the Embedded Systems Model

Size, speed, and functionality are the technical characteristics of a JVM that most impact an embedded system design, and two JVM components are the primary differentiators between embedded JVMs: the JVM classes included within the JVM, and the execution engine that contains components needed to successfully process Java code (see Figure 2-9).

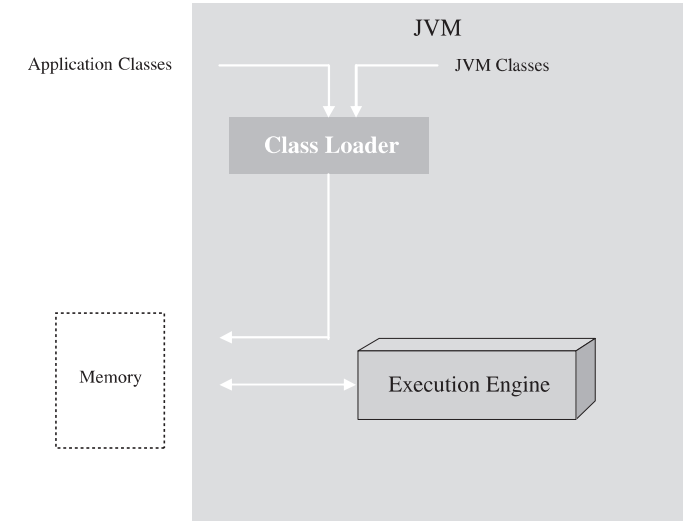


Figure 2-9: Internal JVM components

The JVM classes shown in Figure 2-9 are compiled libraries of Java byte code, commonly referred to as ***Java APIs*** (application program interfaces). Java APIs are application independent libraries provided by the JVM to, among other things, allow programmers to execute system functions and reuse code. Java applications require the Java API classes, in addition to their own code, to successfully execute. The size, functionality, and constraints provided by these APIs differ according to the Java specification they adhere to, but can include memory management features, graphics support, networking support, and so forth. Different standards with their corresponding APIs are intended for different families of embedded devices (see Figure 2-10).

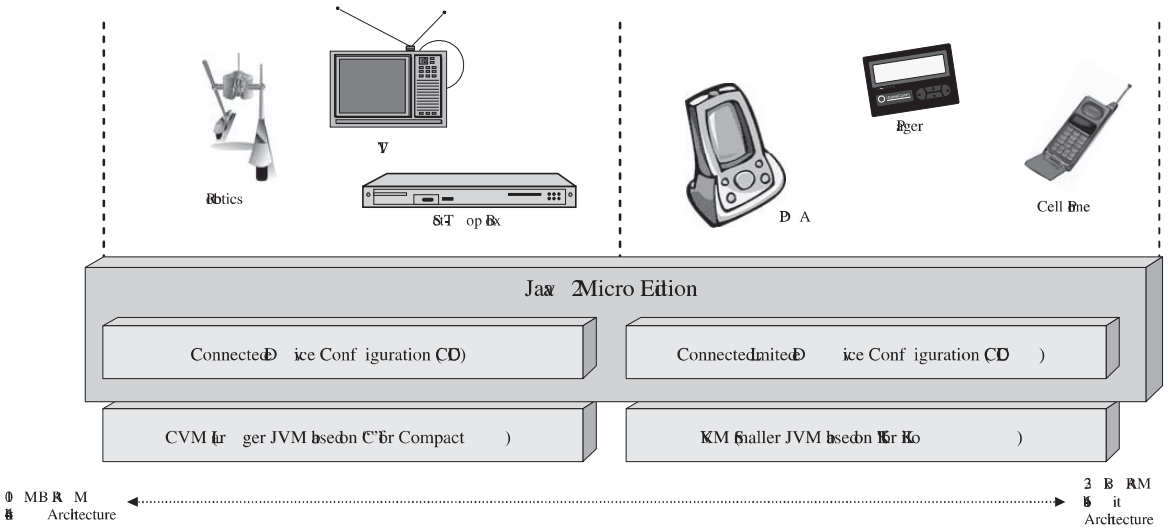


Figure 2-10: J2ME family of devices

In the embedded market, recognized embedded Java standards include J Consortium's *Real-Time Core Specification*, and *Personal Java* (pJava), *Embedded Java, Java 2 Micro Edition* (J2ME) and *The Real-Time Specification for Java* from Sun Microsystems.

Figures 2-11a and b show the differences between the APIs of two different embedded Java standards.

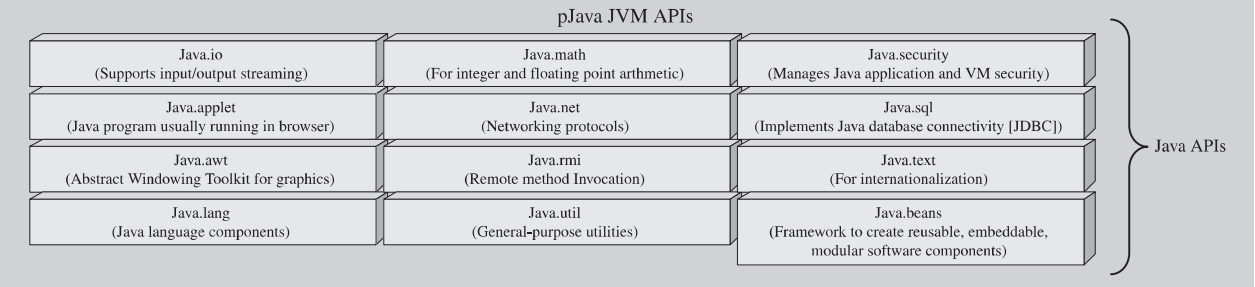


Figure 2-11a: pJava 1.2 API components diagram

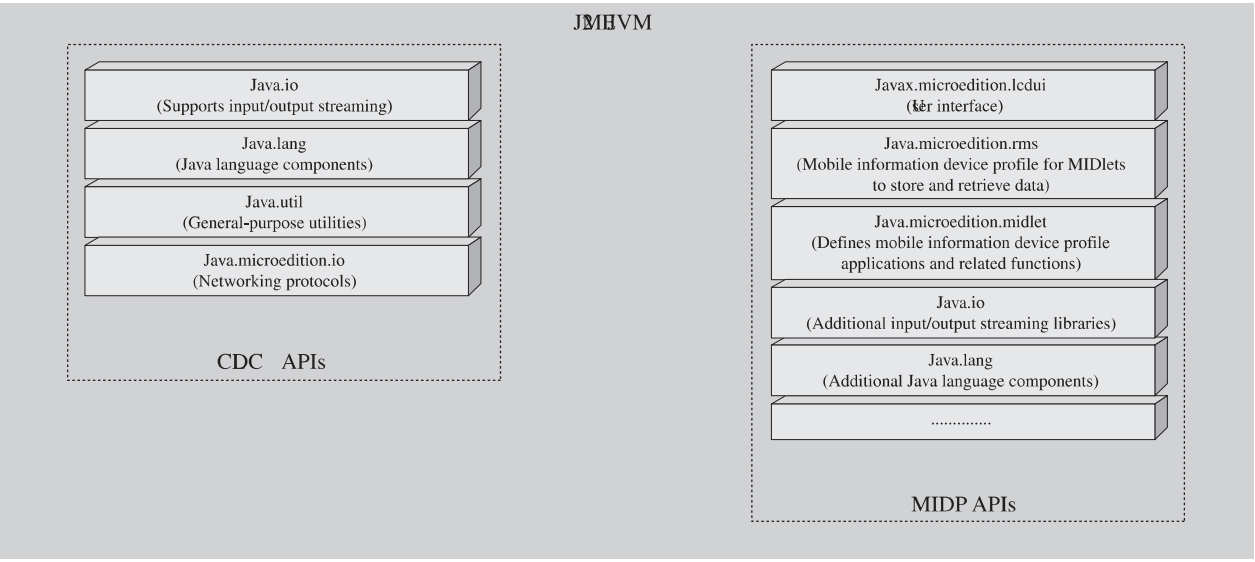


Figure 2-11b: J2ME CLDC 1.1/MIDP 2.0 API components diagram

Table 2-3 shows several real-world JVMs and the standards they adhere to.

Table 2-3: Real-world examples of JVMs based on embedded Java standards

Embedded Java Standards	Java Virtual Machines
Personal Java (pJava)	Tao Group's Intent (www.tao-group.com)
	Insignia's pJava Jeode (www.insignia.com)
	NSICom CrE-ME (www.nsicom.com)
	Skelmir's pJava Cee-J (www.skelmir.com)
Embedded Java	Esmertec Embedded Java Jeode (www.esmertec.com)
Java 2 Micro Edition (J2ME)	Esmertec's Jbed for CLDC/MIDP and Insignia's CDC Jeode (www.esmertec.com and www.insignia.com)
	Skelmir's Cee-J CLDC/MIDP and CDC (www.skelmir.com)
	Tao Group's Intent (www.tao-group.com) CLDC & MIDP

Note: Information in table was gathered at the time this book was written and is subject to change; check with specific vendors for latest information.

Within the execution engine (shown in Figure 2-12), the main differentiators that impact the design and performance of JVMs that support the same specification are:

- The ***garbage collector (GC)***, which is responsible for deallocating any memory no longer needed by the Java application.
- The unit that ***processes byte codes***, which is responsible for converting Java byte codes into machine code through *interpretation*, *compilation* (commonly referred to as way-ahead-of-time or WAT), or *just-in-time (JIT)*, an algorithm that combines both compiling and interpreting. A JVM can implement one or more of these byte code processing algorithms within its execution engine.

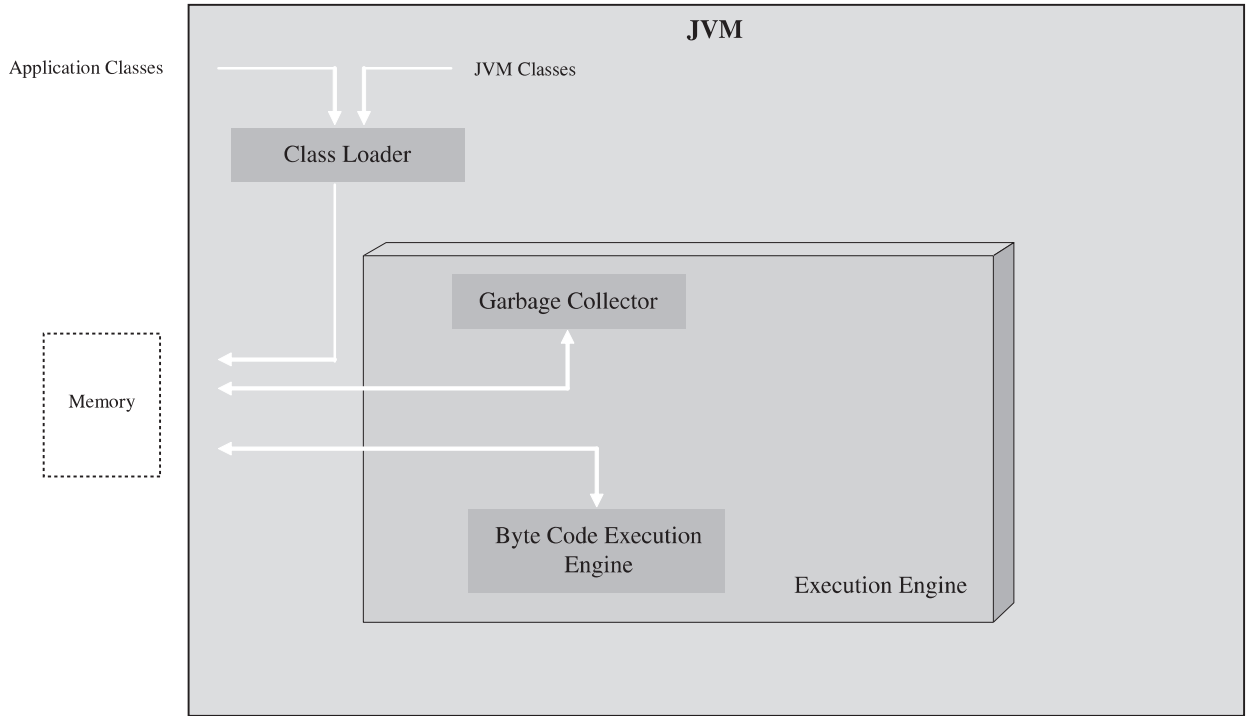


Figure 2-12: Internal execution engine components

Garbage Collection

Why Talk About Garbage Collection?

While this section discusses garbage collection within the context of Java, I use it as a separate example because garbage collection isn't unique to the Java language. A garbage collector can be implemented in support of other languages, such as C and C++[2-24], that don't typically add an additional component to the system. When creating a garbage collector to support any language, it becomes part of an embedded system's architecture.

An application written in a language such as Java cannot deallocate memory that has been allocated for previous use (as can be done in native languages, such as using "free" in the C language, though as mentioned above, a garbage collector can be implemented to support any language). In Java, only the GC (garbage collector) can deallocate memory no longer in use by Java applications. GCs are provided as a safety mechanism for Java programmers so they do not accidentally deallocate objects that are still in use. While there are several garbage collection schemes, the most common are based upon the copying, mark and sweep, and generational GC algorithms.

The copying garbage collection algorithm (shown in Figure 2-13) works by copying referenced objects to a different part of memory, and then freeing up the original memory space of unreferenced objects. This algorithm uses a larger memory area in order to work, and usually cannot be interrupted during the copy (it blocks the system). However, it does ensure that what memory is used is used efficiently by compacting objects in the new memory space.

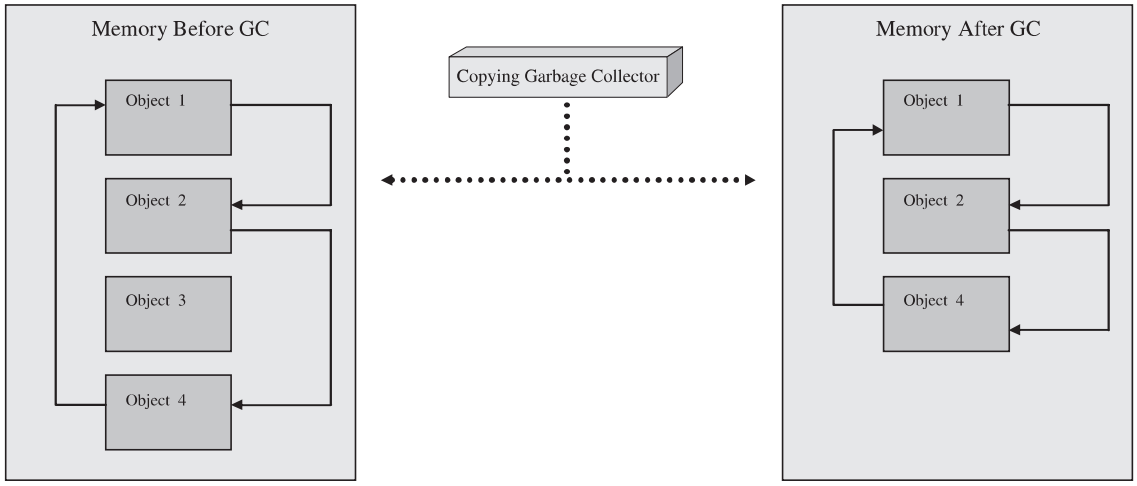


Figure 2-13: Copying GC

The mark and sweep garbage collection algorithm (shown in Figure 2-14) works by "marking" all objects that are used, and then "sweeping" (deallocating) objects that are unmarked. This algorithm is usually nonblocking, meaning the system can interrupt the garbage collector to execute other functions when necessary. However, it doesn't compact memory the way a copying garbage collector does, leading to *memory fragmentation*, the existence of small, unusable holes where deallocated objects used to exist. With a mark and sweep garbage collector, an additional memory compacting algorithm can be implemented, making it a mark (sweep) and compact algorithm.

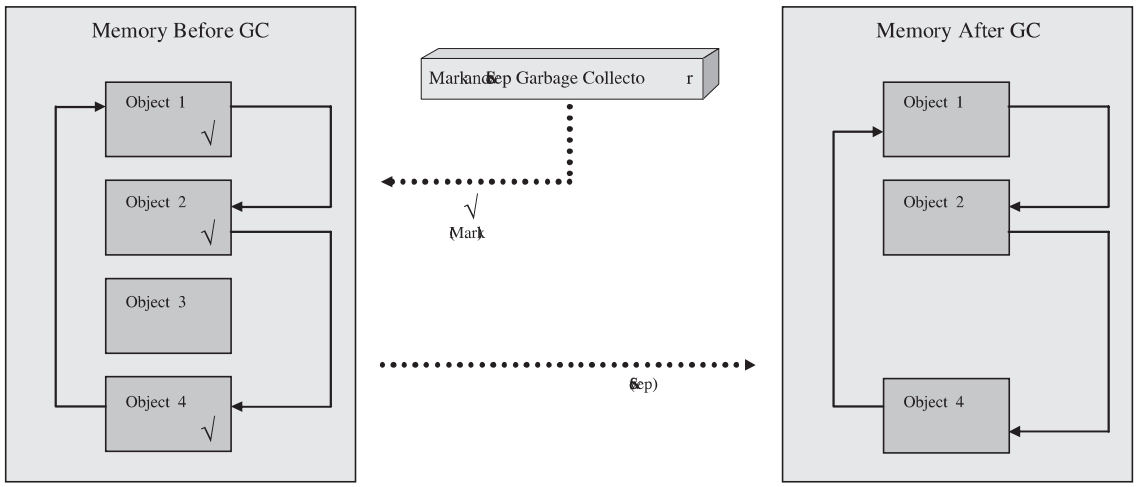


Figure 2-14: Mark and sweep (no compaction) garbage collector diagram

Finally, the generational garbage collection algorithm (shown in Figure 2-15) separates objects into groups, called *generations*, according to when they were allocated in memory. This algorithm assumes that most objects that are allocated by a Java program are short-lived, thus copying or compacting the remaining objects with longer lifetimes is a waste of time. So, it is objects in the younger generation group that are cleaned up more frequently than objects in the older generation groups. Objects can also be moved from a younger generation to an older generation group. Different generational garbage collectors also may employ different algorithms to deallocate objects within each generational group, such as the copying algorithm or mark and sweep algorithms described previously.

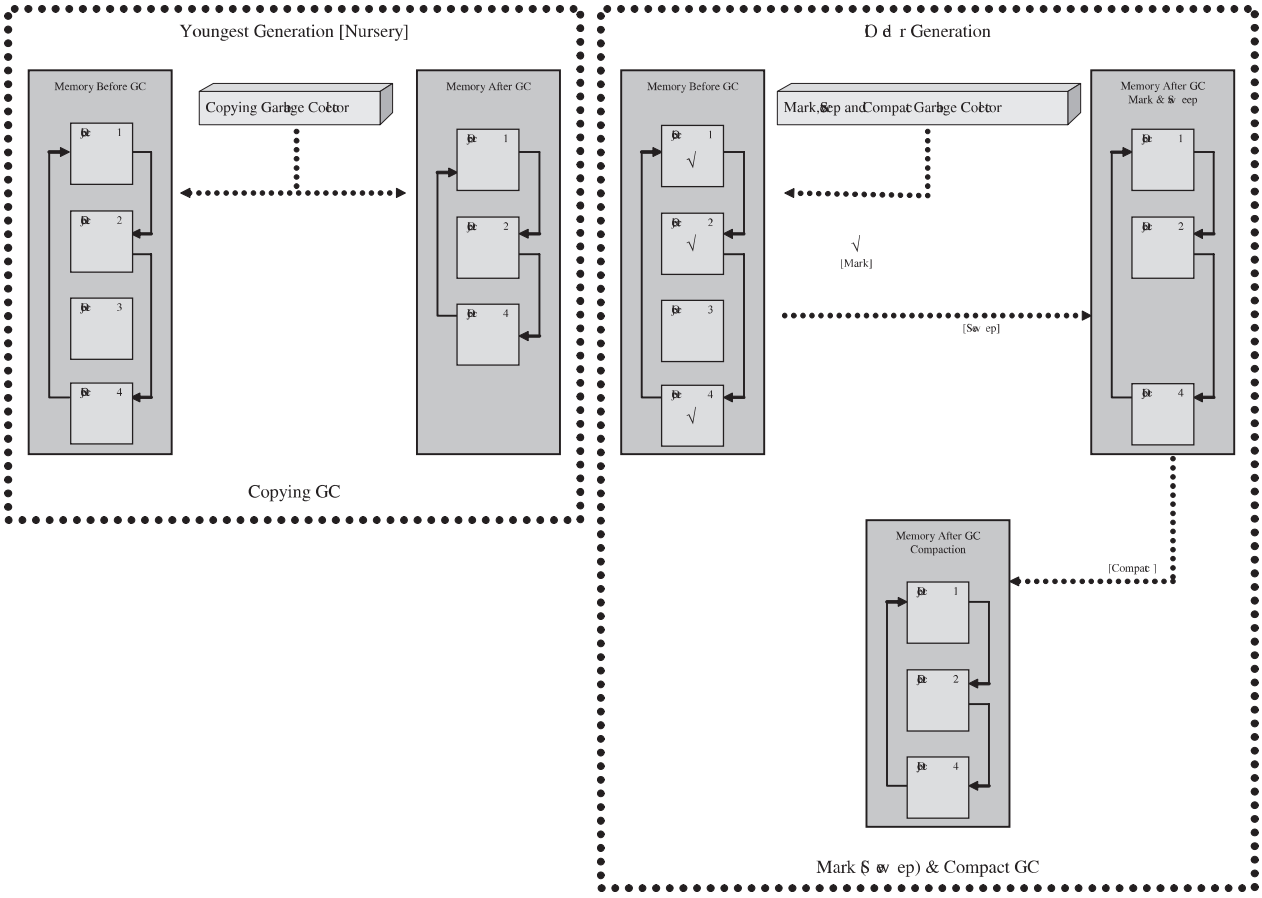


Figure 2-15: Generational garbage collector diagram

As mentioned at the start of this section, most real-world embedded JVMs implement some form of either the copying, mark and sweep, or generational algorithms (see Table 2-4).

Table 2-4: Real-world examples of JVMs based on the garbage collection algorithms

Garbage Collector	Java Virtual Machine
Copying	NewMonic's Perc (www.newmonics.com)
Mark & Sweep	Skelmir's Cee-J (www.skelmir.com)
	Esmertec's Jbed (www.esmertec.com)
	NewMonics' Perc (www.newmonics.com)
	Tao Group's Intent (www.tao-group.com)
Generational	Skelmir's Cee-J (www.skelmir.com)

Note: Information in table was gathered at the time this book was written and is subject to change; check with specific vendors for latest information.

Processing Java Bytecode

Why Talk About How Java Processes Bytecode?

This section is included because Java is an illustration of many different real-world techniques that are used to translate source code into machine code in a variety of other languages. For example, in assembly, C, and C++, the compilation mechanisms exist on the host machine, whereas HTML scripting language source is interpreted directly on the target (with no compilation needed). In the case of Java, Java source code is compiled on the host into Java byte code, which then can be interpreted or compiled into machine code, depending on the JVM's internal design. Any mechanism that resides on the target, which translates Java byte code into machine code, is part of an embedded system's architecture. In short, Java's translation mechanisms can exist both on the host and on the target, and so act as examples of various real-world techniques that can be used to understand how programming languages in general impact an embedded design.

The JVM's primary purpose in an embedded system is to process platform-independent Java byte code into platform-dependent code. This processing is handled in the execution engine of the JVM. The three most common byte code processing algorithms implemented in an execution engine to date are interpretation, just-in-time (JIT) compiling, and way-ahead-of-time/ahead-of-time compiling (WAT/AOT).

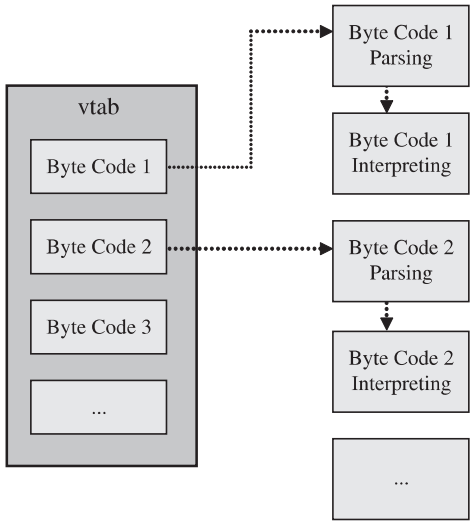


Figure 2-16: Interpreter diagram

With interpretation, every time the Java program is loaded to be executed, every byte code instruction is parsed and converted to native code, one byte code at a time, by the JVM's interpreter. Moreover, with interpretation, redundant portions of the code are reinterpreted every time they are run. Interpretation tends to have the lowest performance of the three algorithms, but it is typically the simplest algorithm to implement and to port to different types of hardware.

A JIT compiler, on the other hand, interprets the program once, and then compiles and stores the native form of the byte code at runtime, thus allowing redundant code to be executed without having to reinterpret. The JIT algorithm performs better for redundant code, but it can have additional runtime overhead while converting the byte code into native code. Additional memory is also used for storing both the Java byte codes and the native compiled code. Variations on the JIT algorithm in real-world JVMs are also referred to as translators or dynamic adaptive compilation (DAC).

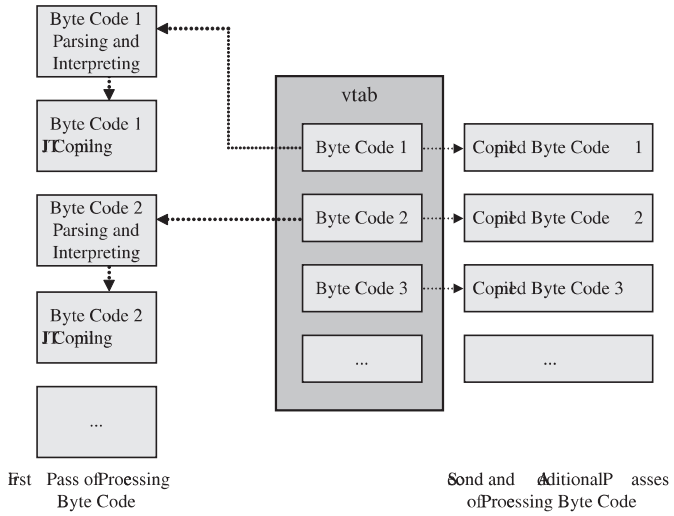
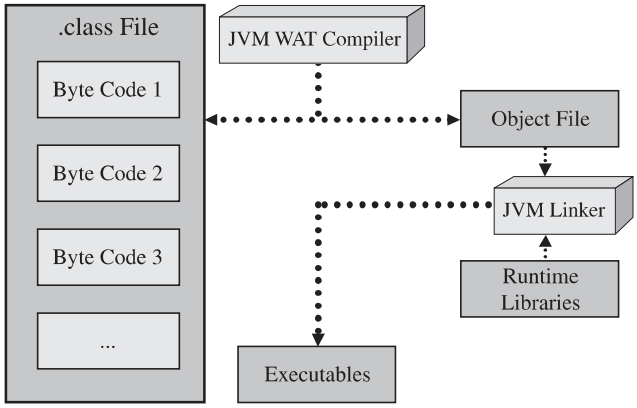


Figure 2-17: JIT diagram

Figure 2-18: WAT/AOT diagram



Finally, in WAT/AOT compiling, all Java byte code is compiled into the native code at compile time, as with native languages, and no interpretation is done. This algorithm performs at least as well as the JIT for redundant code and better than a JIT for nonredundant code, but as with the JIT, there is additional runtime overhead when additional Java classes dynamically downloaded at runtime have to be compiled and introduced to the system. WAT/AOT can also be a more complex algorithm to implement.

As seen in Table 2-5, there are real-world JVM execution engines that implement each of these algorithms, as well as execution engine hybrids that implement some or all of these algorithms.

Table 2-5: Real-world examples of JVMs based on the various byte code processing algorithms

Byte Code Processing	Java Virtual Machine
Interpretation	Skelmir Cee-J (www.skelmir.com)
	NewMonics Perc (www.newmonics.com)
	Insignia's Jeode (www.insignia.com)
JIT	Skelmir Cee-J (two types of JITS) (www.skelmir.com)
	Tao Group's Intent (www.tao-group.com) – translation
	NewMonics Perc (www.newmonics.com)
	Insignia's Jeode DAC (www.insignia.com)
WAT/AOT	NewMonics Perc (www.newmonics.com)
	Esmertec's Jbed (www.esmertec.com)

Note: Information in table was gathered at the time this book was written and is subject to change; check with specific vendors for latest information.

Scripting languages and Java aren't the only high-level languages that can automatically introduce an additional component within an embedded system. The ***.NET Compact Framework*** from Microsoft allows applications written in almost any high-level programming language (such as C#, Visual Basic and Javascript) to run on any embedded device, independent of hardware or system software design. Applications that fall under the .NET Compact Framework must go through a compilation and linking procedure that generates a CPU-independent intermediate language file, called MSIL (Microsoft Intermediate Language), from the original source code file (see Figure 2-19). For a high-level language to be compatible with the .NET Compact Framework, it must adhere to Microsoft's ***Common Language Specification***, a publicly available standard that anyone can use to create a compiler that is .NET compatible.

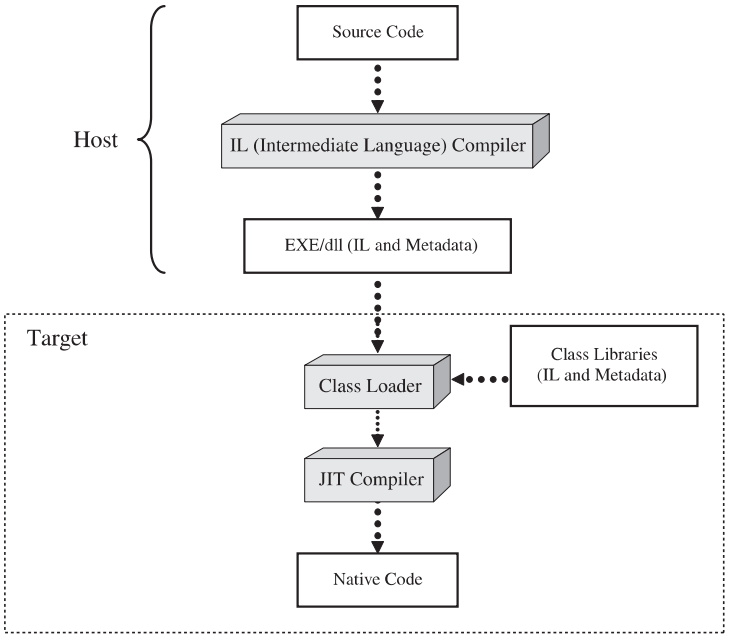


Figure 2-19: .NET Compact Framework execution model

The .NET Compact Framework is made up of a ***common language runtime*** (CLR), a class loader, and platform extension libraries. The CLR is made up of an execution engine that processes the intermediate MSIL code into machine code, and a garbage collector. The platform extension libraries are within the base class library (BCL), which provides additional functionality to applications (such as graphics, networking and diagnostics). As shown in Figure 2-20, in order to run the intermediate MSIL file on an embedded system, the .NET Compact Framework must exist on that embedded system. At the current time, the .NET Compact Framework resides in the system software layer.

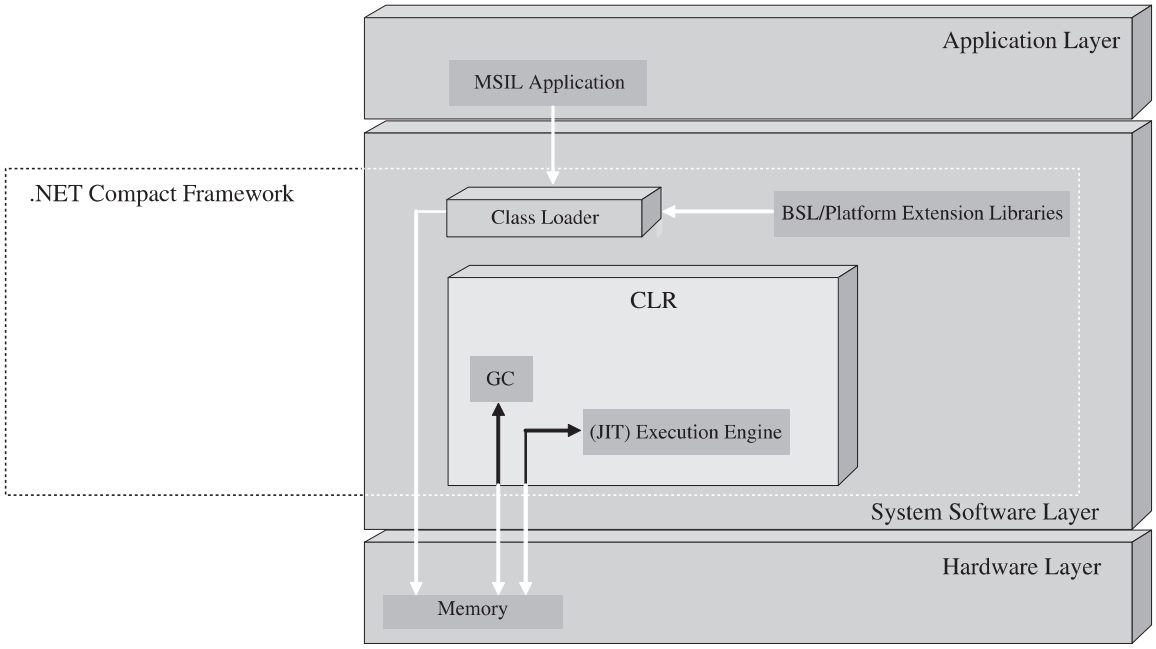


Figure 2-20: .NET Compact Framework and the Embedded Systems Model

2.2 Standards and Networking

Why Use Networking as a Standards Example?

A network, by definition, is two or more connected devices that can send and/or receive data. If an embedded system needs to communicate with any other system, whether a development host machine, a server, or another embedded device, it must implement some type of connection (networking) scheme. In order for communication to be successful, there needs to be a scheme that interconnecting systems agree upon, and so networking protocols (standards) have been put in place to allow for interoperability. As shown in Table 2-1, networking standards can be implemented in embedded devices specifically for the networking market, as well as in devices from other market segments that require networking connectivity, even if just to debug a system during the development phase of a project.

Understanding what the required networking components are for an embedded device requires two steps:

- Understanding the entire network into which the device will connect.
- Using this understanding, along with a networking model, such as the OSI (Open Systems Interconnection) model discussed later in this section, to determine the device's networking components.

Understanding the entire network is important, because key features of the network will dictate the standards that need to be implemented within an embedded system. Initially, an embedded engineer should, at the very least, understand three features about the entire network the device will plug into: the distance between connected devices, the physical medium connecting the embedded device to the rest of the network, and the network's overall structure (see Figure 2-21).

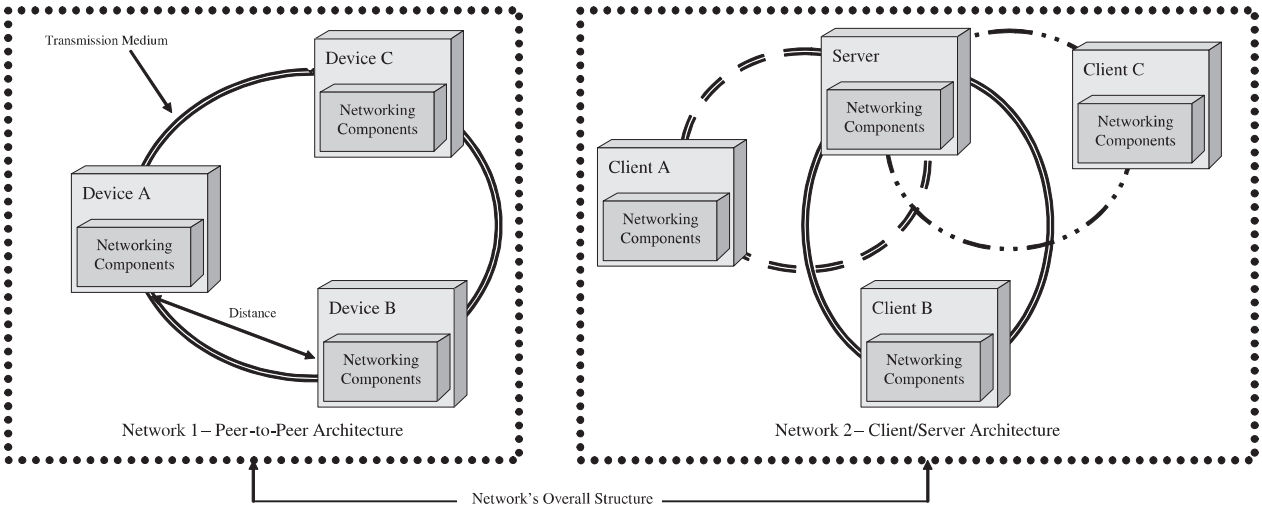


Figure 2-21: Network block diagram

Distance Between Connected Devices

Networks can be broadly defined as either local area networks (LANs) or wide area networks (WANS). A LAN is a network in which all devices are within close proximity to each other, such as in the same building or room. A WAN is a network that connects devices and/or LANs that are scattered over a wider geographical area, such as in multiple buildings or across the globe. While there are other variations of WANs (such as Metropolitan Area Networks (MAN) for inter-city networks, Campus Area Networks (CAN) for school-based networks, etc.) and LANs (i.e., short-distance wireless Personal Area Networks (PANs)) that exist, networks are all essentially either WANs or LANs.

Warning!

Watch those acroynyms! Many look similar but in fact can mean very different things. For example, a WAN (wide area network) should not be confused with WLAN (wireless local area network).

Physical Medium

In a network, devices are connected with transmission mediums that are either bound or unbound. Bound transmission mediums are cables or wires and are considered "guided" mediums since electromagnetic waves are guided along a physical path (the wires). Unbound transmission mediums are wireless connections, and they are considered unguided mediums because the electromagnetic waves that are transmitted are not guided via a physical path, but are transmitted through a vacuum, air, and/or water.

In general, the key characteristics that differentiate all transmission mediums whether wired or wireless are:

- The type of data the medium can carry (i.e., analog or digital).
- How much data the medium can carry (capacity).
- How fast the medium can carry the data from source to destination (speed).
- How far the medium can carry the data (distance). For example, some mediums are lossless, meaning no energy is lost per unit distance traveled, while others are lossy mediums, which means a significant amount of energy is lost per unit distance traveled. Another example is the case of wireless networks, that are inherently subject to the laws of propagation, where given a constant amount of power, signal strengths decrease by the *square* of a given distance from the source (i.e., if distance = 2 ft., the signal becomes four times weaker; if distance = 10 ft., the signal is 100 times weaker, and so on).
- How susceptible the medium is to external forces (interference such as electromagnetic interference (EMI), radio frequency interference (RFI), weather, and so forth).

Note: the direction a transmission medium can transmit data (that is, data being able to travel in only one direction vs. bidirectional transmission) is dependent on the hardware and software components implemented within the device, and is typically not dependent on the transmission medium alone. This will be discussed later in this section.

Understanding the features of the transmission medium is important, because these impact the overall network's performance, affecting such variables as the network's bandwidth (data rate in bits per second) and latency (the amount of time it takes data to travel between two given points, including delays). Tables 2-6a and b summarize a few examples of wired (bound) and wireless (unbound) transmission mediums, as well as some of their features.

Table 2-6a: Wired transmission mediums [2-25]

Medium	Features
Unshielded Twisted Pair (UTP)	Copper wires are twisted into pairs and used to transmit analog or digital signals. Limits in length (distance) depending on the desired bandwidth. UTP used in telephone/telex networks; can support both analog and digital. Different categories of cables (3, 4, and 5) where CAT3 supports a data rate of up to 16 Mbps, CAT4 up to 20 MHz, and CAT5 up to 100 Mbps. Requires amplifiers every 5–6 km for analog signals, and repeaters every 2–3 km for digital signals (over long distances signals lose strength and are mistimed). Relatively easy and cheap to install, but with a security risk (can be tapped into). Subject to external electromagnetic interference. Can act as antennas receiving EMI/RFI from sources such as electric motors, high-voltage transmission lines, vehicle engines, radio or TV broadcast equipment. These signals, when superimposed on a data stream, may make it difficult for the receiver to differentiate between valid data and EMI/RFI-induced noise (especially true for long spans that incorporate components from multiple vendors). Crosstalk occurs when unwanted signals are coupled between "transmit" and "receive" copper pairs, creating corrupted data and making it difficult for the receiver to differentiate between normal and coupled signals. Lightning can be a problem when it strikes unprotected copper cable and attached equipment (energy may be coupled into the conductor and can propagate in both directions).
Coaxial	Baseband and broadband coaxial cables differ in features. Generally, coaxial cables are copper-wire and aluminum-wire connections that are used to transmit both analog and digital signals. Baseband coaxial commonly used for digital—cableTV/cable modems. Broadband used in analog (telephone) communication. Coaxial cable cannot carry signals beyond several thousand feet before amplification is required (i.e., by repeaters or boosters); higher data rates than twisted-pair cabling (several hundred Mbps and up to several km). Coaxial cables not secure (can be tapped into), but are shielded to reduce interference and therefore allow higher analog transmissions.
Fiber Optic	Clear, flexible tubing that allows laser beams to be transmitted along the cable for digital transmissions. Fiber optic mediums have a GHz (bandwidth) transmission capability up to 100 km. Because of their dielectric nature, optical fibers are immune to both EMI and RFI and cross-talk rarely occurs. Optical-fiber cables do not use metallic conductors for transmission, so all-dielectric, optical-fiber communications are less susceptible to electrical surges even if struck directly by lightning. More secure—difficult to tap into, but typically more costly than other terrestrial solutions.

Table 2-6b: Wireless transmission mediums [2-26]

Medium	Features
Terrestrial Microwave	Classified as SHF (super high frequency). Transmission signal must be line of sight, meaning high-frequency radio (analog or digital) signals are transmitted via a number of ground stations and transmission between stations must be in a straight line that is unobstructed ground—often used in conjunction with satellite transmission. The distance between stations is typically 25–30 miles, with the transmission dishes on top of high buildings or on high points like hill tops. The use of the low GHz frequency range 2–40 GHz provides higher bandwidths (i.e., 2 GHz band has approx 7 MHz bandwidth whereas 18 GHz band has approx 220 MHz bandwidth) than those available using lower frequency radio waves.
Satellite Microwave	Satellites orbit above the earth and act as relay stations between different ground stations and embedded devices (their antennas) that are within the satellite's line-of-sight and area covered (footprint), where the size and shape of the footprint on the earth's surface vary with the design of the satellite. The ground station receives analog or digital data from some source (internet service provider, broadcaster, etc.) and modulates it onto a radio signal that it transmits to the satellite, as well as controls the position and monitors the operations of the satellite. At the satellite, a transponder receives the radio signal, amplifies it and relays it to a device's antenna inside its footprint. Varying the footprint changes the transmission speeds, where focusing the signal on a smaller footprint increases transmission speeds. The large distances covered by a signal can also result in propagation delays of several seconds. A typical GEO (geosynchronous earth orbit) satellite (a satellite that orbits about 36,000 kilometers above the equator—the speed of the satellite is matched to the rotation of the Earth at the equator) contains between 20 and 80 transponders, each capable of transmitting digital information of up to about 30–40 Mbps.
Broadcast Radio	Uses a transmitter and a receiver (of the embedded device) tuned to a specific frequency for the transfer of signals. Broadcast communication occurs within a local area, where multiple sources receive one transmission. Subject to frequency limitations (managed by local communications companies and government) to ensure no two transmissions on the same frequency. Transmitter requires large antennas; frequency range of 10 kHz–1 GHz subdivided into LF (low frequency), MF (medium frequency), HF (high frequency), UHF (ultra high frequency) or VHF (very high frequency) bands. Higher frequency radio waves provide larger bandwidths (in the Mbps) for transmission, but they have less penetrating power than lower frequency radio waves (with bandwidths as low as in the kbps).
IR (Infrared)	Point-to-point link of two IR lasers lined up; blinking of laser beam reflects bit representations. THz (1,000GHz – 2×10^{11} Hz – 2×10^{14} Hz) range of frequencies, with up to 20 Mbps bandwidth. Cannot have obstructions, more expensive, susceptible to bad weather (clouds, rain) and diluted by sunlight, difficult to "tap" (more secure). Used in small, open areas with a typical transmission distance of up to 200 meters.
Cellular Microwave	Works in UHF band—variable—depends on whether there are barriers. Signals can penetrate buildings/barriers, but degradation occurs and reduces the required distance the device must be from the cell site.

The Network's Architecture

The relationship between connected devices in a network determines the network's overall architecture. The most common architecture types for networks are ***peer-to-peer***, ***client/server***, and ***hybrid*** architectures.

Peer-to-peer architectures are network implementations in which there is no centralized area of control. Every device on the network must manage its own resources and requirements. Devices all communicate as equals, and can utilize each other's resources. Peer-to-peer networks are usually implemented as LANs because, while simpler to implement, this architecture creates security and performance issues related to the visibility and accessibility of each device's resources to the rest of the network.

Client/server architectures are network implementations in which there is a centralized device, called the ***server***, in control that manages most of the network's requirements and resources. The other devices on the network, called ***clients***, contain fewer resources and must utilize the server's resources. The client/server architecture is more complex than the peer-to-peer architecture and has the single critical point of failure (the server). However, it is more secure than the peer-to-peer model, since only the server has visibility into other devices. Client/server architectures are also usually more reliable, since only the server has to be responsible for providing redundancy for the network's resources in case of failures. Client/server architectures also have better performance, since the server device in this type of network usually needs to be more powerful in order to provide the network's resources. This architecture is implemented in either LANs or WANs.

Hybrid architectures are a combination of the peer-to-peer and client/server architecture models. This architecture is also implemented in both LANs and WANs.

*Note: A network's architecture is **not** the same as its topology. A network's topology is the physical arrangement of the connected devices, which is ultimately determined by the architecture, the transmission medium (wired vs. wireless), and the distance between the connected devices of the particular network.*

Open Systems Interconnection (OSI) Model

To demonstrate the dependencies between the internal networking components of an embedded system and the network's architecture, the distance between connected devices, and the transmission medium connecting the devices, this section associates networking components with a universal networking model, in this case the Open Systems Interconnection (OSI) Reference model. All the required networking components in a device can be grouped together into the OSI model, which was created in the early 1980s by the International Organization for Standardization (ISO). As shown in Figure 2-22, the OSI model represents the required hardware and software components within a networked device in the form of seven layers: physical, data-link, network, transport, session, presentation, and application layers. In relation to the Embedded Systems Model (Figure 1-1), the physical layer of the OSI model maps to the hardware layer of the Embedded Systems Model, the application, presentation, and session layers of the OSI model map to the application software layer of the Embedded Systems Model, and the remaining layers of the OSI model typically map to the system software layer of the embedded systems model.

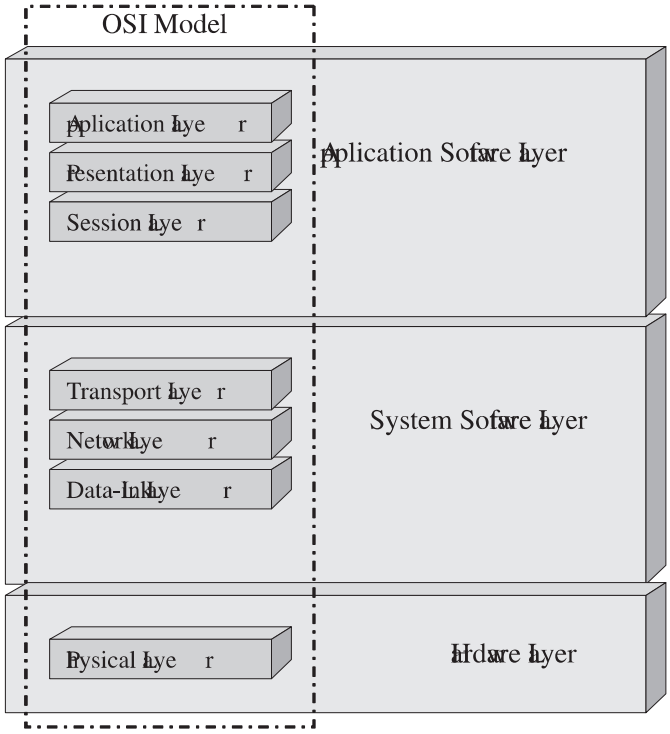


Figure 2-22: OSI and Embedded Systems Model block diagram

The key to understanding the purpose of each layer in the OSI model is to grasp that networking is not simply about connecting one device to another device. Instead, networking is primarily about the data being transmitted between devices or, as shown in Figure 2-23, between the different layers of each device.

In short, a networking connection starts with data originating at the application layer of one device and flowing downward through all seven layers, with each layer adding a new bit of information to the data being sent across the network. Information, called the ***header*** (shown in Figure 2-24), is appended to the data at every layer (except for the physical and application layers) for peer layers in connected devices to process. In other words, the data is wrapped with information for other devices to unwrap and process.

The data is then sent over the transmission medium to the physical layer of a connected device, and then up through the connected device's layers. These layers then process the data (that is, strip the headers, reformat, etc.) as the data flows upward. The functionality and methodologies implemented at each layer based on the OSI model are also commonly referred to as ***networking protocols***.

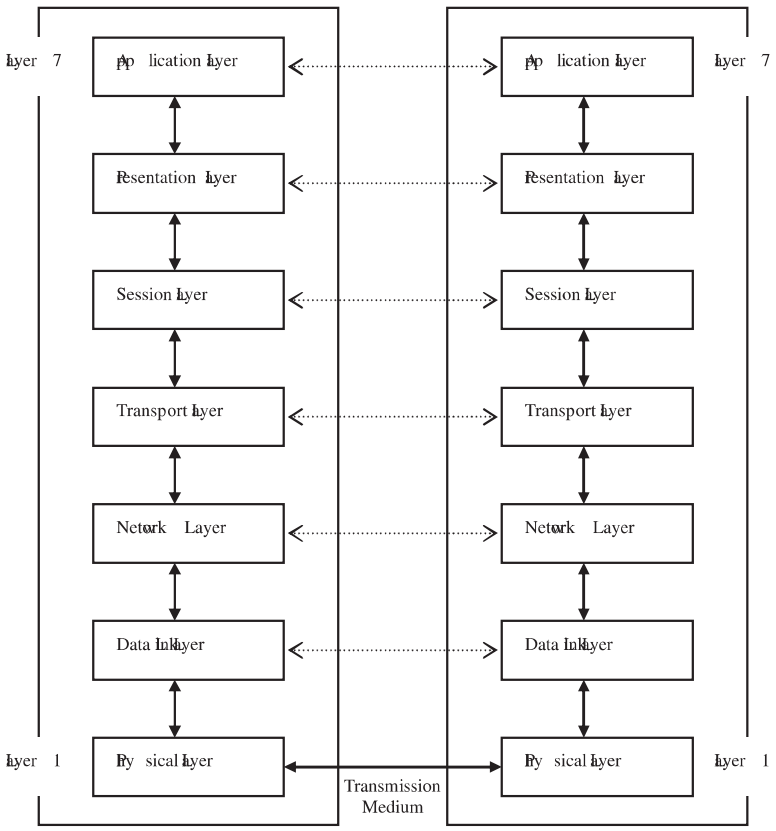


Figure 2-23: OSI model data flow diagram

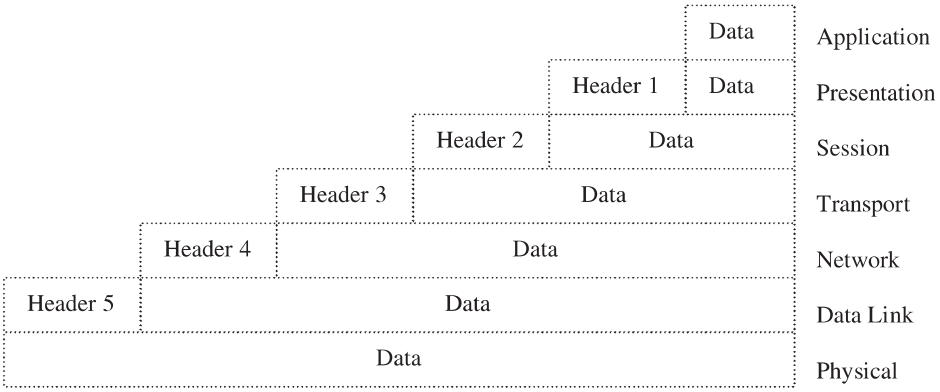


Figure 2-24: Header diagram

The OSI Model and Real-World Protocol Stacks

Remember that the OSI model is simply a reference tool to use in understanding real-world networking protocols implemented in embedded devices. Thus, it isn't always the case that there are seven layers, or that there is only one protocol per layer. In reality, the functionality of one layer of the OSI model can be implemented in one protocol, or it can be implemented across multiple protocols and layers. One protocol can also implement the functionality of multiple OSI layers as well. While the OSI model is a very powerful tool to use to understand networking, in some cases a group of protocols may have their own name and be grouped together in their own proprietary layers. For example, shown in Figure 2-25 is a TCP/IP protocol stack that is made up of four layers: the network access layer, internet layer, transport layer, and the application layer. The TCP/IP application layer incorporates the functionality of the top three layers of the OSI model (the application, presentation, and session layers), and the network access layer incorporates two layers of the OSI model (physical and data link). The internet layer corresponds to the network layer in the OSI model, and the transport layers of both models are identical.

As another example, the wireless application protocol (WAP) stack (shown in Figure 2-26) provides five layers of upper layer protocols. The WAP application layer maps to the application layer of the OSI model, as do the transport layers of both models. The session and transaction layers of the WAP model map to the OSI session layer, and WAP's security layer maps to OSI's presentation layer.

The final example in this section is the Bluetooth protocol stack (shown in Figure 2-27), which is a three-layer model made up of Bluetooth-specific as well as adopted protocols from other networking stacks, such as WAP and/or TCP/IP. The physical and lower data-link layers of the OSI model map to the transport layer of the Bluetooth model. The upper data-link, network, and transport layers of the OSI model map to the middleware layer of the Bluetooth model, and the remaining layers of the OSI model (session, presentation, and application) map to the application layer of the Bluetooth model.

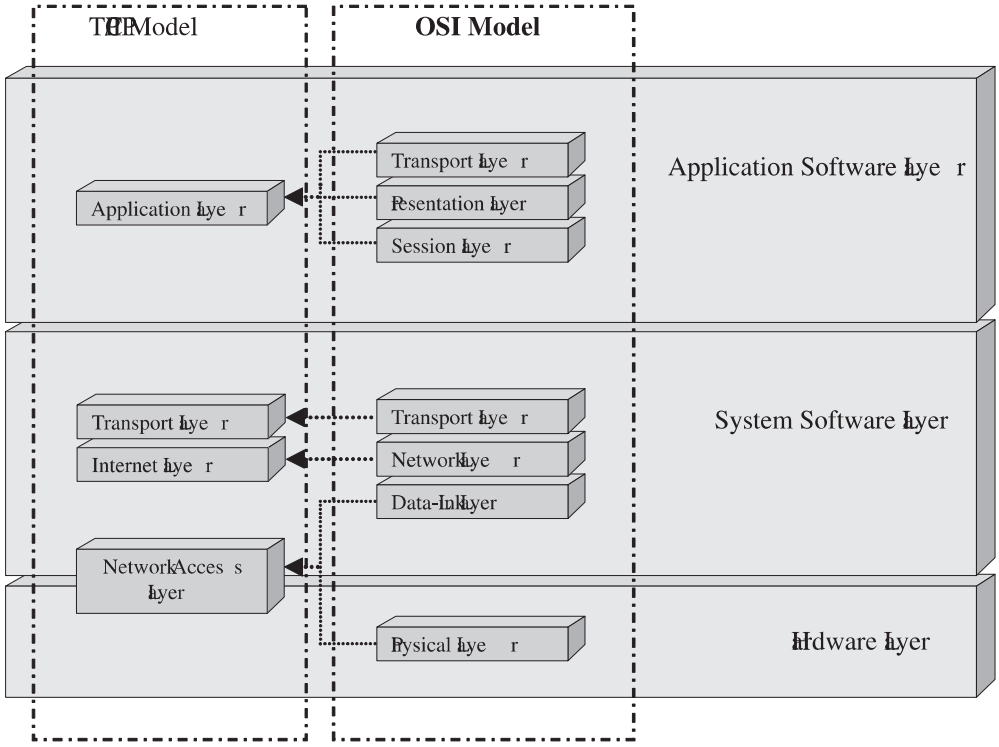


Figure 2-25: TCP/IP, OSI models and Embedded Systems Model block diagram

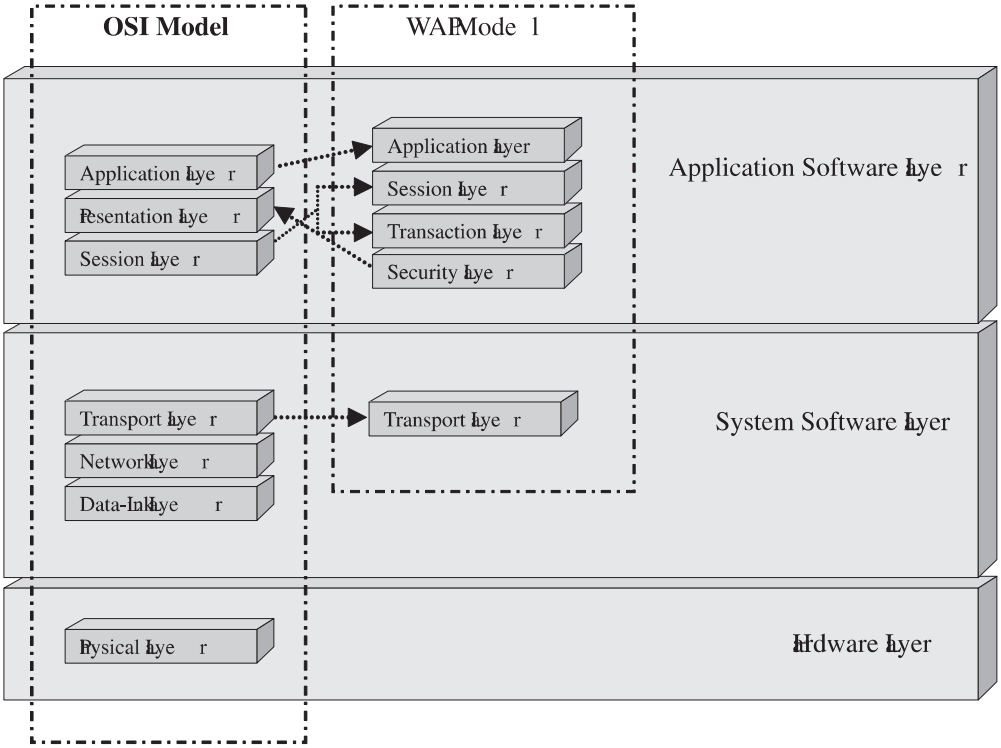


Figure 2-26: WAP, OSI and Embedded Systems Model block diagram

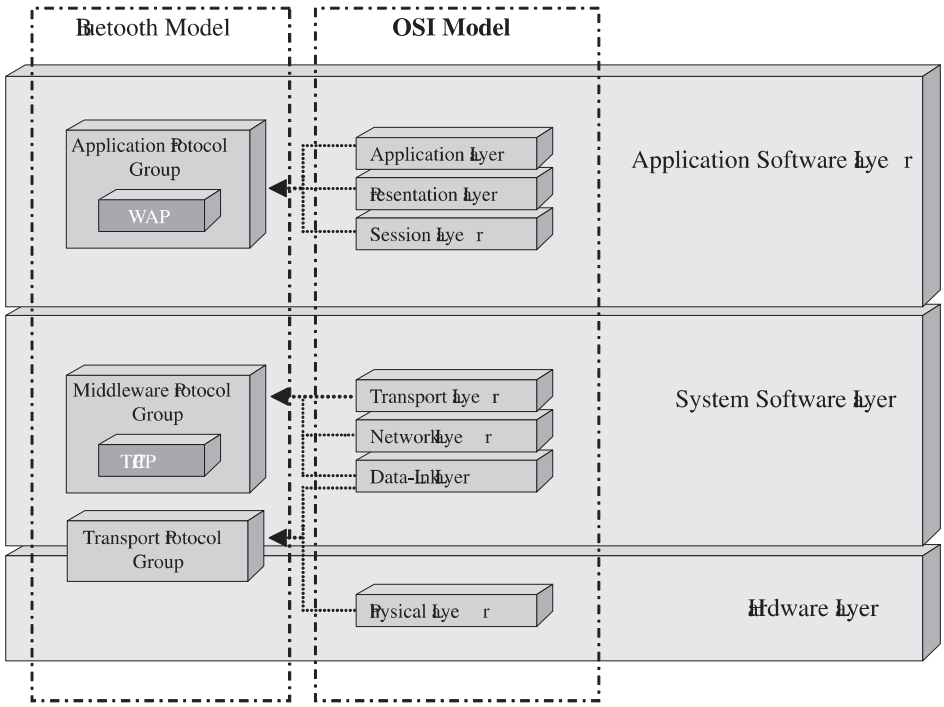


Figure 2-27: Bluetooth, OSI and Embedded Systems Model block diagram

OSI Model Layer 1: Physical Layer

The physical layer represents all of the networking hardware physically located in an embedded device. Physical layer protocols defining the networking hardware of the device are located in the hardware layer of the Embedded Systems Model (see Figure 2-28). Physical layer hardware components connect the embedded system to some transmission medium. The distance between connected devices, as well as the network's architecture, are important at this layer, since physical layer protocols can be classified as either LAN protocols or WAN protocols. LAN and WAN protocols can then be further subdivided according to the transmission medium connecting the device to the network (wired or wireless).

The physical layer defines, manages, and processes, via hardware, the data signals—the actual voltage representations of 1's and 0's—coming over the communication medium. The physical layer is responsible for physically transferring the data bits over the medium received from higher layers within the embedded system, as well as for reassembling bits received over the medium for higher layers in the embedded system to process (see Figure 2-29).

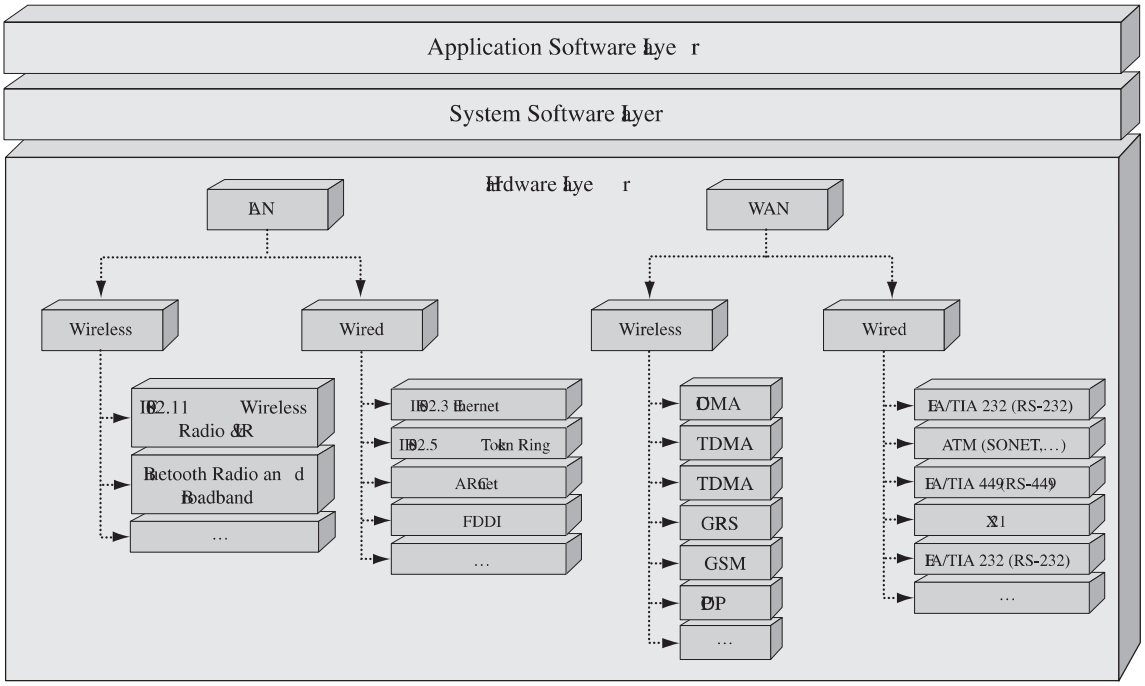


Figure 2-28: Physical layer protocols in the Embedded Systems Model

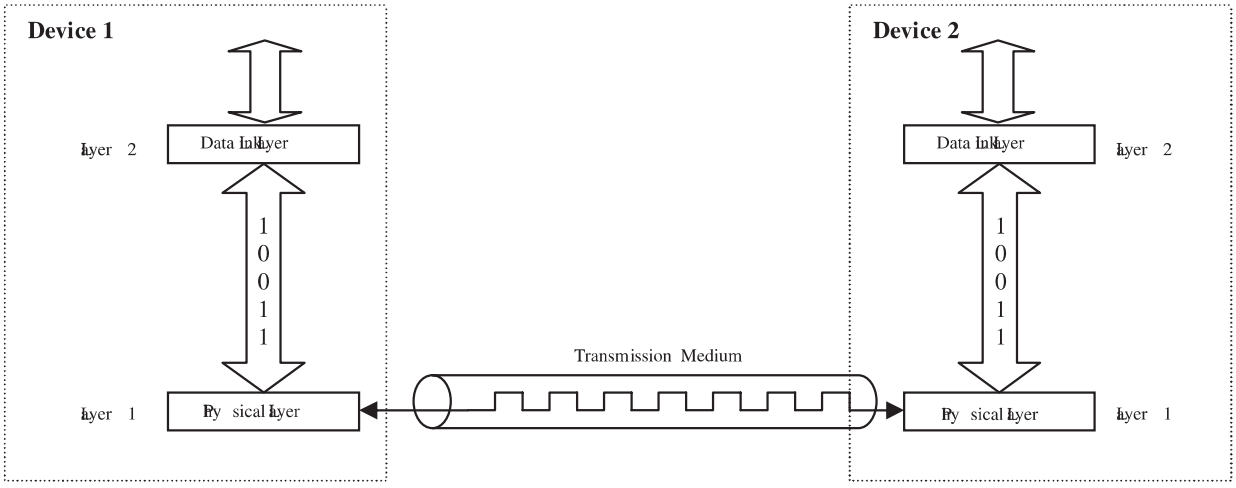


Figure 2-29: Physical layer data flow block diagram

OSI Model Layer 2: Data-Link Layer

The data-link layer is the software closest to the hardware (physical layer). Thus, it includes, among other functions, any software needed to control the hardware. Bridging also occurs at this layer to allow networks interconnected with different physical layer protocols—for example, Ethernet LAN and an 802.11 LAN—to interconnect.

Like physical layer protocols, data-link layer protocols are classified as either LAN protocols, WAN protocols, or protocols that can be used for both LANs and WANs. Data-link layer protocols that are reliant on a specific physical layer may be limited to the transmission medium involved, but in some cases (for instance, PPP over RS-232 or PPP over Bluetooth's RFCOMM), data-link layer protocols can be ported to very different mediums if there is a layer that simulates the original medium the protocol was intended for, or if the protocol supports hardware-independent upper-data-link functionality. Data-link layer protocols are implemented in the system software layer, as shown in Figure 2-30.

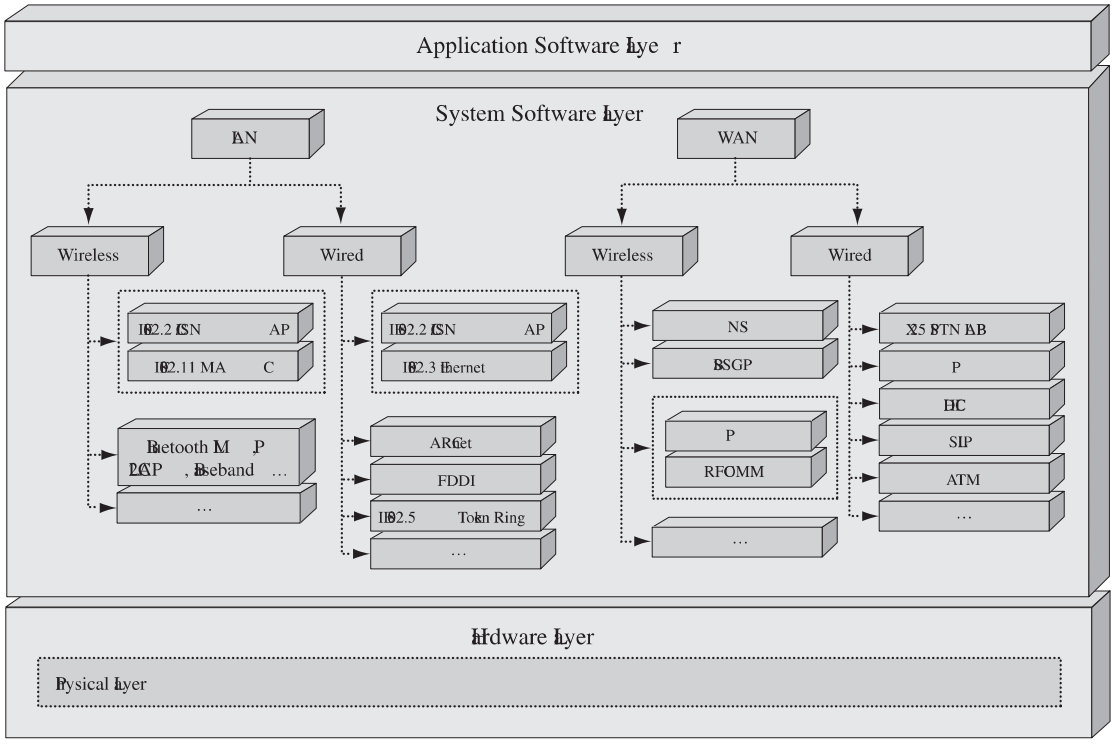


Figure 2-30: Data-link layer protocols

The data-link layer is responsible for receiving data bits from the physical layer and formatting these bits into groups, called data-link frames. Different data-link standards have varying data-link frame formats and definitions, but in general this layer reads the bit fields of these frames to ensure that entire frames are received, that these frames are error free, that the frame is meant for this device by using the physical address retrieved from the networking hardware on the device, and where this frame came from. If the data is meant for the device, then all data-link layer headers are stripped from the frame, and the remaining data field, called a ***datagram***, is passed up to the networking layer. These same header fields are appended to data coming down from upper layers by the data-link layer, and then the full data-link frame is passed to the physical layer for transmission (see Figure 2-31).

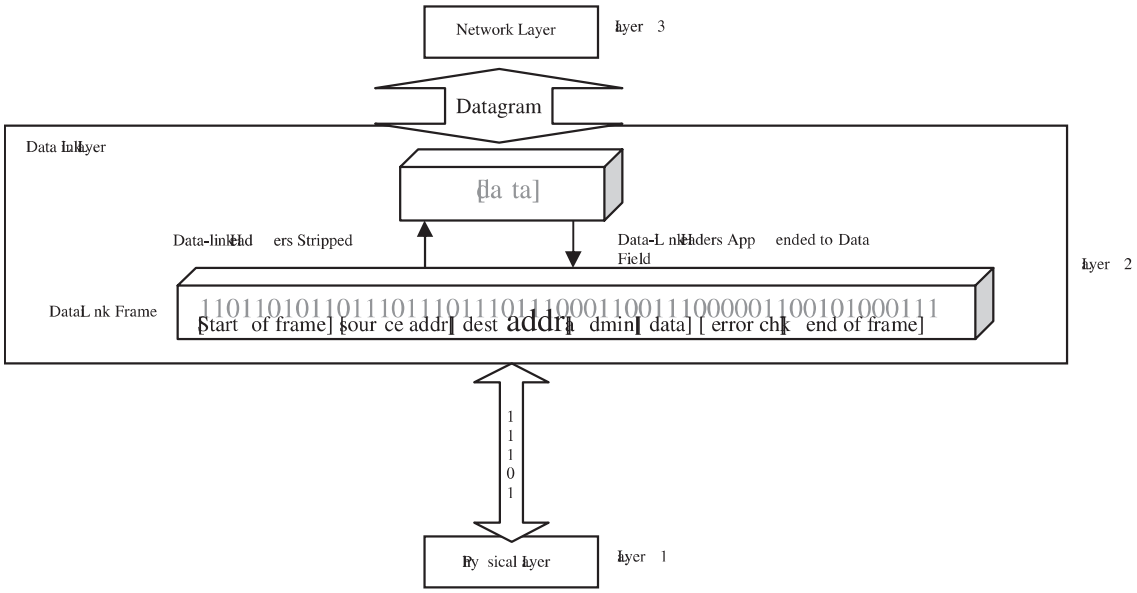


Figure 2-31: Data-link layer data flow block diagram

OSI Model Layer 3: Network Layer

Network layer protocols, like data-link layer protocols, are implemented in the system software layer, but unlike the lower data-link layer protocols, the network layer is typically hardware independent and only dependent on the data-link layer implementations (see Figure 2-32).

At the OSI network layer, networks can be divided into smaller sub-networks, called ***segments***. Devices within a segment can communicate via their physical addresses. Devices in different segments, however, communicate through an additional address, called the ***network address***. While the conversion between physical addresses and network addresses can occur in data-link layer protocols implemented in the device (i.e., ARP, RARP, etc.), network layer protocols can also convert between physical and networking addresses, as well as assign networking addresses. Through the network address scheme, the network layer manages datagram traffic and any routing of datagrams from the current device to another.

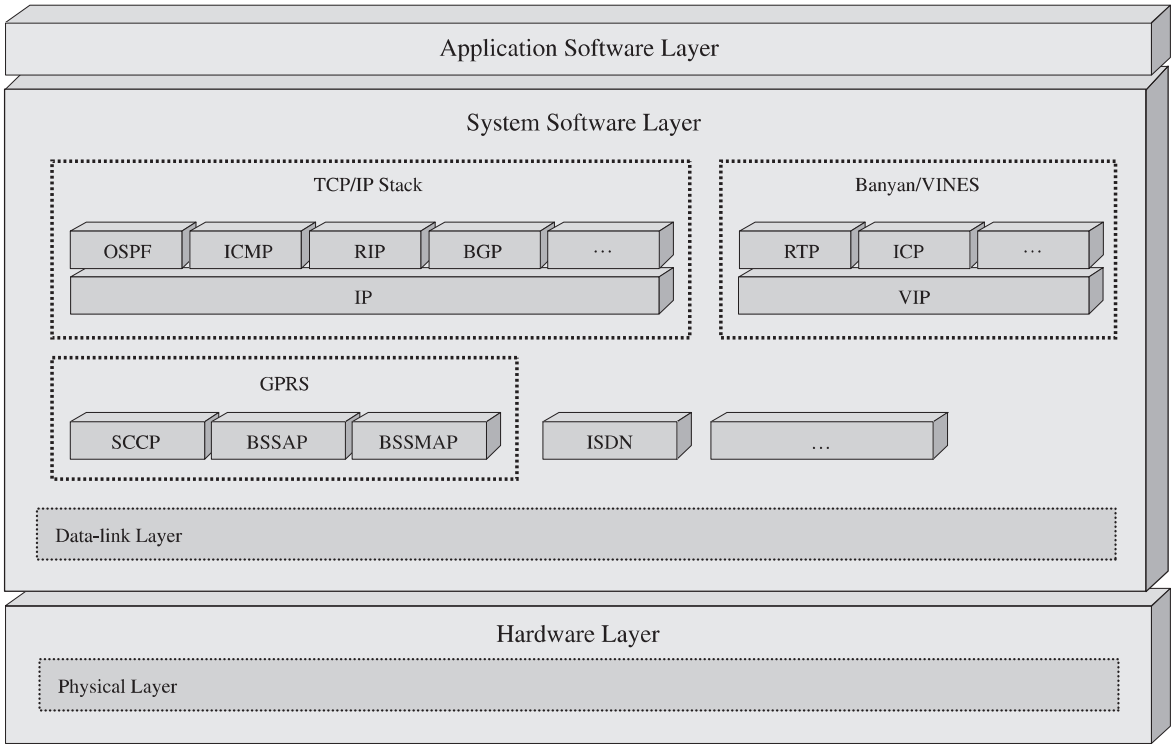


Figure 2-32: Network Layer protocols in the Embedded Systems Model

Like the data-link layer, if the data is meant for the device, then all network layer headers are stripped from the datagrams, and the remaining data field, called a ***packet***, is passed up to the transport layer. These same header fields are appended to data coming down from upper layers by the network layer, and then the full network layer datagram is passed to the data-link layer for further processing (see Figure 2-33). Note that the term "packet" is sometimes used to discuss data transmitted over a network, in general, in addition to data processed at the transport layer.

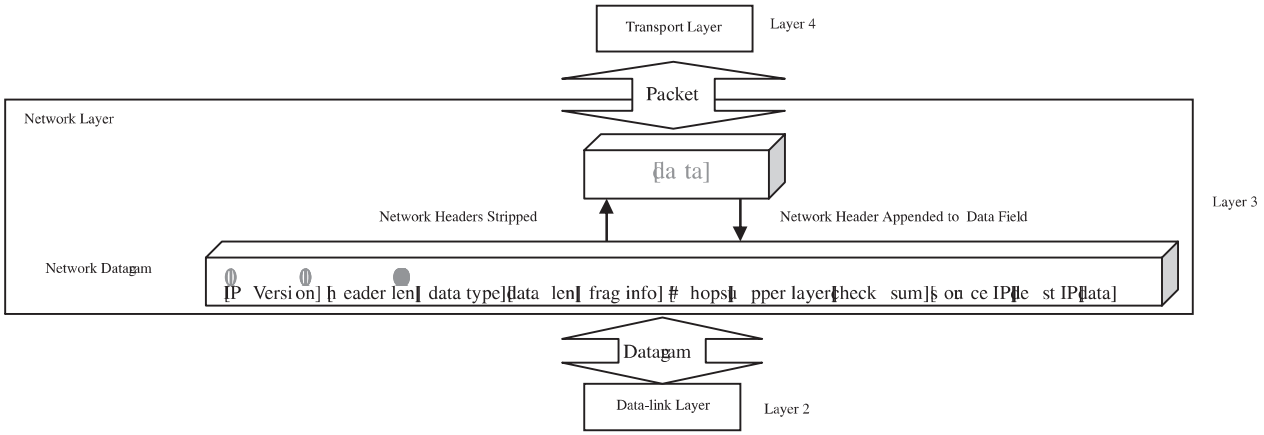


Figure 2-33: Network layer data flow block diagram

OSI Model Layer 4: Transport Layer

Transport layer protocols (shown in Figure 2-34) sit on top of and are specific to the network layer protocols. They are typically responsible for establishing and dissolving communication between two specific devices. This type of communication is referred to as ***point-to-point*** communication. Protocols at this layer allow for multiple higher-layer applications running on the device to connect point-to-point to other devices. Some transport layer protocols can also ensure reliable point-to-point data transmission by ensuring that packets are received and transmitted in the correct order, are transmitted at a reasonable rate (flow control), and the data within the packets are not corrupted. Transport layer protocols can provide the acknowledgments to other devices upon receiving packets, and request packets to be retransmitted if an error is detected.

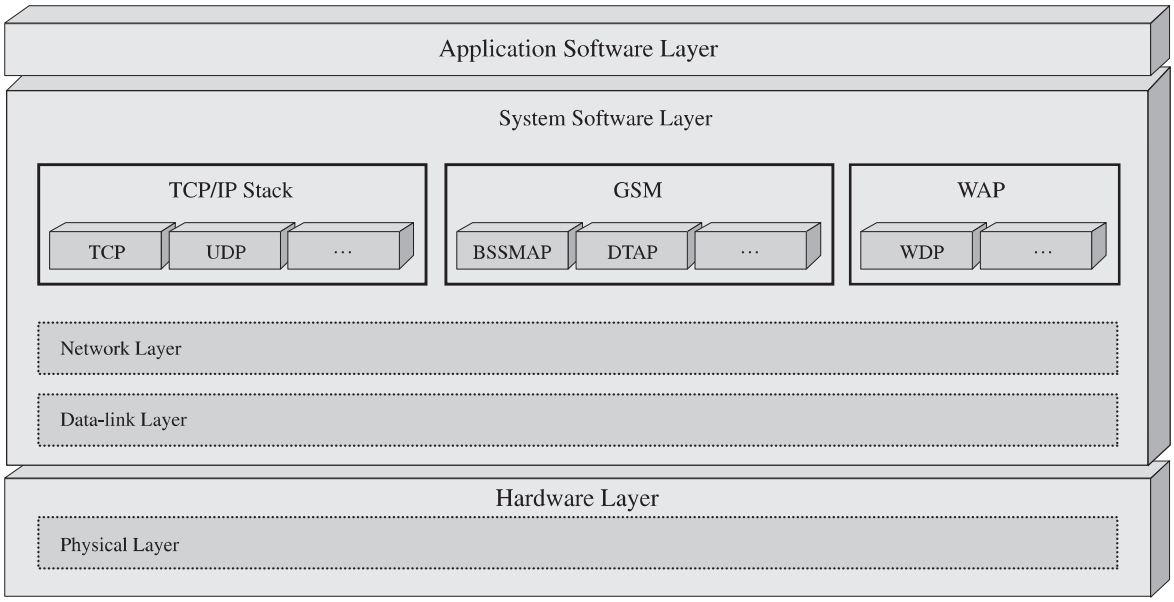


Figure 2-34: Transport layer protocols in the Embedded Systems Model

In general, when the transport layer processes a packet received from lower layers, then all transport layer headers are stripped from the packets, and the remaining data fields from one or multiple packets are reassembled into other packets, also referred to as ***messages,*** and passed to upper layers. Messages/packets are received from upper layers for transmission, and are divided into separate packets if too long. The transport layer header fields are then appended to the packets, and passed down to lower layers for further processing (see Figure 2-35).

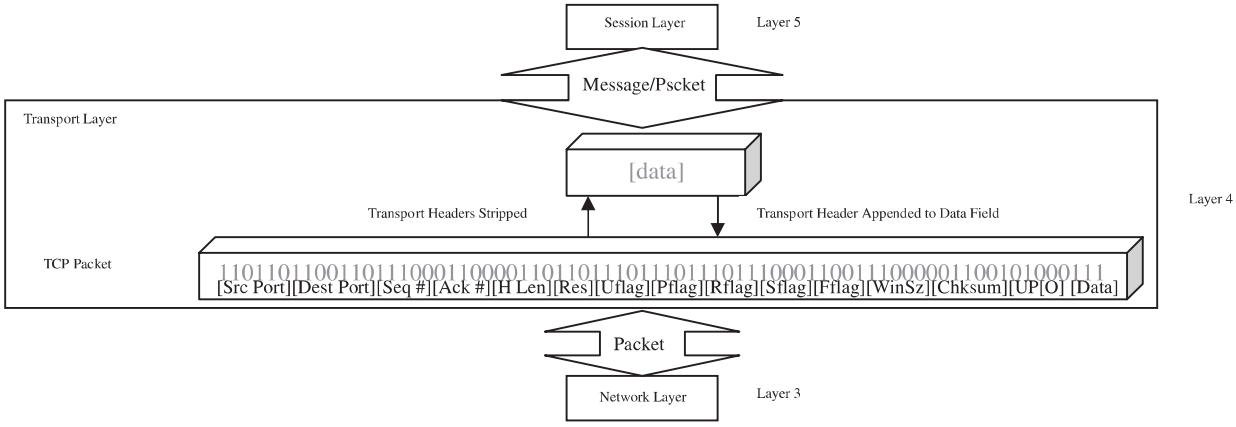


Figure 2-35: Transport layer data flow block diagram

OSI Model Layer 5: Session Layer

The connection between two networking applications on two different devices is called a ***session***. Where the transport layer manages the point-to-point connection between devices for multiple applications, the management of a session is handled by the session layer, as shown in Figure 2-36. Generally, sessions are assigned a port (number), and the session layer protocol must separate and manage each session's data, regulate the data flow of each session, handle any errors that arise with the applications involved in the session, and ensure the security of the session—for example, that the two applications involved in the session are the right applications.

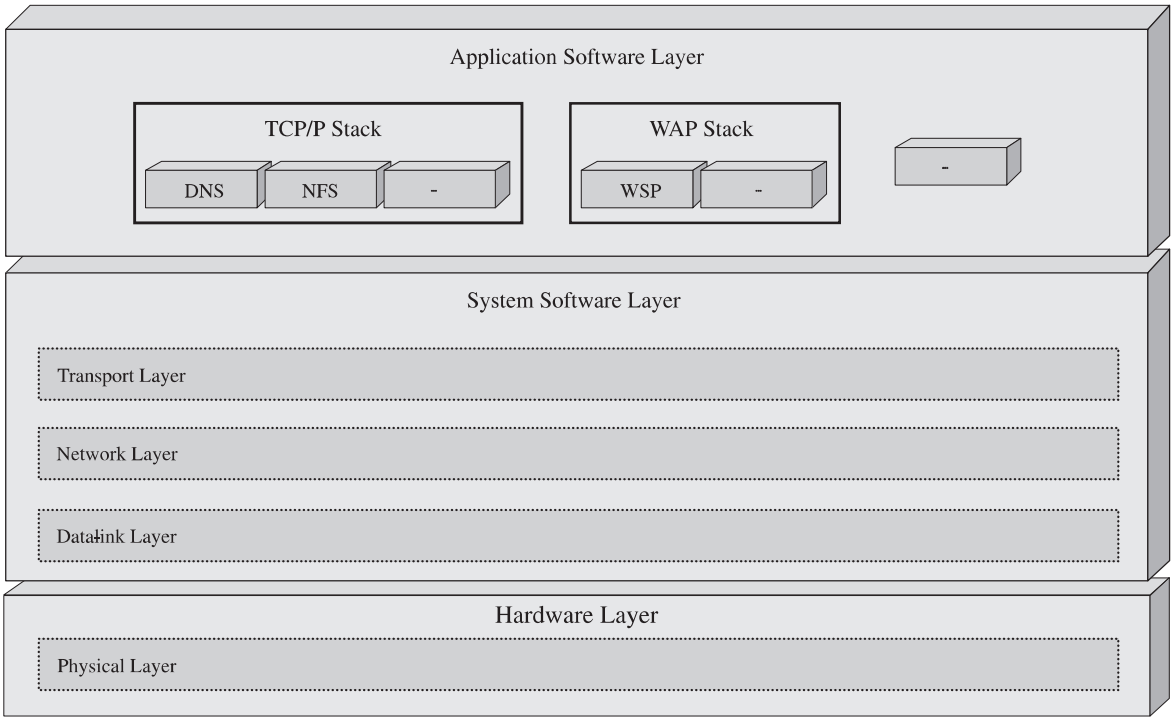


Figure 2-36: Session layer protocols in the Embedded Systems Model

When the session layer processes a message/packet received from lower layers, then all session layer headers are stripped from the messages/packets, and the remaining data field is passed to upper layers. Messages that are received from upper layers for transmission are appended with session layer header fields and passed down to lower layers for further processing (see Figure 2-37).

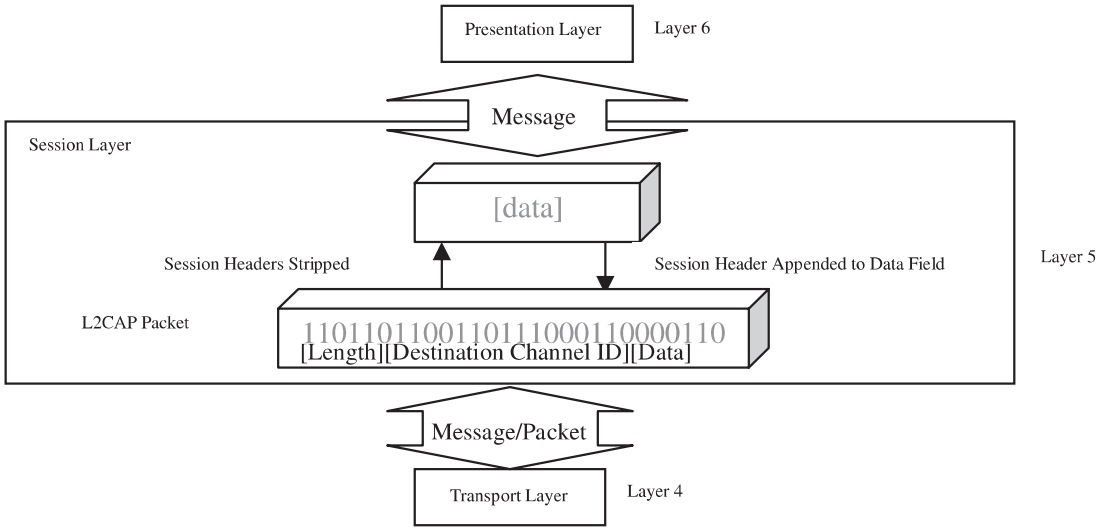


Figure 2-37: Session layer data flow block diagram

OSI Model Layer 6: Presentation Layer

Protocols at the presentation layer are responsible for translating data into formats that higher applications can process, or translating data going to other devices into a generic format for transmission. Generally, data compression/decompression, data encryption/decryption, and data protocol/character conversions are implemented in presentation layer protocols. Relative to the Embedded Systems Model, presentation layer protocols are usually implemented in networking applications found in the application layer as shown in Figure 2-38.

Basically, a presentation layer processes a message received from lower layers, and then all presentation layer headers are stripped from the messages, and the remaining data field is passed to upper layers. Messages that are received from upper layers for transmission are appended with presentation layer header fields and passed down to lower layers for further processing (see Figure 2-39).

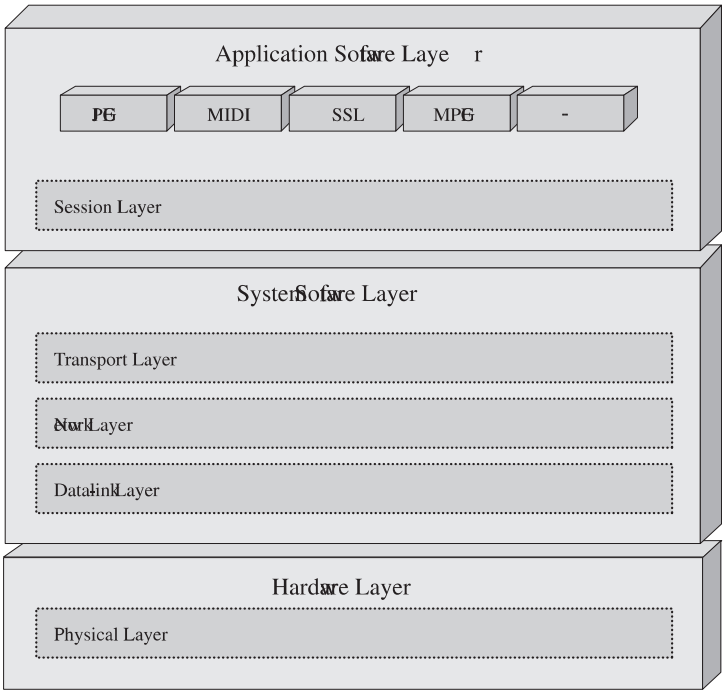


Figure 2-38: Presentation layer protocols in the Embedded Systems Model

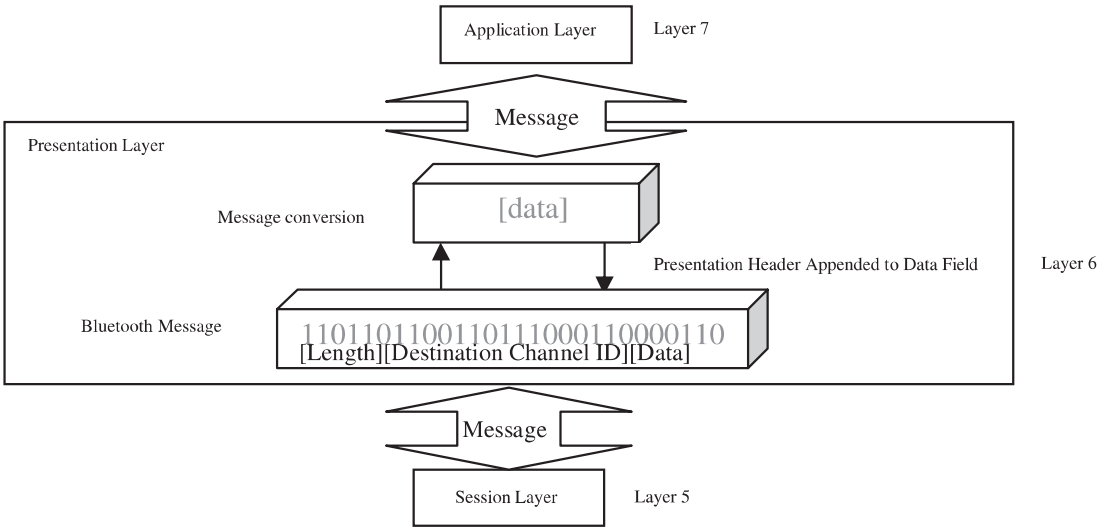


Figure 2-39: Presentation layer data flow block diagram

OSI Model Layer 7: Application Layer

A device initiates a network connection to another device at the application layer (shown in Figure 2-40). In other words, application layer protocols are either used directly as network applications by end-users or the protocols are implemented in end-user network applications (see Chapter 10). These applications "virtually" connect to applications on other devices.

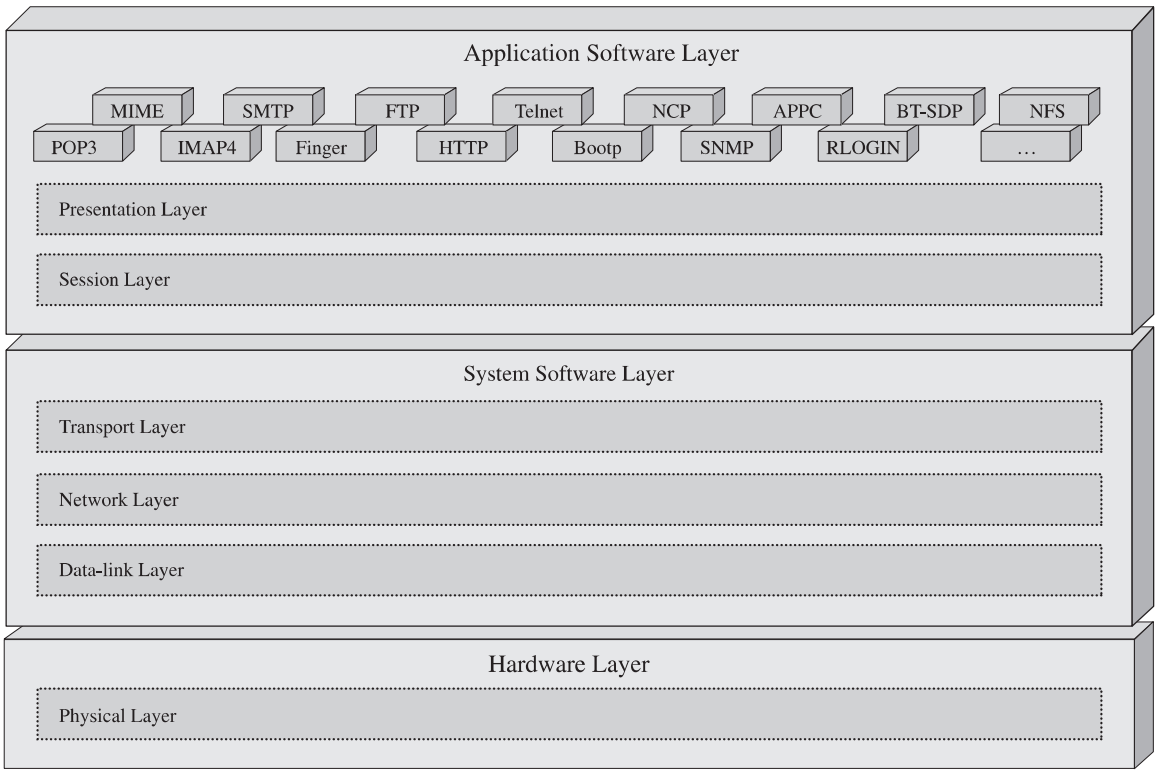


Figure 2-40: Application layer protocols in the Embedded Systems Model

2.3 Multiple Standards-Based Device Example: Digital Television (DTV)

Why Use Digital TV as a Standards Example?

"There's a frenzy about portals on the internet, but the average person spends about an hour online. The average consumer spends seven (7) hours a day watching TV, and TV is in 99% of U.S. homes."

—Forrester Research

Analog TVs process incoming analog signals of traditional TV video and audio content, whereas digital TVs (DTVs) process both incoming analog and digital signals of TV video/audio content, as well as application data content that is embedded within the entire digital

data stream (a process called data broadcasting or data casting). This application data can either be unrelated to the video/audio TV content (noncoupled), related to video/audio TV content in terms of content but not in time (loosely coupled), or entirely synchronized with TV audio/video (tightly coupled).

The type of application data embedded is dependent on the capabilities of the DTV receiver itself. While there are a wide variety of DTV receivers, most fall under one of three categories: enhanced broadcast receivers, which provide traditional broadcast TV enhanced with graphics controlled by the broadcast programming; interactive broadcast receivers, capable of providing e-commerce, video-on-demand, e-mail, and so on through a return channel on top of "enhanced" broadcasting; and multi-network receivers that include internet and local telephony functionality on top of interactive broadcast functionality. Depending on the type of receiver, DTVs can implement general-purpose, market-specific, and/or application-specific standards all into one DTV/set-top box (STB) system architecture design (shown in Table 2-7).

Table 2-7: Examples of DTV standards

Standard Type	Standard
Market Specific	Digital video broadcasting (DVB) – multimedia home platform (MHP)
	Java TV
	Home audio/video interoperability (HAVi)
	Digital Audio Video Council (DAVIC)
	Advanced Television Standards Committee (ATSC)/Digital TV Applications Software Environment (DASE)
	Advanced Television Enhancement Forum (ATVEF)
	Digital Television Industrial Alliance of China (DTVIA)
	Association of Radio Industries and Business of Japan (ARIB-BML)
	OpenLabs OpenCable application platform (OCAP)
	Open services gateway initiative (OSGi)
	OpenTV
	MicrosoftTV
General Purpose	HTTP (hypertext transfer protocol) – in browser applications
	POP3 (post office protocol) – in e-mail application
	IMAP4 (Internet message access protocol) – in e-mail application
	SMTP (simple mail transfer protocol) – in e-mail application
	Java
	Networking (terrestrial, cable, and satellite)
	POSIX

These standards then can define several of the major components that are implemented in all layers of the DTV Embedded Systems Model, as shown in Figure 2-41.

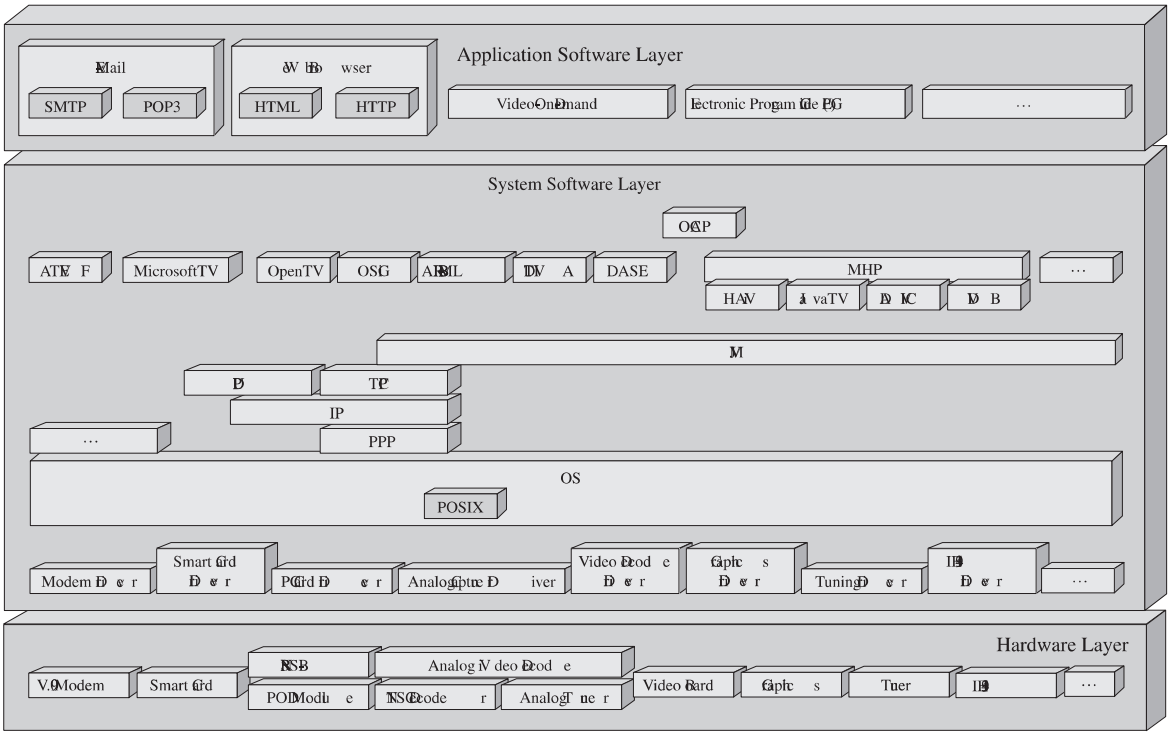


Figure 2-41: DTV standards in the Embedded Systems Model

2.4 Summary

The purpose of this chapter was to show the importance of industry-supported standards when trying to understand and implement embedded system designs and concepts. The programming language, networking, and DTV examples provided in this chapter demonstrated how standards can define major elements within an embedded architecture. The programming language example provided an illustration of general-purpose standards that can be implemented in a wide variety of embedded devices. These examples specifically included Java, showing how a JVM was needed at the application, system, or hardware layer, and the .NET Compact Framework, for languages such as C# and Visual Basic, which demonstrated a programming language element that must be integrated into the systems software layer. Networking provided an example of standards that can be general purpose, specific to a family of devices (market driven), or specific to an application (http in a browser, for instance). In the case of networking, both the embedded systems and the OSI models were referenced to demonstrate where certain networking protocols fit into an embedded architecture. Finally, the digital TV STB example illustrated how one device implemented several standards that defined embedded components in all layers.

Chapter 3, *Embedded Hardware Building Blocks and the Embedded Board*, is the first chapter of *Section II: Embedded Hardware*. Chapter 3 introduces the fundamental elements found on an embedded board and some of the most common basic electrical components that make up these elements, as well as those that can be found independently on a board.

Chapter 2 Problems

1. How can embedded system standards typically be classified?

2. [a] Name and define four groups of market-specific standards.
   [b] Give three examples of standards that fall under each of the four market-specific groups.

3. [a] Name and define four classes of general-purpose standards.
   [b] Give three examples of standards that fall under each of the four general-purpose groups.

4. Which standard below is neither a market-specific nor a general-purpose embedded systems standard?
   A. HTTP – Hypertext Transfer Protocol.
   B. MHP – Multimedia Home Platform.
   C. J2EE – Java 2 Enterprise Edition.
   D. All of the above.
   E. None of the above.

5. [a] What is the difference between a high-level language and a low-level language?
   [b] Give an example of each.

6. [Select] A compiler can be located on:
   A. a target.
   B. a host.
   C. on a target and/or on a host.
   D. None of the above.

7. [a] What is the difference between a cross-compiler and a compiler?
   [b] What is the difference between a compiler and an assembler?

8. [a] What is an interpreter?
   [b] Give two examples of interpreted languages.

9. [T/F] All interpreted languages are scripting languages but not all scripting languages are interpreted.

10. [a] In order to run Java, what is required on the target?
    [b] How can the JVM be implemented in an embedded system?

11. Which standards below are embedded Java standards?
    A. pJava – Personal Java.
    B. RTSC – Real Time Core Specification.
    C. HTML – Hypertext Markup Language.
    D. A and B only.
    E. A and C only.

12. What are the two main differences between all embedded JVMs?

13. Name and describe three of the most common byte processing schemes.

14. [a] What is the purpose of a GC?
    [b] Name and describe two common GC schemes.

15 [a] Name three qualities that Java and scripting languages have in common.
    [b] Name two ways that they differ.

16. [a] What is the .NET Compact Framework?
    [b] How is it similar to Java?
    [c] How is it different?

17. What is the difference between LANs and WANs?

18. What are the two types of transmission mediums that can connect devices?

19. [a] What is the OSI model?
    [b] What are the layers of the OSI model?
    [c] Give examples of two protocols under each layer.
    [d] Where in the Embedded Systems Model does each layer of the OSI model fall? Draw it.

20. [a] How does the OSI model compare to the TCP/IP model?
    [b] How does the OSI model compare to Bluetooth?

Section I: Endnotes

Chapter 1: *Endnotes*

[1-1] *Embedded Microcomputer Systems*, Valvano, p. 3; *Embedded Systems Building Blocks*, Labrosse, p. 61.

[1-2] The Embedded Systems Design and Development Lifecycle Model is specifically derived from the SEI's Evolutionary Delivery Lifecycle Model and the Software Development Stages Model.

[1-3] The six stages of creating an architecture outlined and applied to embedded systems in this book are inspired by the Architecture Business Cycle developed by SEI. For more on this brain child of SEI, read "Software Architecture in Practice," by Bass, Clements, and Kazman which does a great job in capturing and articulating the process that so many of us have taken for granted over the years, or not even have bothered to think about. While SEI focuses on software in particular, their work is very much applicable to the entire arena of embedded systems, and I felt it was important to introduce and recognize the immense value of their contributions as such.

[1-4] *Software Architecture in Practice*, Bass, Clements, Kazman; *Real-Time Design Patterns,* Douglass.

[1-5] *Software Testing*, Ron Patton, pp. 31–36.

Chapter 2: *Endnotes*

[2-1] *Embedded System Building Blocks*, Labrosse, Jean, p. 61; "Embedded Microcomputer Systems," Volvano, Jean, p. 3, Table 1.1.

[2-2] http://www.mhp.org/what_is_mhp/index.html.

[2-3] http://java.sun.com/products/javatv/overview.html.

[2-4] http://www.davic.org.

[2-5] http://www.atsc.org/standards.html.

[2-6] http://www.atvef.com.

[2-7] http://java.sun.com/pr/2000/05/pr000508-02.html and http://netvision.qianlong.com/8737/2003-6-4/39@878954.htm.

[2-8] http://www.arib.or.jp.

[2-9] http://www.osgi.org/resources/spec_overview.asp.

[2-10] http://www.opentv.com.

[2-11] http://www.microsoft.com/tv/default.mspx.

[2-12] "HAVi, the A/V digital network revolution," Whitepaper, p. 2.

[2-13] http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfStandards/search.cfm

[2-14] http://europa.eu.int/smartapi/cgi/sga_doc?smartapi!celexapi!prod!CELEXnumdoc&lg = E N&numdoc=31993L0042&model=guichett"

[2-15] http://www.ieee1073.org.

[2-16] *Digital Imaging and Communications in Medicine (DICOM): Part 1: Introduction and Overview*, p. 5.

[2-17] http://www.ce.org/standards/default.asp.

[2-18] http://europa.eu.int/comm/enterprise/mechan_equipment/machinery/.

[2-19] https://domino.iec.ch/webstore/webstore.nsf/artnum/025140.

[2-20] http://www.iso.ch/iso/en/CatalogueListPage.CatalogueList? ICS1=25&ICS2=40&ICS3=1.

[2-21] "Bluetooth Protocol Architecture," whitepaper. Riku Mettala, p. 4.

[2-22] *Systems Analysis and Design*, Harris, David, p. 17.

[2-23] "I/Opener," Morin and Brown, Sun Expert Magazine, 1998.

[2-24] "Boehm-Demers-Weiser conservative garbage collector: A garbage collector for C and C++", Hans Boehm, http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[2-25] "Selecting the Right Transmission Medium Optical fiber is moving to the forefront of transmission media choices leaving twisted copper and coaxial cable behind," Curt Weinstein.

[2-26] "This Is Microwave" Whitepaper, Stratex Networks. http://www.stratexnet.com/about_us/our_technology/tutorial/This_is_Microwave_expanded.pdf, "Satellite, Bandwidth Without Borders" whitepaper. Scott Beardsley, Pär Edin and Alan Miles.

[2-27] http://www.ihs.com/standards/vis/collections.html and "IHS: Government Information Solutions," p. 4.

[2-28] http://standards.ieee.org/reading/ieee/std_public/description/busarch/1284.1-1997_desc.html.

[2-29] http://www.aimglobal.org/aimstore/linearsymbologies.htm.

[2-30] http://www.iaob.org/iso_ts.html.

[2-31] http://www.praxiom.com/iso-intro.htm and the "Fourth-Generation Languages" whitepaper. Sheri Cummings. p. 1.

[2-32] "Spread Spectrum: Regulation in Light of Changing Technologies," whitepaper, Carter, Garcia, and Pearah, p. 7.