Defining the System—Creating the Architecture and Documenting the Design

● ●

In This Chapter

▶ *Defining the stages of creating an embedded systems architecture*
▶ *Introducing the architecture business cycle and its effect on architecture*
▶ *Describing how to create and document an architecture*
▶ *Introducing how to evaluate and reverse engineer an architecture*

● ●

This chapter is about giving the reader some practical processes and techniques that have proven useful over the years. Defining the system and its architecture, if done correctly, is the phase of development which is the *most difficult* and the *most important* of the entire development cycle. Figure 11-1 shows the different phases of development as defined by the **Embedded System Design and Development Lifecycle Model**. [11-1]
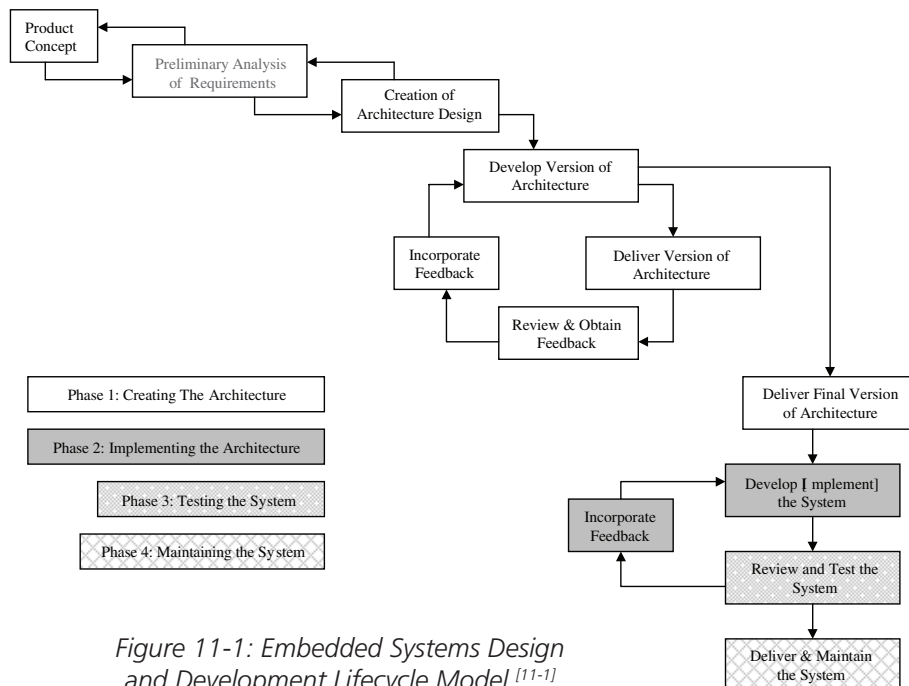

Figure 11-1: Embedded Systems Design and Development Lifecycle Model [11-1]

509

This model indicates that the process of designing an embedded system and taking that design to market has four phases:

- *Phase 1. Creating the Architecture,* which is the process of planning the design of the embedded system.
- *Phase 2. Implementing the Architecture*, which is the process of developing the embedded system.
- *Phase 3. Testing the System*, which is the process of testing the embedded system for problems, and then solving those problems.
- *Phase 4. Maintaining the System*, which is the process of deploying the embedded system into the field, and providing technical support for users of that device for the duration of the device's lifetime.

This model also indicates that the most important time is spent in phase 1, creating the architecture. At this phase of the process, no board is touched and no software is coded. It is about putting full attention, concentration and investigative skills into gathering information about the device to be developed, understanding what options exist, and documenting those findings. If the right preparation is done in defining the system's architecture, determining requirements, understanding the risks, and so on, then the remaining phases of development, testing and maintaining the device will be simpler, faster, and cheaper. This, of course assumes that the engineers responsible have the necessary skills.

In short, if phase 1 is done correctly, then less time will be wasted on deciphering code that doesn't meet the system requirements, or guessing what the designers' intentions were, which most often results in more bugs, and more work. That is not to say that the design process is always smooth sailing. Information gathered can prove inaccurate, specifications can change, and so on, but if the system designer is technically disciplined, prepared, and organized, new hurdles can be immediately recognized and resolved. This results in a development process that is much less stressful, with less time and money spent and wasted. Most importantly, the project will, from a technical standpoint, almost certainly end in success.

11.1 Creating an Embedded System Architecture

Several industry methodologies can be adopted in designing an embedded system's architecture, such as the Rational Unified Process (RUP), Attribute Driven Design (ADD), the object-oriented process (OOP), and the model driven architecture (MDA), to name a few. Within this book, I've taken a pragmatic approach by introducing a process for creating an architecture that combines and simplifies many of the key elements of these different methodologies. This process consists of six stages, where each stage builds upon the results of the previous stages. These stages are:

- *Stage 1*. Having a solid technical base
- *Stage 2*. Understanding the architecture business cycle
- *Stage 3*. Defining the architectural patterns and reference models
- *State 4*. Creating the architectural structures

- *Stage 5*. Documenting the architecture
- *Stage 6*. Analyzing and evaluating the architecture

These six stages can serve as a basis for further study of one of the many, more complex, architectural design methodologies in the industry. However, if given a limited amount of time and resources to devote to full-time architectural methodology studies of the many industry schemes before having to begin the design of a real product, these six stages can be used directly as a simple model for creating an architecture. The remainder of this chapter will provide more details on the six stages.

> *Author note: This book attempts to give a pragmatic process for creating an embedded systems architecture based upon some of the mechanisms that exist in the more complex industry approaches. I try to avoid using a lot of the specific terminology associated with these various methodologies because the same terms across different approaches can have different definitions as well as different terms can have identical meanings.*

Stage 1: Have a Solid Technical Foundation

In short, stage 1 is about understanding the material presented in Chapters 2 through 10 of this book. Regardless of what portion of the embedded system an engineer or programmer will develop or work on, it is useful and practical to understand at a systems engineering level *all of the elements that can be implemented in an embedded system*. This includes the possible permutations of both the hardware and software as represented in the embedded systems model, such as the von Neumann model, reflecting the major components that can be found on an embedded board (shown in Figure 11-2a) or the possible complexities that can exist in the system software layer (shown in Figure 11-2b).
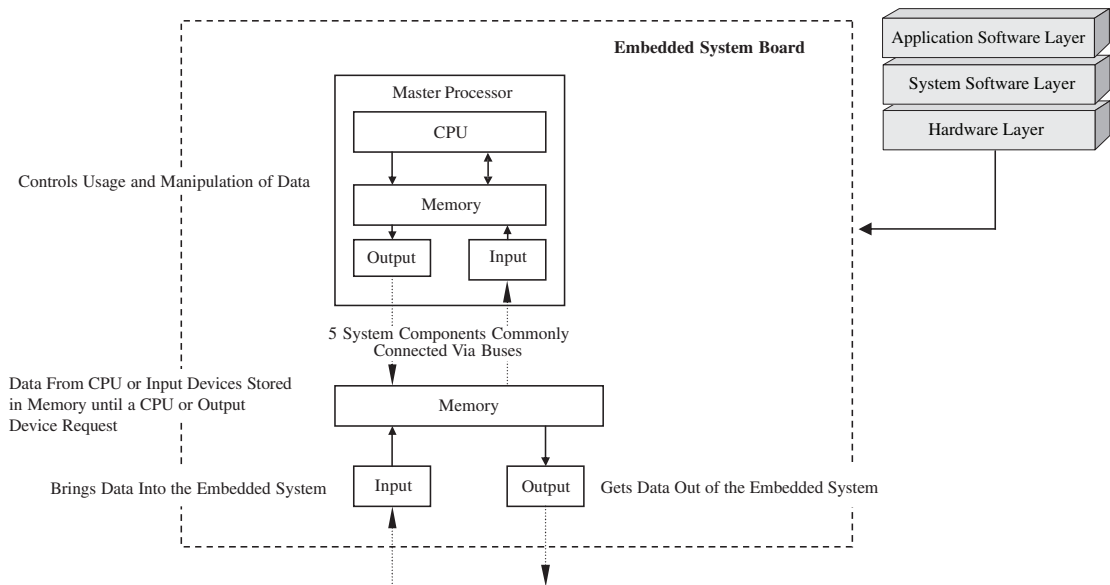

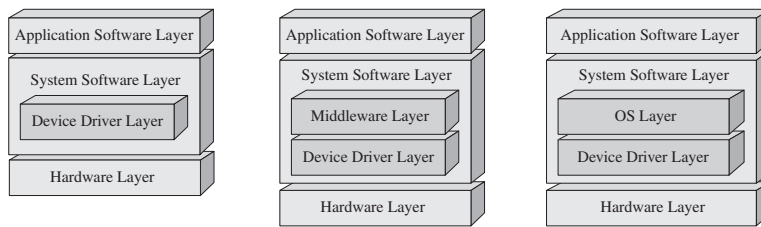*Figure 11-2a: von Neumann and Embedded Systems Model diagrams*

Figure 11-2b: System Software Layer and Embedded Systems Model diagrams

Stage 2: Know the ABCs (Architecture Business Cycles) of Embedded Systems

The Architecture Business Cycle (ABC)[11-2] of an embedded device, shown in Figure 11-3, is the cycle of influences that impact the architecture of an embedded system, and the influences that the embedded system in turn has on the environment in which it is built. These influences can be technical, business-oriented, political, or social. In short, the *ABCs of embedded systems* are the *many* different types of influences that generate the requirements of the system, the requirements in turn generate the architecture, the architecture then produces the system, and the resulting system in turn provides requirements and capabilities back to the organization for future embedded designs.
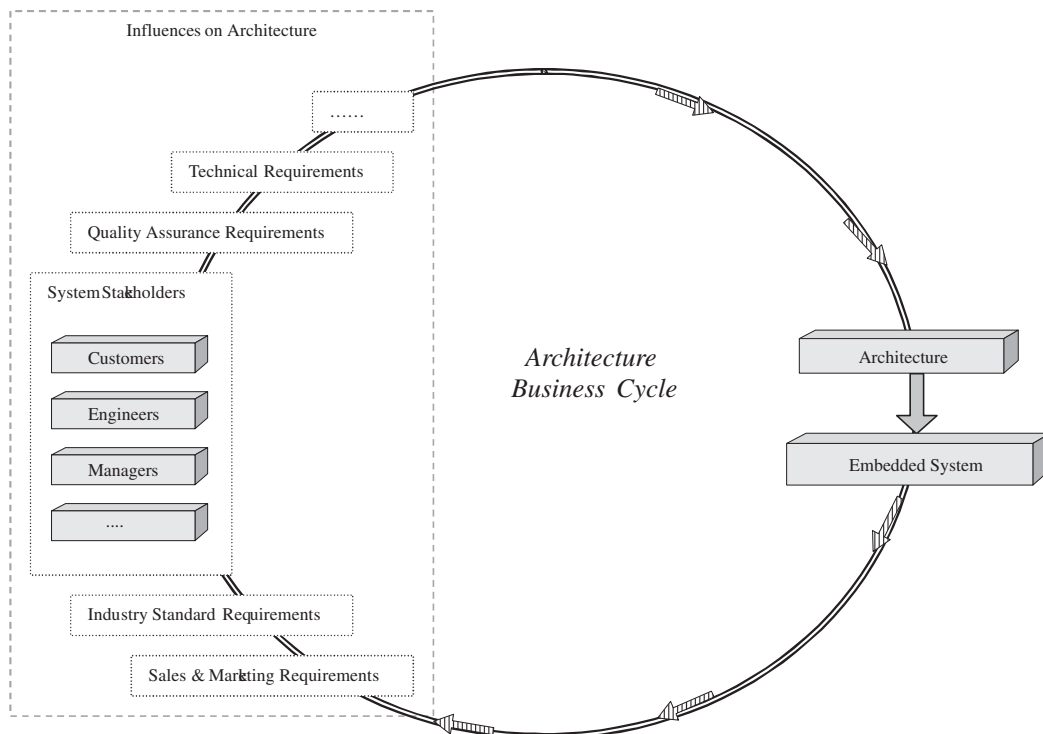

Figure 11-3: Architecture Business Cycle [11-2]

What this model implies is that, for better or worse, architectures are not designed on technical requirements alone. For example, given the same type of embedded system, such as a cell phone or TV, with the exact same technical requirements designed by different design teams, the different architectures produced that incorporate different processors, OSes, and other elements. An engineer that recognizes this from the start will have much more success in creating the architecture for an embedded system. If the responsible architects of an embedded system identify, understand, and engage the various influences on a design at the start of the project, it is less likely that any of these influences will later demand design changes or delays after a lot of time, money, and effort has gone into developing the original architecture.

The steps of Stage 2 include:

- *Step 1*. Understanding that ABC influences drive the requirements of an embedded system, and that these influences are not limited to technical ones.

- *Step 2*. Specifically identifying all the ABC influences on the design, whether technical, business, political and/or social.

- *Step 3*. Engaging the various influences as early as possible in the design and development lifecycle and gathering the requirements of the system.

- *Step 4*. Determining the possible hardware and/or software elements that would meet the gathered requirements.

The previous page introduced steps 1 and 2 in detail, and the next few sections of this chapter will discuss steps 3 and 4 in more depth.

Gathering the Requirements

Once the list of influences has been determined, then the architectural requirements of the system can be gathered. The process by which information is obtained from the various influences in the ABC can vary, depending on the project, from the *informal*, such as word-of-mouth (not recommended), to *formal* methods in which requirements are obtained from finite-state machine models, formal specification languages, and/or scenarios, to name a few. Regardless of the method used for gathering requirements, what is important to remember is that information should be gathered *in writing*, and any documentation, no matter how informal (even if it is written on a napkin), should be *saved*. When requirements are in writing, it decreases the probability of confusion or past communication disagreements involving requirements, because the written documentation can be referenced to resolve related issues.

The kind of information that must be gathered includes both the functional and nonfunctional requirements of the system. Because of the wide variety of embedded systems, it is difficult in this book to provide a list of functional requirements that could apply to all embedded systems. Nonfunctional requirements, on the other hand, can apply to a wide variety of embedded systems and will be used as real-world examples later in this chapter. Furthermore, from nonfunctional requirements certain functional requirements can be derived. This can be useful for those that have no specific functional requirements at the start of a project, and only

have a general concept of what the device to be designed should be able to do. Some of the most useful methods for deriving and understanding nonfunctional requirements are through outlining **general ABC features** and utilizing **prototypes**.

General ABC features are the characteristics of a device that the various influence types require. This means that the nonfunctional requirements of the device are based upon general ABC features. In fact, because most embedded systems commonly require some combination of general ABC features, they can be used as a starting point in defining and capturing system requirements for just about any embedded system. Some of the most common features acquired from various general ABC influences are shown in Table 11-1.

Table 11-1: Examples of General ABC features

Influence	Feature	Description
Business [Sales, Marketing, Executive Management, etc.]	Sellability	How the device will sell, will it sell, how many will it sell, etc.
	Time-to-Market	When will the device be delivered with what technical features, etc.
	Costs	How much will the device cost to develop, how much can it sell for, is there overhead, how much for technical support once device is in field, etc.
	Device Lifetime	How long the device will be on the market, how long will the device be functionable in the field, etc.
	Target Market	What type of device is it, who will buy it, etc.
	Schedule	When will it be in production, when will it be ready to be deployed to the marketplace, when does it have to be completed, etc.
	Capability	Specifying the list of features that the device needs to have for the target market, understanding what the device actually can do once it ships from production, are there any serious bugs in the shipping product, etc.
	Risks	Risks of lawsuits over device features or malfunctions, missing scheduled releases, not meeting customer expectations, etc.
Technical	Performance	Having the device appear to the user to be functioning fast enough, having the device do what it is supposed to be doing, throughput of processor, etc.
	User Friendliness	How easy it is to use, pleasant or exciting graphics, etc.
	Modifiability	How fast it can be modified for bug fixes or upgrades, how simple it is to modify, etc.
	Security	Is it safe from competitors, hackers, worms, viruses, and even idiot-proof, etc.
	Reliability	Does it crash or hang, how often does it crash or hang, what happens if it crashes or hangs, what can cause it to crash or hang, etc.
	Portability	How simple is it to run the applications on different hardware, on different system software, etc.
	Testability	How easily can the system be tested, what features can be tested, how can they be tested, are there any built in features to allow testing, etc.
	Availability	Will any of the commercial software or hardware implemented in the system be available when needed, when will they be available, what are the reputations of vendors, etc.
	Standards	(See Industry below.)
	Schedule	(See Business above.)

Table 11-1: Examples of General ABC features (continued)

Influence	Feature	Description
Industry	Standards	Industry standards (introduced in Chapter 2), which may be market specific (i.e., TV standards, medical device standards, etc.) or general purpose across different families of devices (programming language standards, networking standards, etc.)
Quality Assurance	Testability	(See Technical above.)
	Availability	When is the system available to be tested
	Schedule	(See Business above.)
	Features	(See Business above.)
	QA Standards	ISO9000, ISO9001, and so on. (See industry above.)
Customers	Cost	How much will the device cost, how much will it cost to repair or upgrade, etc.
	User Friendliness	(See Technical above.)
	Performance	(See Technical above.)

Another useful tool in understanding, capturing, and modeling system requirements is through utilizing a system *prototype*, a physically running model containing some combination of system requirements. A prototype can be used to define hardware and software elements that could be implemented in a design, and indicate any risks involved in using these elements. Using a prototype in conjunction with the general ABC features allows you to accurately determine early in the project what hardware and software solutions would be the most feasible for the device to be designed.

A prototype can be developed from scratch or can be based upon a currently deployed product in the market, which could be any similar devices or more complex devices that incorporate the desirable functionality. Even devices in other markets may have the desired look and feel even if the applications are not what you are looking for. For example, if you want to design a wireless medical handheld device for doctors, consumer PDAs (shown in Figure 11-4) that have been successfully deployed into the market could be studied and adapted to support the requirements and system architecture of the medical device.
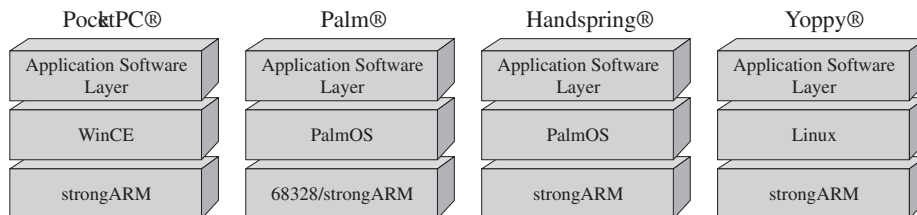

Figure 11-4: PDAs

When comparing your product to similar designs already on the market, also notice when what was adopted in the design wasn't necessarily the best technical solution for that product. Remember, there could have been a multitude of nontechnical reasons, from nontechnical influences, that a particular hardware or software component was implemented.

Some of the main reasons for looking at similar solutions is to save time and money by gathering ideas of what is feasible, what problems or limitations were associated with a particular solution, and, if technically the prototype is a reasonable match, understanding why that is the case. If there really is no device on the market that mirrors any of the requirements of your system, using an *off-the-shelf reference board* and/or *off-the-shelf system software* is another quick method to create your own prototype. Regardless of how the prototype is created, it is a useful tool in modeling and analyzing the design and behavior of a potential architecture.

Deriving the Hardware and Software from the Requirements

Understanding and applying the requirements to derive feasible hardware and/or software solutions for a particular design can be accomplished through:

1. Defining a set of scenarios that outlines each of the requirements.

2. Outlining tactics for each of the scenarios that can be used to bring about the desired system response.

3. Using the tactics as the blueprint for what **functionality** is needed in the device, and then deriving a list of specific hardware and/or software elements that contain this functionality.

As shown in Figure 11-5, outlining a scenario means defining:

- the external and internal *stimulus sources* that interact with the embedded system

- the actions and events, or *stimuli*, that are caused by the stimulus sources

- the *environment* which the embedded system is in when the stimulus takes place, such as in the field under normal stress, in the factory under high stress, outdoors exposed to extreme temperature, indoors, and so on.

- the *element*s of the embedded system that could be affected by the stimulus, whether it is the entire system or the general hardware or software element within such as memory, the master processor, or data, for example.

- the desired *system response* to the stimulus, which reflects one or more system requirements.

- how the system response can be *measured*, meaning how to prove the embedded system meets the requirements.

After outlining the various scenarios, the tactics can then be defined that bring about the desired system response. These tactics can be used to determine what type of functionality is needed in the device. These next several examples demonstrate how hardware and software components can be derived from nonfunctional requirements based upon performance, security, and testability general ABC features.
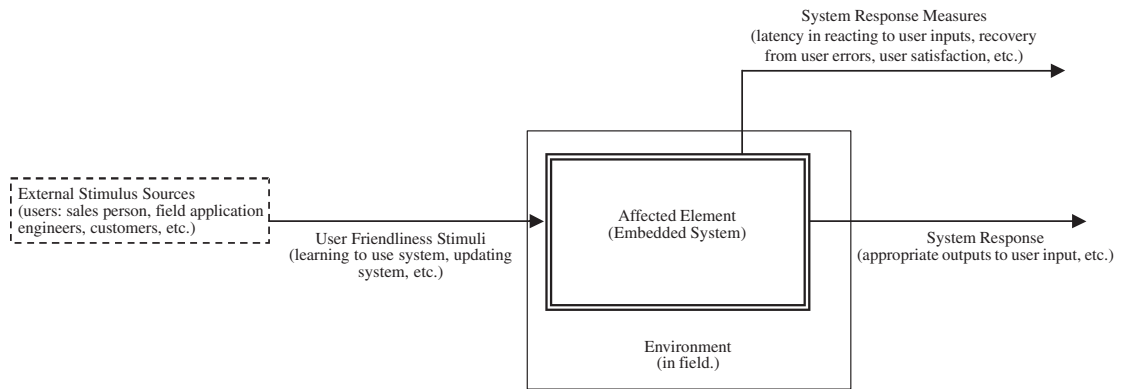
System Response Measures
(latency in reacting to user inputs, recovery
from user errors, user satisfaction, etc.)

External Stimulus Sources
(users: sales person, field application
engineers, customers, etc.)

Affected Element
(Embedded System)

System Response
(appropriate outputs to user input, etc.)

User Friendliness Stimuli
(learning to use system, updating
system, etc.)

Environment
(in field.)

Figure 11-5: General ABC user friendliness *scenario* [11-2]

EXAMPLE *1: Performance*

Figure 11-6a is one possible scenario for a performance-based requirement. In this example, the *stimulus sources* that can impact performance are internal and/or external sources to the embedded system. These stimulus sources can generate one-time and/or periodic asynchronous *events*. According to this scenario, the *environment* in which these events take place occurs when there is normal to a high level of data for the embedded system to process. The stimulus sources generate events that impact the performance of the *entire embedded device*, even if it is only one or a few specific elements within the system that are directly manipulated by the events. This is because typically any performance bottlenecks within the system are perceived by the user as being a performance issue with the entire system.

In this scenario, a desirable system response is for the device to process and respond to events in a timely manner, a possible indicator that the system meets the desired performance requirements. To prove that the performance of the embedded system meets specific performance-based requirements, the system response can be measured and verified via throughput, latency or data loss system response measures.
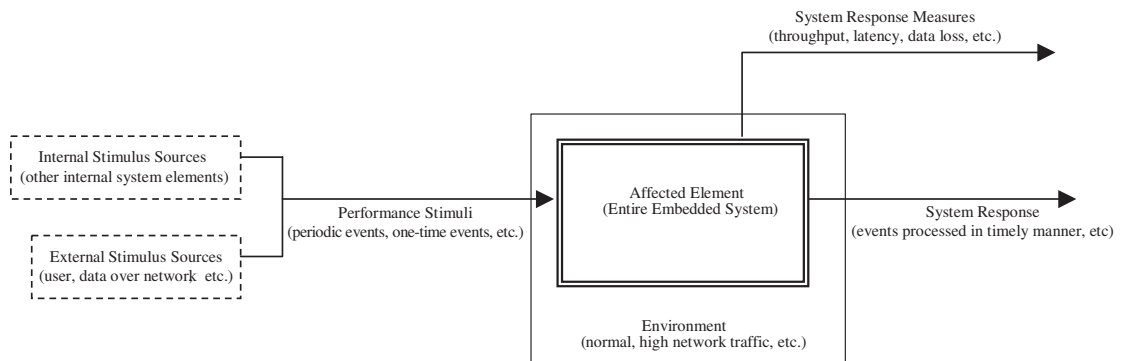
System Response Measures
(throughput, latency, data loss, etc.)

Internal Stimulus Sources
(other internal system elements)

Affected Element
(Entire Embedded System)

System Response
(events processed in timely manner, etc)

Performance Stimuli
(periodic events, one-time events, etc.)

External Stimulus Sources
(user, data over network etc.)

Environment
(normal, high network traffic, etc.)

Figure 11-6a: General ABC performance *scenario* [11-2]

Given the performance scenario shown in Figure 11-6a, a method by which to bring about the desired system response is to control the *time period* in which the stimuli is processed and responses generated. In fact, by defining the specific variables that impact the time period, you can then define the tactics needed to control these variables. The tactics can then be used to define the specific elements within an architecture that would implement the functionality of the tactic in order to allow for the desired performance of the device.

For example, *response time*, a system response measure in this scenario, is impacted by the availability and utilization of resources within a device. If there is a lot of contention between multiple events that want access to the same resource, such as events having to block and wait for other events to finish using a resource, the time waiting for the resource impacts the response time. Thus, a *resource management tactic* shown in Figure 11-6b that arbitrates and manages the requests of events allowing for fair and maximum utilization of resources could be used to decrease response time, and increase a system's performance.
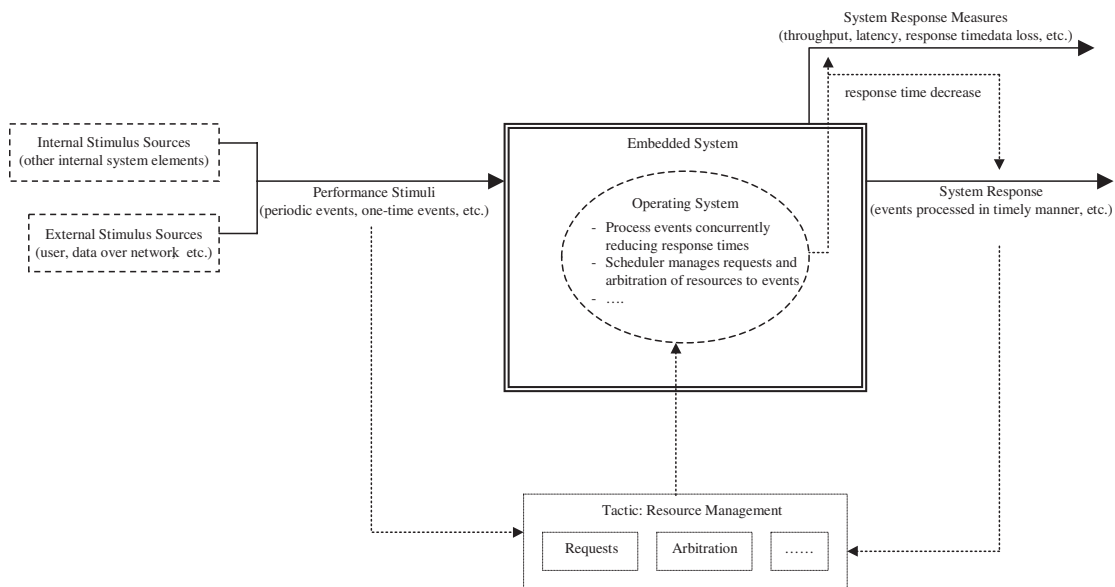

Figure 11-6b: Performance tactics and architectural elements [11-2]

A scheduler, such as that found within an operating system, is an example of a specific software element that can provide resource management functionality. Thus, it is the operating system with the desired scheduling algorithm that is derived for the architecture in this scenario example. In short, this example demonstrates that given the stimuli (events) and a desired system response (good performance), a tactic can be derived (resource management) to achieve the desired system response (good performance) measurable via a system response measure (response time). The functionality behind this tactic, resource management, can then be implemented via an operating system through its scheduling and process management schemes.

> *Author note: It is at this point where Stage 1, "Having a Solid Technical Foundation," is critical. In order to determine what software or hardware elements could support a tactic, one needs to be familiar with the hardware and software elements available to go into an embedded system and the functionality of these elements. Without this knowledge, the results of this stage could be disastrous to a project.*

EXAMPLE 2: Security

Figure 11-7a is a possible scenario for a security-based requirement. In this example, the *stimulus sources* that can impact security are external, such as a hacker or a virus. These external sources can generate *events* that would access system resources, such as the contents of memory. According to this scenario, the *environment* in which these events can take place occurs when the embedded device is in the field connected to a network, doing uploads/downloads of data. In this example, these stimulus sources generate events that impact the security of anything in *main memory* or any *system resource* accessible to the stimulus sources.

In this scenario, desirable system responses for the embedded device include defending against, recovering from, and resisting a system attack. The level and effectiveness of system security is measured in this example by such system response measures as determining how often (if any) security breaches occur, how long it takes the device to recover from a security breach, and its ability to detect and defend against future security attacks. Given the security scenario shown in Figure 11-7a, one method by which to manipulate an embedded system's response so that it can resist a system attack is to control the *access* external sources have to internal system resources.
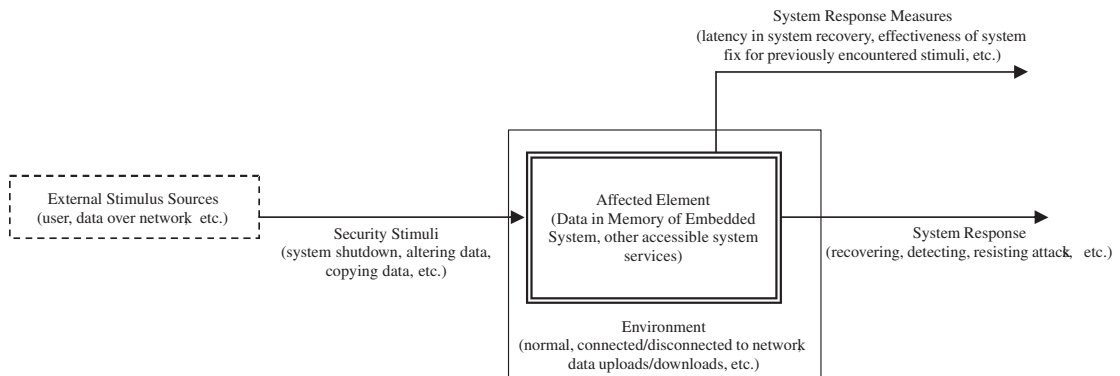

Figure 11-7a: General ABC security *scenario* [11-2]

To manipulate access to system resources, one could control the variables that impact system access through the authentication of external sources accessing the system, and through limiting access to a system's resources to unharmful external sources. Thus, the *authorization* and *authentication* tactics shown in Figure 11-7b could be used to allow a device to track external sources accessing the device and then deny access to harmful external sources, thereby increasing system security.

Given a device resource impacted by a security breach, such as main memory for example, memory and process management schemes such as those found within an operating sys-
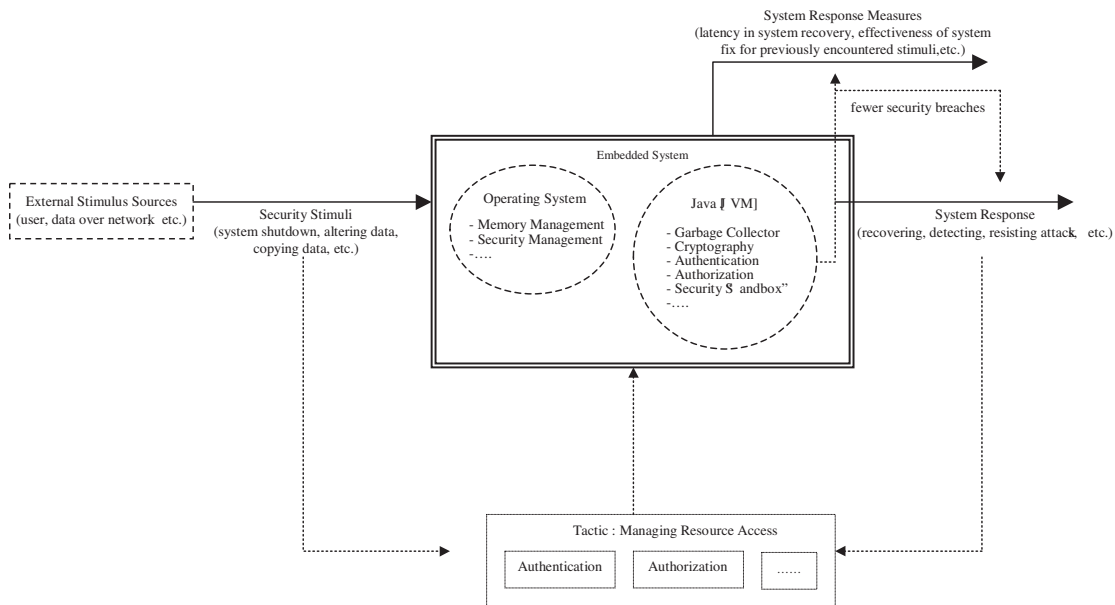

Figure 11-7b: Security tactics and architectural elements [11-2]

tem, security APIs and memory allocation/garbage collection schemes included when using certain higher-level programming languages such as Java, and network security protocols are examples of software and hardware elements that can support managing access to memory resources. In short, this example shows that given the stimuli (attempt to access/delete/create unauthorized data) and a desired system response (detect, resist, recover from attacks), a tactic can be derived (managing access to resources) to achieve the desired system response (detect, resist, recover from attacks) measured via a system response measure (occurrences of security breaches).

EXAMPLE 3: Testability

Figure 11-8a shows a possible scenario for a testability-based requirement. In this example, the *stimulus sources* that can impact testability are internal and external. These sources can generate *events* when hardware and software elements within the embedded system have been completed or updated, and are ready to be tested. According to the scenario in this example,

the *environment* in which these events take place occurs when the device is in development, during manufacturing, or when it has been deployed to the field. The *affected elements* within the embedded system can be any individual hardware or software element, or the entire embedded device as a whole.

In this scenario, the desired system response is the easily controlled and observable responses to tests. The testability of the system is measured by such system response measures as the number of tests performed, the accuracy of the tests, how long tests take to run, verifying data in registers or in main memory, and whether the actual results of the tests match the specifications. Given the testability scenario in Figure 11-8a, a method in which to manipulate an embedded system's response so that the responses to tests are controllable and observable is to provide accessibility for stimulus sources to the internal workings of the embedded system.
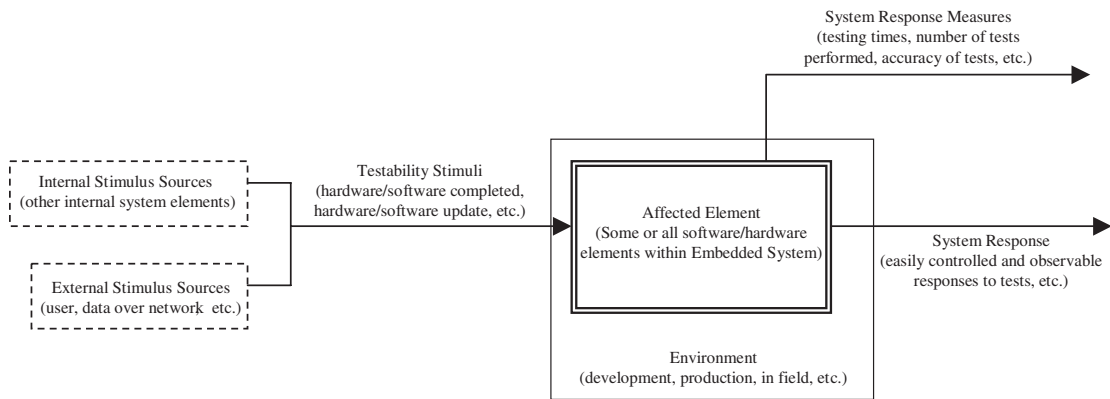
Figure 11-8a: General ABC testability scenario [11-2]

To provide accessibility into the internal workings of the system, one could control the variables that impact the desired system response, such as the ability to do runtime register and memory dumps to verify data. This means that the internal workings of the system have to be visible to and manipulatable by stimulus sources to allow for requesting internal control and status information (i.e., states of variables, manipulating variables, memory usage, etc.), and receiving output based on the requests.

Thus, an internal monitoring tactic, as shown in Figure 11-8b, could be used to provide stimulus sources with the ability to monitor the internal workings of the system and allow this internal monitoring mechanism to accept inputs and provide outputs. This tactic increases the system's testability, since how testable a system is typically depends on how visible and accessible the internal workings of the system are.
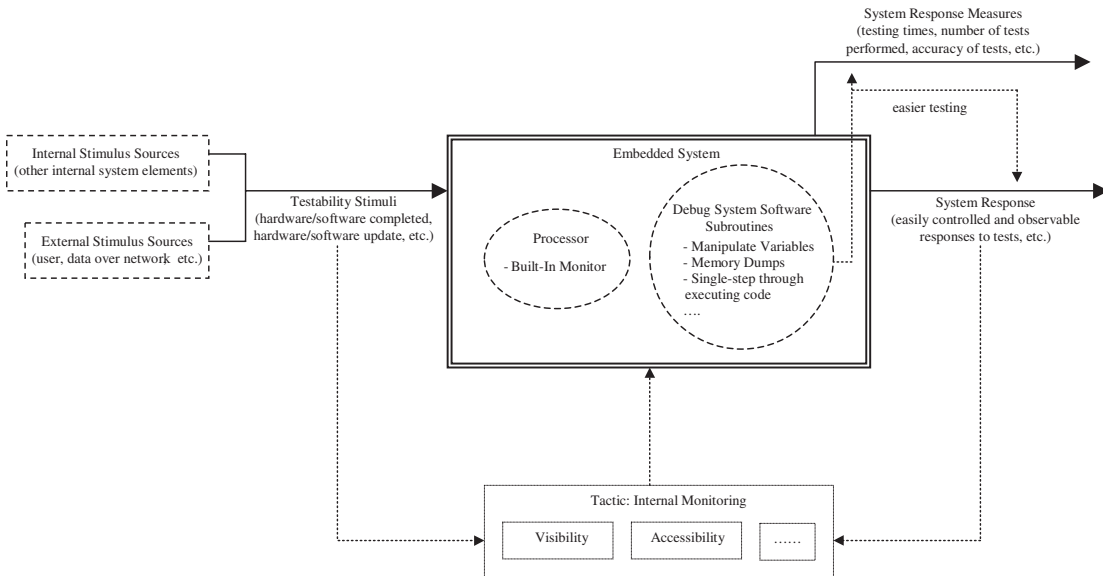

Figure 11-8b: General ABC testability *scenario* [11-2]

Built-in monitors such as those found on various processors, or debugging software subroutines integrated into system software that can be called by a debugger on the development system to perform various tests, are examples of elements that can provide internal monitoring of a system. These hardware and software elements are examples of what could be derived from this scenario. In short, this example demonstrates that given the stimuli (element completed and ready to be tested) and a desired system response (easily test the element and observe results), a tactic can then be derived (internal monitoring of system) to achieve the desired system response (easily test the element and observe results) measurable via a system response measure (testing results, test times, testing accuracy, etc.).

Note: While these examples explicitly demonstrate how elements within an architecture can be derived from general requirements (via their scenarios and tactics), they also implicitly demonstrate that the tactics for one requirement may be counterproductive to another requirement. For example, the functionality that allows for security can impact performance or the functionality that allows for testing accessibility can impact the system's security. Also, note that:

- *a requirement can have multiple tactics*
- *a tactic isn't limited to one requirement*
- *the same tactic can be used across a wide variety of requirements*

Keep these points in mind when defining and understanding the requirements of any system.

Stage 3: Define the Architectural Patterns and Reference Models

An architectural pattern (also referred to as *architectural idioms* or *architectural styles*) for a particular device is essentially a high-level **profile** of the embedded system. This profile is a description of the various types of software and hardware elements the device could consist of, the functions of these elements within the system, a topological layout of these elements (a.k.a. *a reference model*), and the interrelationships and external interfaces of the various elements. Patterns are based upon the hardware and elements derived from the functional and nonfunctional requirements via prototypes, scenarios, and tactics.

Figure 11-9 is an example of architectural pattern information. It starts from the top down in defining the elements for a digital TV set-top box (DTV-STB). This means that it starts with the types of applications that will run on the device, then outlines what system software and hardware these applications implicitly or explicitly require, any constraints in the system, and so on.
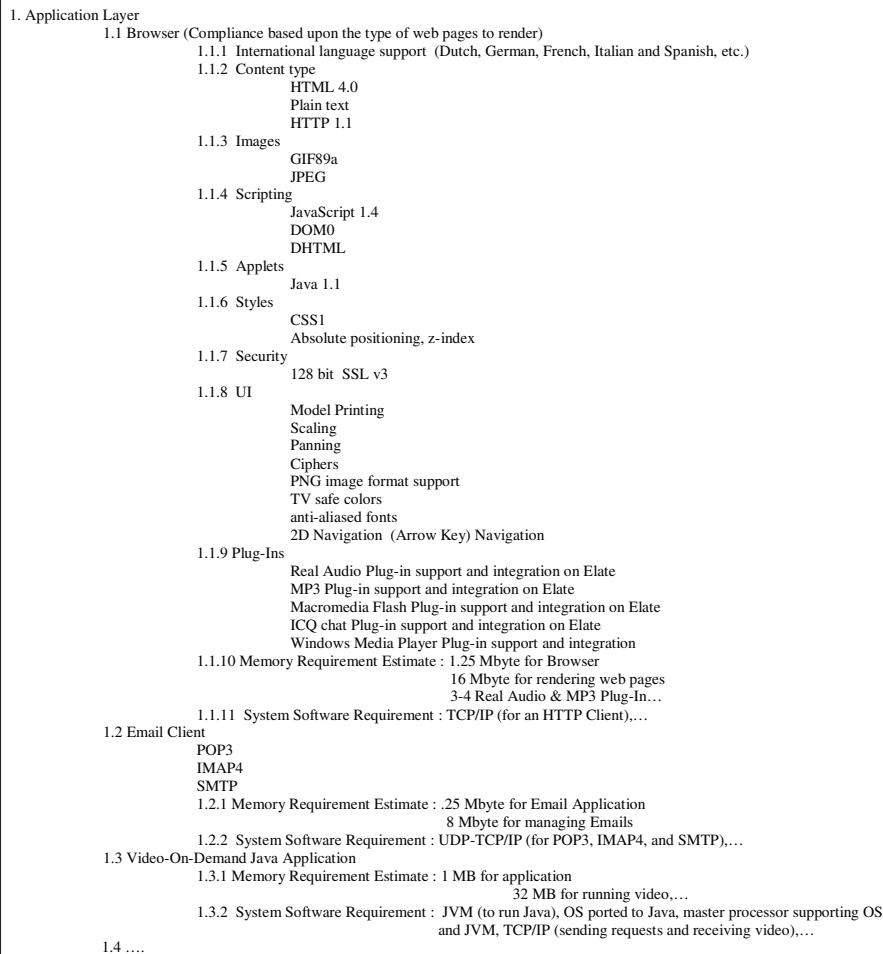
```
1. Application Layer
        1.1 Browser (Compliance based upon the type of web pages to render)
                1.1.1  International language support  (Dutch, German, French, Italian and Spanish, etc.)
                1.1.2  Content type
                        HTML 4.0
                        Plain text
                        HTTP 1.1
                1.1.3  Images
                        GIF89a
                        JPEG
                1.1.4  Scripting
                        JavaScript 1.4
                        DOM0
                        DHTML
                1.1.5  Applets
                        Java 1.1
                1.1.6  Styles
                        CSS1
                        Absolute positioning, z-index
                1.1.7  Security
                        128 bit  SSL v3
                1.1.8  UI
                        Model Printing
                        Scaling
                        Panning
                        Ciphers
                        PNG image format support
                        TV safe colors
                        anti-aliased fonts
                        2D Navigation  (Arrow Key) Navigation
                1.1.9  Plug-Ins
                        Real Audio Plug-in support and integration on Elate
                        MP3 Plug-in support and integration on Elate
                        Macromedia Flash Plug-in support and integration on Elate
                        ICQ chat Plug-in support and integration on Elate
                        Windows Media Player Plug-in support and integration
                1.1.10 Memory Requirement Estimate : 1.25 Mbyte for Browser
                                                16 Mbyte for rendering web pages
                                                3-4 Real Audio & MP3 Plug-In…
                1.1.11  System Software Requirement : TCP/IP (for an HTTP Client),…
        1.2 Email Client
                        POP3
                        IMAP4
                        SMTP
                1.2.1 Memory Requirement Estimate : .25 Mbyte for Email Application
                                                8 Mbyte for managing Emails
                1.2.2  System Software Requirement : UDP-TCP/IP (for POP3, IMAP4, and SMTP),…
        1.3 Video-On-Demand Java Application
                1.3.1 Memory Requirement Estimate : 1 MB for application
                                                32 MB for running video,…
                1.3.2  System Software Requirement :  JVM (to run Java), OS ported to Java, master processor supporting OS
                                                and JVM, TCP/IP (sending requests and receiving video),…
        1.4 ….
```

Figure 11-9: DTV-STB profile – application layer

The system profile can then be leveraged to come up with possible hardware and software reference models for the device that incorporates the relative elements. Figure 11-10 shows a possible reference model for the DTV-STB used as an example in Figure 11-9.
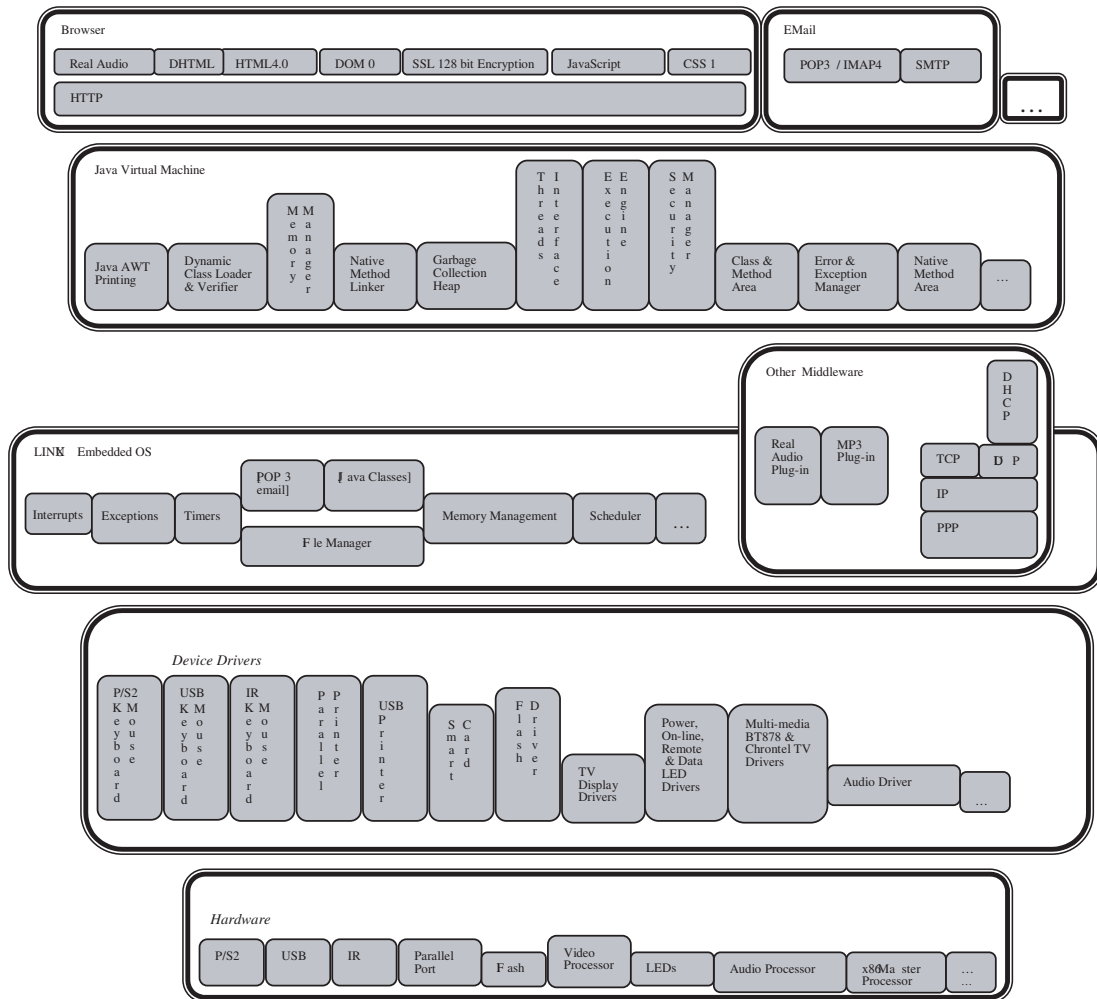

Figure 11-10: DTV-STB reference model

Author Recommendation: If the software requirements are known, map and decompose as much of the major software elements (i.e., OS, JVM, applications, networking, etc.) as possible before finalizing the hardware elements. This is because the hardware will limit (or enhance) what can be done via software, and typically the less that is done in hardware, the cheaper the hardware. Then, while matching the software configuration with the possible master processor and board, come up with different models. This could include removing some functionality or implementing some software components within hardware, for example.

Selecting Commercially Available Hardware and Software

Regardless of what elements make their way into an architecture, all typically have to meet a basic set of criteria, both functional and nonfunctional, as shown in stage 2, such as:

- *Cost*. Does the purchasing (versus creating internally), integrating, and deploying of the element meet cost restrictions?

- *Time-to-market*. Will the element meet the requirements in the required time frame (i.e. at various milestones during development, at production time, etc.)?

- *Performance.* Will the element be fast enough for user satisfaction and/or for other dependent elements?

- *Development and Debugging Tools*. What tools are available to make designing with the element faster and easier?

- …..

While all elements can be designed internally that support architectural requirements, many elements can be purchased commercially, off the shelf. Regardless of what the specific list of criteria is, the most common way of selecting between commercially available components is to build a **matrix** of required features for each element, and then fill in the matrix with the products that fulfill the particular requirements (see Figure 11-11). Matrices for different elements that rely on each other can then be cross-referenced.

	Requirement 1	Requirement 2	Requirement 3	Requirement …	Requirement "N"
Product 1	YES Features …	NO	NOT YET Next Year	…	…
Product 2	YES Features …	YES Features …	YES Features …	…	…
Product 3	NO	YES Features …	NO	…	…
Product 4	YES Features …	NOT YET In 3 Months	NOT YET In 6 Months	…	…
Product …	….	….	….	….	…
Product "N"	…	…	…	…	…

Figure 11-11: Sample matrix

While all off-the-shelf available elements (i.e., networking stack, device drivers, etc.) in an embedded architecture design are important in making the design a success, some of the most critical design elements that typically impact the other design decisions the most are the *programming languages* selected, the use of an *operating system*, and what *master processor* the embedded board is based upon. In this section, these elements are used as examples in providing suggestions for how to select between commercially available options and creating the relative matrix in each of these areas.

EXAMPLE 1: Programming Language Selection

All languages require a compiler to translate source into machine code for that processor, whether it's a 32-bit, 16-bit, 8-bit, etc., processor. With 4-bit and 8-bit based designs that contain kilobytes of memory (total ROM and RAM), assembly language has traditionally been the language of choice. In systems with more powerful architectures, assembly is typically used for the low-level hardware manipulation or code that must run very fast. Writing code in assembly is always an option, and in fact, most embedded systems implement some assembly code. While assembly is fast, it is typically more difficult to program than higher-level languages; in addition, there is a different assembly language set to learn for each ISA.

C is typically the basis of more complex languages used in embedded systems, such as C++, Java, Perl, etc. In fact, C is often the language used in more complex embedded devices that have an operating system, or that use more complex languages, such as a JVM or scripting languages. This is because operating systems, JVMs, and scripting language interpreters, excluding those implemented in non-C applications, are usually written in C.

Using higher-level object-oriented languages, such as C++ or Java, in an embedded device is useful for larger embedded applications where modular code can simplify the design, development, testing, and maintenance of larger applications over procedural coding (of C, for instance). Also, C++ and Java introduce additional out-of-the-box mechanisms, such as security, exception handling, namespace, type-safety, etc., not traditionally found in the C language. In some cases, market standards may actually require the device to support a particular language (for example, the Multimedia Home Platform specification, or MHP, implemented in certain DTVs requires using Java).

For implementing embedded applications that are hardware and system software independent, Java, .Net languages (C#, Visual Basic, etc.), and scripting languages are the most common higher-level languages of choice. In order to be able to run applications written in any of these languages, a JVM (for Java), the .NET Compact Framework (for C#, Visual Basic, etc.), and an interpreter (for scripting languages, like JavaScript, HTML, Perl, etc.) all need to be included within the architecture. Given the desire for hardware and system software independent applications, the correct APIs must be supported on the device in order for the applications to run. For example, Java applications targeted for larger, more complex devices are typically based upon the pJava or J2ME CDC APIs, whereas Java applications targeted for smaller, simpler systems would typically expect a J2ME CLDC supported implementation. Also, if the element is not implemented in hardware (i.e., JVM in a Java processor), then it must either be implemented in the system software stack, either as part of the OS, ported to the OS and to the master processor (as is the case with .NET to WinCE, or a JVM to vxWorks, Linux, etc. on x86, MIPS, strongARM, and so on), or it needs to be implemented as part of the application (i.e., HTML, JavaScript in a browser, a JVM in a .exe file on WinCE or .prc file on PalmOS, etc.). Also, note, as with any other significant software element introduced in the architecture, the minimal processing power and memory requirements have to be met in hardware in systems that contain these higher-level language elements in order for the code written in these languages to perform in a reasonable manner, if at all.

In short, what this example attempts to portray is that an embedded system can be based upon a number of different languages (assembly, C, Java, and HTML in an MHP-based DTV with a browser on an x86 board, for example), or based upon only one language (assembly in a 21" analog TV based on an 8-bit TV microcontroller, for example). As shown in Figure 11-12, the key is creating a matrix that outlines what available languages in the industry meet the functional and nonfunctional requirements of the device.

	Real-Time	Fast Performance	MHP-Spec	ATVEF-Spec	Browser Application	...
Assembly	YES	YES	NOT Required	NOT Required	NOT Required	...
C	YES	YES Slower than assembly	NOT Required	NOT Required	NOT Required	...
C++	YES	YES Slower than C	NOT Required	NOT Required	NOT Required	...
.NetCE (C#)	NO WinCE NOT RTOS	Depends on processor, slower than C on less powerful processors	NOT Required	NOT Required	NOT Required	...
JVM (Java)	Depends on JVM's Garbage Collector and is OS ported to RTOS	Depends on JVM's byte code processing scheme (WAT almost as fast as C where interpretation requires more power-ful processor i.e. 200+ MHz), slower than C on slower processors	YES	NOT Required	NOT Required	...
HTML (Scripting)	Depends on what language written in, and the OS (an RTOS in C/assem-bly OK, .NetCE platform no, Java depends on JVM	Slower because of the interpretation that needs to be done but depends on what language interpreter written in (see above cells of this column)	NOT Re-quired	YES	YES	...

Figure 11-12: Programming language matrix

Example 2: Selecting an Operating System

The main questions surrounding the use of an operating system (OS) within an embedded design are:

1. What type of systems typically use or require an OS?
2. Is an OS needed to fulfill system requirements?
3. What is needed to support an OS in a design?
4. How to select the OS that best fits the requirements?

Embedded devices based upon 32-bit processors (and up) typically have an OS, because these systems are usually more complex and have many more megabytes of code to manage than their 4-bit, 8-bit, and 16-bit based counterparts. Sometimes other elements within the architecture require an OS within the system, such as when a JVM or the .NET Compact Framework is implemented in the system software stack. In short, while any master CPU can support some type of kernel, the more complex the device, the more likely it is that a kernel will be used.

Whether an OS is needed or not depends on the requirements and complexity of the system. For example, if the ability to multitask, to schedule tasks in a particular scheme, to manage resources fairly, to manage virtual memory, and to manage a lot of application code is important, then using an OS will not only simplify the entire project, but may be critical to completing it. In order to be able to introduce an OS into any design, the overhead needs to be taken into account (as with any software element introduced), including: processing power, memory, and cost. This also means that the OS needs to support the hardware (i.e., master processor).

Selecting an off-the-shelf OS, again, goes back to creating a matrix with requirements and the OS features. The features in this matrix could include:

- *Cost*. When purchasing an OS, many costs need to be taken into consideration. For example, if development tools come with an OS package, there is typically a fee for the tools (which could be a per team or per developer fee), as well as a license fee for the OS. Some OS companies charge a one-time license OS fee, whereas others charge an upfront fee along with royalties (a fee per device being manufactured).

- *Development and Debugging Tools*. This includes tools that are compatible with or are included in the OS package, such as technical support (a website, a support engineer, or Field Applications Engineer to call for help), an IDE (Integrated Development Environment), ICEs (In-circuit Emulators), compilers, linkers, simulators, debuggers, and so on.

- *Size*. This includes the footprint of the OS on ROM, as well as how much main memory (RAM) is needed when OS is loaded and running. Some OSes can be configured to fit into much less memory by allowing the developer to scale out unnecessary functionality.

- *Nonkernel Related Libraries.* Many OS vendors entice customers by including additional software with the OS package or optional packages available with the OS (i.e., device drivers, file systems, networking stack, etc.) that are integrated with the OS and (usually) ready to run out-of-the-box.

- *Standards Support.* Various industries may have specific standards for software (such as an OS) that need to meet some safety or security regulations. In some cases, an OS may need to be formally certified. There are also general OS standards (i.e., POSIX) which many embedded OS vendors support.

- *Performance.* (See Chapter 9, Section 9.6 on OS performance).

Of course, there are many other desirable features that can go into the matrix such as process management scheme, processor support, portability, vendor reputation, and so on, but essentially it comes down to taking the time to create the matrix of desired features and OSes as shown in Figure 11-13.

	Tools	Portability	Non-kernel	Processor	Scheduling Scheme	...
vxWorks	Tornado IDE, SingleStep debugger, ...	BSP	Device Drivers w/ BSP, graphics, networking, ...	x86, MIPS, 68K, ARM, strongARM, PPC ...	Hard Real-Time, Priority-based ...	...
Linux	Depends on vendor for development IDE, gcc, ...	Depends on vendor, some with no BSP	Device Drivers graphics, networking, ...	Depends on vendor (x86, PPC, MIPS, ...)	Depends on vendor, some are hard real-time, others soft-real time ...	...
Jbed	Jbed IDE , Sun Java compiler, ...	BSP	Device Drivers – the rest depends on JVM specification (graphics, networking, ...)	PPC, ARM, ...	EDF Hard Real Time Scheduling ...	...

Figure 11-13: Operating system matrix

EXAMPLE 3: Selecting a Processor

The different ISA designs—application-specific, general purpose, instruction-level parallelism, etc.—are targeted for various types of devices. However, different processors that fall under different ISAs could be used for the same type of device, given the right components on the board and in the software stack. As the names imply, general purpose ISAs are used in a wide variety of devices, and application-specific ISAs are targeted for specific types of devices or devices with specific requirements, where their purposes are typically implied by their names, such as: TV microcontroller for TVs, Java processor for providing Java support, DSPs (digital signal processors) that repeatedly perform fixed computations on data, and so on. As mentioned in chapter 4, instruction-level parallelism processors are typically general

purpose processors with better performance (because of their parallel instruction execution mechanisms). Furthermore, in general, 4-bit/8-bit architectures have been used for lower-end embedded systems, and 16-bit/32-bit architectures for higher-end, larger, and more expensive embedded systems.

As with any other design decision, selecting a processor means both selecting it based on the requirements as well as identifying its impact on the remainder of the system, including software elements, as well as other hardware elements. This is especially important with hardware, since it is the hardware that impacts what enhancements or constraints are implementable in software. What this means is creating the matrix of requirements and processors, and cross-referencing this matrix with the other matrices reflecting the requirements of other system components. Some of the most common features that are considered when selecting a processor include: the cost of the processor, power consumption, development and debugging tools, operating system support, processor/reference board availability and lifecycle, reputation of vendor, technical support and documentation (data sheets, manuals, etc.). Figure 11-14 shows a sample matrix for master processor selection for implementing a Java-based system.

	Tools	Java-specific Features	OS Support	…
aJile aj100 Java Processor (Application Specific ISA)	JEMBuilder, Charade debugger,	J2ME/CLDC JVM	NOT Needed	…
Motorola PPC823 (General Purpose ISA)	Tornado tools, Jbed Tools, Sun tools, Abatron BDM…	Implemented in software (Jbed, PERC, CEE-J, …)	Coming Soon — Linux, vxWorks, Jbed, Nucleus Plus, OSE, …	…
Hitachi Camelot Superscaler SoC(Instruction Level Parallel ISA)	Tornado Tools, QNX Tools, JTAG, …	Coming Soon — Implemented in software (IBM, OTI, Sun VMs..)	Coming Soon — QNX, vxWorks, WinCE, Linux, ...	…

Figure 11-14: Processor matrix

Stage 4: Define the Architectural Structures

After stages 1 through 3 have been completed, the architecture can be created. This is done by decomposing the entire embedded system into hardware and/or software elements, and then further decomposing the elements that need breaking down. These decompositions are represented as some combination of various types of structures (see Chapter 1, Table 1-1 for examples of structure types). The patterns defined under stage 3 that most satisfy the system requirements (the most complete, the most accurate, the most buildable, highest conceptual integrity, etc.) should be used as the foundations for the architectural structures.

It is ultimately left up to the architects of the system to decide which structures to select and how many to implement. While different industry methodologies have different recommendations, a technique favored by some of the most popular methodologies, including the Rational Unified Process, Attribute Driven Design, and others, is the "4+1" model shown in Figure 11-15.
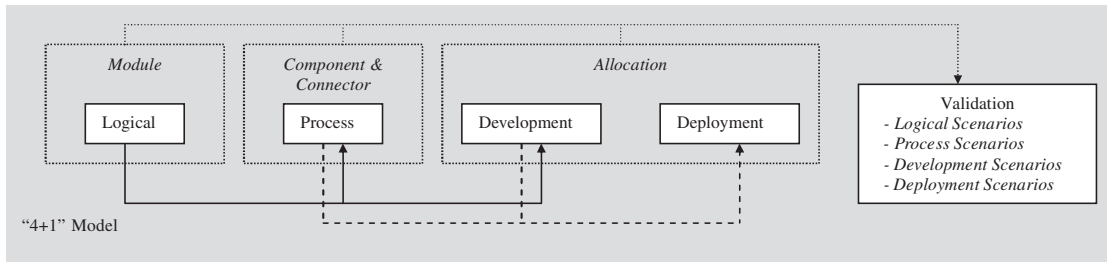
Figure 11-15: "4+1" model [11-2]

The "4+1" model states that a system architect should create **five** concurrent structures per architecture at the very least, and each structure should represent a different viewpoint of the system. What is literally meant by "4+1" is that *four* of the structures are responsible for capturing the various requirements of the system. The fifth structure is used to validate the other four, insuring that there are no contentions between the structures and that all structures describe the exact same embedded device, from their various viewpoints.

What is specifically shown in Figure 11-15 is that the four base structures of the "4+1" model should fall under the *module*, *component and connector*, and *allocation* structural types. Within each of these structural family types, this model after being adapted for use in embedded systems by this author specifically recommends:

- **Structure 1**. One *logical* structure, which is a modular structure of the key functional hardware and software elements in the system, such as objects for an object-oriented based structure, or a processor, OS, etc. A logical modular structure is recommended because it demonstrates how the key functional hardware and software requirements of the system are being met by displaying the elements that meet the requirements, as well as their interrelationships. This information can then be leveraged to build the actual system through these functional elements, outlining which functional elements need to be integrated with which other functional elements, as well as outlining what functional requirements are needed by various elements within the system to execute successfully.

- **Structure 2**. One *process* structure, which is the component and connector structure reflecting the concurrency and synchronization of processes in a system that would contain an OS. A process structure is recommended because it demonstrates how nonfunctional requirements, such as performance, system integrity, resource availability, etc., are being met by an OS. This structure provides a snapshot of the system from an OS process standpoint, outlining the processes in the system, the scheduling mechanism, the resource management mechanisms, and so on.

Two Allocation Structures

- **Structure 3**. A *development* structure describing how the hardware and software map into the development environment. A development structure is recommended because it provides support for nonfunctional requirements related to the buildability of the

531

hardware and software. This includes information on any constraints of the development environment like the integrated development environment (IDE), debuggers, compilers, and so forth; complexity of the programming language(s) to be used; and other requirements. It demonstrates this buildability through the mapping of the hardware and software to the development environment.

- **Structure 4.** A *deployment/physical* structure showing how software maps into the hardware. A *deployment/physical* structure is recommended because, like the process structure, it demonstrates how nonfunctional requirements such as hardware resource availability, processor throughput, performance, and reliability of the hardware are met by demonstrating how all the software within the device maps to the hardware. This essentially defines the hardware requirements based on the software requirements. This includes the processors executing code/data (processing power), memory storing the code/data, buses transmitting code/data, and so on.

As seen from the definitions of these structures, this model assumes that, first, the system has software (development, deployment, and process structures), and, second, the embedded device will include some type of operating system (process structure). Essentially, modular structures apply universally regardless of what software components are in the system, or even if there is no software, as is the case with some older embedded system designs. With embedded designs that don't require an operating system, then other component and connector structures, such as system resource structures like memory or I/O, can be substituted that represent some of the functionality that is typically found in an OS, such as memory management or I/O management. As with embedded systems with no OS, for embedded devices with no software, hardware-oriented structures can be substituted for the software-oriented structures.

The "+1" structure, or fifth structure, is the mapping of a subset of the most important scenarios and their tactics that exist in the other four structures. This insures that the various elements of the four structures are not in conflict with each other, thus validating the entire architectural design. Again, keep in mind that these specific structures are *recommended* for particular types of embedded systems, not *required* by the "4+1" model. Furthermore, implementing five structures, as opposed to implementing fewer or more structures, is also a recommendation. These structures can be altered to include additional information reflecting the requirements. Additional structures can be added if they are needed to accurately reflect a view of the system not captured by any of the other structures created. The important thing that the model is trying to relay regarding the number of structures is that it is very difficult to reflect all the information about the system in only one type of structure.

Finally, the arrows to and from the four primary structures shown in Figure 11-15 of the "4+1" module represent the fact that, while the various structures are different perspectives of the same embedded system, they are not *independent* of each other. This means that at least one element of a structure is represented as a similar element or some different manifestation in another structure, and it is the sum of all of these structures that makes up the architecture of the embedded system.

Author note: While the "4+1" model was originally created to address the creation of a software architecture, it is adaptable and applicable to embedded systems architectural hardware and software design as a whole. In short, the purpose of this model is to act as a tool to determine how to select the right structures and how many to select. Essentially, the same fundamentals of the "4+1" model concerning the number, types, and purposes of the various structures can be applied to embedded systems architecture and design regardless of how structural elements are chosen to be represented by the architect, or how strictly an architect chooses to abide by the various methodology notations (i.e., symbols representing various architectural elements within a structure) and styles (i.e., object oriented, hierarchical layers, etc.).

Many architectural structures and patterns have been defined in various architecture books (do your research), but some useful books include Software Architecture in Practice (Bass, Clements, Kazman, 2003), A System of Patterns: Pattern-Oriented Software Architecture (Buschmann, Meunier, Rohnert, Sommerlad, Stal, 1996), and Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems (Douglass, 2003). These can all be applied to embedded systems design.

Stage 5: Document the Architecture

Documenting the architecture means documenting all of the structures—the elements, their functions in the system, and their interrelationships—in a coherent manner. How an architecture is actually documented ultimately depends on the standard practices decided by the team and/or management. A variety of industry methodologies provide recommendations and guidelines for writing architectural specifications. These popular guidelines can be summarized in the following three steps:

- *Step 1: A document outlining the entire architecture.*

  This step involves creating a table of contents that outlines the information and documentation available for the architecture, such as: an overview of the embedded system, the actual requirements supported by the architecture, the definitions of the various structures, the inter-relationships between the structures, outlining the documentation representing the various structures, how these documents are laid out (i.e., modeling techniques, notation, semantics, etc.), and so on.

- *Step 2: A document for each structure.*

  This document should indicate which requirements are being supported by the structure and how these requirements are being supported by the design, as well as any relative constraints, issues, or open items. This document should also contain graphical and nongraphical (i.e., tabular, text, etc.) representation of each of the various elements within the structure. For instance, a graphical representation of the structural elements and relationships would include an index containing a textual summary of the various elements, their behavior, their interfaces, and their relationships to other structural elements.

It is also recommended that this document or a related subdocument outline any interfaces or protocols used to communicate to devices outside the embedded system from the point-of-view of the structure.

While in embedded systems there is no one template for documenting the various structures and related information, there are popular industry techniques for modeling the various structural related information. Some of the most common include the *Universal Modeling Language* (UML) by the Object Management Group (OMG) that defines the notations and semantics for creating state charts and sequence diagrams that model the behavior of structural elements, and the *attribute driven design* (ADD) that, among other templates, provides a template for writing the interface information. Figures 11-16a, b, and c below are examples of templates that can be used to document the information of an architectural design.

UML Sequence Diagram   *UML Use Case Diagram*
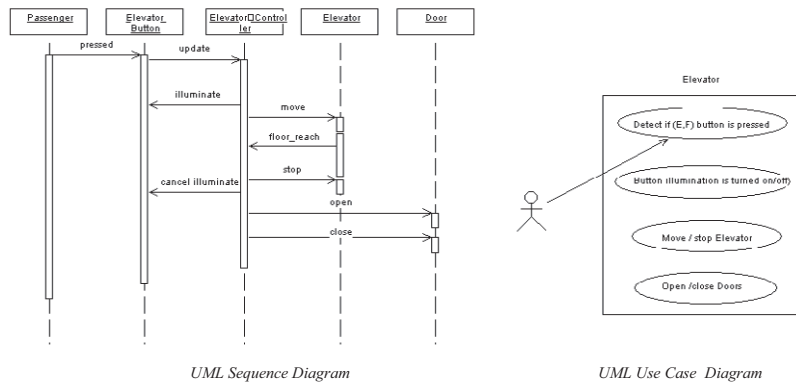
UML is mainly a set of object-oriented graphical techniques

Figure 11-16a: UML diagrams [11-3]

Figure 11-16b: ADD interface template [11-2]

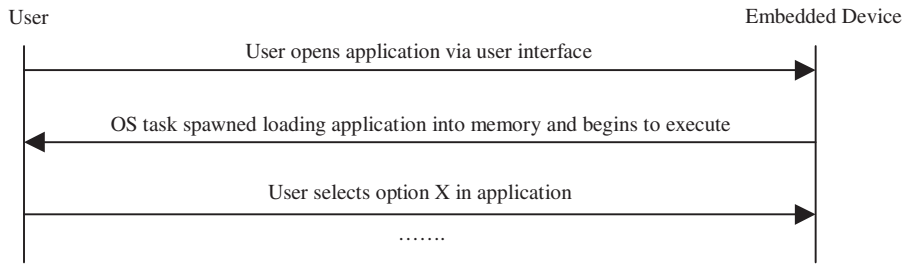User   Embedded Device

Figure 11-16c: (Rough and informal) sequence diagram

- *Step 3: An architecture glossary.*

  This document lists and defines all of the technical terms used in all of the architectural documentation.

Regardless of whether the architecture documentation is made up of text and informal diagrams or is based upon a precise UML template, the documentation should reflect the various readers' points-of-view, not only the writer's. This means that it should be useful and unambiguous regardless of whether the reader is a beginner, nontechnical, or very technical (that is, it should have high-level "use case" models outlining the various users and how the system can be used, sequence diagrams, state charts, etc.). Also, the various architectural documentation should include the different kinds of information the various readers (stakeholders) need in order to do their analysis and provide feedback.

Stage 6: Analyze and Evaluate the Architecture

While there are many purposes for reviewing an architecture, primarily it is to determine if an architecture meets the requirements, and to evaluate the potential risks, and possible failure, of a design long before it has been built. When evaluating an architecture, both ***who*** reviews the architecture must be established, as well as the process of ***how*** evaluations should occur. In terms of the "who," outside of the architects and stakeholders, the evaluation team should include an engineer outside of the ABC influences to provide impartial perspectives on the design.

There are many techniques for analyzing and evaluating architectures that can be adapted and used in an embedded systems design process. The most common of these approaches typically fall under an *architecture-oriented* approach, a *quality attribute*-based approach, or some *combination* of these two approaches. In **architecture-oriented** approaches, scenarios to be evaluated are implemented by the system stakeholders and/or an evaluation team (with stakeholder representatives as a subset of the team).

A **quality attribute** approach is typically considered either *qualitative* or *quantitative*. Under the qualitative-analysis approach, different architectures with the same *quality attributes* (a.k.a. features of a system that nonfunctional requirements are based upon) are compared by an architect and/or by an evaluation team depending on the specific approach. Quantitative-analysis techniques are measurement-based, meaning particular quality attributes of an

architecture and associated information are analyzed, as well as models associated with the quality attributes and related information are built. These models, along with associated characteristics, can then be used to determine the best approach to building the system. There is a wide variety of both quality-attribute and architecture-oriented approaches, some of which are summarized in Table 11-2.

Table 11-2: Architecture analysis approaches [11-2]

Methodology	Description
Architecture Level Prediction of Software Maintenance [ALPSM]	Maintainability evaluated via scenarios.
The Architecture Tradeoff Analysis Method [ATAM]	Quality Attribute (quantitative) approach that defines problem areas and technical implications of an architecture via question and measurement techniques. Can be used for evaluating many quality attributes.
Cost Benefit Analysis Method [CBAM]	Extension of ATAM for defining the economical implications of an architecture.
ISO/IEC 9126-1 thru 4	Architecture analysis standards using internal and external metric models for evaluation (relevant to the functionality, reliability, usability, efficiency, maintainability and portability of devices).
Rate Monotonic Analysis (RMA)	Approach which evaluates the real-time behavior of a design.
Scenario-Based Architecture Analysis Method [SAAM]	Modifiability evaluated through scenarios defined by stakeholders (an architecture-oriented approach).
SAAM Founded on Complex Scenarios [SAAMCS]	SAAM extension — Flexibility evaluated via scenarios defined by stakeholders (an architecture-oriented approach).
Extending SAAM by Integration in the Domain [ESAAMI]	SAAM extension — Modifiability evaluated via scenarios defined by stakeholders (an architecture-oriented approach).
Scenario-Based Architecture Reengineering [SBAR]	Variety of quality attributes evaluated via mathematical modeling, scenarios, simulators, objective reasoning (depends on attribute).
Software Architecture Analysis Method for Evolution and Reusability [SAAMER]	Evaluation of evolution and reusability via scenarios.
A Software Architecture Evaluation Model [SAEM]	A quality model that evaluates via different metrics depending on GQM technique.

As seen in Table 11-2, some of these approaches analyze only certain types of requirements, whereas others are meant to analyze a wider variety of quality attributes and scenarios. In order for the evaluation to be considered successful, it is important that 1) the members of the evaluation team understand the architecture, such as the patterns and structures, 2) these members understand how the architecture meets the requirements, and 3) everyone on the

team agree that the architecture meets the requirements. This can be accomplished via mechanisms introduced in these various analytic and evaluation approaches (see Table 11-2 for examples), general steps including:

- *Step 1*. Members of the evaluation team obtain copies of the architecture documentation from the responsible architect(s), and it is explained to the various team members the evaluation process, as well as the architecture information within the documentation to be evaluated.

- *Step 2*. A list of the architectural approaches and patterns is compiled based upon feedback from the members of the evaluation team after they have analyzed the documentation.

- *Step 3*. The architect(s) and evaluation team members agree upon the exact scenarios derived from the requirements of the system (the team responding with their own inputs of the architect's scenarios: changes, additions, deletions, etc.), and the priorities of the various scenarios are agreed upon in terms of both importance and difficulty of implementation.

- *Step 4*. The (agreed upon) more difficult and important scenarios are where the evaluation team spends the most evaluation time because these scenarios introduce the greatest risks.

- *Step 5*. Results the evaluation team should include (at the very least) the 1) uniformly agreed upon list of requirements/scenarios, 2) benefits (i.e., the return-on-investment (ROI) or the ratio of benefit to cost), 3) risks, 4) strengths, 5) problems, and 6) any of the recommended changes to the evaluated architectural design.

11.2 Summary

This chapter introduced a simple process for creating an embedded systems architecture that included six major stages: having a solid technical background (stage 1), understanding the architectural business cycle of the system (stage 2), defining the architectural patterns and respective models (stage 3), defining the architectural structures (stage 4), documenting the architecture (stage 5), and analyzing and evaluating the architecture (stage 6). In short, this process uses some of the most useful mechanisms of the various popular industry architectural approaches. The reader can use these mechanisms as a starting point for understanding the variety of approaches, as well as for creating an embedded system architecture based upon this simplified, pragmatic approach.

The next and final chapter in this text, Chapter 12, discusses the remaining phases of embedded system design: the implementation of the architecture, the testing of the design, and the maintainability issues of a design after deployment.

Chapter 11 Problems

1. Draw and describe the four phases of the Embedded System Design and Development Lifecycle Model.

2. [a]   Of the four phases, which phase is considered the most difficult and important?
   [b]   Why?

3. What are the six stages in creating an architecture?

4. [a] What are the ABCs of embedded systems?
   [b]   Draw and describe the cycle.

5. List and define the four steps of Stage 2 of creating the architecture.

6. Name four types of influences on the design process of an embedded system.

7. Which method is least recommended for gathering information from ABC influences?
   A.   finite-state machine models.
   B.   scenarios.
   C.   by phone.
   D.   in an e-mail.
   E.   All of the above.

8. Name and describe four examples of general ABC features from five different influences.

9. [a]   What is a prototype?
   [b]   How can a prototype be useful?

10. What is the difference between a scenario and a tactic?

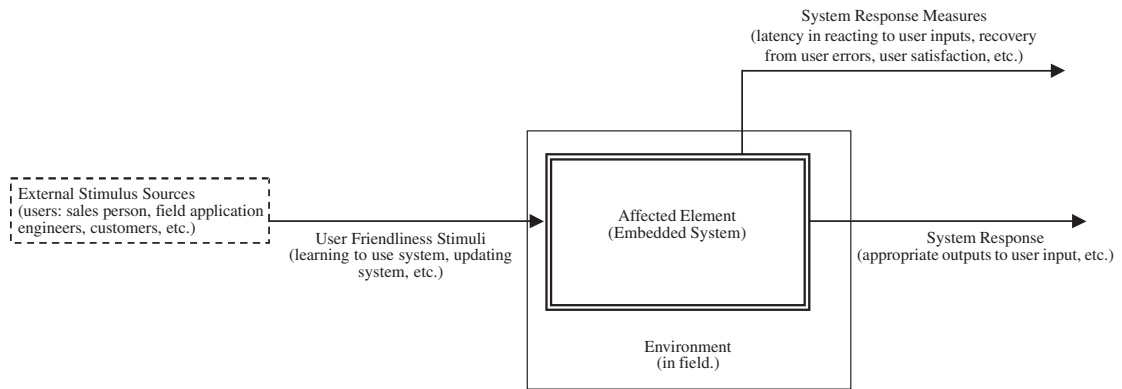11. In Figure 11-17, list and define the major components of a scenario.

Figure 11-17 General ABC user friendliness scenario [11-2]

12. [T/F] A requirement can have multiple tactics.

13. What is the difference between an architectural pattern and a reference model?

14. [a] What is the "4+1" model?
    [b] Why is it useful?
    [c] List and define structures that correspond to the "4+1" model.

15. [a] What is the process for documenting an architecture?
    [b] How can a particular structure be documented?

16. [a] List and define two common approaches for analyzing and evaluating an architecture?
    [b] Give at least five real-world examples of either.

17. What is the difference between a qualitative and quantitative quality attribute approach?

18. What are the five steps introduced in the text as a method by which to review an architecture?