CHAPTER 11

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]

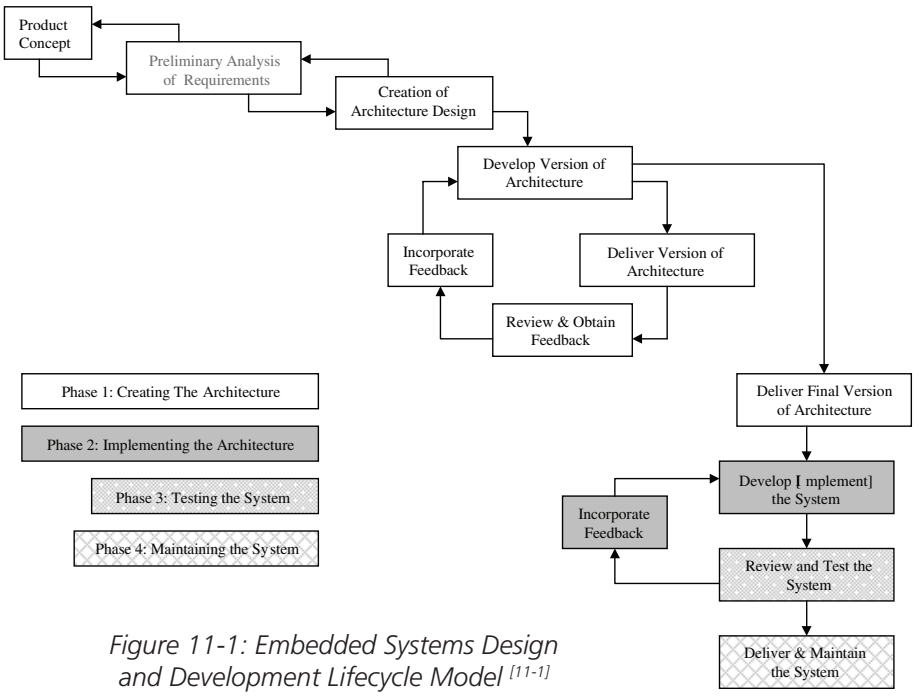


Figure 11-1: Embedded Systems Design and Development Lifecycle Model [11-1]

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).

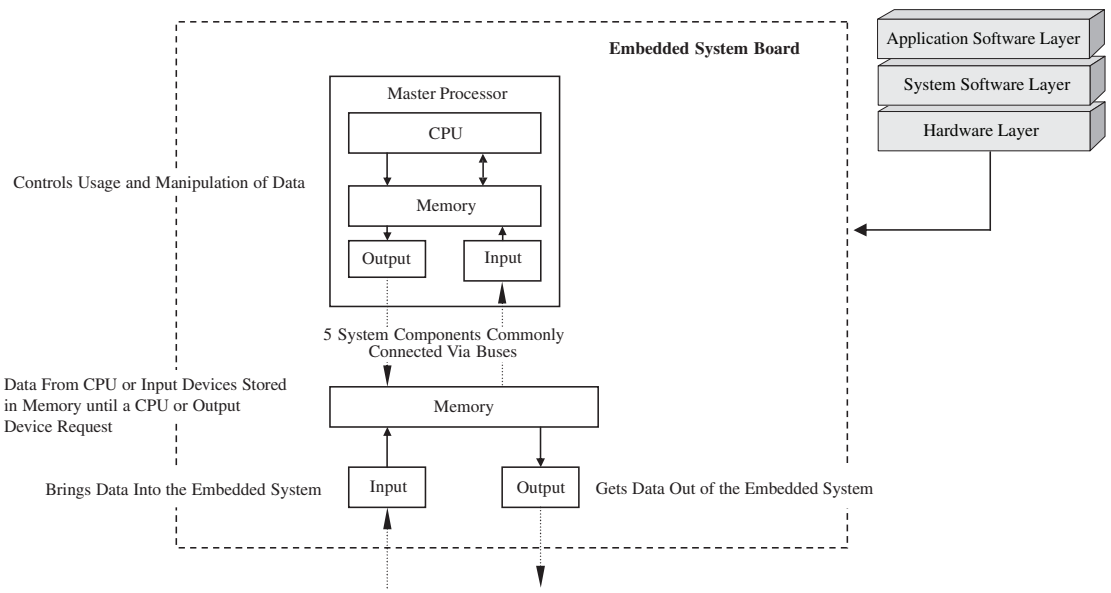


Figure 11-2a: von Neumann and Embedded Systems Model diagrams

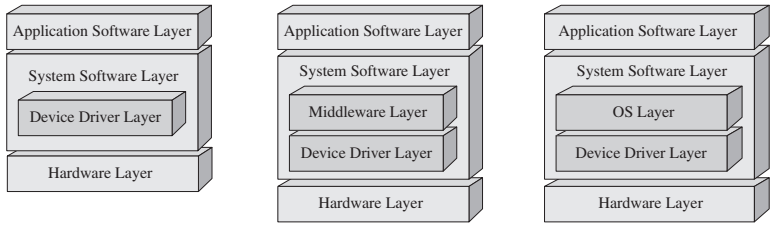


Figure 11-2b: System Software Layer and Embedded Systems Model diagrams

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.

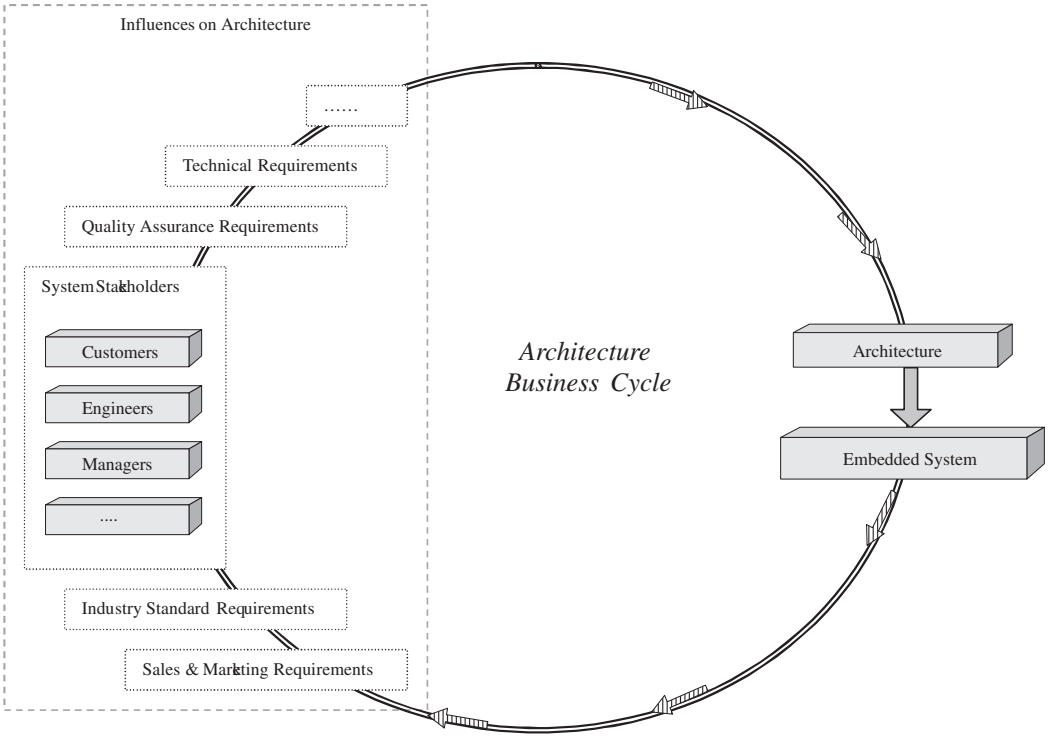


Figure 11-3: Architecture Business Cycle [11-2]

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.

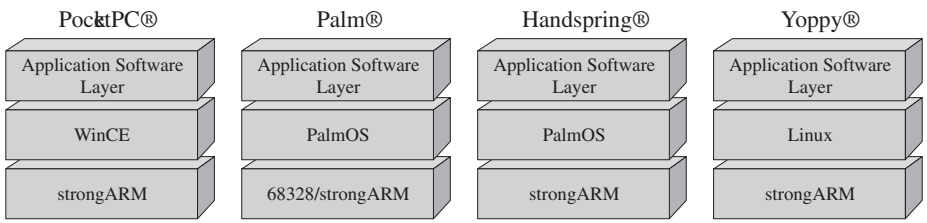


Figure 11-4: PDAs

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.

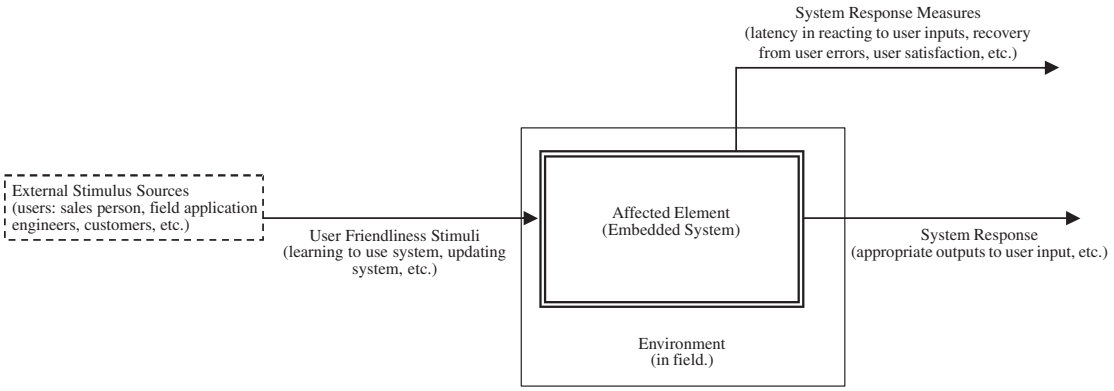


Figure 11-5: General ABC user friendliness *scenario [11-2]*

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.

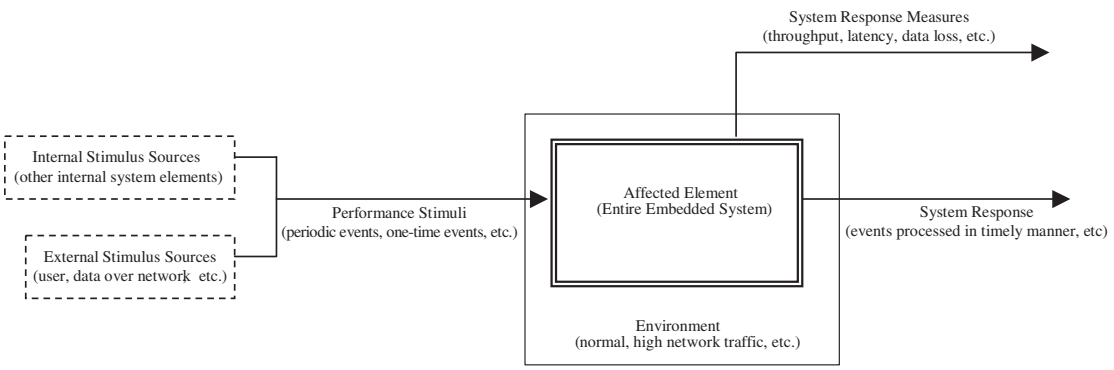


Figure 11-6a: General ABC performance *scenario [11-2]*

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.

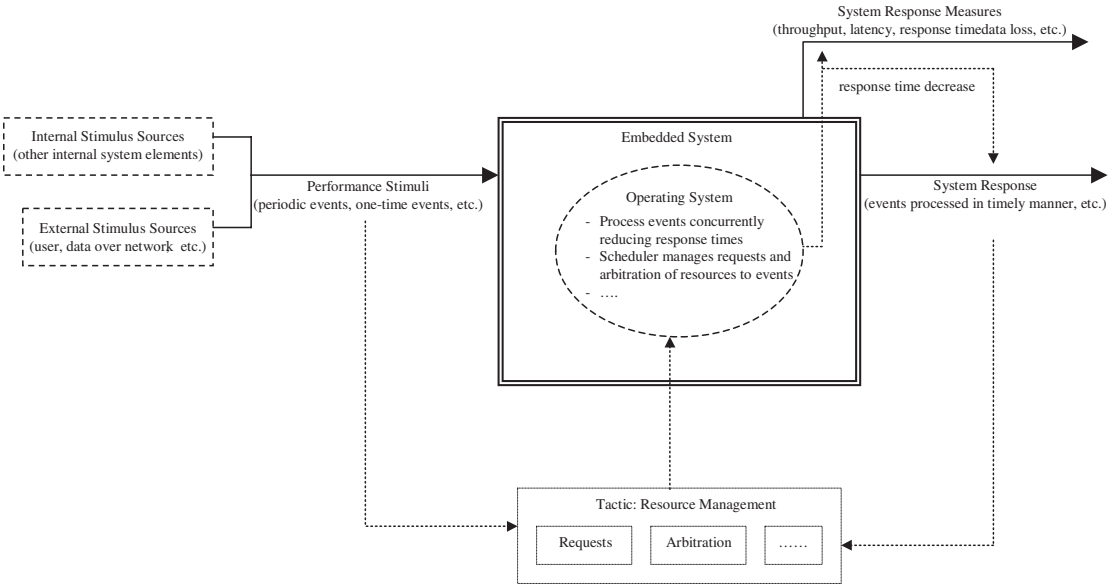


Figure 11-6b: Performance tactics and architectural elements [11-2]

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.

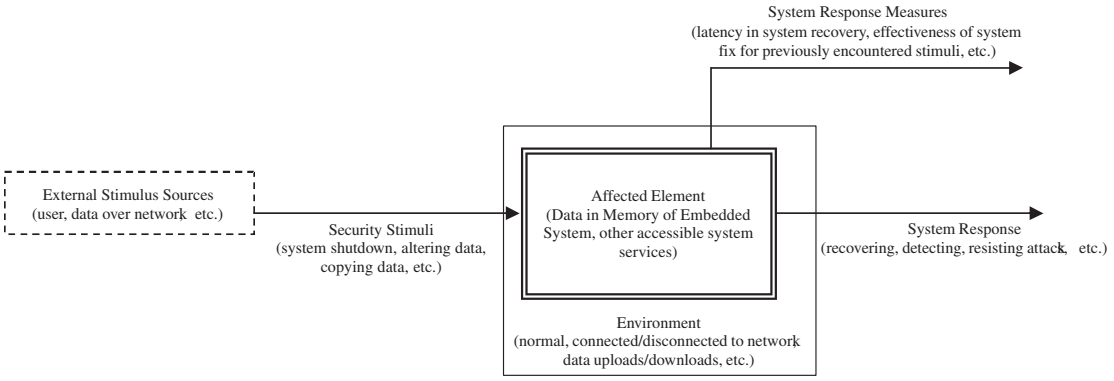


Figure 11-7a: General ABC security scenario [11-2]

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-

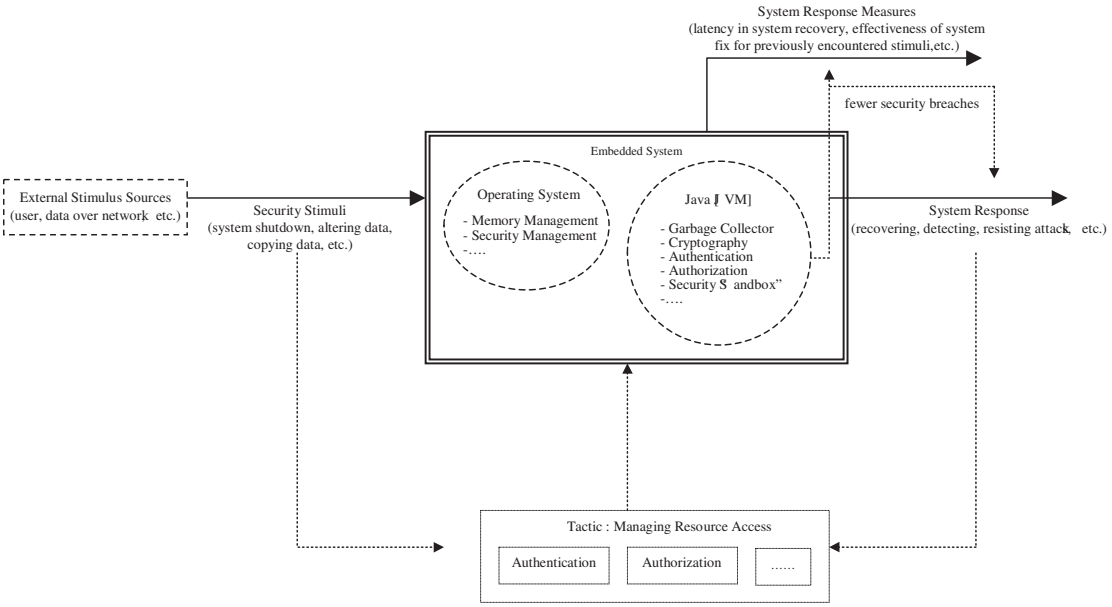


Figure 11-7b: Security tactics and architectural elements [11-2]

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.

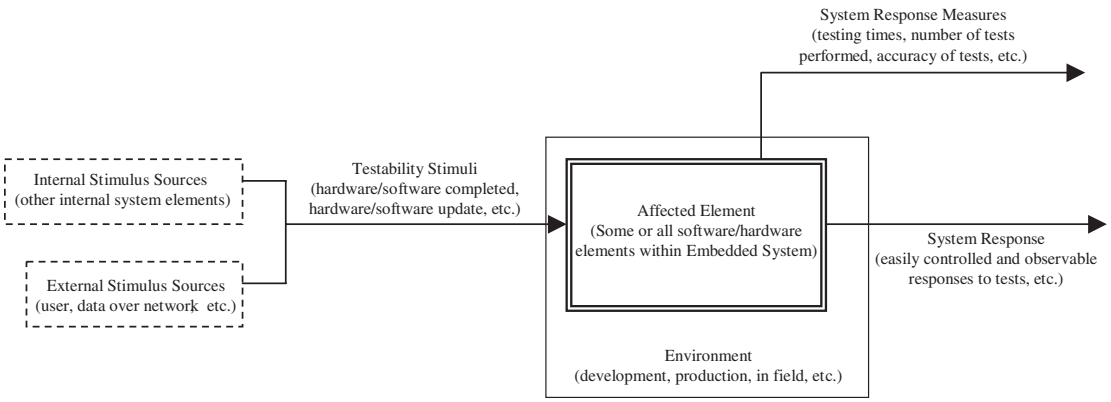


Figure 11-8a: General ABC testability *scenario* [11-2]

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.

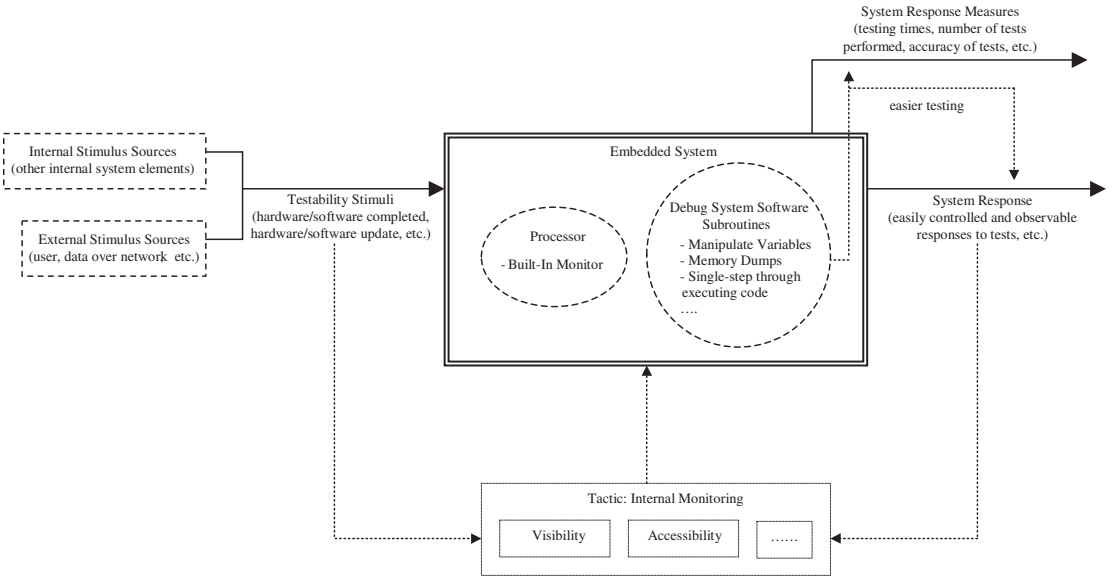


Figure 11-8b: General ABC testability *scenario* [11-2]

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.

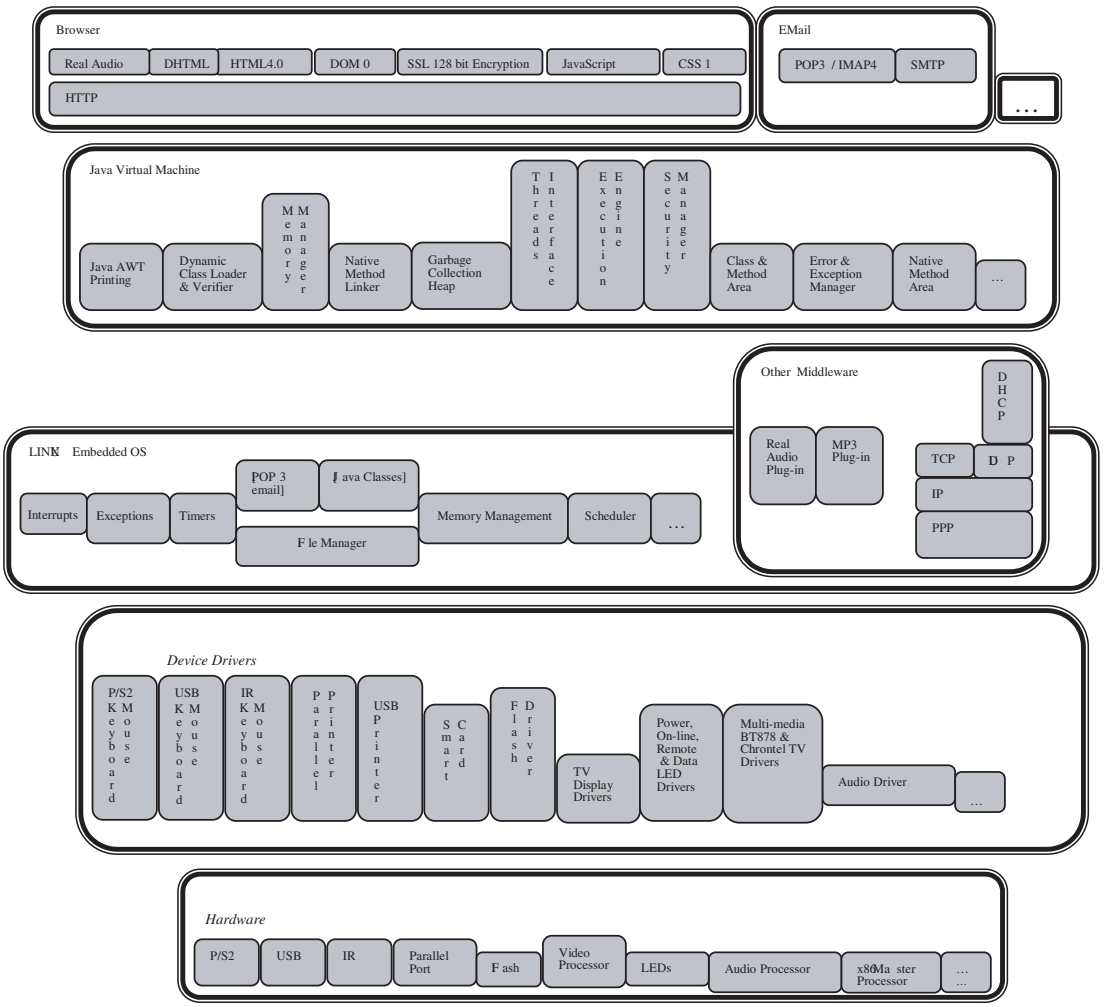


Figure 11-10: DTV-STB reference model

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 powerful 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/assembly 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 Required	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.

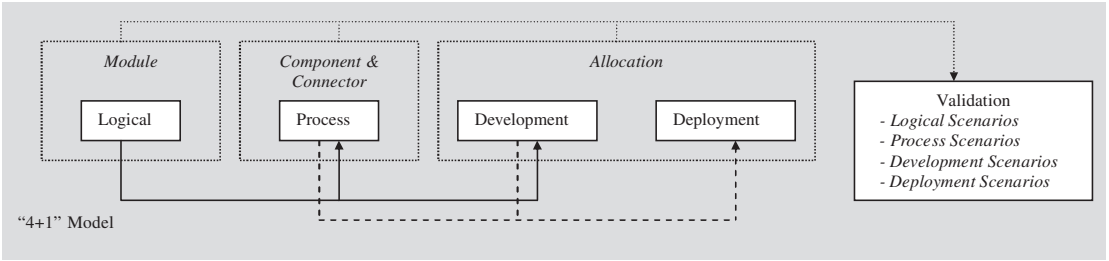


Figure 11-15: "4+1" model [11-2]

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

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.

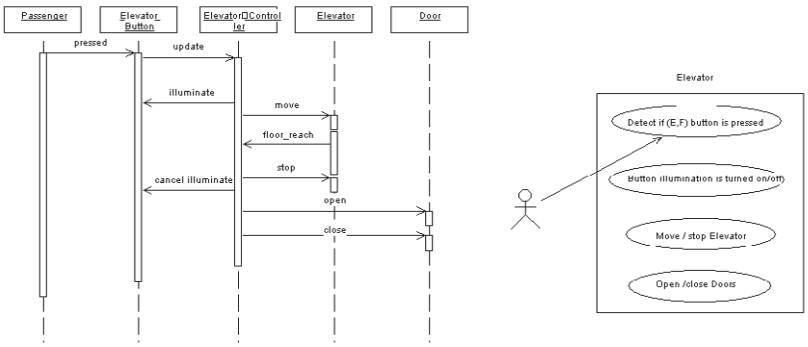


Figure 11-16a: UML diagrams [11-3]

Section 1 - Interface Identity
Section 2 - Resources Provided
  Section 2.1 - Resource Syntax
  Section 2.2 - Resource Semantics
  Section 2.3 - Resource Usage Restrictions
Section 3 - Locally defined data types
Section 4 - Exception Definitions
Section 5 - Variability Provided
Section 6 - Quality Attribute Characteristics
Section 7 - Element Requirements
Section 8 - Rationale and Design Issues
Section 9 - Usage Guide

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

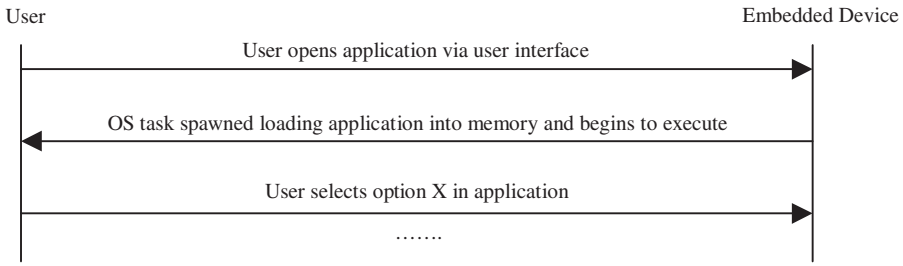


Figure 11-16c: (Rough and informal) sequence diagram

- *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.

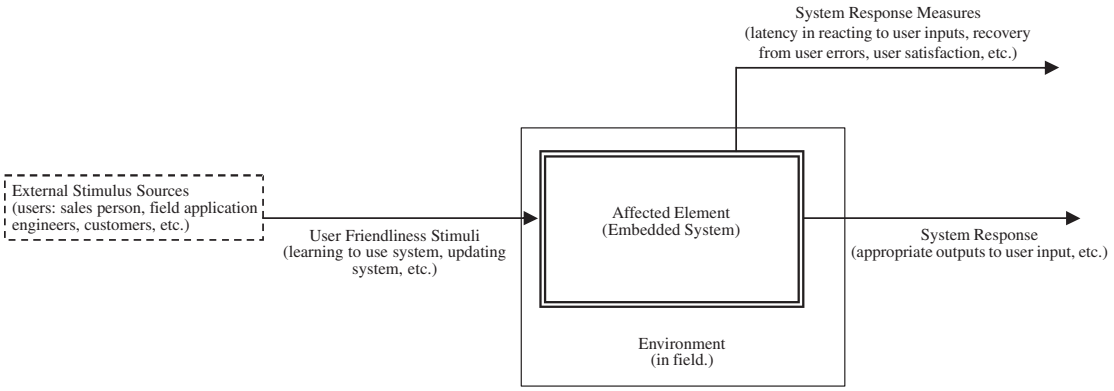


Figure 11-17 General ABC user friendliness scenario [11-2]

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?

This Page Intentionally Left Blank

CHAPTER 12

The Final Phases of Embedded Design: Implementation and Testing

In This Chapter

- *Defining the key aspects of implementing an embedded systems architecture*
- *Introducing quality assurance methodologies*
- *Discussing the maintenance of an embedded system after deployment*
- *Conclusion of book*

12.1 Implementing the Design

Having the explicit architecture documentation helps the engineers and programmers on the development team to implement an embedded system that conforms to the requirements. Throughout this book, real-world suggestions have been made for implementing various components of a design that meet these requirements. In addition to understanding these components and recommendations, it is important to understand what ***development tools*** are available that aid in the implementation of an embedded system. The development and integration of an embedded system's various hardware and software components are made possible through development tools that provide everything from loading software into the hardware to providing complete control over the various system components.

Embedded systems aren't typically developed on one system alone—for example, the hardware board of the embedded system—but usually require *at least* one other computer system connected to the embedded platform to manage development of that platform. In short, a development environment is typically made up of a ***target*** (the embedded system being designed) and a ***host*** (a PC, Sparc Station, or some other computer system where the code is actually developed). The target and host are connected by some transmission medium, whether serial, Ethernet, or other method. Many other tools, such as utility tools to burn EPROMs or debugging tools, can be used within the development environment in conjunction with host and target. (See Figure 12-1.)

The key development tools in embedded design can be located on the host, on the target, or can exist stand-alone. These tools typically fall under one of three categories: ***utility***, ***translation***, and ***debugging*** tools. Utility tools are general tools that aid in software or hardware development, such as editors (for writing source code), VCS (Version Control Software) that manages software files, ROM burners that allow software to be put onto ROMs, and so

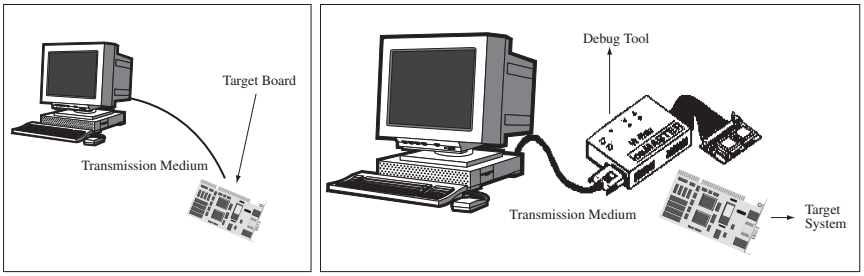


Figure 12-1: Development environments

on. Translation tools convert code a developer intends for the target into a form the target can execute, and debugging tools can be used to track down and correct bugs in the system. Development tools of all types are as critical to a project as the architecture design, because without the right tools, implementing and debugging the system would be very difficult, if not impossible.

Real-World Advice

The Embedded Tools Market

The embedded tools market is a small, fragmented market, with many different vendors supporting some subset of the available embedded CPUs, operating systems, JVMs, and so on. No matter how large the vendor, there is not yet a "one-stop-shop" where all tools for most of the same type of components can be purchased. Essentially there are many different distributions from many different tool vendors, each supporting their own set of variants or supporting similar sets of variants. Responsible system architects need to do their research and evaluate available tools *before* finalizing their architecture design to ensure that both the right tools are available for developing the system, and the tools are of the necessary quality. Waiting several months for a tool to be ported to your architecture, or for a bug fix from the vendor after development has started, is not a good situation to be in.

—Based on the article "The Trouble with The Embedded Tools Market" by Jack Ganssle

—Embedded Systems Programming. April 2004

12.1.1 The Main Software Utility Tool: Writing Code in an Editor or IDE

Source code is typically written with a tool such as a standard ASCII text editor, or an ***Integrated Development Environment*** (IDE) located on the host (development) platform, as shown in Figure 12-2. An IDE is a collection of tools, including an ASCII text editor, integrated into one application user interface. While any ASCII text editor can be used to write any type of code, independent of language and platform, an IDE is specific to the platform and is typically provided by the IDE's vendor, a hardware manufacturer (in a starter kit that bundles the hardware board with tools such as an IDE or text editor), OS vendor, or language vendor (Java, C, etc.).

Figure 12-2: IDEs [12-1]

12.1.2 Computer-Aided Design (CAD) and the Hardware

Computer-Aided Design (CAD) tools are commonly used by hardware engineers to simulate circuits at the electrical level in order to study a circuit's behavior under various conditions before they actually build the circuit.

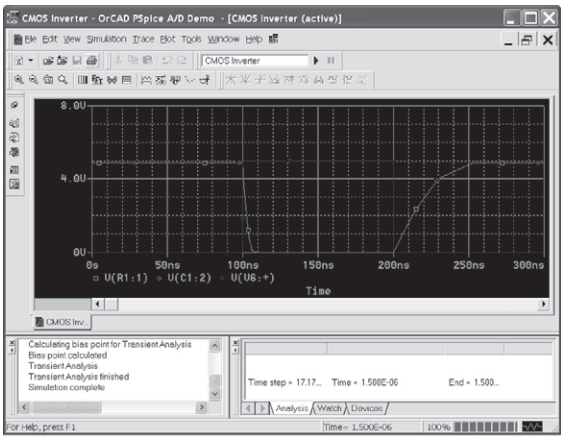


Figure 12-3a: Pspice CAD simulation sample [12-2]

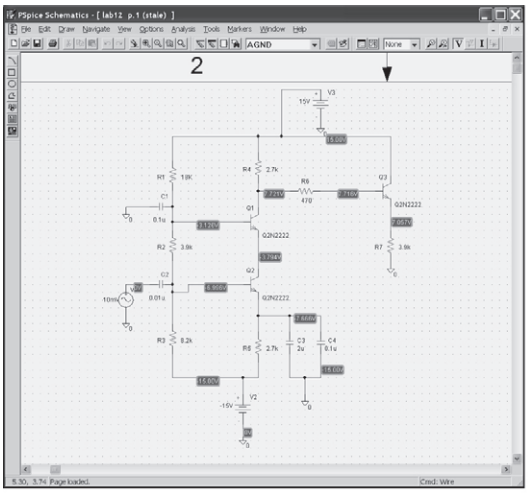


Figure 12-3b: Pspice CAD circuit sample [12-2]

Figure 12-3a is a snapshot of a popular standard circuit simulator, called PSpice. This circuit simulation software is a variation of another circuit simulator that was originally developed at University of California, Berkeley called SPICE (Simulation Program with Integrated Circuit Emphasis). PSpice is the PC version of SPICE, and is an example of a simulator that can do several types of circuit analysis, such as nonlinear transient, nonlinear dc, linear ac, noise, and distortion to name a few. As shown in Figure 12-3b, circuits created in this simulator can be made up of a variety of active and/or passive elements. Many commercially available

electrical circuit simulator tools are generally similar to PSpice in terms of their overall purpose, and mainly differ in what analysis can be done, what circuit components can be simulated, or the look and feel of the user interface of the tool.

Because of the importance of and costs associated with designing hardware, there are many industry techniques in which CAD tools are utilized to simulate a circuit. Given a complex set of circuits in a processor or on a board, it is very difficult, if not impossible, to perform a simulation on the whole design, so a hierarchy of *simulators* and *models* are typically used. In fact, the use of models is one of the most critical factors in hardware design, regardless of the efficiency or accuracy of the simulator.

At the highest level, a behavioral model of the entire circuit is created for both analog and digital circuits, and is used to study the behavior of the entire circuit. This behavioral model can be created with a CAD tool that offers this feature, or can be written in a standard programming language. Then depending on the type and the makeup of the circuit, additional models are created down to the individual active and passive components of the circuit, as well as for any environmental dependencies (temperature, for example) that the circuit may have.

Aside from using some particular method for writing the circuit equations for a specific simulator, such as the tableau approach or modified nodal method, there are simulating techniques for handling complex circuits that include one or some combination of: [12-1]

- dividing more complex circuits into smaller circuits, and then combining the results.
- utilizing special characteristics of certain types of circuits.
- utilizing vector-high speed and/or parallel computers.

12.1.3 Translation Tools—Preprocessors, Interpreters, Compilers, and Linkers

Translating code was first introduced in Chapter 2, along with a brief introduction to some of the tools used in translating code, including preprocessors, interpreters, compilers, and linkers. As a review, after the source code has been written, it needs to be translated into machine code, since machine code is the only language the hardware can directly execute. All other languages need development tools that generate the corresponding machine code the hardware will understand. This mechanism usually includes one or some combination of *preprocessing*, *translation*, and/or *interpretation* machine code generation techniques. These mechanisms are implemented within a wide variety of translating development tools.

Preprocessing is an optional step that occurs either before 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. The preprocessor can be 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, to target code through the use of a ***compiler***, a program which generates some target language, such as machine code, Java byte code, etc., from the source language, such as assembly, C, Java, etc. (see Figure 12-4).

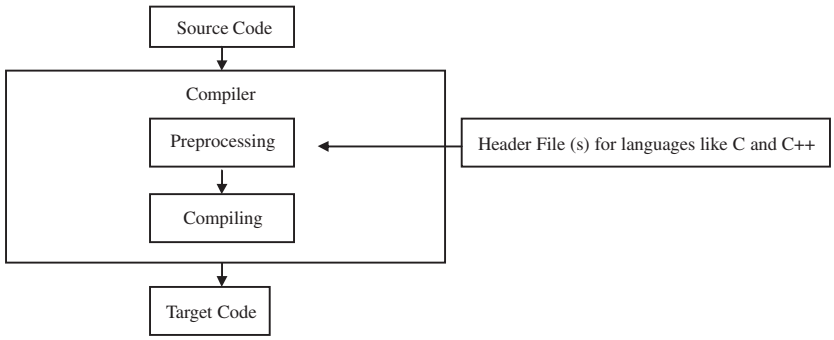


Figure 12-4: Compilation diagram

A compiler typically translates all of the source code to a target code at one time. As is usually the case in embedded systems, most 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, an assembly compiler is a specialized cross-compiler referred to as an ***assembler***, and will always generate machine code. Other high-level language compilers are commonly referred to by the language name plus "compiler" (i.e., Java compiler, C compiler). High-level language compilers can vary widely in terms of what is generated. Some generate machine code while others generate other high-level languages, which then require

what is produced to be run through at least one more compiler. Still 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, depending on the programming language used. As shown in Figure 12-5, a ***linker*** integrates this object file with any other required system libraries, creating what is commonly referred to as an ***executable*** binary file, either directly onto the board's memory or ready to be transferred to the target embedded system's memory by a ***loader***.

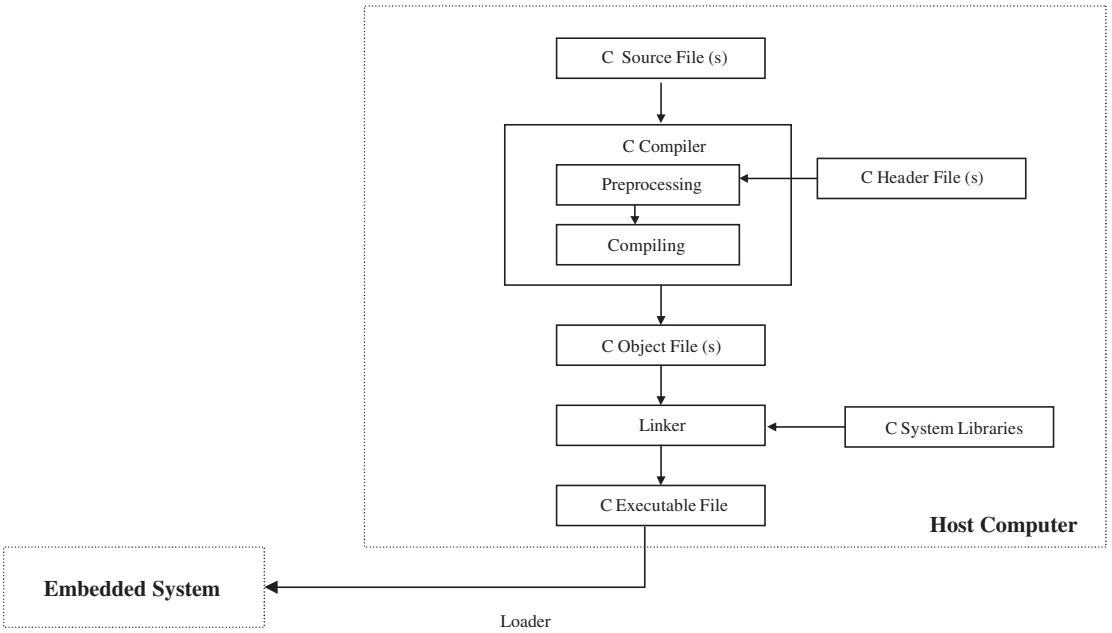


Figure 12-5: C example compilation/linking steps and object file results

One of the fundamental strengths of a translation process is based upon the concept of software placement (also referred to as *object placement*), the ability to divide the software into modules and relocate these modules of code and data anywhere in memory. This is an especially useful feature in embedded systems, because: (1) embedded designs can contain several different types of physical memory; (2) they typically have a limited amount of memory compared to other types of computer systems; (3) memory can typically become very fragmented and defragmentation functionality is not available out-of-the-box or too expensive; and (4) certain types of embedded software may need to be executed from a particular memory location.

This software placement capability can be supported by the master processor, which supplies specialized instructions that can be used to generate "position independent code," or it could be separated by the software translation tools alone. In either case, this capability depends

on whether the assembler/compiler can process only absolute addresses, where the starting address is fixed by software before the assembly processes code, or whether it supports a relative addressing scheme in which the starting address of code can be specified later and where module code is processed relative to the start of the module. Where a compiler/assembler produces relocatable modules, process instruction formats, and may do some translation of relative to physical (absolute) addresses, for example, the remaining translation of relative addresses into physical addresses, essentially the software placement, is done by the linker.

While the IDE, preprocessors, compilers, linkers and so on reside on the host development system, some languages, such as Java and scripting languages, have compilers or ***interpreters*** located on the target. An interpreter generates (interprets) machine code one source code line at a time from source code or target code generated by a intermediate compiler on the host system (see Figure 12-6 below).

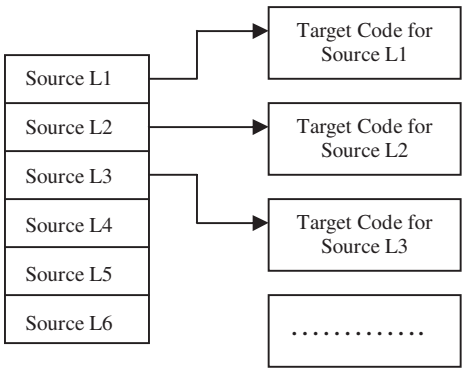


Figure 12-6: Interpretation diagram

An embedded developer can make a big impact in terms of selecting translation tools for a project by understanding how the compiler works and, if there are options, by selecting the strongest possible compiler. This is because the compiler, in large part, determines the size of the *executable* code by how well it translates the code.

This not only means selecting a compiler based on support of the master processor, particular system software, and the remaining toolset (a compiler can be acquired separately, as part of a starter kit from a hardware vendor, and/or integrated within an IDE). It also means selecting a compiler based upon a feature set that optimizes the code's simplicity, speed, and size. These features may, of course, differ between compilers of different languages, or even different compilers of the same language, but as an example would include allowing in-line assembly within the source and standard library functions that make programming embedded code a little easier. Optimizing the code for performance means that the compiler understands and makes use of the various features of a particular ISA, such as math operations, the register set, knowing the various types of on-chip ROM and RAM, the number of clock cycles for various types of accesses, etc. By understanding how the compiler translates the code, a developer

can recognize what support is provided by the compiler and learn, for example, how to program in the higher-level language supported by the compiler in an efficient manner ("*compiler-friendly* code"), and when to code in a lower-level, faster language, such as assembly.

Real-World Advice

The Ideal Embedded Compiler

Embedded systems have unique requirements and constraints atypical of the non-embedded world of PCs and larger systems. In many ways, features and techniques implemented in many embedded compiler designs evolved from the designs of non-embedded compilers. These compilers work fine for non-embedded system development, but don't address the different requirements of embedded systems development, such as limited speed and space. One of the main reasons that assembly is still so prevalent in embedded devices that use higher level languages is that developers have no *visibility* into what the compiler is doing with the higher-level code. Many embedded compilers provide no information about how the code is generated. Thus, developers have no basis to make programming decisions when using a higher-level language to improve on size and performance. Compiler features that would address some of the needs, such as the size and speed requirements unique to embedded systems design, include:

• A compiler listing file that tags each line of code with estimates of expected execution times, an expected range of execution time, or some type of formula by which to do the calculation (gathered from the target-specific information of the other tools integrated with the compiler).

• A compiler tool that allows the developer to see a line of code in its compiled form, and tag any potential problem areas.

• Providing information on size of the code via a precise size map, along with a browser that allows the programmer to see how much memory is being used by particular subroutines.

Keep these useful features in mind when designing or shopping for an embedded compiler.

—Based on the article "Compilers Don't Address Real-Time Concerns" by Jack Ganssle Embedded Systems Programming, March 1999

12.1.4 Debugging Tools

Aside from creating the architecture, debugging code is probably the most difficult task of the development cycle. Debugging is primarily the task of locating and fixing errors within the system. This task is made simpler when the programmer is familiar with the various types of debugging tools available and how they can be used (the type of information shown in Table 12-1).

As seen from some of the descriptions in Table 12-1, debugging tools reside and interconnect in some combination of standalone devices, on the host, and/or on the target board.

A Quick Comment on Measuring System Performance with Benchmarks

Aside from debugging tools, once the board is up and running, benchmarks are software programs that are commonly used to measure the performance (latency, efficiency, etc.) of individual features within an embedded system, such as the master processor, the OS, or the JVM. In the case of an OS, for example, performance is measured by how efficiently the master processor is utilized by the scheduling scheme of the OS. The scheduler needs to assign the approriate time quantum—the time a process gets access to the CPU—to a process, because if the time quantum is too small, thrashing occurs.

The main goal of a benchmark application is to represent a real workload to the system. There are many benchmarking applications available. These include EEMBC (Embedded Microprocessor Benchmark Consortium) benchmarks, the industry standard for evaluating the capabilities of embedded processors, compilers, and Java; Whetstone, which simulates arithmetic-intensive science applications; and Dhrystone, which simulates systems programming applications, used to derive MIPS introduced in Section II. The drawbacks of benchmarks are that they may not be very realistic or reproducible in a real world design that involves more than one feature of a system. Thus, it is typically much better to use real embedded programs that will be deployed on the system to determine not only the performance of the software, but the overall system performance.

In short, when interpreting benchmarks, insure you understand exactly what software was run and what the benchmarks did or did not measure.

Table 12-1: Debug tools

Tool Type	Debugging Tools	Descriptions	Examples of Uses and Drawbacks
Hardware	In-Circuit Emulator (ICE)	Active device replaces microprocessor in system	• typically most expensive debug solution, but has a lot of debugging capabilities • can operate at the full speed of the processor (depends on ICE) and to the rest of the system it is the microprocessor • allows visibility and modifiability of internal memory, registers, variables, etc. real-time • similar to debuggers, allows setting breakpoints, single stepping, etc. • usually has overlay memory to simulate ROM • processor dependent …..
	ROM Emulator	Active tool replaces ROM with cables connected to dual port RAM within ROM emulator, simulates ROM. It is an intermediate hardware device connected to the target via some cable (i.e. BDM), and connected to the host via another port	• allows modification of contents in ROM (unlike a debugger) • can set breakpoints in ROM code, and view ROM code real-time • usually doesn't support on-chip ROM, custom ASICs, etc. • can be integrated with debuggers …..

Table 12-1: Debug tools (continued)

Tool Type	Debugging Tools	Descriptions	Examples of Uses and Drawbacks
Hardware	Background Debug Mode (BDM)	BDM hardware on board (port and integrated debug monitor into master CPU), and debugger on host, connected via a serial cable to BDM port. The connector on cable to BDM port, commonly referred to as wiggler. BDM debugging sometimes referred to as On-Chip Debugging (OCD)	• usually cheaper than ICE, but not as flexible as ICE • observe software execution unobtrusively in real time • can set breakpoints to stop software execution • allows reading and writing to registers, RAM, I/O ports, etc • processor/target dependent, Motorola proprietary debug interface …..
	IEEE 1149.1 Joint Test Action Group (JTAG)	JTAG-compliant hardware on board	• similar to BDM, but not proprietary to specific architecture (is an open standard) ….
	IEEE-ISTO Nexus 5001	Options of JTAG port, Nexus-compliant port, or both, several layers of compliance (depending on complexity of master processor, engineering choice, etc.)	• offers scalable debug functions depending on level of compliance of hardware …..
	Oscilloscope	Passive analog device that graphs voltage (on vertical axis) versus time (on horizontal axis), detecting the exact voltage at a given time	• monitor up to 2 signals simultaneously • can set a trigger to capture voltage given specific conditions • used as voltmeter (though a more expensive one) • can verify circuit is working by seeing signal over bus or I/O ports • capture changes in a signal on I/O port to verify segments of software are running, calculate timing from one signal change to next, etc. • processor independent ….
	Logic Analyzer	Passive device that captures and tracks multiple signals simultaneously and can graph them	• can be expensive • typically can only track 2 voltages (VCC and ground); signals in-between are graphed as either one or the other • can store data (whereas only storage oscilloscopes can store captured data) • 2 main operating modes (timing, state) to allow triggers on changes of states of signal (i.e., high-to-low or low-to-high) • capture changes in a signal on I/O port to verify segments of software are running, calculate timing from one signal change to next, etc. (timing mode) • can be triggered to capture data from a clock event off the target or an internal logic analyzer clock

Table 12-1: Debug tools (continued)

Tool Type	Debugging Tools	Descriptions	Examples of Uses and Drawbacks
Hardware	Logic Analyzer *(continued)*	Passive device that captures and tracks multiple signals simultaneously and can graph them	• can trigger if processor accesses off-limits section of memory, writes invalid data to memory, or accesses a particular type of instruction (state mode) • some will show assembly code, but usually cannot set break point and single-step through code using analyzer • logic analyzer can only access data transmitted externally to and from processor, not the internal memory, registers, etc. • processor independent and allows view of system executing in real time with very little intrusion …
	Voltmeter	Measures voltage difference between 2 points on circuit	• to measure for particular voltage values • to determine if circuit has any power at all • cheaper then other hardware tools …
	Ohmmeter	Measures resistance between 2 points on circuit	• cheaper than other hardware tools • to measure changes in current/voltage in terms of resistance (Ohm's Law V=IR) • …
	Multimeter	measures both voltage and resistance	• same as volt and ohm meters • …
Software	Debugger	Functional debugging tool	Depends on the debugger – in general: • loading/singlestepping/tracing code on target • implementing breakpoints to stop software execution • implementing conditional breakpoints to stop if particular condition is met during execution • can modify contents of RAM, typically cannot modify contents of ROM ….
	Profiler	Collects the timing history of selected variables, registers, etc.	• capture time dependent (when) behavior of executing software • to capture execution pattern (where) of executing software ….

Table 12-1: Debug tools (continued)

Tool Type	Debugging Tools	Descriptions	Examples of Uses and Drawbacks
Software	Monitor	Debugging interface similar to ICE, with debug software running on target and host. Part of monitor resides in ROM of target board (commonly called debug agent or target agent), and a debugging kernel on the host. Software on host and target typically communicate via serial or Ethernet (depends on what is available on target).	• similar to print statement but faster, less intrusive, works better for soft real-time deadlines, but not for hard real-time • similar functionality to debugger (breakpoints, dumping registers and memory, etc.) • embedded OSes can include monitor for particular architectures ….
	Instruction Set Simulator	Runs on host and simulates master processor and memory (executable binary loaded into simulator as it would be loaded onto target) and mimics the hardware	• typically does not run at exact same speed of real target, but can estimate response and throughput times by taking in consideration the differences between host and target speeds • verify assembly code is bug free • usually doesn't simulate other hardware that may exist on target, but can allow testing of built-in processor components • can simulate interrupt behavior • capture variable, memory and register values • more easily port code developed on simulator to target hardware • will not precisely simulate the behavior of the actual hardware in real-time • typically better suited for testing algorithms rather than reaction to events external to an architecture or board (waveforms and such need to be simulated via software) • typically cheaper than investing in real hardware and tools …..
Manual	Readily available, free or cheaper than other solutions, effective, simpler to use but usually more highly intrusive than other types of tools, not enough control over event selection, isolation, or repeatability. Difficult to debug real-time system if manual method takes too long to execute.		
	Print Statements	Functional debugging tool, printing statements inserted into code that print variable information, location in code information, etc.	• to see output of variables, register values, etc. while the code is running • to verify segment of code is being executed • can significantly slow down execution time • can cause missed deadlines in real-time system. ….

Table 12-1: Debug tools (continued)

Tool Type	Debugging Tools	Descriptions	Examples of Uses and Drawbacks
Manual	Dumps	Functional debugging tool that dumps data into some type of storage structure at runtime	• same as print statements but allows faster execution time in replacing several print statements (especially if there is a filter identifying what specific types of information to dump or what conditions need to be met to dump data into the structure) • see contents of memory at runtime to determine if any stack/heap overruns ….
	Counters/Timers	Performance and efficiency debugging tool in which counters or timers reset and incremented at various points of code	• collect general execution timing information by working off system clock or counting bus cycles, etc. • some intrusiveness ….
	Fast Display	Functional debugging tool in which LEDs are toggled or simple LCD displays present some data	• similar to print statement but faster, less intrusive, working well for real-time deadlines • allows confirmation that specific parts of code running …
	Ouput ports	Performance, efficiency, and functional debugging tool in which output port toggled at various points in software	• with an oscilloscope or logic analyzer, can measure when port is toggled and get execution times between toggles of port • same as above but can see on oscilloscope that code is being executed in first place • in multitasking/multithreaded system assign different ports to each thread/task to study behavior ….

Some of these tools are active debugging tools and are intrusive to the running of the embedded system, while other debug tools passively capture the operation of the system with no intrusion as the system is running. Debugging an embedded system usually requires a combination of these tools in order to address all of the different types of problems that can arise during the development process.

Real-World Advice

The Cheapest Way To Debug

Even with all the available tools, developers should still try to reduce debugging time and costs, because 1) the cost of bugs increases the closer to production and deployment time the schedule gets, and 2) the cost of a bug is logarithmic (it can increase tenfold when discovered by a customer versus if it had been found during development of the device). Some of the most effective means of reducing debug time and cost include:

• Not developing too quickly and sloppily. The cheapest and fastest way to debug is to *not insert any bugs in the first place*. Fast and sloppy development actually delays the schedule with the amount of time spent on debugging mistakes.

• *System Inspections*. This includes hardware and software inspections throughout the development process that ensures that developers are designing according to the architecture specifications, and any other standards required of the engineers. Code or hardware that doesn't meet standards will have to be "debugged" later if system inspections aren't used to flush them out quickly and cheaply (relative to the time spent debugging and fixing all that much more hardware and code later).

• *Don't use faulty hardware or badly written code*. A component is typically ready to be redesigned when the responsible engineer is fearful of making any changes to the offending component.

• *Track the bugs* in a general text file or using one of the many bug tracking off-the-shelf software tools. If components (hardware or software) are continuously causing problems, it may be time to redesign that component.

• *Don't skimp on the debugging tools*. One good (albeit more expensive) debugging tool that would cut debug time is worth more than a dozen cheaper tools that, without a lot of time and headaches, can barely track down the type of bugs encountered in the process of designing an embedded system.

And finally what I (the author of this book) believe is one of the best methods by which to reduce debug times and costs—*read the documentation* provided by the vendor and/or responsible engineers first, before trying to run or modify anything. I have heard many, many excuses over the years—from "I didn't know what to read" to "Is there documentation?"—as to why an engineer hasn't read any of the documentation. These same engineers have spent hours, if not days, on individual problems with configuring the hardware or getting a piece of software running correctly. I know that if these engineers had read the documentation in the first place, the problem would have been resolved in seconds or minutes – or might not have occurred at all.

If you are overwhelmed with documentation and don't know what to read first, anything titled along the lines of "Getting Started…", "Booting up the system…", or "README" are good indicators of a place to begin. ☺ Moreover, take the time to read *all* of the documentation provided with any hardware or software to become familiar with what type of information is there, in case it's needed later.

—Based on the article "Firmware Basics for the Boss" by Jack Ganssle, Embedded Systems Programming, February 2004

12.1.5 System Boot-Up

With the development tools ready to go, and either a reference board or development board connected to the development host, it is time to start up the system and see what happens. System boot-up means that some type of power-on or reset source, such as an internal/external hard reset (i.e., generated by a check-stop error, the software watchdog, a loss of lock by the PLL, debugger, etc.) or an internal/external soft reset (i.e., generated by a debugger, application code, etc.), has occurred. When power is applied to an embedded board (because of a reset), start-up code, also referred to as ***boot code***, ***bootloader***, ***bootstrap*** code, or ***BIOS*** (basic input output system) depending on the architecture, in the system's ROM is loaded and executed by the master processor. Some embedded (master) architectures have an internal program counter that is automatically configured with an address in ROM in which the start of the boot-up code (or table) is located, while others are hardware wired to start executing at a specific location in memory.

Boot code differs in length and functionality depending on where in the development cycle the board is, as well as the components of the actual platform that need initialization. The same (minimal) general functions are performed by boot code across the various platforms, which are basically initializing the hardware, which includes disabling interrupts, initializing buses, setting the master and slave processors in a specific state, and initializing memory. This first hardware initialization portion of boot-up code is essentially the executing of the initialization device drivers, as discussed in Chapter 8. How initialization is actually done—that is, the order in which drivers are executed—is typically outlined by the master architecture documentation or in documentation provided by the manufacturers of the board. After the hardware initialization sequence, executed via initialization device drivers, the remaining system software, if any, is then initialized. This additional code may exist in ROM, for a system that is being shipped out of the factory, or loaded from an external host platform (see the callout box with bootcodeExample).

```
bootcodeExample ()

{
….
  // Serial Port Initialization Device Driver
   initializeRS232(UART,BAUDRATE,DATA_BITS,STOP_BITS,PARITY);

  // Initialize Networking Device Driver
  initializeEthernet(IPAddress,Subnet, GatewayIP, ServerIP);

  //check for host development system for down loaded file of rest of code to RAM
  // through ethernet
  // start executing rest of code(i.e. define memory map, load OS, etc.)

…
}
```

MPC823-Based Board Booting Example

The MPC823 processor contains a reset controller that is responsible for responding to all reset sources. The actions taken by the reset controller differ depending on the source of a reset event, but in general the process includes reconfiguring the hardware, and then sampling the data pins or using an internal default constant to determine the initial reset values of system components.

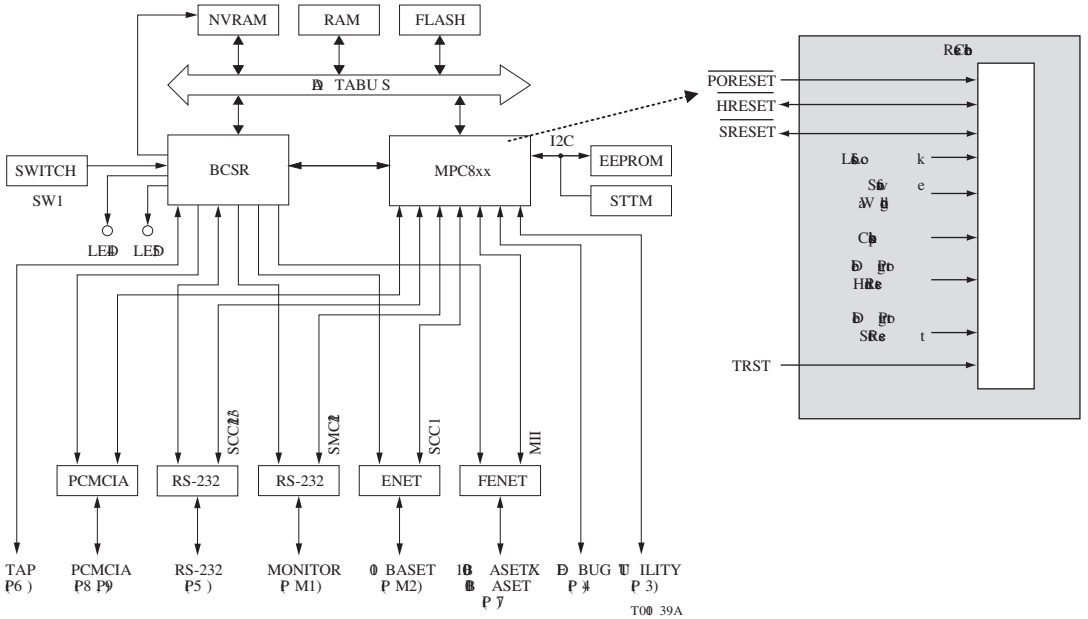


Figure 12-7a: Interpretation diagram [12-3]

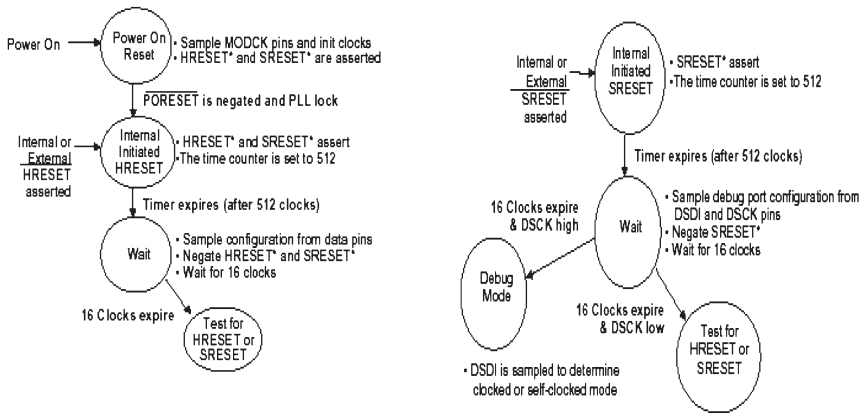


Figure 12-7b: Interpretation diagram [12-3]

The data pins sample represents initial configuration (setup) parameters as shown in Figure 12-7c.

If..		then..	
No external arbitration	SIUMCR.EARB = 0	D0 = 0	D0
External arbitration	SIUMCR.EARB = 1	D0 = 1	
EVT at 0	MSR.IP = 0	D1 = 1	D1
EVT at 0xFFF00000	MSR.IP = 1	D1 = 0	
Do not activate memory controller	BR0.V = 0	D3 = 1	D3
Enable CS0	BR0.V = 1	D3 = 0	
Boot port size is 32	BR0.PS = 00	D4 = 0, D5 = 0	D4 D5
Boot port size is 8	BR0.PS = 01	D4 = 0, D5 = 1	
Boot port size is 16	BR0.PS = 10	D4 = 1, D5 = 0	
Reserved	BR0.PS = 11	D4 = 1, D5 = 1	
DPR at 0	immr = 0000xxxx	D7 = 0, D8 = 0	D7 D8
DPR at 0x00F00000	immr = 00F0xxxx	D7 = 0, D8 = 1	
DPR at 0xFF000000	immr = FF00xxxx	D7 = 1, D8 = 0	
DPR at 0xFFF00000	immr = FFF0xxxx	D7 = 1, D8 = 1	
Select PCMCIA functions, Port B	SIUMCR.DBGC = 0	D9 = 0, D10 = 0	D9 D10
Select Development Support functions	SIUMCR.DBGC = 1	D9 = 0, D10 = 1	
Reserved	SIUMCR, DBGC = 2	D9 = 1, D10 = 0	
Select program tracking functions	SIUMCR.DBGC = 3	D9 = 1, D10 = 1	
Select as in DBGC + Dev. Supp. comm pins	SIUMCR.DBPC = 0	D11 = 0, D12 = 0	D11 D12
Select as in DBGC + JTAG pins	SIUMCR.DBPC = 1	D11 = 0, D12 = 1	
Reserved	SIUMCR.DBPC = 2	D11 = 1, D12 = 0	
Select Dev. Supp. comm and JTAG pins	SIUMCR.DBPC = 3	D11 = 1, D12 = 1	
CLKOUT is GCLK2 divided by 1	SCCR.EBDF = 0	D13 = 0, D14 = 0	D13 D14
CLKOUT is GCLK2 divided by 2	SCCR.EBDF = 1	D13 = 0, D14 = 1	
Reserved	SCCR.EBDF = 2	D13 = 1, D14 = 0	
Reserved	SCCR.EBDF = 3	D13 = 1, D14 = 1	

D0 specifies whether external arbitration or the internal arbiter is to be used.

D1 controls the initial location of the exception vector table, and the IP bit in the machine state register is set accordingly.

D3 specifies whether Chip Select 0 is active on reset.

If Chip Select 0 is active on reset pins D4 and D5 specify the port size of theboot ROM, with a choice of 8, 16, or 32 bits.

D7 and D8 specify the initial value for the IMMR registers. There are four different possible locations for the internal memory map.

D9 and D10 select the configuration for the debug pins.

D11 and D12 select the configuration of the debug port pins. This selection involves configuring these pins either as JTAG pins, or development support communication pins.

D13 and D14 determine which clock scheme is in use; one clock scheme Implements GCLK2 divided by one, and the second implements GCLK2 divided by two.

Figure 12-7c: Interpretation diagram [12-3]

The Embedded Planet RPXLite Board assumes that onboard ROM (FLASH) contains the bootloader monitor/program, called PlanetCore, originally created by Embedded Planet. The PowerPC processor and on-board memory start up in a default configuration set via hardware (CS0 is an output pin that can be configured to be the global chip select for the boot device, HRESET/SRESET, data pins,…) , and has no dedicated accessible PC register.

Chip Select	Port Size	Function/Address	Comment
CS0#	x32	FLASH (x32) FFFF FFFF minus actual FLASH size	Reset vector at IP = 1: 0000 0100 Vector set at IP = 1 in hardware BR0 set at FFFF minus FLASH size 2, 4, 8, or 16 Mbytes
CS1#	x32	SDRAM (x32) 0000	16, 32, or 64 Mbytes
CS2#		Expansion Header UUUU	Routed to expansion receptacle
CS3#	x32	Control and Status Registers FA40	Byte and/or word accessible
CS4#	x8	NVRAM/RTC or SRAM/RTC FA00	0K, 32K, 128K, or 512 Kbytes Also available at Expansion Receptacle
CS5#		Expansion Header UUUU	Routed to expansion receptacle
CS6#	x16 or U	PCMCIA Slot B Chip Select Even Bytes or Chip Select 6 to I/O Header UUUU	OP2 in MPC850 PCMCIA control register selects mode: L = PCMCIA Slot B enabled H = CS6# to expansion header enabled
CS7#	x16 or U	PCMCIA Slot B Chip Select Odd Bytes or Chip Select 7 to I/O Header UUUU	OP2 in MPC850 PCMCIA control register selects mode: L = PCMCIA Slot B enabled H = CS7# to I/O expansion header enabled
IMMR	x32	Value at reset = FF00 0000, then set to FA20 0000	

Figure 12-7d: Interpretation diagram [12-3]

The default configuration executed via hardware includes the configuration of only one bank of memory, whose base address is determined by D7 and D8 where 00 = 0x00000000, 01 = 0x00F000000, 10 = 0xFF000000, 11 = 0xFFF00000. This bank is in some type of ROM (i.e., Flash) and is where the boot code resides. After the board has been powered on, the PowerPC processor executes the boot code in this memory bank to complete the initialization and configuration sequence. In fact, all the MPC8xx processor series (not just the MPC823) require either a high or low boot, depending on the specific board and revision, meaning PlanetCore is located either at the high end of Flash or at the low end of Flash. PlanetCore starts at virtual address 0xFFF00000 if it is located at the high end of Flash. On the other hand, PlanetCore is in the first sectors of the Flash—i.e., located at virtual address 0xFC000000 for 64 Mbytes of Flash—if it is located at the low end of Flash.

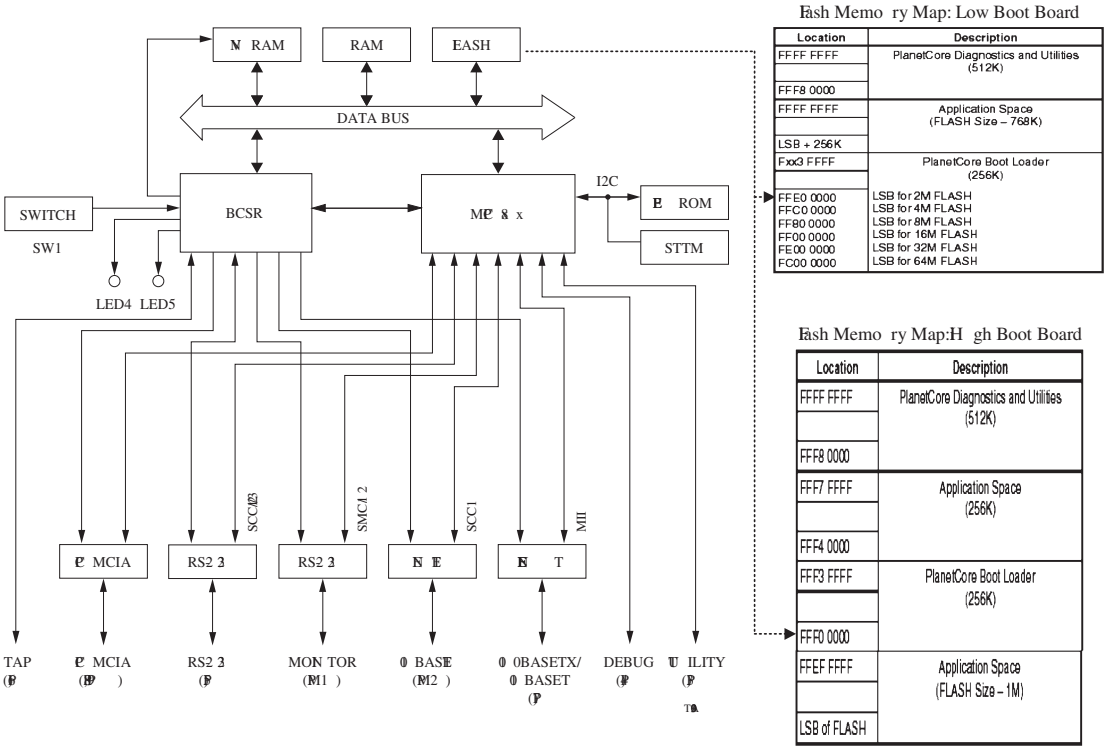


Figure 12-7e: Interpretation diagram [12-3]

On this MPC823-based board, after the hardware initialization sequence initializing the processor, the CPU begins executing the PlanetCore bootloader code. As shown in the following callout box, all of the hardware specific to the MPC823 architecture as well as specific to the board is initialized (i.e., serial, networking, etc.) via this type of boot code.

```
/*********************************************************************
 *                    c_entry
 * Description :
 * -------------
 *
 * First C-function
 *
 * Return values :
 * ---------------
 *
 * Never returns
 *********************************************************************/
int c_entry(void){

BootLoader
-       Board initialization for custom BSP

Initializing the MPC823, itself (not board initialization), involves about 24 steps, which includes :

        1. disable the data cache to prevent a machine check error from occuring
        2. Initialize the Machine State Register and the Save and Restore Register 1 with a
        value of 0x1002.
        3. Initialize the Instruction Support Control Register, or ICTRL, modifying it so that
        the core is not serialized (which has an impact on performance).
        4. Initialize the Debug Enable Register, DER.
        5. Initialize the Interrupt Cause Register, ICR. T
        6. Initialize the Internal Memory Map Register, or IMMR.
        7. Initialize the Memory Controller Base and Options registers as required.
        8. Initialize the Memory Periodic Timer Pre-scalar Register, or MPTPR.
        9. Initialize the Machine Mode Registers, MAMR and MBMR.
        10. Initialize the SIU Module Configuration Register, or SIUMCR. Note that this step
        configures many of the pins shown on the right hand side of the main pin diagram in
        the User Manual.
        11. Initialize the System Protection Register, or SYPCR. This register contains settings
        for the bus monitor and the software watchdog.
        12. Initialize the Time Base Control and Status Register, TBSCR.
        13. Initialize the Real Time Clock Status and Control Register, RTCSC.
        14. Initialize the Periodic Interrupt Timer Register, PISCR.
        15. Initialize the UPM RAM arrays using the Memory Command Register and the
        Memory Data Register. We also discuss this routine in the chapter regarding the
        memory controller.
        16. Initialize the PLL Low Power and Reset Control Register, or PLPRCR.
        17. is not required, although many programmers implement this step. This step moves
        the ROM vector table to the RAM vector table.
```

18. changes the location of the vector table. The example shows this procedure byget-
ting the Machine State Register, setting or clearing the IP bit, and writing the Machine
State Register back again.
19. Disable the instruction cache.
20. Unlock the instruction cache.
21. Invalidate the instruction cache.
22. Unlock the data cache.
23. Verify whether the cache was enabled, and if so, flush it.
24. invalidates the data cache.

- Initialization of all components: processor, clocks, EEPROM, I2C, serial, Ethernet 10/100, chip selects, UPM machine, DRAM initialization, PCMCIA(Type I and II), SPI,UART, video encoder, LCD, audio, touch screen, IR,…

Flash Burner
Diagnostics and Utilities
- Test DRAM
- Command line interface

}

[12-3]

MIPS32-Based Booting Example

The Ampro Encore M3 Au1500 based board assumes that on-board ROM (i.e., Flash) contains the bootloader monitor/program, called YAMON, originally created by MIPS Technologies. Where this boot ROM is mapped on the Au1500 is based upon the requirements of the MIPS architecture itself, which specifies that upon a reset, a MIPS processor must fetch the Reset exception vector from address 0xBFC00000. Basically, when a cold boot occurs on a MIPS32-based processor, a reset exception occurs which performs a full reset "hardware" initialization sequence that (in general) puts the processor in a state of executing instructions from unmapped, uncached memory, initializes registers (such as Rando, Wired, Config, and Status) for reset, and then loads the PC with 0xBFC0_0000, the Reset Exception Vector.

0xBFC0_0000 is a virtual address, not a physical address. All addresses under the MIPS32 architecture are virtual addresses, meaning the actual physical memory address on the board is translated when processing, such as instruction fetches and data loading and storing. The upper bits of the virtual address define the different regions in the memory map; for example:

- KUSEG (2 GBytes of virtual memory ranging from 0x0000000–0x7FFFFFFF).
- KSEG0 (512 MB virtual memory from 0x8000000 to 9FFFFFFF) which is a direct map to physical addresses and inherently cacheable.
- KSEG1 (512 MB virtual memory from 0xA000000 to BFFFFFFF) which is a direct map to physical addresses and inherently non-cacheable.

This means virtual addresses (KSEG0) 0x80000000 (a cacheable view of physical memory) and (KSEG1) 0xA00000000 (a non-cacheable view of physical memory) both map directly onto physical address 0x00000000. The MIPS32 Reset Exception Vector (0xBFC0_0000) is located in the last 4 Mbytes of the KSEG1 region of memory, a non-cacheable region that can execute even if other board components are not yet initialized. This means that a physical address of 0x1FC00000 is generated for the first instruction fetch from the boot ROM. Basically, the programmer puts the start of the boot code (i.e., YAMON) at 0x1FC0_0000, which is the value the PC is set to start executing upon power-on, and effectively could occupy the entire 4MB of space (0x1FC00000 thru 0x1FFFFFFF) or more.

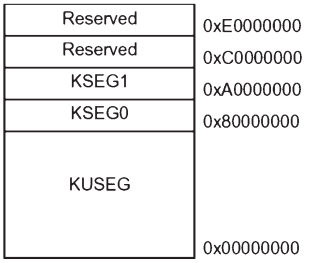
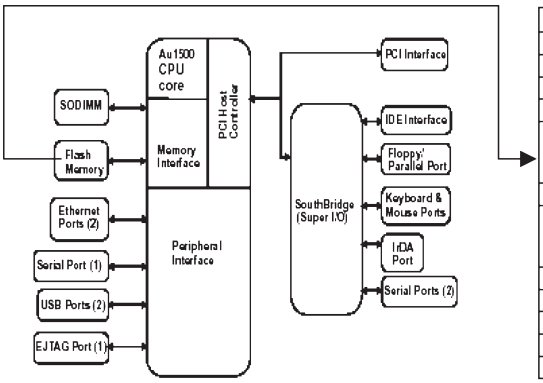


Figure 12-8a: Interpretation diagram [12-4]



Start Address	End Address	Size (MB)	Function
0x0 00000000	0x0 0FFFFFFF	256	Memory KSEG 0/1
0x0 10000000	0x0 11FFFFFF	32	I/O Devices on Peripheral Bus
0x0 12000000	0x0 13FFFFFF	32	Reserved
0x0 14000000	0x0 17FFFFFF	64	I/O Devices on System Bus
0x0 18000000	0x0 1FFFFFFF	128	Memory Mapped 0x0 1FC00000 must contain the boot vector so this is typically where Flash or ROM is located.
0x0 20000000	0x0 7FFFFFFF	1536	Memory Mapped
0x0 80000000	0x0 EFFFFFFF	1792	Memory Mapped Currently this space is memory mapped, but it should be considered reserved for future use.
0x0 F0000000	0x0 FFFFFFFF	256	Debug Probe
0x1 00000000	0x3 FFFFFFFF	4096 * 3	Reserved
0x4 00000000	0x4 FFFFFFFF	4096	PCI Uncached Memory Space
0x5 00000000	0x5 FFFFFFFF	4096	PCI I/O Space
0x6 00000000	0x6 FFFFFFFF	4096	PCI Configuration Space
0x7 00000000	0xC FFFFFFFF	4096 * 7	Reserved
0xD 00000000	0xD FFFFFFFF	4096	I/O Device
0xE 00000000	0xE FFFFFFFF	4096	External LCD Controller Interface
0xF 00000000	0xF FFFFFFFF	4096	PCMCIA Interface

Figure 12-8b: Interpretation diagram [12-4]

Sys Address	Flash Address	Sectors	Description
bfC00000 – bfC03FFF	00000000 – 00003FFF	0	Reset Image (16kB)
bfC04000 – bcC05FFF	00004000 – 00005FFF	1	Boot Line (8kB0
bfC06000 – bfC07FFF	00006000 – 00007FFF	2	Parameter Flash (8kB0
bfC08000 – bfC0FFFF	00008000 – 0000FFFF	3	User NVRAM (32kB)
bfC10000 – bfC8FFFF	00010000 – 0008FFFF	4–11	YAMON Little Endian (512kB)
bfC90000 – bfD0FFFF	00090000 – 0010FFFF	12–19	YAMON Big Engian (512kB)
bfD10000 – bfDEFFFF	00110000 – 001EFFFF	20–33	System Flash (896kB)
bfDF0000 – bfDFFFFF	001F0000 – 001FFFFF	34	Environmental Flash (64kB0

Figure 12-8c: Interpretation diagram [12-4]

The physical address 0x1FC000000 is fixed for the Reset Exception Vector (the start of YAMON) on the MIPS32, regardless of how much physical memory is actually on the board—meaning the Flash chip on the board has to be integrated so that it correlates to this physical address. In the case of the Ampro Encore M3 Board, there are 2 MB of Flash memory on the board.

On this MIPS32 based board, after the hardware initialization sequence initializing the processor, the CPU begins executing the code located at the address within the PC register. In this case it is the YAMON bootloader code that has been ported to the Ampro Encore M3 Board. All of the hardware specific to the MIPS32 architecture (i.e.: initialization of interrupts) as well as specific to the board (i.e., serial, networking, etc.) is initialized via the YAMON program, which is proprietary software available from MIPS.

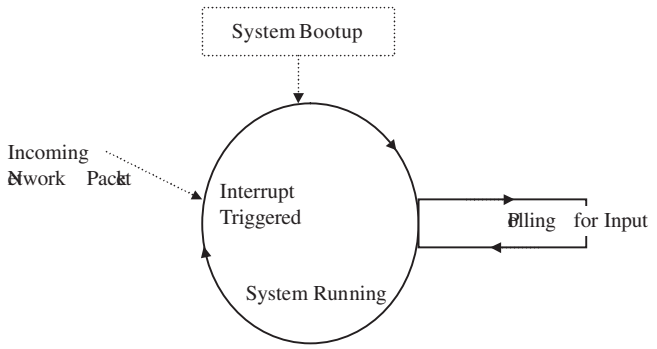
System Booting with an OS

Typically, 32-bit architectures include a more complex system software stack that includes an OS and, depending on the OS, may also include a BSP. Regardless of where the additional bootstrapping code comes from, if the system includes an OS, then it is the OS (with its BSP if there is one) that is initialized and loaded. While the boot sequence of a particular OS may vary, all architectures essentially execute the same steps when initializing and loading different embedded OSes.

For example, the boot sequence for a Linux kernel on an *x86 architecture* occurs via a BIOS that is responsible for searching for, loading and executing the Linux kernel, which is the central part of the Linux OS that controls all other programs. This is basically the "init" parent process that is started (executed) next. Code within the init process takes care of setting up the remainder of the system, such as forking tasks to manage networking/serial port, etc. The vxWorks RTOS boot sequence on most architectures, on the other hand, occurs via a vxWorks boot ROM that performs the architecture and board-specific initialization, and then starts the multitasking kernel with a user booting task as its only activity.

After the completion of the start-up sequence, an embedded system then typically enters an infinite loop, waiting for events that trigger interrupts, or actions triggered after some components are polled (see Figure 12-9 below).

Figure 12-9: System running



12.2 Quality Assurance and Testing of the Design

Among the goals of testing and assuring the quality of a system are finding bugs within a design and tracking whether the bugs are fixed. Quality assurance and testing is similar to debugging, discussed earlier in this chapter, except that the goals of debugging are to actually fix discovered bugs. Another main difference between debugging and testing the system is that debugging typically occurs when the developer encounters a problem in trying to complete a portion of the design, and then typically *tests-to-pass* the bug fix (meaning tests only to ensure the system minimally works under normal circumstances). With testing, on the other hand, bugs are discovered as a result of trying to break the system, including both testing-to-pass and *testing-to-fail,* where weaknesses in the system are probed.

Under testing, bugs usually stem from either the system not adhering to the architectural specifications—i.e., behaving in a way it shouldn't according to documentation, not behaving in a way it should according to the documentation, behaving in a way not mentioned in documentation—or the inability to test the system. The types of bugs encountered in testing depend on the type of testing being done. In general, testing techniques fall under one of four models: *static black box* testing, *static white box* testing, *dynamic black box* testing, or *dynamic white box* testing (see the matrix in Figure 12-9). Black box testing occurs with a tester that has no visibility into the internal workings of the system (no schematics, no source code, etc.). Black box testing is based on general product requirements documentation, as opposed to white box testing (also referred to *clear box* or *glass box* testing) in which the tester has access to source code, schematics, and so on. Static testing is done while the system is not running, whereas dynamic testing is done when the system is running.

	Black Box Testing	White Box Testing
Static Testing	Testing the product specifications by: 1. looking for high-level fundamental problems, oversights, omissions (i.e., pretending to be customer, research existing guidelines/standards, review and test similar software, etc.). 2. low-level specification testing by insuring completeness, accuracy, preciseness, consistency, relevance, feasibility, etc.	Process of methodically reviewing hardware and code for bugs without executing it.
Dynamic Testing	Requires definition of what software and hardware does, includes: • *data testing*, which is checking info of user inputs and outputs • *boundary condition testing*, which is testing situations at edge of planned operational limits of software • *internal boundary testing*, which is testing powers-of-two, ASCII table • *input testing*, which is testing null, invalid data • *state testing*, which is testing modes and transitions between modes software is in with state variables i.e., race conditions, repetition testing (main reason is to discover memory leaks), stress (starving software = low memory, slow cpu slow network), load (feed software = connect many peripherals, process large amount of data, web server have many clients accessing it, etc.), and so on.	Testing running system while looking at code, schematics, etc. Directly testing low-level and high-level based on detailed operational knowledge, accessing variables and memory dumps. Looking for data reference errors, data declaration errors, computation errors, comparison errors, control flow errors, subroutine parameter errors, I/O errors, etc.

Figure 12-10: Testing model matrix [12-5]

Within each of the models (shown in Figure 12-10), testing can be further broken down to include *unit/module testing* (incremental testing of individual elements within the system), *compatibility testing* (testing that the element doesn't cause problems with other elements in the system), *integration testing* (incremental testing of integrated elements), *system testing* (testing the entire embedded system with all elements integrated), *regression testing* (rerunning previously passed tests after system modification), and *manufacturing testing* (testing to ensure that manufacturing of system didn't introduce bugs), just to name a few.

From these types of tests, an effective set of test cases can be derived that verify that an element and/or system meets the architectural specifications, as well as validate that the element and/or system meets the actual requirements, which may or may not have been reflected correctly or at all in the documentation. Once the test cases have been completed and the tests are run, how the results are handled can vary depending on the organization, but typically vary between *informal*, where information is exchanged without any specific process being followed, and *formal* design reviews, or peer reviews where fellow developers exchange elements to test, walkthroughs where the responsible engineer formally walks through the schematics and source code, inspections where someone other than the responsible engineer does the walk through, and so on. Specific testing methodologies and templates for test cases, as well as the entire testing process, have been defined in several popular industry quality assurance and testing standards, including ISO9000 Quality Assurance standards, Capability Maturity Model (CMM), and the ANSI/IEEE 829 Preparation, Running, and Completion of Testing standards.

Finally, as with debugging, there are a wide variety of automation and testing tools and techniques that can aid in the speed, efficiency, and accuracy of testing various elements. These include load tools, stress tools, interference injectors, noise generators, analysis tools, macro recording and playback, and programmed macro, including tools listed in Table 12-1.

Real-World Advice

The Potential Legal Ramifications (in the United States) of NOT Testing

The US laws of product liabilities are considered very strict, and it is recommended that those responsible for the quality assurance and testing of systems receive training in products liability law in order to recognize when to use the law to ensure that a critical bug is fixed, and when to recognize that a bug could pose a serious legal liability for the organization.

The general areas of law under which a consumer can sue for product problems are:

- Breach of Contract (i.e., if bug fixes stated in contract are not forthcoming in timely manner)
- Breach of Warranty and Implied Warranty (i.e., delivering system without promised features)
- Strict and Negligence liability for personal injury or damage to property (i.e., bug causes injury or death to user)
- Malpractice (i.e., customer purchases defective product)
- Misrepresentation and Fraud (i.e., product released and sold that doesn't meet advertised claims, whether intentionally or unintentionally)

Remember, these laws apply whether your "product" is embedded consulting services, embedded tools, an actual embedded device, or software/hardware that can be integrated into a device.

—Based on the chapter "Legal Consequences of Defective Software" by Cem Kaner —Testing Computer Software. 1999

12.3 Conclusion: Maintaining the Embedded System and Beyond

This chapter introduced some key requirements behind implementing an embedded system design, such as understanding utility, translation, and debugging development tools. These tools include IDE and CAD tools, as well as interpreters, compilers, and linkers. A wide range of debugging tools useful for both debugging and testing embedded designs were discussed, from hardware ICEs, ROM emulators, and oscilloscopes to software debuggers, profilers, and monitors, just to name a few. This chapter also discussed what can be expected when booting up a new board, providing a few real-world examples of system bootcode.

Finally, even after an embedded device has been deployed, there are responsibilities that typically need to be met, such as user training, technical support, providing technical updates, bug fixes, and so on. In the case of user training, for example, architecture documentation can be leveraged relatively quickly as a basis for technical, user, and training manuals. Architecture documentation can also be used to assess the impact involved in introducing updates (i.e., new features, bug fixes, etc.) to the product while it is in the field, mitigating the risks of costly recalls or crashes, or on-site visits by FAEs that could be required at the customer site. Contrary to popular belief, the responsibilities of the engineering team last throughout the lifecycle of the device, and do **not** end when the embedded system has been deployed to the field.

To ensure success in embedded systems design, it is important to be familiar with the phases of designing embedded systems, especially the importance of first creating an architecture. This requires that all engineers and programmers, regardless of their specific responsibilities and tasks, have a strong technical foundation by understanding at the systems level all the major components that can go into any embedded system's design. This means that hardware engineers understand the software, and software engineers understand the hardware at the systems level, at the very least. It is also important that the responsible designers adopt, or come up with, an agreed-upon methodology to implement and test the system, and then have the discipline to follow through on the required processes.

It is the hope of the author that you appreciated the architectural approach of this book, and found it a useful tool as a comprehensive introduction to the world of embedded systems design. There are unique requirements and constraints related to designing an embedded system, such as those dictated by cost and performance. Creating an architecture addresses these requirements very early in a project, allowing a design team to mitigate risks. For this reason alone, the architecture of an embedded device will continue to be one of the most critical elements of any embedded system project.

Chapter 12 Problems

1. What is the difference between a host and a target?

2. What high-level categories do development tools typically fall under?

3. [T/F] An IDE is used on the target to interface with the host system.

4. What is CAD?

5. In addition to CAD, what other techniques are used to design complex circuits?

6. [a] What is a preprocessor?
   [b] Provide a real-world example of how a preprocessor is used in relation to a programming language.

7. [T/F] A compiler can reside on a host or a target, depending on the language.

8. What are some features that differentiate compiling needs in embedded systems versus in other types of computer systems?

9. [a] What is an object file?
   [b] What is the difference between a loader and a linker?

10. [a] What is an interpreter?
    [b] Name three real-world languages that require an interpreter.

11. An interpreter resides on:
    A. the host.
    B. the target and the host.
    C. in an IDE.
    D. A and C Only.
    E. None of the above.

12. [a] What is debugging?
    [b] What are the main types of debugging tools?
    [c] List and describe four real-world examples of each type of debugging tool.

13. What are five of the cheapest techniques to use in debugging?

14. Boot code is
    A. Hardware that powers on the board.
    B. Software that shuts down the board.
    C. Software that starts-up the board.
    D. All of the above.
    E. None of the above.

15. What is the difference between debugging and testing?

16. [a] List and define the four models under which testing techniques fall.
    [b] Within each of these models, what are five types of testing that can occur?

17. [T/F] Testing-to-pass is testing to insure that system minimally works under normal circumstances.

18. What is the difference between testing-to-pass and testing-to-fail?

19. Name and describe four general areas of law under which a customer can sue for product problems.

20. [T/F] Once the embedded system enters the manufacturing process, the design and development team's job is done.