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 wit hin 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

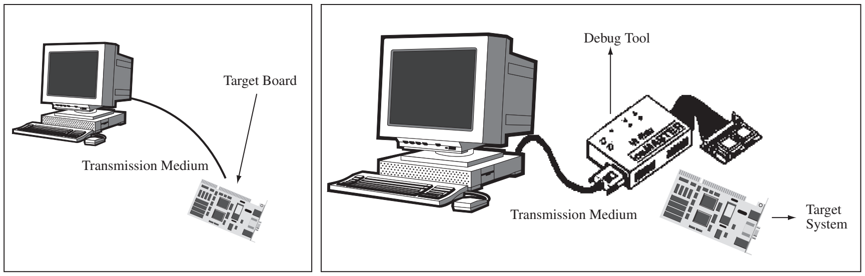


Figure 12-1: Development environments

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

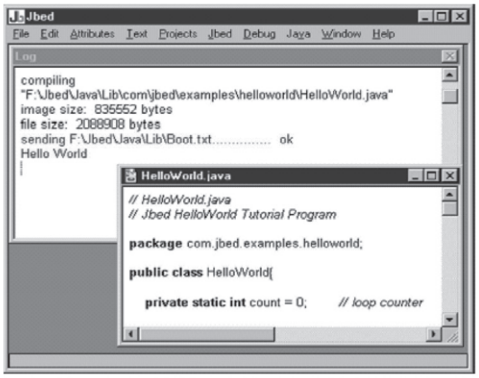
The embedded tools market is a small, fragmented market, with many different vendors supporting some subset of the available embedded CPUs, operating systems, JVMs, and so on. No matter how large the vendor, there is not yet a "one-stop-shop" where all tools for most of the same type of components can 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.).


Jbed
File Edit Attributes Text Projects Jbed Debug Java Window Help
Log
HelloWorld.java

```
compiling
"F:\Jbed\Java\Lib\com\jbed\examples\helloworld\HelloWorld.java"
image size: 835552 bytes
file size: 2088908 bytes
sending F:\Jbed\Java\Lib\Boot.txt............... ok
Hello World
```

```
// HelloWorld.java
// Jbed HelloWorld Tutorial Program

package com.jbed.examples.helloworld;

public class HelloWorld{

    private static int count = 0;       // loop counter
```

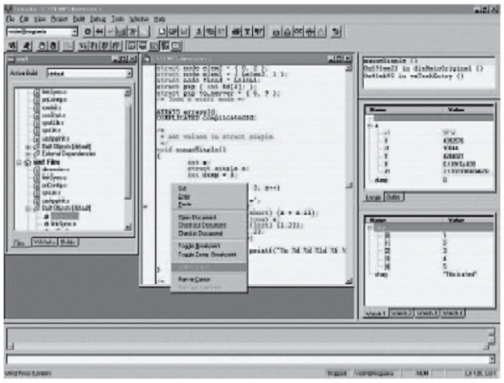
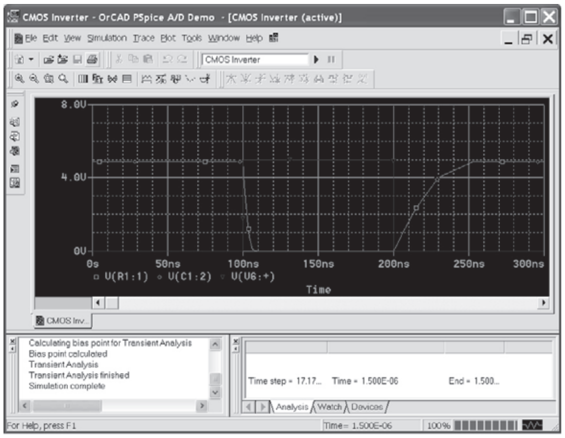


Figure 12-2: IDEs [12-1]

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.



```
Calculating bias point for Transient Analysis
Bias point calculated
Transient Analysis
Transient Analysis finished
Simulation complete
```

```
Time step = 17.17...     Time = 1.500E-06       End = 1.500...
```

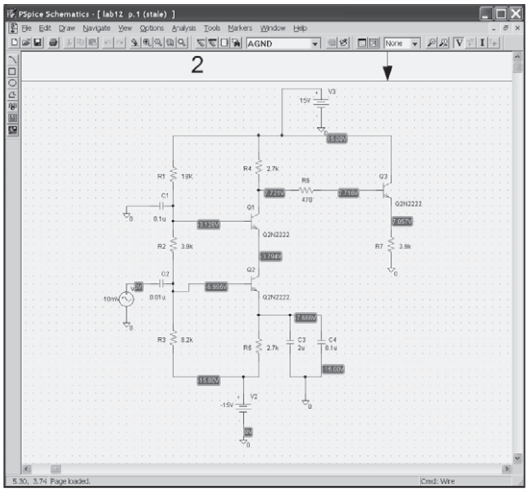


Figure 12-3a: Pspice CAD simulation sample [12-2]

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).

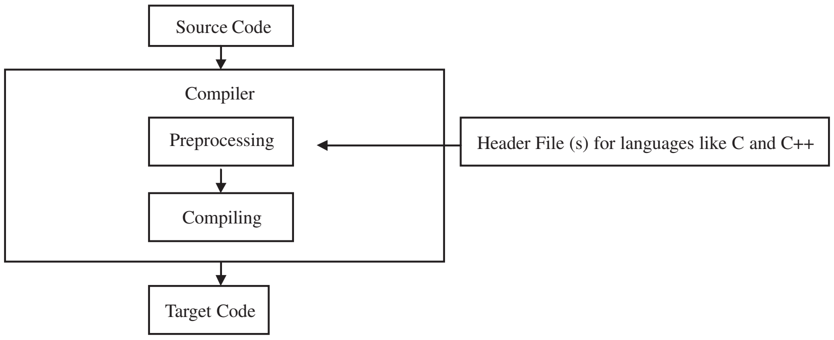


Figure 12-4: Compilation diagram

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***.

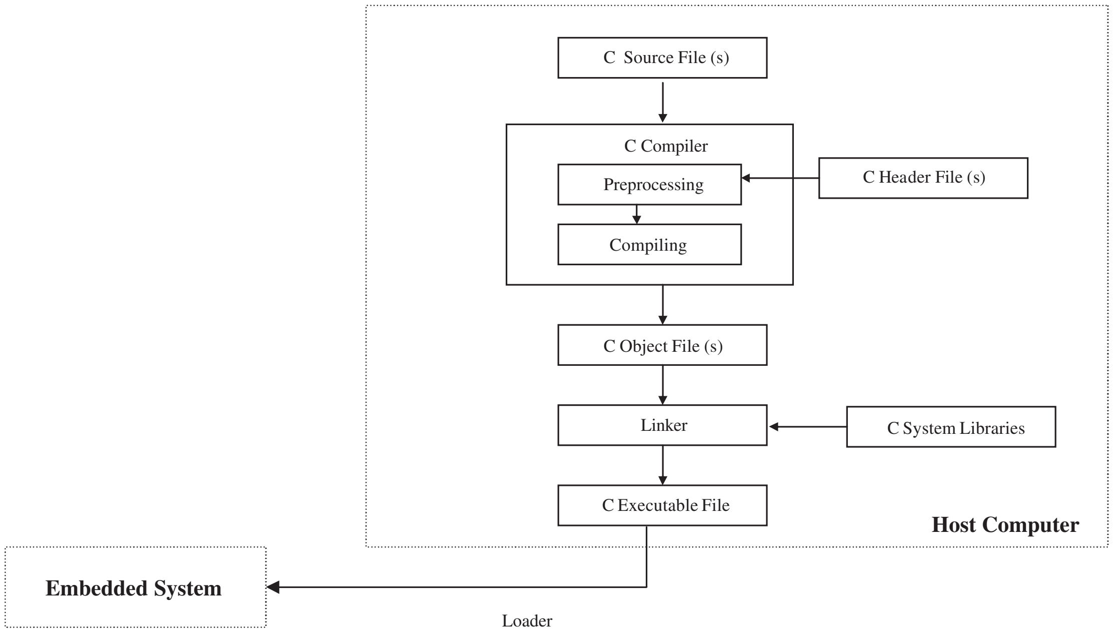


Figure 12-5: C example compilation/linking steps and object file results

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).

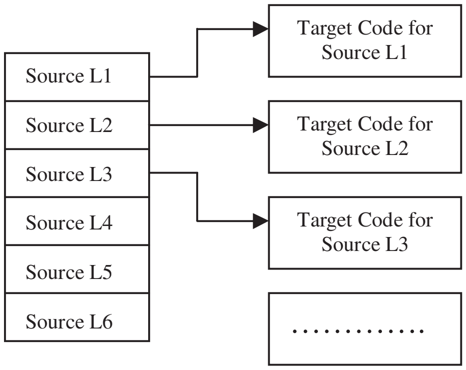

Source L1
Source L2
Source L3
Source L4
Source L5
Source L6
Target Code for Source L1
Target Code for Source L2
Target Code for Source L3
....

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 micro-processor 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 break-points, 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 (un-like 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.

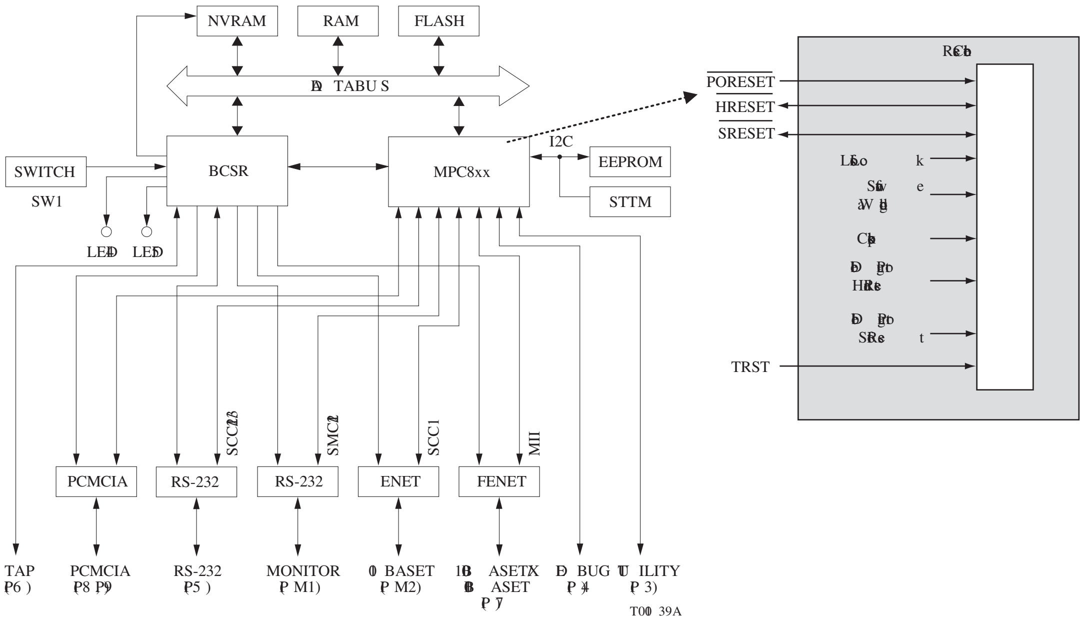


Figure 12-7a: Interpretation diagram [12-3]

Copyright of Freescale Semiconductor, Inc. 2004. Used by permission.

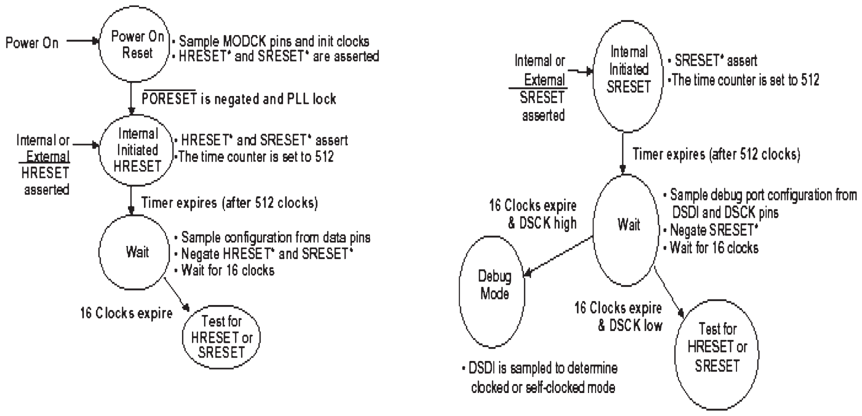

Power On
Power On Reset
• Sample MODCK pins and init clocks
• HRESET* and SRESET* are asserted
PORESET is negated and PLL lock
Internal or External HRESET asserted
Internal Initiated HRESET
• HRESET* and SRESET* assert
•The time counter is set to 512
Timer expires (after 512 clocks)
Wait
• Sample configuration from data pins
• Negate HRESET* and SRESET*
• Wait for 16 clocks
16 Clocks expire
Test for HRESET or SRESET
Internal or External SRESET asserted
Internal Initiated SRESET
• SRESET* assert
•The time counter is set to 512
Timer expires (after 512 clocks)
16 Clocks expire & DSCK high
Wait
• Sample debug port configuration from DSDI and DSCK pins
• Negate SRESET*
• Wait for 16 clocks
Debug Mode
16 Clocks expire & DSCK low
Test for HRESET or SRESET
• DSDI is sampled to determine clocked or self-clocked mode

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 0×00F00000	immr = 00F0xxxx	D7 = 0, D8 = 1	
DPR at 0×FF000000	immr = FF00xxxx	D7 = 1, D8 = 0	
DPR at 0×FFF00000	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.

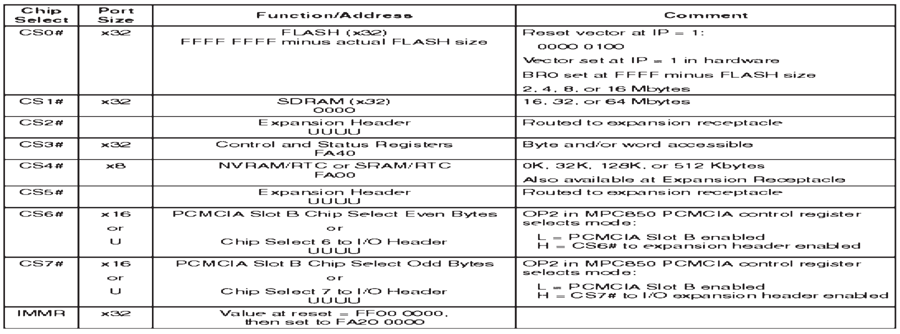
D9 and D10 select the configuration for the debug pins.

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.



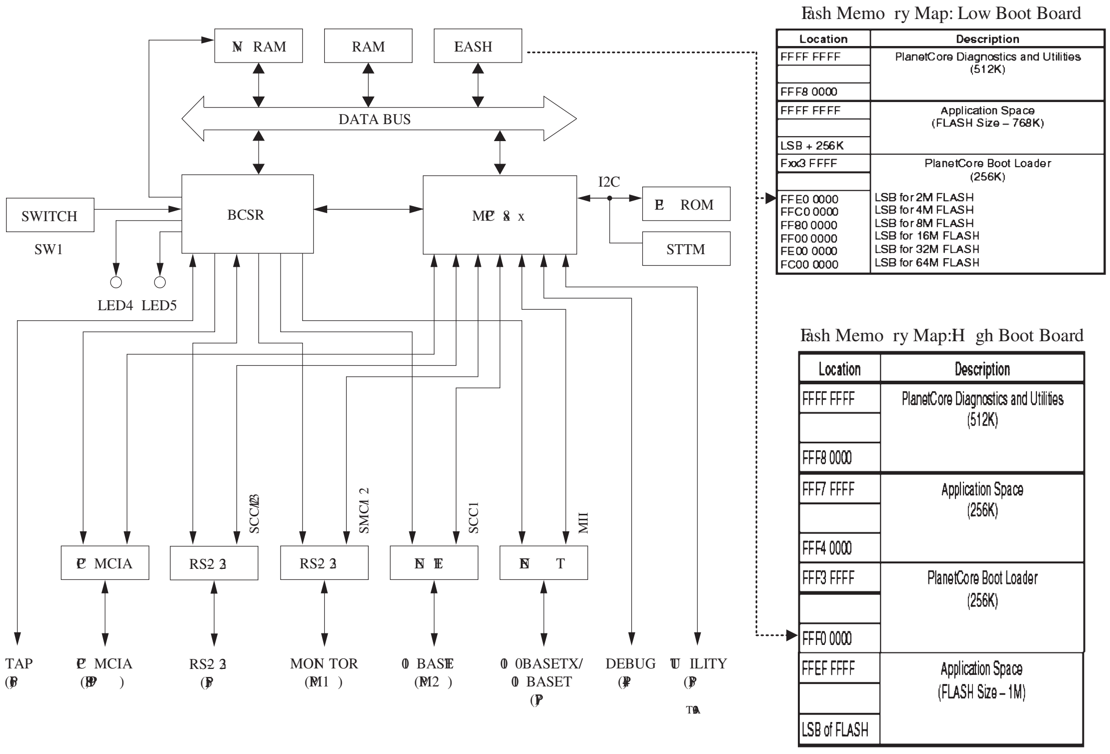
Chip Select	Port Size	Function/Address	Comment
CS0#	x32	FLASH (x32) FFFF FFFF minus actual FLASH size	Reset vector at IP = 1: 0000 0100 Vector set at IP = 1 in hardware BR0 set at FFFF minus FLASH size 2, 4, 8, or 16 Mbytes
CS1#	x32	SDRAM (x32) 0000	16, 32, or 64 Mbytes
CS2#		Expansion Header UUUU	Routed 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.



Fash Memo ry Map: Low Boot Board

Location	Description
FFFF FFFF	PlanetCore Diagnostics and Utilities (512K)
FFF8 0000	
FFFF FFFF	Application Space (FLASH Size – 768K)
LSB + 256K	
Fxx3 FFFF	PlanetCore Boot Loader (256K)
FFE0 0000	LSB for 2M FLASH
FFC0 0000	LSB for 4M FLASH
FF80 0000	LSB for 8M FLASH
FF00 0000	LSB for 16M FLASH
FE00 0000	LSB for 32M FLASH
FC00 0000	LSB for 64M FLASH

Fash Memo ry Map:H gh Boot Board

Location	Description
FFFF FFFF	PlanetCore Diagnostics and Utilities (512K)
FFF8 0000	
FFF7 FFFF	Application Space (256K)
FFF4 0000	
FFF3 FFFF	PlanetCore Boot Loader (256K)
FFF0 0000	
FFEF FFFF	Application Space (FLASH Size – 1M)
LSB of FLASH	

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

Copyright of Freescale Semiconductor, Inc. 2004. Used by permission.

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.

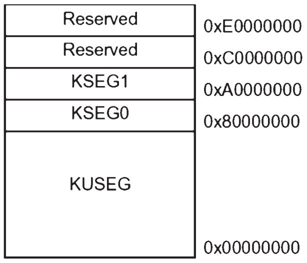
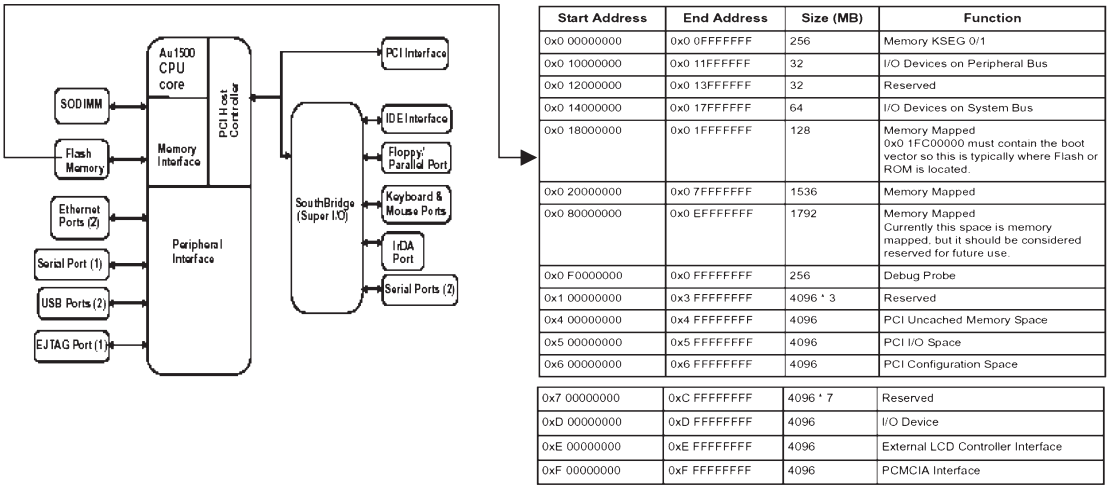


Figure 12-8a: Interpretation diagram [12-4]

This means virtual addresses (KSEG0) 0x80000000 (a cacheable view of physical memory) and (KSEG1) 0xA0000000 (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.



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.

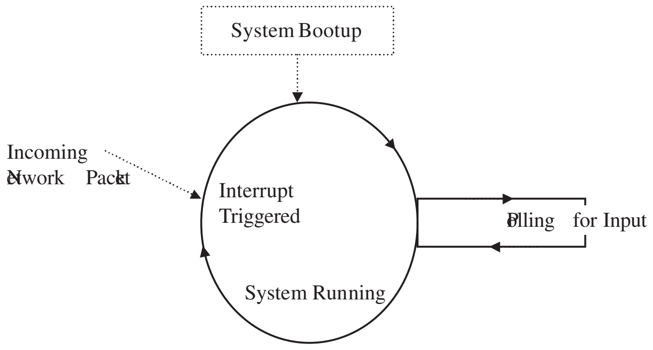
System Booting with an OS

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, sub-routine 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.

Section IV: Endnotes

Chapter 11: *Endnotes*

[11-1] The Embedded Systems Design and Development Lifecycle Model is specifically derived from the SEI's Evolutionary Delivery Lifecycle Model and the Software Development Stages Model.

[11-2] Based on the software architectural brainchildren of the Software Engineering Institute (SEI), read "Software Architecture in Practice." Bass, Clements, and Kazman. 2003 or go to http://www.sei.cmu.edu/ for more information. Whitepapers: "A Survey on Software Architecture Analysis Methods," Dobrica and Niemela. IEEE Transactions on Software Engineering, Vol. 28, No. 7, July 2002. "The 4+1 View Model of Architecture," Kruchten, IEEE Software, 1995."

[11-3] http://mini.net/cetus/oo_uml.html#oo_uml_examples.

Chapter 12: *Endnotes*

[12-1] *The Electrical Engineering Handbook*, Dorf, Chapter 27.

[12-2] "Short Tutorial on Using PSPice," Bill Rison, http://www.ee.nmt.edu/~rison/ee321_fall02/Tutorial.html

[12-3] Embedded Planet RPXLite Board Documentation and Freescale's PowerPC MPC823 User's Manual.

[12-4] Ampro Encore M3 Au150 Documentation.

[12-5] "Software Testing," Ron Patton, 2001.