Zynq-7000 All Programmable SoC Software Developers Guide

UG821 (v5.0) June 19, 2013

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/20/13	v4.0	Added Hardware Design Tools. Added notes about early access on IP integrator. Added RSA_SUPPORT. Added FSBL Multiboot. Expanded the definition. Added FSBL Hooks. Added DDR ECC Enable Added Secure Boot Support. Added Authentication Certificate. Added Table 3-1 and Table 3-2. Changed: Figure 3-14, Boot Image Format. Changed Figure 3-15. Changes and additions throughout Boot Image Creation. Added reference to Zynq and removed PPC processor in U-Boot Moved all Bootgen and BIF file description to Appendix A, Using Bootgen.
06/19/13	v5.0	Added Figure 3-2, page 28. Added additional Fallback information in FSBL Fallback Feature, page 32. Added eMMC Flash Devices, page 30. Added an additional compilation flag to Setting FSBL Compilation Flags, page 31. Added NAND Boot Mode, page 41, Added QSPI Boot Mode, page 41. Modified the information in Table 3-1, page 46, and Table 3-2, page 46.. Added Appendix B, eFUSE for Zynq-7000 AP SoC Devices. Removed Appendix B; eFUSE information - this content moved to OS and Libraries document collection. Added references to the eFUSE library. Added references to the Zynq-7000 quick take videos. Added additional BIF options to the table of BIF File Attributes, page 52. Removed Bootgen Command options -intstyle, -split. Added <filename> to UDP option. Changed verbiage and figures in FSBL Fallback Feature, page 32.

Appendix B: Additional Resources

XILINX®

Chapter 1

Introduction

1.1 Overview

This document summarizes the software-centric information required for designing with Xilinx® Zynq®-7000 All Programmable SoC devices. It assumes that you are:

- Experienced with embedded software design
- Familiar with ARM® development tools
- Familiar with Xilinx FPGA devices, intellectual property (IP), development tools, and tool environments

1.2 Introduction

This document contains the following content:

Architectural Decisions, page 4, describes the necessary architectural decisions that you must make prior to starting an All Programmable SoC design.

Operating System (OS) Considerations, page 6 provides a brief description of a "bare-metal" software system (no operating system), the Linux operating system, and real-time operating systems.

The addition of hardware programmability to the hardware and software interface imposes new requirements on design flows.

Certain hardware features are unique to Xilinx, such as hardware co-simulation and co-debug functionality that make it possible to verify custom logic implemented on Zynq-7000 AP SoC devices or in a logic simulation environment while applications execute on a Zynq-7000 AP SoC processor on a physical board or an emulator. See Software Tools Overview, page 8.

Chapter 2, Software Application Development Flows, describes software application development, beginning with an overview of the Xilinx-provided tools for developing and debugging applications for Zynq-7000 AP SoC devices. The chapter also provides the typical steps to develop bare-metal applications (using the Xilinx SDK tool), and lists the typical steps to develop an embedded Linux application.

Chapter 3, Boot and Configuration, describes the boot process for Zynq-7000 AP SoC devices. It details the possible boot modes, then documents the boot stages. This chapter also covers how to create a boot image and how to program a flash device.

Chapter 4, Linux provides an overview of using Git and the Xilinx public Git server, a diagram of the Linux Kernel, and a description of U-Boot, and provides links for more information on these topics.

Appendix A, Using Bootgen, describes the Bootgen utility.

Appendix B, Additional Resources, lists all relevant documentation, and provides links to that documentation (where available).

For a step-by-step explanation on designing a Zynq-based Embedded System using EDK, see the *Zynq-7000 All Programmable SoC Concepts, Tools, and Techniques Guide (UG873)* [Ref 7] .

For a step-by-step explanation on designing a Zynq-based Embedded System using the Vivado® Design Suite see the *Vivado Design Suite Tutorial: Embedded Processor Hardware Design (UG940)* [Ref 12]. The *Vivado Design Suite User Guide: Embedded Hardware Design* (UG898) [Ref 11] describes the process of embedded hardware design.

IMPORTANT: *The Vivado IP integrator is the replacement for Xilinx Platform Studio (XPS) for embedded processor designs, including designs targeting Zynq-7000 devices and MicroBlaze processors. XPS only supports designs targeting MicroBlaze™ processors, not Zynq-7000 devices. Both IP integrator and XPS are available from the Vivado IDE.*

See http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/smarter-system.html for quick-take videos on the Zynq-7000 AP SoC devices.

1.3 Architectural Decisions

You must make several architectural decisions before beginning embedded development on applications to run on the Zynq-7000 AP SoC.

Because the Zynq-7000 AP SoC devices have dual-core ARM Cortex™-A9 processors, you must determine whether to use Asymmetric Multiprocessing (AMP) or Symmetric Multiprocessing (SMP).

The same decision must be made for all embedded software projects: which operating system(s) to use (if any). This introduction defines both AMP and SMP, and provides an assessment of the trade-offs and concerns with each method.

1.3.1 Multiprocessing Considerations

The following subsections describe the two multiprocessing considerations.

Asymmetric Multiprocessing

Asymmetric multiprocessing (AMP) is a processing model in which each processor in a multiple-processor system executes a different operating system image while sharing the same physical memory. Each image can be of the same operating system, but more typically, each image is a different operating system, complementing the other OS with different characteristics:

- A full-featured operating system, such as Linux, lets you connect to the outside world through networking and user interfaces.
- A smaller, light-weight operating system can be more efficient with respect to memory and real-time operations.

A typical example is running Linux as the primary operating system along with a smaller, light-weight operating system, such as FreeRTOS or a bare-metal system, which is described in Bare-Metal System, page 6, as the secondary operating system.

The division of system devices (such as the UART, timer-counter, and Ethernet) between the processors is a critical element in system design. In general:

- Most devices must be dedicated to their assigned processor.
- The interrupt controller is designed to be shared with multiple processors.
- One processor is designated as the interrupt controller master because it initializes the interrupt controller.

Communication between processors is a key element that allows both operating systems to be effective. It can be achieved in many different ways, including inter-processor interrupts, shared memory, and message passing.

Symmetric Multiprocessing

Symmetric multiprocessing (SMP) is a processing model in which each processor in a multiple-processor system executes a single operating system image. The scheduler of the operating system is responsible for scheduling processes on each processor.

This is an efficient processing model when the selected single operating system meets the system requirements. The operating system uses the processing power of multiple processors automatically and is consequently transparent to the end user. Programmers can:

- Specify a specific processor to execute a process
- Handle interrupts with any available processor
- Designate one processor as the master for system initialization and booting other processors

1.4 Operating System (OS) Considerations

1.4.1 Bare-Metal System

Bare-metal refers to a software system without an operating system. This software system typically does not need many features (such as networking) that are provided by an operating system. An operating system consumes some small amount of processor throughput and tends to be less deterministic than simple software systems. Some system designs might not allow the overhead and lack of determinism of an operating system. As processing speed has continued to increase for embedded processing, the overhead of an operating system has become mostly negligible in many system designs. Some designers choose not to use an operating system due to system complexity.

1.4.2 Operating System: Linux

Linux is an open-source operating system used in many embedded designs. It is available from many vendors as a distribution, or it can be built from the open-source repositories. Linux is not inherently a real-time operating system, but it has taken on more real-time characteristics.

It is a full-featured operating system that takes advantage of the Memory Management Unit (MMU) in the processor, and is consequently regarded as a protected operating system. Linux also provides SMP capabilities to take advantage of multiple processors.

1.4.3 Real-Time Operating System

Some system designers use a *Real-Time Operating System* (RTOS) from Xilinx third party partners.

An RTOS offers the deterministic and predictable responsiveness required by timing sensitive applications and systems.

For information on the latest third party tools, contact your nearest Xilinx office.

1.4.4 Zynq-7000 Operating Systems From Partners

You can choose you own favorite embedded solutions based on past experience, new standards, unique requirements, and legacy designs, as well as corporate agreements.

For a detailed list of operating systems supported on Zynq-7000 devices from Xilinx partners, see:

http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/ecosystem/index.htm

XILINX

Chapter 2

Software Application Development Flows

2.1 Introduction

The Zynq®-7000 All Programmable (AP) SoC software application development flows let you create software applications using a unified set of Xilinx® tools, and leverage a broad range of tools offered by third-party vendors for the ARM® Cortex™-A9 processors.

This chapter focuses on Xilinx tools and flows; however, the concepts are generally applicable to third party tools, and the Zynq-7000 AP SoC device solutions incorporate familiar components such as an Eclipse-based integrated development environment (IDE) and the GNU compiler toolchain.

This chapter also provides an overview of bare-metal and Linux software application development flows using Xilinx tools, which mirror support available for other Xilinx embedded processors, with differences as noted. This chapter also references boot, device configuration, and OS usage within the context of application development flows. Those topics are covered in-depth in other chapters and references to other material.

Figure 2-1, page 8 shows a block diagram of the Zynq-7000 AP SoC processor.

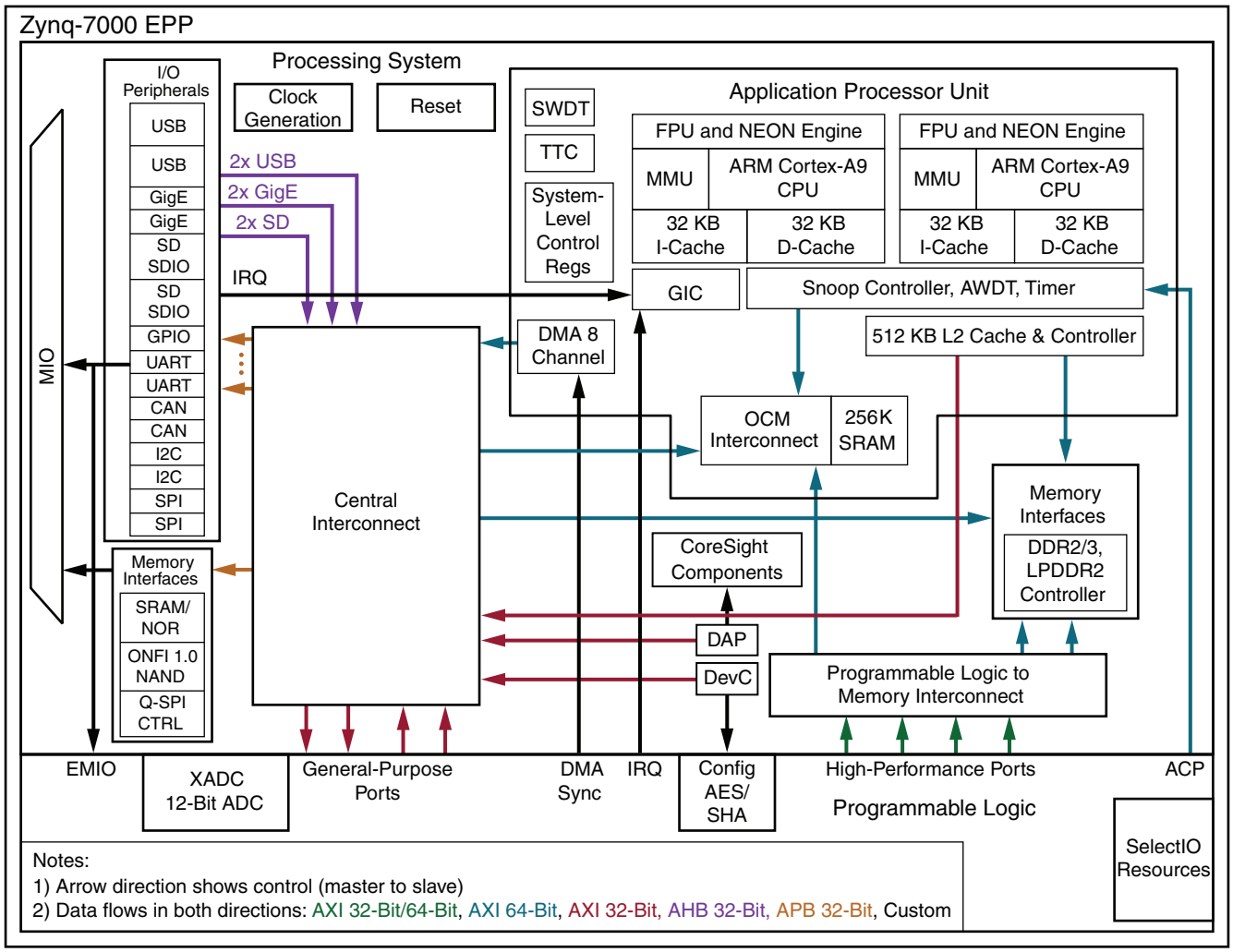

Figure 2-1: **Zynq-7000 AP SoC Processor System High-Level Diagram**

2.2 Software Tools Overview

The coupling of ARM-based Processing System (PS) and Programmable Logic (PL) creates unique opportunities to add custom peripherals and co-processors. Custom logic implemented in the PL can be used to accelerate time-critical software functions, reduce application latency, reduce system power, or provide solution-specific hardware features.

The addition of hardware programmability to the hardware and software interface imposes new requirements on design flows. Certain hardware features are unique to Xilinx, such as hardware co-simulation and co-debug functionality that make it possible to verify custom logic implemented on Zynq-7000 AP SoC devices or in a logic simulation environment while applications execute on a Zynq-7000 AP SoC processor on a physical board or an emulator.

Xilinx provides design tools for developing and debugging software applications for Zynq-7000 AP SoC devices, that include:

- Software IDE
- GNU-based compiler toolchain
- JTAG debugger
- Associated utilities

These tools let you develop both:

- Bare-metal applications that do not require an OS
- Applications for the open source Linux OS

Custom logic and user software can run various combinations of physical hardware or simulation, with the ability to monitor hardware events. For example:

- Custom logic running in hardware or in a simulation tool
- User software running on the target or in a software emulator
- PL and processor cross-triggering on events

Software solutions are also available from third-party sources that support Cortex-A9 processors, including, but not limited, to:

- Software IDEs
- Compiler toolchains
- Debug and trace tools
- Embedded OS and software libraries
- Simulators
- Models and virtual prototyping tools

Third party tool solutions vary in the level of integration and direct support for Zynq-7000 AP SoC devices. Xilinx does not provide tools that target Kernel development and debug, but those tools can be obtained from third party vendors.

The following subsections provide a summary of the available Xilinx development tools. Tools are available on 32- and 64-bit Windows and x86 Linux host computing platforms.

2.2.1 Hardware Configuration Tools

Xilinx provides hardware configuration tools, as follows:

- ISE® Design Suite Embedded Development Kit (EDK) Xilinx Platform Studio (XPS) which capture information about the PS and peripherals, including configuration settings, the register memory map, and the bitstream for PL initialization.

- Vivado IP integrator lets you use a block diagram to configure IP that is related to the PL and the Zynq-7000 processor.

IMPORTANT: *The Vivado IP integrator is the replacement for Xilinx Platform Studio (XPS) for embedded processor designs, including designs targeting Zynq-7000 devices and MicroBlaze processors. XPS only supports designs targeting MicroBlaze processors, not Zynq-7000 devices. Both IP integrator and XPS are available from the Vivado IDE.*

IMPORTANT: *When either XPS or the IP integrator tool is applicable, this document refers to both as the Xilinx hardware configuration tools.*

Xilinx Platform Studio

XPS captures hardware platform information in XML format along with other data files which are then used by software design tools to create and configure Board Support Package (BSP) libraries, infer compiler options, program the PL, define JTAG settings, and automate other operations that require information about the hardware.

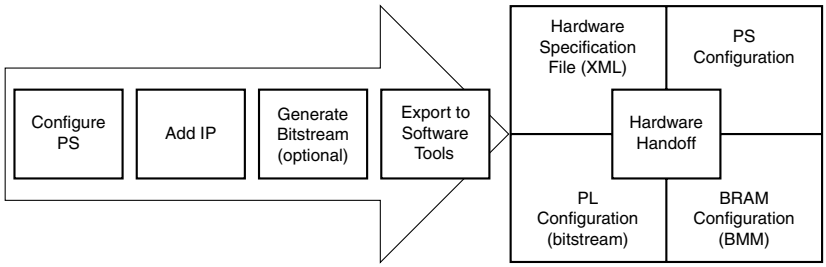

Figure 2-2: **XPS Hardware Handoff to Software Tools**

Vivado IP Integrator

The Vivado Design Suite IP integrator provides a block diagram for the Zynq-7000 AP SoC wherein you can set Programmable Logic (PL) information in an XML file, INIT files (`.h`, `.c`, and `.tcl`), which are then used by software design tools to create and configure Board Support Package (BSP) libraries, infer compiler options, define JTAG settings, and automate other operations that require information about the hardware.

For more information, see the following documents:

- *Vivado Design User Guide: Embedded Processor Hardware Design (UG898)* [Ref 11]
- *Vivado Design Suite Tutorial: Embedded Hardware Design* (UG940) [Ref 12]
- *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 14]
- *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893) [Ref 13]

2.2.2 Software Development Kit

The Xilinx Software Development Kit (SDK) provides a complete environment for creating software applications targeted for Xilinx embedded processors. It includes a GNU-based compiler toolchain (GCC compiler, GDB debugger, utilities, and libraries), JTAG debugger, flash programmer, drivers for Xilinx IPs and bare-metal board support packages, middleware libraries for application-specific functions, and an IDE for C/C++ bare-metal and Linux application development and debugging. Based upon the open source Eclipse platform, SDK incorporates the C/C++ Development Toolkit (CDT). Features include:

- C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic `makefile` generation
- Error navigation
- Integrated environment for debugging and profiling embedded targets
- Additional functionality available using third-party plug-ins, including source code version control

SDK Availability

SDK is available from any Xilinx design tool that can pass hardware information (the ISE® Design Suite installation package, the Xilinx Embedded Development Kit (EDK), and the Vivado IDE) or as a standalone application. SDK also includes an application template for creating a First Stage Bootloader (FSBL), as well as a graphical interface for building a boot image. SDK contains a complete help system that describes concepts, tasks, and reference information.

You can launch SDK from Vivado when you export a hardware definition, as shown in Figure 2-3.

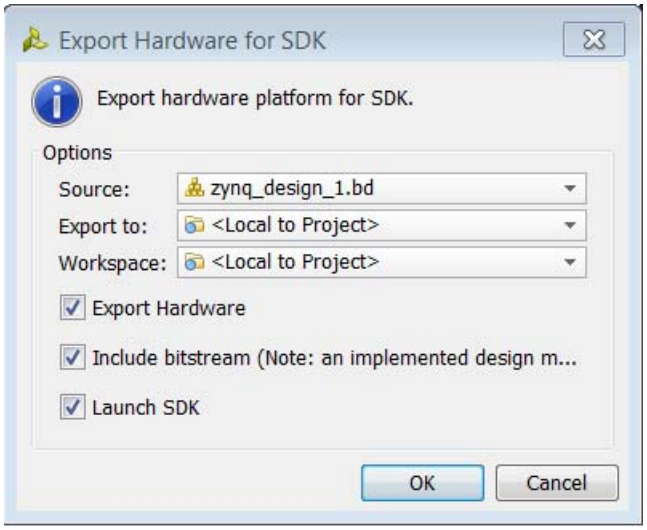

Figure 2-3: **Export Hardware for SDK Dialog Box**

2.2.3 Microprocessor Debugger

The Xilinx Microprocessor Debugger (XMD) is a JTAG debugger that can be invoked on the command line to download, debug, and verify programs. It includes a Tool Command Language (Tcl) interface that supports scripting for repetitive or complex tasks. XMD is not a source-level debugger, but serves as the GDB server for GDB and SDK when debugging bare-metal applications.

When debugging Linux applications, SDK interacts with a GDB server running on the target. Debuggers can connect to XMD running on the same host computer or on a remote computer on the network. For more information, see the *Embedded System Tools Reference Manual (UG111)* [Ref 16].

2.2.4 Sourcery CodeBench Lite Edition for Xilinx Cortex-A9 Compiler Toolchain

SDK includes the Sourcery CodeBench Lite Edition for Xilinx Cortex-A9 compiler toolchain for bare-metal Embedded Application Binary Interface (EABI) and Linux application development.

The Xilinx Sourcery CodeBench Lite toolchain in SDK contains the same GNU tools, libraries and documentation as the standard Sourcery CodeBench Lite Edition EABI and Linux compiler toolchains, but adds the following enhancements:

- Default toolchain settings for the Xilinx Cortex-A9 processors
- Bare-metal (EABI) start up support and default linker scripts for the Xilinx Cortex-A9 processors

- Vector Floating Point (VFP) and NEON™ optimized libraries

2.2.5 Analysis Tools

ChipScope Pro Analyzer

The ChipScope™ Pro analyzer inserts logic analyzer, system analyzer, and virtual I/O cores into custom logic in your PL design, so you can view any internal signal or node.

Signals are captured at the speed of operation and you can display and analyze those signals using the ChipScope debugging tool. Events (changes) in the custom logic can trigger breakpoints in the software debugger, halting a program running the processor, and vice versa. Many co-debugging flows are supported using SDK, ChipScope Pro Analyzer, and other Xilinx tools. See the *ChipScope Pro Software and Cores User Guide* (UG029) for more information about debugging an ISE Design Suite project.

Vivado Lab Tool

The Vivado IDE has integrated debugging capability. See *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 17] for more information.

2.2.6 System Generator for DSP

The System Generator™ for DSP tool can be used to develop DSP and data flow-centric, hardware-based coprocessors, working within the MATLAB®/Simulink® environment.

System Generator supports rapid simulation of the DSP hardware, reducing overall development time, and automates the generation of co-processors that can be connected to the PS. The SDK co-debug feature lets you run and debug programs running on the processor in SDK, while retaining visibility and control over the hardware under development in System Generator.

2.2.7 ISim Simulator

ISE® Design Suite and PlanAhead™ tools contain the ISim HDL simulator that lets you verify and debug custom hardware logic implemented in the PL without requiring a physical board. Using the ISim Hardware Co-Simulation (HWCoSim) technology, you can debug programs running on the target processor in an emulator or on the target device in SDK concurrently with the custom logic in the ISE simulator.

2.3 Bare-Metal Device Driver Architecture

The bare-metal device drivers are designed with a layered architecture as shown in Figure 2-4, page 14. The layered architecture accommodates the many use cases of device drivers while at the same time providing portability across operating systems, toolsets, and processors.

The layered architecture provides seamless integration with:

- A Layer 2 (RTOS Adapter), an abstract device driver interface that is full-featured and portable across operating systems
- Processors Layer 1 (Device Driver)
- A direct hardware interface for simple use cases or those wishing to develop a custom device driver

The following subsections describe the layers.

IMPORTANT: The direct hardware interface does not add additional overhead to the device driver function call overhead, as it is typically implemented as a set of manifest constants and macros.

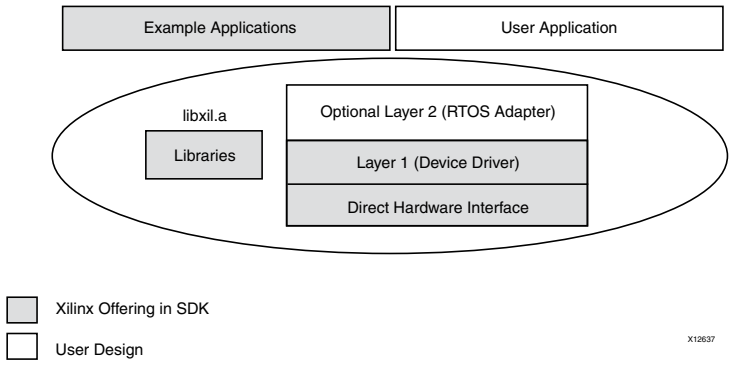

Figure 2-4: **Bare-Metal Drivers Architecture**

2.3.1 Layer 2 (RTOS Adapter)

Layer 2 is an adapter between an RTOS and a device driver. It converts a Layer 1 device driver to an interface that matches the requirements of the driver model for an RTOS. Unique adapters might be necessary for each RTOS.

Adapters typically:

- Communicate directly to the RTOS as well as the Layer 1 interface of the device driver
- Reference functions and identifiers specific to the RTOS. This layer is therefore not portable across operating systems
- Can use memory management
- Can use RTOS services such as threading and inter-task communication
- Can be simple or complex depending upon the RTOS interface and requirements for the device driver

2.3.2 Layer 1 (Device Driver)

Layer 1 is an abstract device driver interface that shields you from potential changes to the underlying hardware. It is implemented with macros and functions and designed to allow you to use all features of a device. The device driver is independent of operating systems and processors, making it highly portable. This interface typically has:

- Consistent API that gives you an "out-of-the-box" solution. The abstract API helps isolate the your project from hardware changes.
- Lack of RTOS or processor dependencies makes the device driver highly portable
- Run-time error checking such as assertion of input arguments that provides the ability to compile away asserts
- Device feature support
- Support for device configuration parameters to handle FPGA-based parameterization of hardware devices
- Support for multiple instances of a device while managing unique characteristics on a per instance basis
- Polled and interrupt-driven I/O
- Non-blocking function calls to aid complex applications
- A potentially large memory footprint
- Buffer interfaces for data transfers as opposed to byte interfaces. This makes the API easier to use for complex applications.
- No direct communication to Layer 2 adapters or application software, by using asynchronous callbacks for upward communication

2.3.3 Direct Hardware Interface

The interface that is contained within the Layer 1 device driver is a direct hardware interface. It is typically implemented as macros and manifest constants, and is designed so you can create a small applications or create a custom device driver. This interface typically has:

- Constants that define the device register offsets and bit fields
- Simple macros that provide access to the hardware registers
- A small memory footprint
- Little to no error checking
- Minimal abstraction so the API typically matches the device registers. The API is therefore less isolated from hardware device changes.
- No support of device configuration parameters
- Support of multiple instances of a device with base address input to the API
- No, or minimal state
- Polled I/O only
- Blocking functions for simple use cases
- Byte interfaces typically provided

2.4 Bare-Metal Application Development

Xilinx software design tools facilitate the development of embedded software applications for many runtime environments.

Xilinx embedded design tools create a set of hardware platform data files that include:

- An XML-based hardware description file describing processors, peripherals, memory maps, and additional system data
- A bitstream file containing optional Programmable Logic (PL) programming data
- A block RAM Memory Map (BMM) file
- PS configuration data used by the Zynq-7000 AP SoC First Stage Bootloader (FSBL).

The bare-metal Board Support Package (BSP) is a collection of libraries and drivers that form the lowest layer of your application.

The runtime environment is a simple, semi-hosted and single-threaded environment that provides basic features, including boot code, cache functions, exception handling, basic file I/O, C library support for memory allocation and other calls, processor hardware access macros, timer functions, and other functions required to support bare-metal applications.

Using the hardware platform data and bare-metal BSP, you can develop, debug, and deploy bare-metal applications using SDK.

Figure 2-5 is an overview flowchart for bare-metal application development.

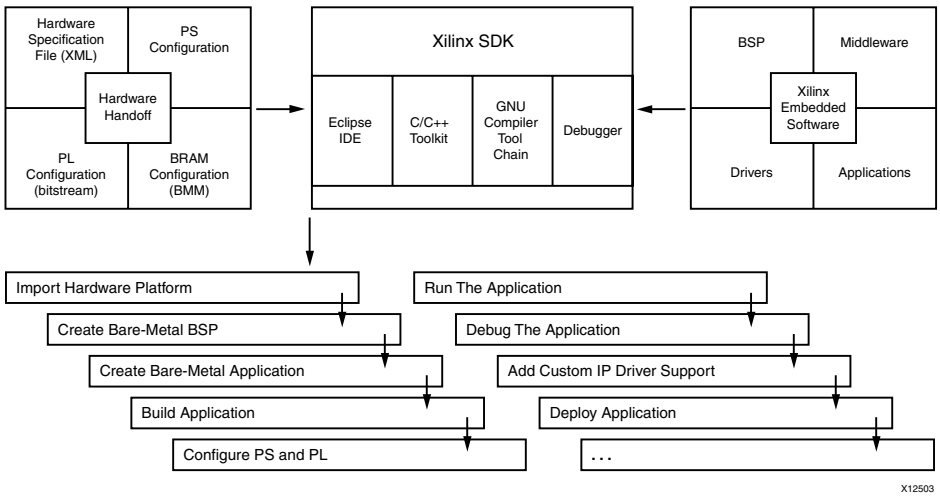

Figure 2-5: **Bare-Metal Application Development Overview**

To develop bare-metal applications using SDK, typical steps include:

1. Importing Hardware Platform Information
2. Creating Bare-Metal BSP
3. Creating Bare-Metal Application
4. Building the Application Project
5. Programming the Device and Running the Application
6. Debugging the Application
7. Adding Custom IP Driver Support
8. Deploying the Application

The following subsections summarize these SDK development flow steps. See the SDK online help, and the *Zynq EDK Concept, Tools, and Techniques Guide (UG873)* [Ref 7] for more details and examples of SDK tool usage.

2.4.1 Importing Hardware Platform Information

Xilinx hardware configuration tools create hardware platform data you can export to SDK to create a hardware platform project. In SDK, the project stores information about the hardware system that includes, but is not limited to, the following:

- Processor and peripheral information for BSP generation
- Memory map information used to generate linker scripts
- Bitstream data used to program the PL with custom logic
- PS configuration data used in the FSBL and the debugger

2.4.2 Creating Bare-Metal BSP

After you create the hardware platform project, you can use SDK to create a bare-metal BSP project. Source files for drivers and libraries are staged, parameterized based on the hardware platform (processor, IP feature set, hardware configuration settings) to create header file parameter definitions, and compiled. The BSP reflects IP enabled in the PS, including Multiplexed I/O (MIO) configuration, and custom logic in the PL. You can modify and re-generate BSP settings. See the *Standalone BSP (UG652)* [Ref 7], which is included in the *OS and Libraries Document Collection (UG643)* [Ref 8].

2.4.3 Creating Bare-Metal BSP Using Third-Party Tools

SDK supports BSP generation for other embedded OS environments and tools by specifying the path to a software repository containing source and meta data files that enable it to configure and build the associated drivers and libraries.

2.4.4 Creating Bare-Metal Application

SDK provides a template-based application generator for included sample programs, from a basic "Hello World" or Dhrystone benchmark application to a FSBL or TCP/IP echo server. A default linker script is created for these applications.

The application generator is invoked by the Xilinx C or C++ Application wizard. You can either create an empty application or import existing applications to port to the bare-metal BSP. Each application project is associated with a BSP project.

Code development tools include editors, search, refactoring, and features available in the base Eclipse platform and CDT plug-in.

2.4.5 Building the Application Project

SDK application projects can be user-managed (user-created `makefiles`) or automatically managed (SDK-created `makefiles`). For user-managed projects, you must maintain the `makefile` and initiate the application builds.

For automatically managed projects, SDK updates the `makefile` as needed when source files are added or removed, source files are compiled when changes are saved and the ELF is built automatically; in Eclipse CDT terminology, the application project is a managed make project.

Where possible, SDK infers or sets default build options based on the hardware platform and BSP used, including compiler, linker, and library path options.

2.4.6 Programming the Device and Running the Application

After building the bare-metal application, SDK can be used to configure the PS, program the PL and run the application. SDK configures the PS using the System-Level Configuration Registers (SLCR) with configuration data also used in the FSBL.

Bitstream (BIT) and block memory map (BMM) data are downloaded to the Zynq-7000 AP SoC to load any custom design logic into the PL, but this step can be omitted when running applications that require only the PS.

Create an SDK configuration run to download and run the application ELF file. A terminal view is available to interact with the application using `STDIN` and `STDOUT`.

2.4.7 Debugging the Application

When you use SDK to debug applications, the steps are similar to those for running an application, except you create a debug configuration instead of a run configuration. A collection of windows (views) provides a complete debugging environment. This debug perspective should be familiar to those who have used Eclipse-based IDEs with the CDT plug-in, and includes a debug window showing the state of the session with a call stack, source viewers, disassembly, memory, register, other views, and console. You can set breakpoints and control execution with familiar debugger commands.

2.4.8 Adding Custom IP Driver Support

The hardware platform data created by Xilinx hardware configuration tools captures the Xilinx IP blocks used in the PL area, and the bare-metal BSP automatically includes driver support for these blocks. Custom IP blocks that include hardware description metadata files can also be captured as part of the hardware platform passed to SDK.

By specifying the path to a software repository containing custom drivers and metadata, SDK can also include them in the bare-metal BSP.

You can also create library projects to manage and build custom driver source files, and build their applications using library projects together with the bare-metal BSP.

As the Hardware Platform changes you might want to configure the Custom IP driver. To customize the software drivers, a Microprocessor Driver Definition (MDD) file along with a Tcl file is used.

The driver parameters to be configured are specified in the MDD file. The procedure to generate the `.h` or `.c` files is present in the Tcl file. For more information, see the *Platform Specification Format Reference Manual, (UG642)* [Ref 9].

2.4.9 Deploying the Application

After developing and debugging the bare-metal application within SDK, you can create a boot image for the application to be deployed on the board. SDK includes an application template for the FSBL that can be modified to create and build the final FSBL. The FSBL, bare-metal application, and bitstream for programming the PL (optional) are combined to generate a boot image that can be programmed to supported devices using the SDK Flash Writer.

For more information about boot image format, see Chapter 3, Boot and Configuration.

2.5 Linux Application Development

In addition to bare-metal applications, Xilinx software design tools facilitate the development of Linux user applications. This section provides an overview of the development flow for Linux application development.

Xilinx embedded design tools create a set of hardware platform data files that include:

- An XML-based hardware description file describing processors, peripherals, memory maps and additional system data
- A bitstream file containing PL programming data (optional)
- A block RAM Memory file (BMM)
- PS configuration data used by the Zynq-7000 AP SoC first stage bootloader (FSBL).

Linux is an open-source operating system. The Xilinx open source solution includes support for a single processor and Symmetric Multiprocessing (SMP). XIlinx provides drivers for the peripherals in the Processor System (PS). (You can add drivers for custom logic in the PL.)

See the *Standalone BSP (UG652)* that is included in the *OS and Libraries Document Collection (UG643)* [Ref 8] for information about the Bare-Metal BSP.

See Chapter 4, Linux for a description of the Linux the U-Boot bootloader, and for links to the Xilinx Open Source Wiki that provide more information.

Using the hardware platform data and Linux Kernel, programmers can develop, debug and deploy Linux user applications with the Xilinx Software Development Kit (SDK). SDK does not support Linux Kernel debug. Linux Kernel configuration and build processes are not discussed in this section.

To develop Linux user applications using SDK, typical steps include:

1. Booting Linux
2. Creating an Application Project
3. Building the Application
4. Running the Application
5. Debugging the Application
6. Adding Driver Support for Custom IP in the PL
7. Profiling the Application
8. Adding Application to Linux File System
9. Modifying the Linux BSP (Kernel or File System)

The flowchart in Figure 2-6 provides an overview of the flow for Linux application development.

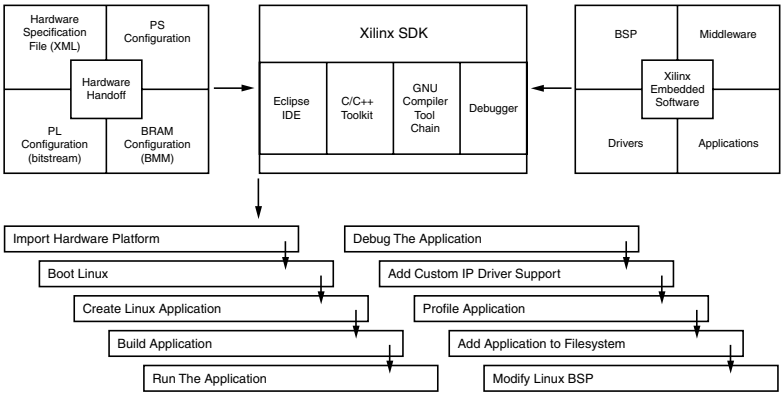

Figure 2-6: **Linux Application Development**

The following subsections describe the steps in this flow. See the *Zynq EDK Concept, Tools, and Techniques Guide (UG873)* [Ref 7], for more details and examples of SDK tool usage.

2.5.1 Booting Linux

You can boot Linux in multiple ways, depending on your preferred work flow:

- Program the boot image into flash and power up or reset the board
- Download and run the FSBL, followed by the U-Boot and then the Linux Kernel
- Use U-Boot to load and run images

With Linux running on the Zynq-7000 AP SoC, SDK can treat the AP platform as a remote Linux host, with functionality that varies depending on the components included in the file system.

Flash memory offsets differ for NAND, NOR and Quad-SPI. Partitions can include FSBL, U-boot, linux kernel, device tree, RAMdisk, and user application.

During the boot process, FSBL is run to set up the PS, followed by U-Boot, which can be used to load the Linux Kernel image and boot Linux. The actual boot sequence and flash image creation process vary depending on the type of flash and other requirements. For example, the FSBL can be used to configure the PL containing custom logic and it is possible for a U-Boot image to include the FSBL.

2.5.2 Creating an Application Project

SDK provides a template-based application generator for included sample programs, from a basic “Hello World” or Dhrystone bootloader, or an FSBL application to a benchmarking application. The application generator is invoked by the Xilinx C or C++ Application wizard.

Users can also create an empty application or import existing Linux applications for porting. Code development tools include editors, search, refactoring and features available in the base Eclipse platform and CDT plug-in.

SDK provides a utility (bootgen) to generate bootable images (.bin/.mcs). You need to provide all the images and the load addresses to the bootgen tool to create the boot image.

SDK also provides a utility to flash images onto the flash device.

2.5.3 Building the Application

SDK application projects can be user-managed (user-created `makefiles`) or automatically managed (SDK created `makefiles`). For user-managed projects, the user maintains the `makefile` and initiates application builds. For automatically managed projects, SDK updates the `makefile` as needed when source files are added or removed, source files are compiled when changes are saved and the ELF is built automatically; in Eclipse CDT terminology, the application project is a managed make project. Where possible, SDK infers or sets default build options based on the hardware platform and BSP used, including compiler, linker, and library path options.

2.5.4 Running the Application

You can create an SDK run configuration to copy the compiled application to the file system and run the application. With Linux running on the Zynq-7000 AP SoC, the run configuration copies the executable to the file system using `sftp` if the Linux environment includes SSH. A terminal view is available to interact with the application using `STDIN` and `STDOUT`.

You can also run the application using a command line shell. Use:

- `sftp` to copy the executable
- `ssh` in Linux to run the executable

2.5.5 Debugging the Application

You can use SDK to debug applications; SDK creates a debug configuration that defines options for the debugger session. A gdbserver runs the application on Linux, and the SDK debugger communicates with it using a TCP connection. A collection of windows (views) provides a complete debugging environment.

This debug perspective should be familiar if you have used Eclipse-based IDEs with the CDT plug-in, and it includes a debug window showing the state of the session with a call stack, source viewers, disassembly, memory, register and other views, and the console. You can set breakpoints and control execution with standard debugger commands.

2.5.6 Adding Driver Support for Custom IP in the PL

SDK supports Linux BSP generation for peripherals in the PS as well as custom IP in the PL. When generating a Linux BSP, SDK produces a device tree, which is a data structure describing the hardware system that is passed to the Kernel at boot time. Device drivers are available as part of the Kernel or as separate modules, and the device tree defines the set of hardware functions available and features enabled.

Additionally, you can add dynamic, loadable drivers. The Linux driver supports these drivers. See the *Zynq-7000 All Programmable Soc Concepts, Tool, and Tools Techniques User Guide (UG821)* [Ref 7].

Custom IP in the PL are highly configurable, and the device tree parameters define both the set of IP available in the system and the hardware features enabled in each IP.

See Chapter 4, Linux for additional details on the Linux Kernel and boot sequence.

2.5.7 Profiling the Application

To profile Linux user applications, use the `-pg` profiling option when building the application. User application profiling is based on the `gprof` utility and an accompanying viewer to display the call graph and other data.

For profiling all running code in the user application, the Kernel, interrupt handlers, and other modules, SDK includes an `OProfile` plug-in that supports visualization of its call profiling capabilities. `OProfile` is an open source system-wide profiler for Linux; it requires a Kernel driver and daemon to collect sample data.

2.5.8 Adding Application to Linux File System

The compiled user application and required shared libraries can be added to the Linux file system, as follows:

- While Linux is running on the Zynq-7000 AP SoC, you can copy the files using `sftp` if the Linux environment includes SSH.
- In SDK, a Remote System Explorer (RSE) plug-in lets you copy files using drag-and-drop.
- In workflows outside of SDK, add the application and libraries to the file system folder before creating the file system image and programming it to flash.

2.5.9 Modifying the Linux BSP (Kernel or File System)

See Chapter 4, Linux, for a description of the Linux U-Boot bootloader, references to the Xilinx Open Source Wiki [Ref 1] that provide more information.

2.6 Additional Information

For additional information related to topics mentioned in this chapter, consult the references listed in the introduction. For further reading, review the following the *Zynq-7000 All Programmable SoC Concept, Tools, and Techniques Guide, (UG873)* [Ref 7].

- "Embedded System Design Using the Zynq Processing System"
- "Adding IPs in Fabric to Zynq PS"

XILINX®

Chapter 3

Boot and Configuration

3.1 Overview

You can boot or configure Zynq®-7000 All Programmable SoC devices in secure mode using static memories only (JTAG disabled) or in non-secure mode using either JTAG or static memories.

- JTAG mode is primarily used for development and debug.
- NAND, parallel NOR, Serial NOR (Quad-SPI), and Secure Digital (SD) flash memories are used for booting the device. *The Zynq-7000 AP SoC Technical Reference Manual (UG585)* [Ref 6] provides the details of these boot modes.

The processor system boot is a two-stage process:

- An internal BootROM stores the stage-0 boot code, which configures one of the ARM® processors and the necessary peripherals to start fetching the First Stage Bootloader (FSBL) boot code from one of the boot devices. The programmable logic (PL) is not configured by the BootROM. The BootROM is not writable.
- The FSBL boot code is typically stored in one of the flash memories, or can be downloaded through JTAG. BootROM code copies the FSBL boot code from the chosen flash memory to On-Chip Memory (OCM). The size of the FSBL loaded into OCM is limited to 192 kilobyte. The full 256 kilobyte is available after the FSBL begins executing.
- Another boot mode supported through FSBL is eMMC boot mode. This boot mode is possible only when the primary boot mode (set through the boot mode pins) is QSPI. This is used when you have a small QSPI flash and would like to store all the other partitions on a larger flash memory like eMMC. In this case, place the FSBL on the QSPI flash, and all the other partitions on eMMC flash.

The FSBL boot code is completely under user control and is referred to as *user boot code*. This provides you with the flexibility to implement whatever boot code is required for your system.

Xilinx® provides sample FSBL boot code that you can tailor to your own needs. The FSBL boot code includes initialization code for the peripherals in the processing system (PS), see the FSBL code provided with SDK for details on the FSBL initialization sequence of the FSBL. The boot image can contain a bitstream for the programmable logic (PL).

The PL is not required to be configured at this stage, because the PS is fully operational when the PL is not configured. You can customize the FSBL boot code to use other PS peripherals such as Ethernet, USB, or STDIO to boot and/or configure the PL.

Note: *DDR and SCU are not enabled by the BootROM. See the Zynq-7000 AP SoC Technical Reference Manual (UG585) [Ref 6] for details.*

3.2 Boot Modes

The following boot modes are available:

- PS Master Non-secure Boot
- PS Master Secure Boot
- JTAG/PJTAG Boot

For details on these boot modes, see "Boot and Configuration" in the *Zynq-7000 AP SoC Technical Reference Manual (UG585)* [Ref 6].

3.3 Boot Stages

Zynq-7000 AP SoC devices support secure and non-secure boot processes, as follows:

- Stage-0 Boot (BootROM)
- First Stage Bootloader
- Second Stage Bootloader (Optional)

3.3.1 Stage-0 Boot (BootROM)

See the section on "BootROM" in the *Zynq-7000 AP SoC Technical Reference Manual (UG585)* [Ref 6].

Figure 3-1, page 27 shows the flow of FSBL loading in OCM by the BootROM code.

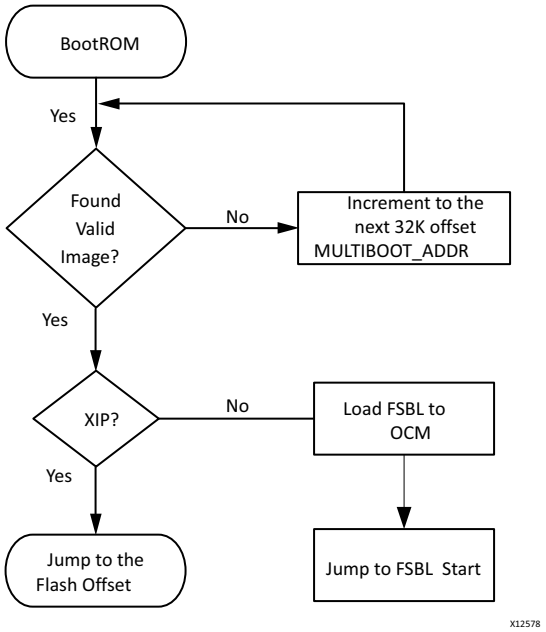

Figure 3-1: **Boot Flow**

FSBL Fallback Feature, page 32 contains more information on the BootROM flow when a valid image is not found.

3.3.2 First Stage Bootloader

The First Stage Bootloader (FSBL) starts after the boot. The BootROM loads FSBL into the OCM.

The FSBL is responsible for:

- Initializing with the PS configuration data that Xilinx hardware configuration tools provide (see Zynq PS Configuration, page 43)
- Programming the PL using a bitstream (if provided)
- Loading second stage bootloader or bare-metal application code into DDR memory
- Handoff to the second stage bootloader or bare-metal application

Note: *Before handoff to the second stage bootloader or bare-metal application, the FSBL invalidates the instruction cache and disables the cache and MMU, because U-Boot assumes it is disabled upon start.*

See the FSBL code provided with SDK for details on the initialization sequence of the FSBL.

Figure 3-2, page 28 shows an example FSBL flow.

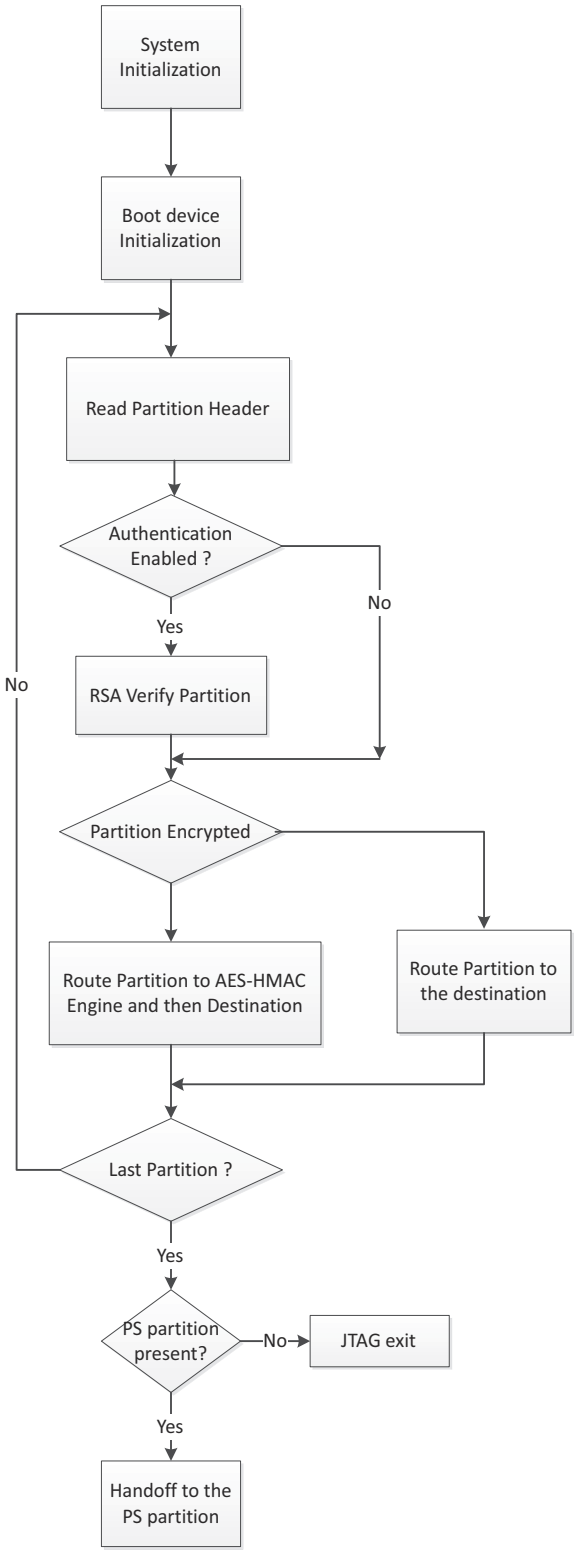

Figure 3-2: **Example FSBL Flow**

The bitstream for the PL and the second stage bootloader or bare-metal application data, as well as other code and data used by the second stage bootloader, Linux (or other operating system), or bare-metal application are grouped into partitions in the flash image. See section 3.4.2 Boot Image Format, for a description of how they are organized.

The FSBL traverses the partition header table to find the bitstream and second stage bootloader or bare-metal application partition. See Appendix A, Using Bootgen, for details.

See 3.4 Boot Image Creation, for details on how the boot image containing these partitions is constructed.

You *stitch* the FSBL with the bitstream and an application using the Bootgen program. SDK has a Create Boot Image wizard option, shown in Figure 3-3, to add the partition images and create a bootable image that you can then flash.

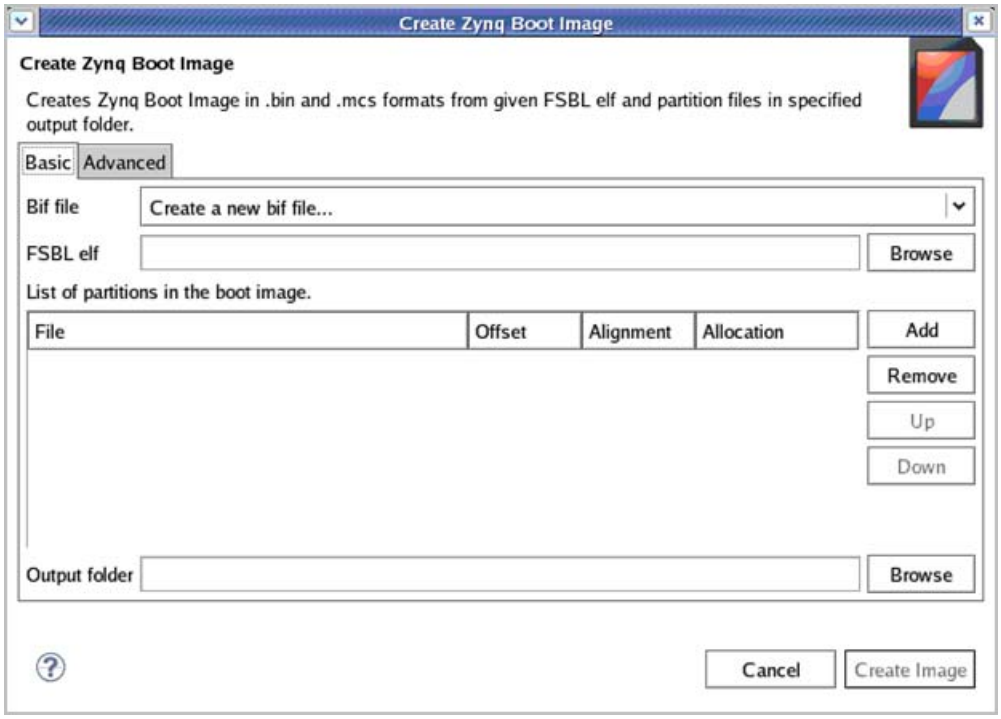

Figure 3-3: **Create Zynq Boot Image Wizard**

The rules are:

- The first partition must be the FSBL ELF followed by the bitstream partition and then the application ELF.
- Bitstream is optional. FSBL does a handoff to the first application in the BIF order.

IMPORTANT: *The order within the BIF file is important. Bitstream must be the partition after FSBL. Bitstream is not mandatory. The bitstream is required only if the PL must be programmed.*

FSBL does not remap the DDR; consequently, DDR that is lower than 1Mb cannot be used.

IMPORTANT: *The application ELF must have an execution address of greater than 1Mb.*

Figure 3-4 shows a simple FSBL flow diagram:

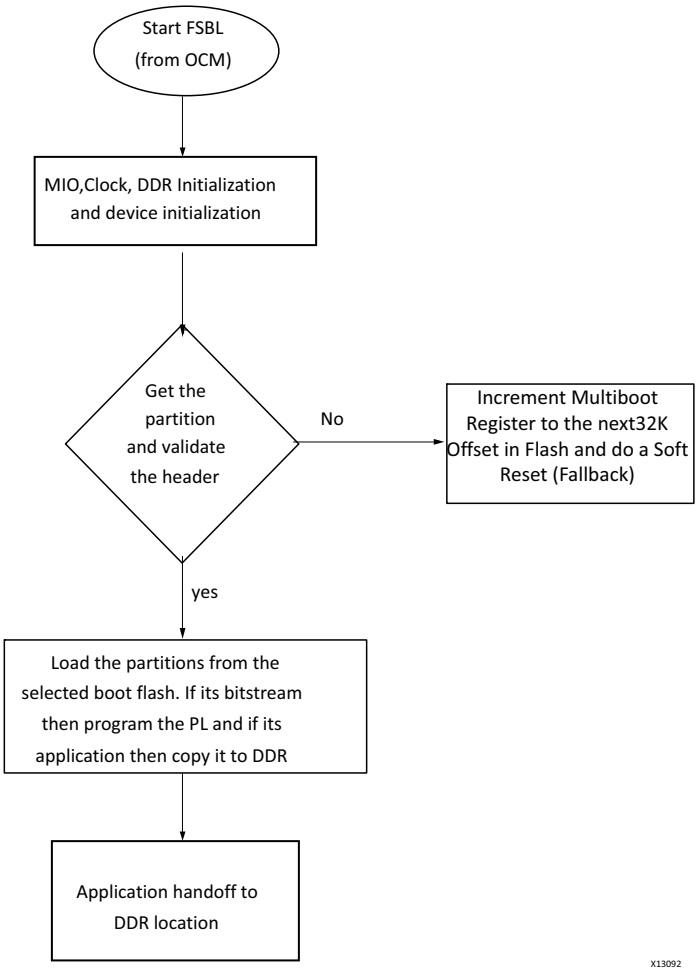

Figure 3-4: **FSBL Flow Diagram**

eMMC Flash Devices

Zynq-7000 devices support eMMC flash devices in MLC and SLC configuration as a secondary boot source. FSBL supports loading the partitions from eMMC. This is possible only when the primary boot mode (set through the boot mode pins) is QSPI. Use this option when there is a small QSPI flash and you would like to store all the other partitions on a larger flash memory like eMMC. In this case, place the FSBL on the QSPI flash and all the other partitions are on eMMC flash.

To enable and use this boot mode:

1. Enable the MMC_SUPPORT flag through SDK and build FSBL.

   The FSBL image build (`fsbl.elf`) now has eMMC support.

2. Stitch the boot image with FSBL as the only partition (using Bootgen).
3. Place the boot image in the QSPI flash.
4. Stitch an image (using bootgen) with all the other required partitions (like the bitstream or the U-Boot) and place it in the eMMC flash.
5. Set the boot mode to QSPI.
6. Power cycle the board.

BootROM comes up, loads the FSBL from QSPI flash to OCM and does a hand-off to FSBL. FSBL then picks all the other partitions from the eMMC device, loads them to DDR, then hands over control to the application.

In this case, FSBL ignores the configured primary boot mode (configured through the boot mode pins on the board) which is QSPI and loads the other partitions from eMMC.

To have FSBL and U-Boot on the QSPI flash, the `MMC_SUPPORT` flag need not be enabled in FSBL; however, the U-Boot auto-configuration file must be updated to indicate to U-Boot to load the rest of the partitions from eMMC flash.

In this case FSBL loads U-Boot to DDR and hands over the control to U-Boot.

U-Boot handles loading the rest of the partitions from the eMMC flash. The limitation here is that the partitions present on the eMMC flash cannot be RSA authenticated because U-Boot does not support RSA authentication.

Setting FSBL Compilation Flags

Compilation flags can be set using the C/C++ settings in SDK FSBL project, as shown in Figure 3-5.

Note: *There is no need to change any of the FSBL source files or header files to include these flags.*

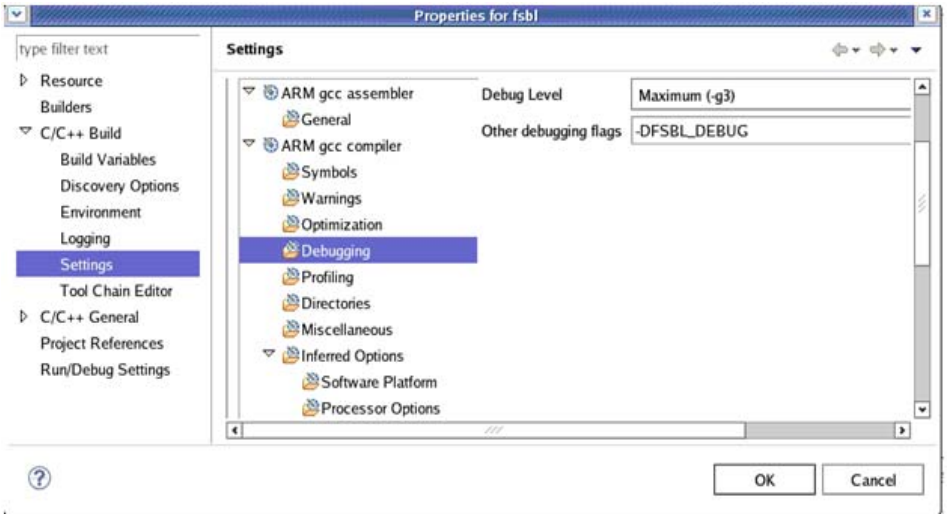

Figure 3-5: **SDK FSBL Properties Settings**

The FSBL compilation flags are:

`FSBL_DEBUG`	Set this flag to enable the logs and message prints.
`FSBL_DEBUG_INFO`	Set this flag to obtain more detailed logs like register and partition header dumps.
`NON_PS_INSTANTIATED_BITSTREAM`	Set this flag when the bitstream does not have a PS component. Then the FSBL does not enable level shifters.
`RSA_SUPPORT`	Set this flag to enable authentication feature in FSBL.
`MMC_SUPPORT`	Set this flag to enable MMC support in FSBL. When this flag is set, FSBL reads all the partitions from the eMMC device, instead of the primary boot device (which is set by the boot mode pins).

FSBL Fallback Feature

To recover from an error condition, FSBL does a *Fallback* and enables BootROM to load another bootable image (the *golden* image that was originally present and in a known good state) if that image is present in the flash memory. FSBL updates a multiboot register and does a soft reset so that Boot ROM executes and loads the next present, valid image.

In the secure boot scenario, with the AES key stored in eFUSE, the Fallback scenario is handled by FSBL without going through a soft reset. The following subsections describe the details.

For more information about eFUSE, see the *LibXil SKey for Zynq-7000 AP SoC Devices (UG996)*, which is located in the *OS and Libraries Document Collection (UG643)* [Ref 8].

Fallback in Non-Secure Cases

In a FSBL non-secure flow, the following actions occur:

- After Power on Reset (POR), BootROM executes and validates the Image 1 Boot header.
  - If there is no corruption, BootROM hands over control to the FSBL, which then loads the other partitions in the image.
  - If there is corruption in the boot header, BootROM does a fallback search to find the next valid image. In the example shown in Figure 3-6, BootROM validates the Image 2 boot header, and, if no corruption, hands over the Image 2 to FSBL, which processes the rest of the partitions in Image 2.
  - In non-secure images, corruption in FSBL and other images is not recognized.

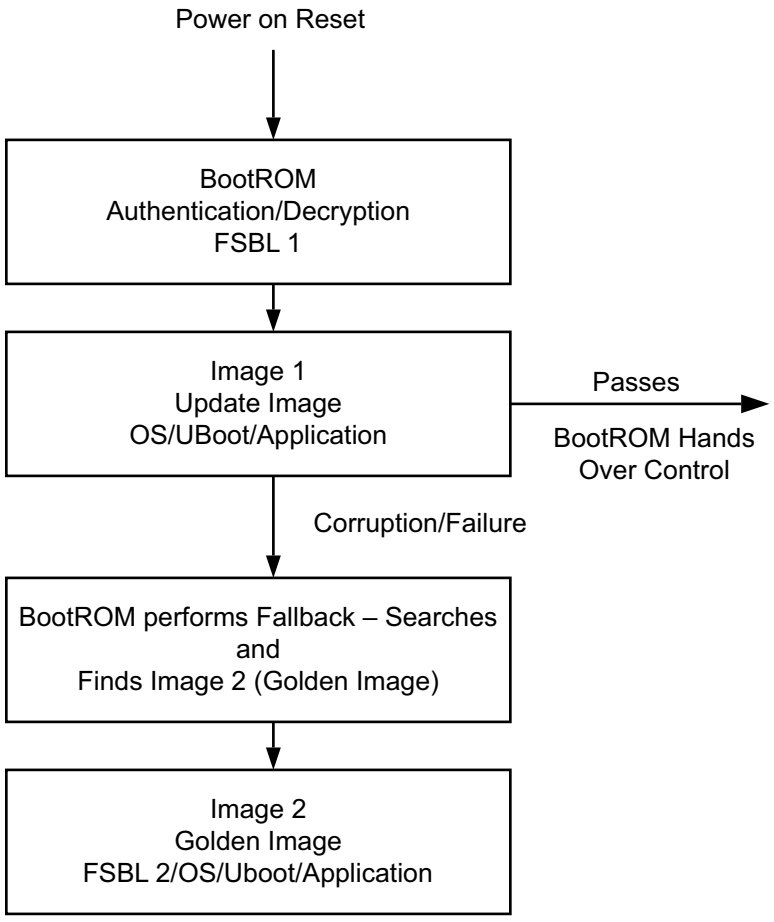

Figure 3-6: **Power on Reset Fallback**

Figure 3-7 represents the Flash image format for non-secure cases.

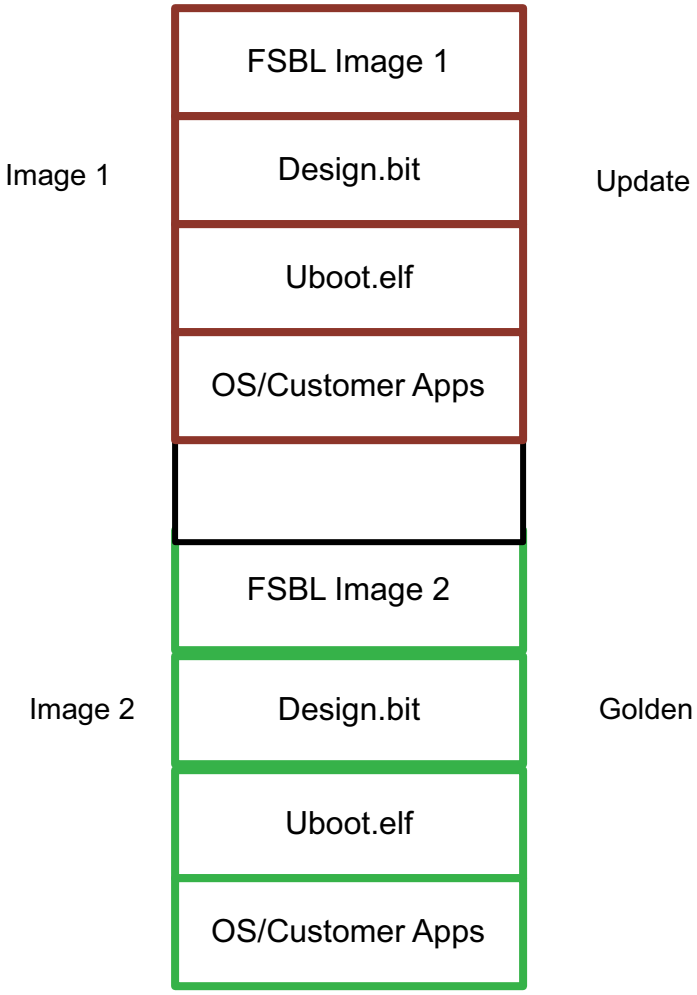

Figure 3-7: **Non-Secure Fallback Image Format**

Fallback Flow for RSA Only

In the case of non-secure Fallback with RSA authentication enabled, the following actions occur:

- After Power On Reset, BootROM executes and validates the Boot Header in Image 1.
- If there is no corruption in the Boot Header, BootROM hand over control to the FSBL, which then authenticates the rest of the partitions and loads those partitions.
- If there is corruption in the Boot Header or the FSBL image, BootROM does a fallback search to find the next valid image. In this example, in Image 2, the BootROM validates the Image 2 boot header. If the boot header validation is successful, then BootROM authenticates the FSBL in Image 2 and hands control over to FSBL.
- In this case, when there is corruption in the bitstream, U-Boot, or the OS, FSBL authentication fails and does fallback by a soft reset of the system and BootROM locates the golden image.

Figure 3-8 shows the Fallback flow for RSA only.

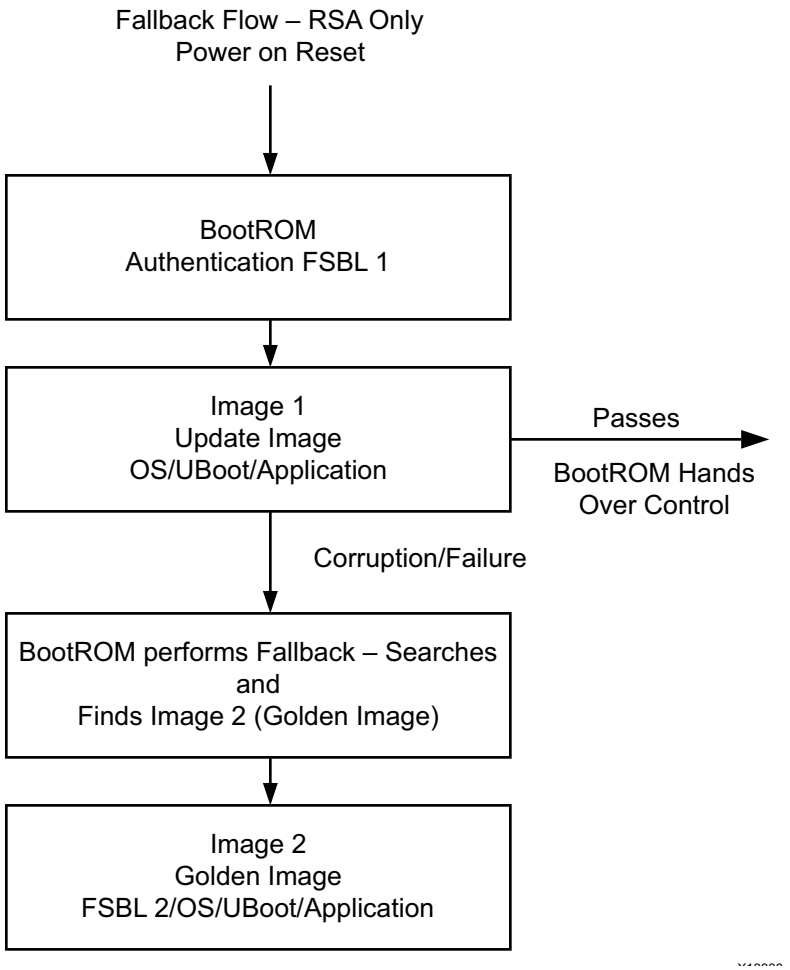

Figure 3-8: **RSA-Only Fallback Flow**

Figure 3-9 represents the Flash image format for non-secure cases.

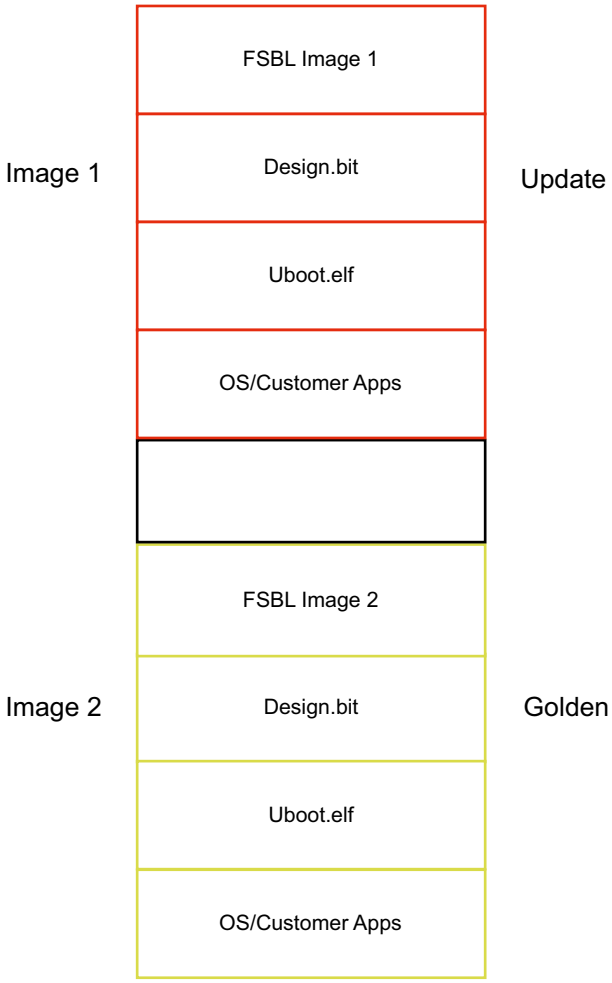

Figure 3-9: **Fallback Partitions for RSA Only**

Secure Fallback Flow with BBRAM

In the secure Fallback flow using BBRAM, the following actions occur:

- BootROM executes and decrypts the FSBL1 and authenticates if RSA is enabled.
  - If the validation is successful, the BootROM hands over the control to FSBL, which then loads, decrypts, and authenticates (if enabled) the other partitions, then hands the control to the OS, U-Boot, and/or application.
  - If the boot header of Image1 is corrupted, BootROM searches for the Image2, decrypts the FSBL, and hands off the decryption to FSBL in Image2. Then the FSBL does any required decryption and authentication (if enabled) of the rest of the partitions and hands over to the U-Boot, OS, or Standalone application. In this process, if FSBL finds any image to be corrupted, then it initiates a Fallback.

Figure 3-10 shows the secure Fallback flow with BBRAM.

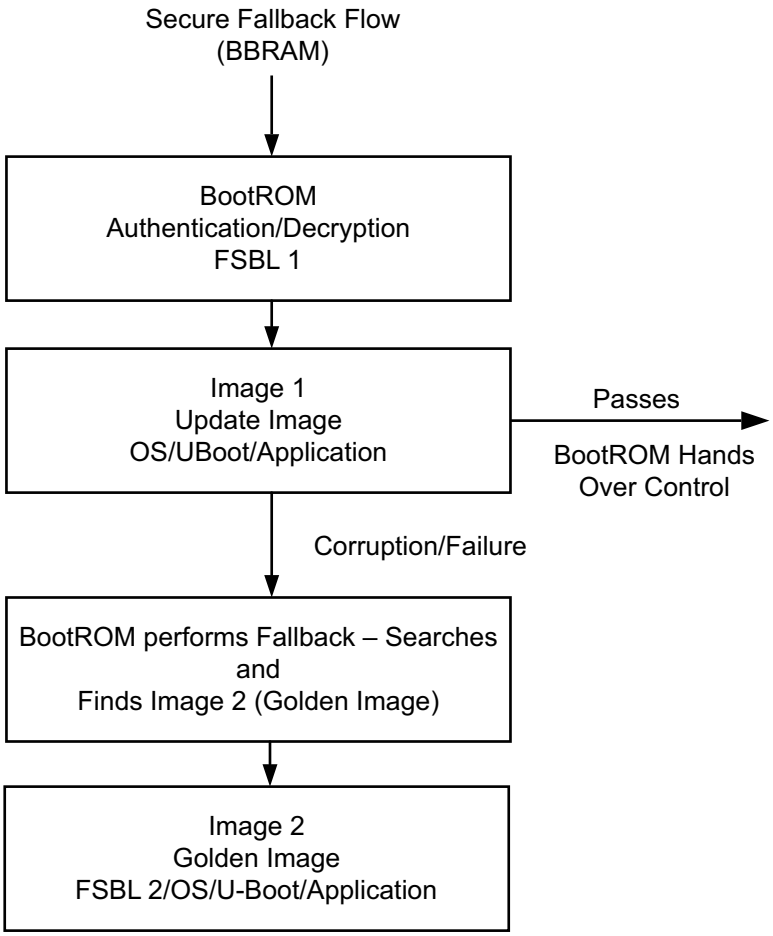

Figure 3-10: **BBRAM Secure Fallback**

Figure 3-11 shows the Flash partitions for secure boot in BBRAM.

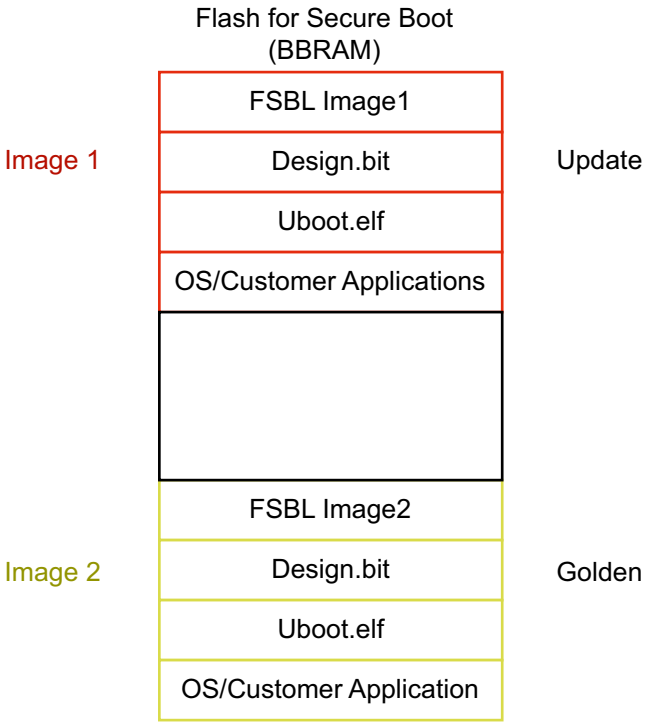

Figure 3-11: **Flash BBRAM Partitions**

Secure Fallback Flow with eFUSE

The secure Fallback flow with eFUSE during Power on Reset (POR) is as follows: In this case FSBL handles the Fallback without going through a soft reset.

- BootROM executes, decrypts the FSBL*, authenticates (if enabled), and passes control back to the FSBL*.
- The FSBL* then:
  - Handles the Encrypted Fallback scenario
  - Finds no other partitions; consequently, does a Fallback, searching for the next valid image.
    - Finds Image 2, and validates the boot header of the Image 2.
    - If valid, skips over the FSBL in Image2 and processes all the other partitions in Image 2, then hands over control to the application in Image 2.

If there is header corruption in Image 2:

- FSBL* to error out with a message to indicate that the Image 2 is corrupt. (FSBL in Image 1 still controls the files and performs Fallback to search for the next Image.)
- FSBL* searches for loads Image 3, then:
  - Validates the boot header

- Authenticates any files if RSA is enabled
- Skips over the FSBL in Image 3 and processes the rest of the partitions in Image 3.

RECOMMENDED: *Use Authentication for secure images.*

Figure 3-12 shows the secure Fallback flow with eFuse.

* This is the FSBL that must be used for Fallback with Encryption

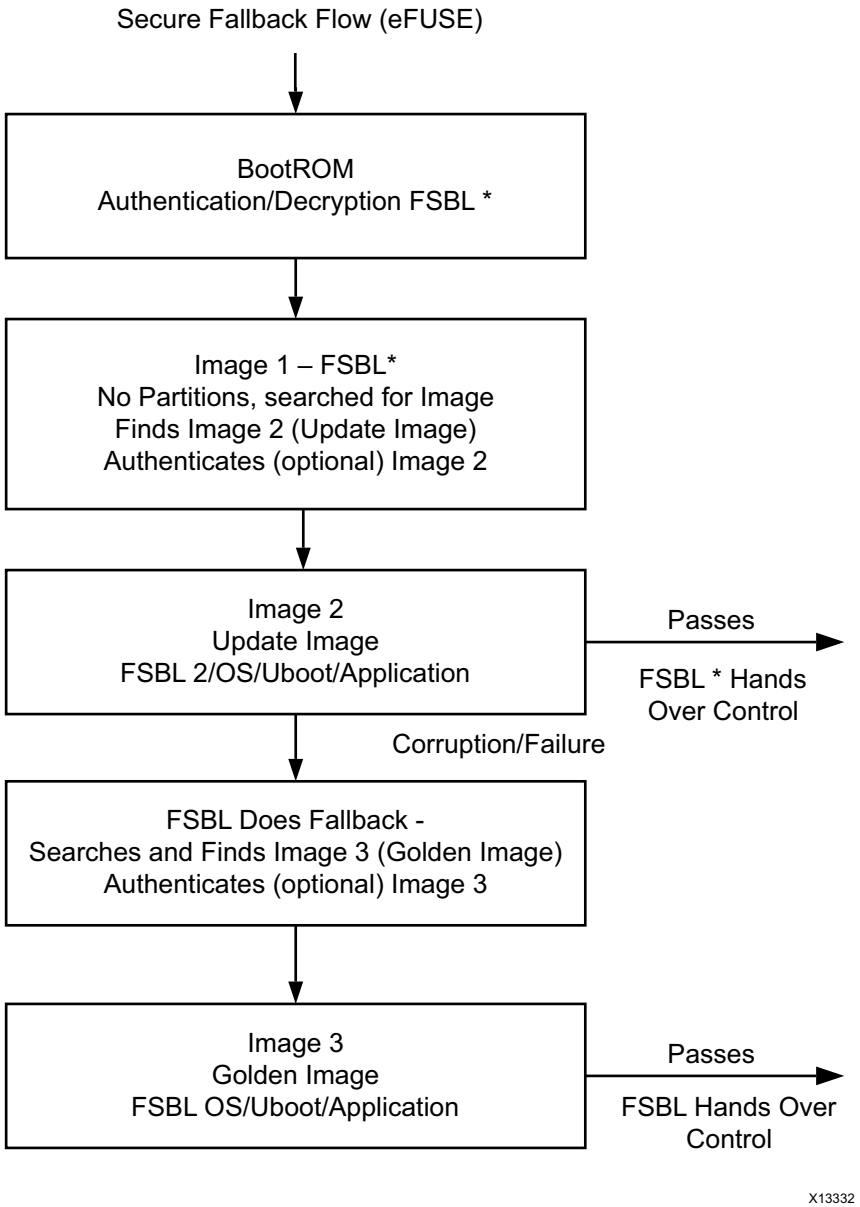

Figure 3-12: **Secure Fallback Flow with eFUSE**

Note: *The Secure flow for Fallback when the AES keys are stored in the PL eFUSE is different than the other flows. RSA authentication in optional.*

- If the FSBL* (FSBL in Image 1) fails authentication, the BootROM goes into a secure lockdown; consequently, you must ensure that Image 1 is not corrupted.
- If the boot header of Image 1 is not valid, the BootROM jumps to Image 2 and the FSBL in Image 2 executes.

RECOMMENDED: *It is recommended in secure mode that you configure Watchdog timers for Interrupt and not SRST. You can route the Watchdog Interrupt to do POR through a GPIO.*

Figure 3-13 shows the FSBL* partitions for Secure Boot with eFUSE.

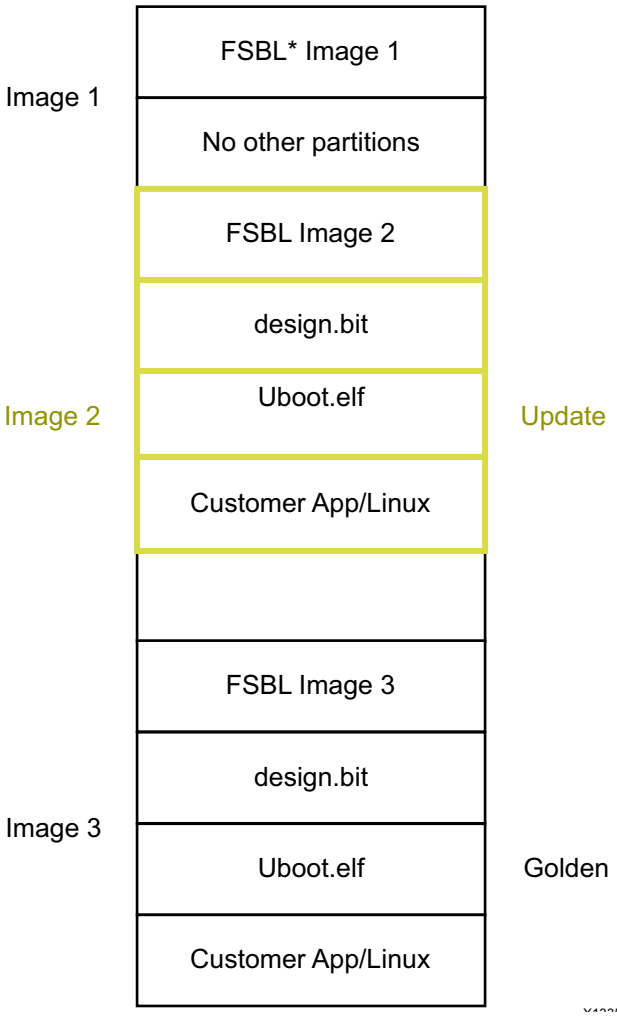

Figure 3-13: **Flash Partitions for Secure Boot with eFUSE**

FSBL Multiboot

Multiboot is the scenario where you want to load some other version of FSBL, other than the one currently executing. For example, you might want to execute a version of FSBL; any image that performs self test and diagnostics, and then jump to the actual application.

In this scenario, after executing the image which performs the diagnostics you can update the multiboot register with the sequence number of the load file which contains the actual application and issue a soft reset.

In the Multiboot scenerio:

- Several images can be used to setup the functionality of a part
- The images are user-selectable, based on what the function the part is supposed to perform at certain times

While the system boots up through the soft reset, the BootROM reads the multiboot register and jumps to that loadable image instead of the first loadable image.

In the secure boot scenario, with the AES key stored in eFUSE, the Multiboot scenario must be handled by the user (without going through a soft reset).

NAND Boot Mode

In NAND boot mode, to use Multiboot, call the `calculate_multiboot()` API that is provided in FSBL. This API calculates the MultiBoot address.

The sequence is:

1. Set the Boot mode to NAND using bootstrap pins.
2. Add the API provided in FSBL to the application that is invoking MultiBoot.
3. From the application that is invoking MultiBoot, call the calculate_multiboot() API to calculate the MultiBoot address.
4. Update the MultiBoot address to the MultiBoot address register, (which is described in Section 6.4.6 of the *Zynq-7000 AP SoC Technical Reference Manual,* (UG585) [Ref 6]) and trigger a soft reset.

QSPI Boot Mode

In QSPI boot mode (where the QSPI device is >128Mb), to use MultiBoot, place the multiple images in such a way that they fit in memory locations less than 128Mb.

To effect this mode, the images should have only (FSBL+U-Boot) to fit in the <128Mb memory. Then, the rest of the partitions, possibly residing in a portion of memory that is >128Mb, must be handled by U-Boot. In this case, update the `zynq_common.h` file to add the commands to load the required partitions. You can find further details on the usage, along with an example, on the Xilinx WIKI pages [Ref 1].

FSBL Hooks

FSBL hooks provide an easy way to plug-in used defined functions, (for example, initializing PL IPs after loading a bitstream). The FSBL hook functions are located in the `fsbl_hook.c` file.

The `fsbl_hook.c` file contains the following functions:

- `FsblHookBeforeBitstreamDload`: This function is called before the PL bitstream download. Any customized code. You can add customized code before the bitstream download in this function.
- FsblHookAfterBitstreamDload: This function is called before the handoff to the application. You can add any customized operations you want to perform before handoff to the application to this function.
- `FsblHookBeforeHandoff`: This function is the hook to call before the FSBL does a handoff to the application. You can add customized code to be executed before the handoff to this routine.
- `FsblHookFallback`: This function is called when the FSBL does a Fallback. You can add customized code, either to print a message, log an error, or do any other intended operation, when Fallback occurs.

By using these hook functions you can plug-in any application-specific customized code into the flow sequence of the FSBL.

DDR ECC Enable

This feature enables ECC support for the DDR.

- In XPS, enable the feature.
- In the Vivado IP integrator Zynq-7000 Block Diagram, use the DDR configuration page.

After the feature is enabled, FSBL does the DDR initialization required to enable the ECC.

FSBL does not provide support for error handling for the ECC errors; you must account for error handling within your program.

DDR starts from 1Mb because FSBL does not remap DDR; consequently, the application program must consider using the DDR from 1Mb. If you need to use a DDR smaller than 1Mb, you must handle the DDR initialization required for supporting ECC.

Secure Boot Support

FSBL provides support for the following secure boot features:

- Advanced Encryption Standard
  - AES-CBC with 256-bit key
  - Encryption key stored on-chip in either eFuse or Battery-backed RAM (BBRAM)

- Keyed-hashed message authentication code (HMAC)
  - SHA-256 authentication engine (FIPS180-2)
- RSA public key authentication
  - 2048-bit public key

FSBL operates in the secure mode, based upon what secure features you enable.

If RSA authentication is enabled, the FSBL uses the public key to authenticate the FSBL before it is decrypted or executed. You can enable the RSA authentication by providing this as an option to Bootgen while generating the bootable image. Based upon the configuration provided in the partition header (Authentication/Encryption/Both), the FSBL performs the required authentication of the image and then the decryption.

For more details about RSA authentication, see the *Zynq-7000 AP SoC Technical Reference Manual (UG585)* [Ref 6].

Zynq PS Configuration

Using the Zynq-7000 configuration interface, the Xilinx hardware configuration tool generates code for initialization of the MIO and SLCR registers. In the project directory, the files of interest are:

- `ps7_init.c` and `ps7_init.h`, which can be used to initialize `CLK`, `DDR`, and `MIO`. The initialization performed by the `ps7_init.tcl` is the same as by the code in `ps7_init.c`.
- `ps7_init.tcl` file, which can be used to initialize `CLK`, `DDR`, and `MIO`. The initialization performed in the `ps7_init.tcl` is the same as the initialization performed by the code in `ps7_init.c`.

  Note: The Tcl file is helpful while debugging an application using XMD. For example, you can run the `ps7_init.tcl` file and then can load your application to DDR and debug. There is no need to run the FSBL in this case.
- `ps7_init.html`, which describes the initialization data.

Note: *The Xilinx hardware configuration tools maintain synchronization between the PL bitstream and this initialization data. It is not advisable to change these settings manually.*

3.3.3 Second Stage Bootloader (Optional)

The second stage bootloader is optional and user-designed. See 4.4 U-Boot, for an example of the second stage bootloader.

3.4 Boot Image Creation

A utility program called *Bootgen* is provided to create a single boot image file suitable for ROM or flash memory programming. It creates the image by building the required boot header, appending tables that describe the following partitions, and processing the input data files (ELF files, FPGA bitstreams, and other binary files) to partitions. It has features for assigning specific destination memory addresses or imposing alignment requirements for each partition. It also supports the encryption, authentication or performing checksums on each partition.

The utility is driven by a configuration file known as the Boot Image Format (BIF) file with a file extension of *.bif. The BIF file lists the input files to the boot image, along with optional attributes for addressing and optional encryption, authentication or checksums. The format of the BIF file is provided in

For advanced authentication flows, Bootgen can be used to output intermediate hash files that can be signed offline. Otherwise, Bootgen uses the provided private keys to sign the authentication certificates included in the boot image.

The format of the boot image conforms to a hybrid of hardware and software requirements. The boot image header is required by the Zynq-7000 BootROM loader which loads a single partition, typically the FSBL. The remainder of the boot image is loaded and processed by the FSBL.

See Appendix A, Using Bootgen, for more information about the utility.

3.4.1 Bootgen Command Example

The following is a simple command line example:

```
bootgen –image myDesign.bif –o i myDesignImage.bin
```

In this example, Bootgen produces the file `myDesignImage.bin` that contains the boot header followed by the data partitions created from the data files described in `myDesign.bif`.

3.4.2 Boot Image Format

The boot image format consists of the following:

- BootROM header
- FSBL image
- One or more partition images
- Unused space, if available

Figure 3-14 shows the layout of the boot image format.

Note: *Encryption is optional in the FSBL.*

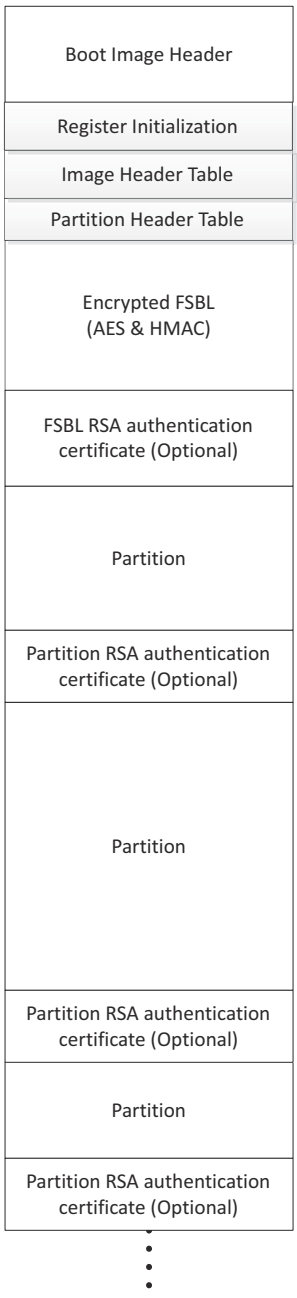

Figure 3-14: **Zynq Secure Boot Image Format**

3.4.3 Authentication Certificate

The Authentication Certificate is appended to the end of each authenticated partition. All integers are stored in little-endian order, including the 2048 bit modulus.

Table 3-1 lists the Offset, Size, Field, and Notes for Authentication Certificate.

Table 3-1: **Authentication Certificate**

Offset	Length	Field	Notes
0x0	0x4	Authentication Certificate Header	See Table 3-2.
0x04	0x4	Authentication Certificate Length	
0x008	0x3C	User Defined Field	56 bytes
0x44	0x100	PPK Modulus	640 bytes little endian
0x144	0x100	PPK Modulus Extension	
0x244	0x04	PPK Exponent	
0x248	0x3C	Padding	480 0s
0x284	0x100	SPK Exponent	640 bytes little endian
0x384	0x100	SPK Modulus Exponent	
0x484	0x04	SPF Modulus Extension	
0x488	0x	Padding	480 0s
0x4C4	0x	SPK Signature	
0x5C4		Partition Signature	

To reduce overhead on the FSBL, BootGen precalculates the modulus extenstion which is used in the Montgomery reduction for modular exponentiation. These values are stored in the certificate after the modulus fields. Table 3-2 shows the Authentication Certificate Bits, Field, and Values.

Table 3-2: **Bit Authenticating Certificate Header**

Bits	Field	Value
31:16	Reserved	0s
15:14	Authentication Certificate Format	00: PKCS #1 v1.5
13:12	Authentication Certificate Version	00: Current AC
11	PPK Key Type	0: Hash Key
10:9	PPK Key Source	0: eFUSE
8	SPK Enable	1: SPK Enable
7:4	Public Strength	0: 2048
3:2	Hash Algorithm	0: SHA256
1:0	Public Algorithm	1: RSA

Figure 3-15 shows an example of the Zynq-7000 AP SoC Linux boot image partitions.

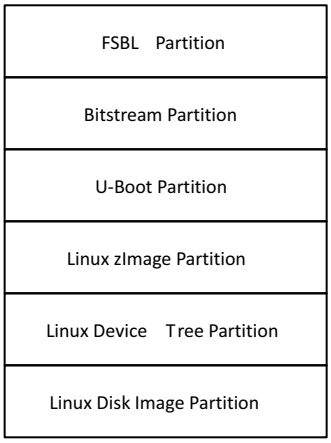

Figure 3-15: **Zynq-7000 AP SoC Example Linux Boot Image Partitions**

3.5 BootRom Header Format

See table 6-3 in the *Zynq-7000 AP SoC Technical Reference Manual, (*UG585) *[Ref 6]* for the BootROM header format.

XILINX

Chapter 4

Linux

4.1 Introduction

Xilinx® Zynq®-7000 AP SoC Linux is based upon open source software (the Kernel from `kernel.org`). Xilinx provides support for Xilinx-specific parts of the Linux Kernel (drivers and Board Support Packages (BSPs)).

Xilinx also supports Linux through the Embedded Linux forum [Ref 1]. As with many open source projects, Xilinx also expects customers to use the open source mailing lists for Linux in areas that are not specific to Xilinx Zynq-7000.

More information about Xilinx Zynq Linux and other Xilinx open source projects is available on the Xilinx Open Source Wiki site [Ref 1] or the most current Linux information.

Xilinx provides a public git server that contains a Linux Kernel, a BSP for Xilinx boards, and drivers for selected IP, which allows third parties to build embedded Linux distributions for Xilinx hardware. In essence, the git server also allows companies who have Linux expertise to develop their own Linux rather than buying a distribution.

Note: Not all Xilinx IP are supported.

4.2 Git Server and Gitk Command

Xilinx uses Git to allow easier interaction with the Linux open source community. For example, patches can be pushed out to the Kernel mainline or patches can be received back from users against the Git tree. Moreover, Git provides some configuration management where the you can see each change to the Kernel.

- The public Git tree is located at http://git.xilinx.com, along with the directions for how to snapshot the repository. You can browse the code from the website.

The main branch of the public repository is the master branch. This is considered the most stable and tested code from Xilinx.

- General information on Git is available at http://git-scm.com
- Git basics are documented at: http://git-scm.com/documentation
- Git can be downloaded from: http://git-scm.com/download

The `gitk` tool provides a graphical display of a git tree. It can be helpful for exploring the branches in a tree. It is installed with git, and can be run using `gitk` from the command line.

Figure 4-1 shows a screen capture of the tool.

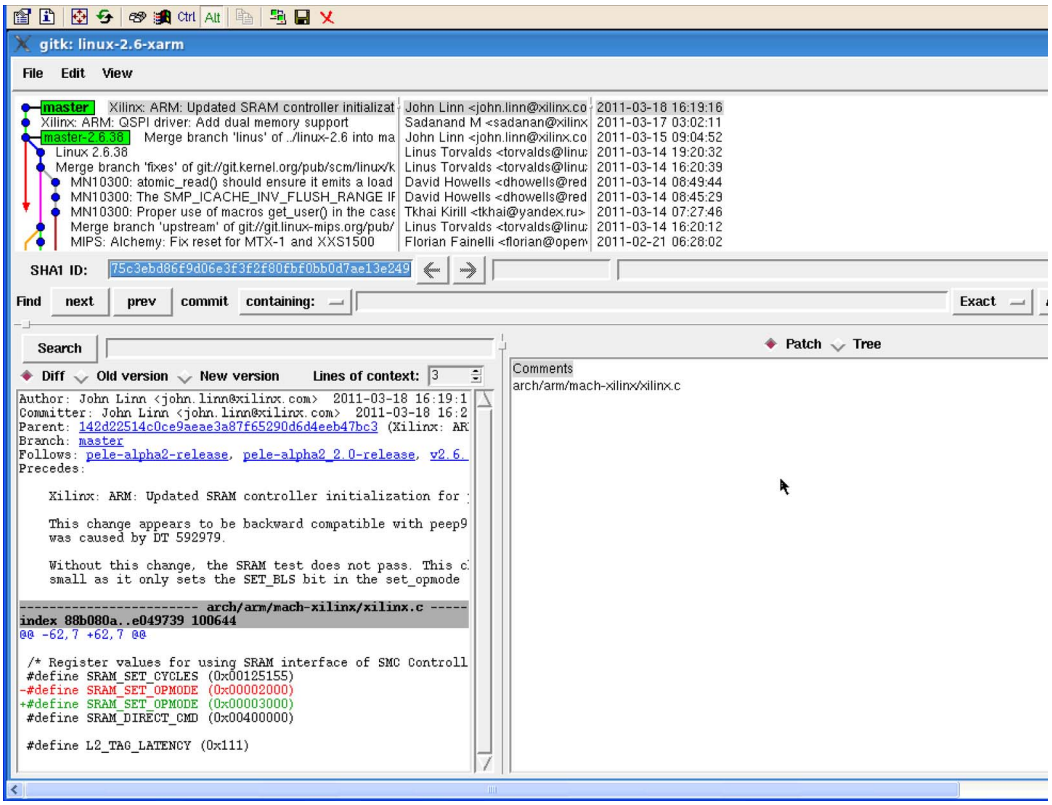

Figure 4-1: **Gitk**

4.3 Linux BSP Contents

4.3.1 Kernel

The Linux Kernel is the Kernel itself together with the Board Support Package (BSP) for boards and the drivers for the system. The Kernel requires a file system, and you must provide a file system to boot the Kernel.

Note: The directory containing the Kernel is referred to as a "Kernel tree." It is assumed that the reader is familiar with the Linux Kernel directory structure.

Figure 4-2, page 50 shows a high order Linux Kernel diagram to help visualize how the different functions relate to the different layers.

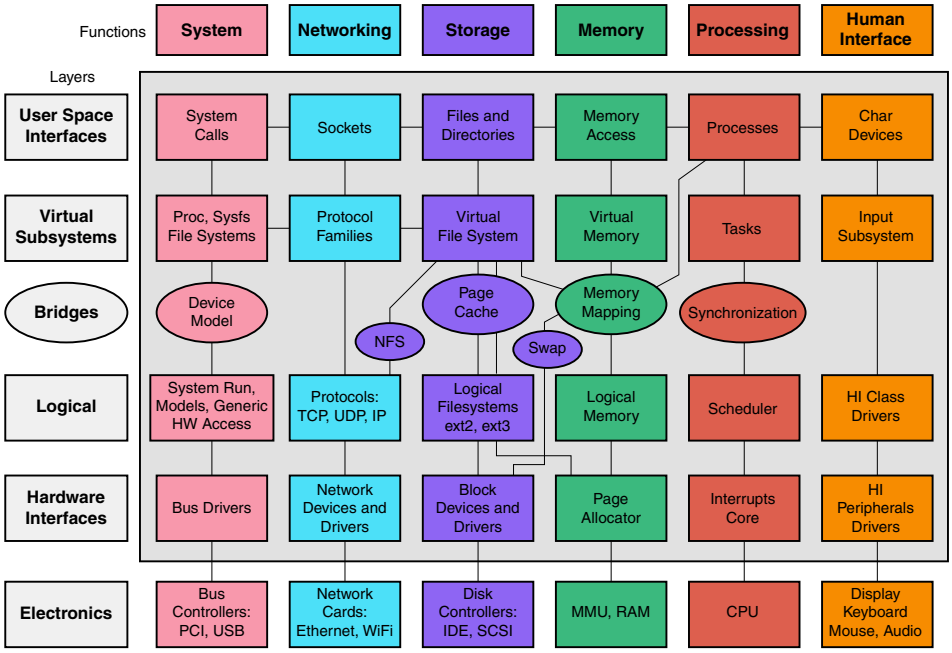

Figure 4-2: **Linux Kernel**

4.3.2 Drivers

See Xilinx SDK online driver documentation.

4.4 U-Boot

Microprocessors can execute code that reside in memory, while operating systems normally reside in large-capacity devices such as hard disks, CD-ROMs, USB disks, network servers, and other permanent storage media. When the processor is powered on, the memory does not contain an operating system, so special software is needed to bring the OS into memory from the media on which it resides. This software is normally a small piece of code called the *bootloader*.

U-Boot is an open source bootloader that is frequently used in the Linux community, and used by Xilinx for the MicroBlaze™ processor and the Zynq-7000 AP processor for Linux. A bootloader initializes hardware that the Linux Kernel does not necessarily initialize (such as the serial port and DDR). System providers often put U-Boot into flash memory. U-Boot is an example of a Second Stage Bootloader, as described in 3.3.3 Second Stage Bootloader (Optional).

This gives it many useful features, including the ability to load and execute images from Ethernet, flash memory, and USB, the ability to start a Kernel image from memory, and the availability of a command interpreter with many commands such as: reading and writing to/from memory, and network operations, such as the `ping` command.

See http://wiki.xilinx.com/zynq-uboot for the most current information.

XILINX®

Appendix A

Using Bootgen

A.1 Introduction

Bootgen is a standalone tool for creating a bootable image suitable for the Zynq®-7000 AP SoC processor. The program assembles the boot image by prefixing a header block to a list of partitions. Optionally, each partition can be encrypted and authenticated. The output is a single file that can be directly programmed into the boot flash memory of the system. Other peripheral files can be generated by the tool to support authentication and encryption as well.

The tool can be integrated into SDK for automatic image generation, or can be used in a command-line oriented script.

A.2 BIF File Syntax

The BIF file specifies each component of the boot image, in order of boot, and allows optional attributes to be applied to each image component. Each image component is usually mapped to a partition, but in some cases an image component can be mapped to more than one partition if the image component is not contiguous in memory.

BIF file syntax takes the following form:

```
name ":" "{" "["attributes"]" datafile... "}"
```

- The name and the {...} grouping brackets the files that are to be made into partitions in the ROM image. One or more data files are listed in the {...} brackets.
- The type of image data (ELF, BIT, RBT, or INT - data files with the `[init]` attribute) is inferred from the file extension, and any special preparations needed are applied based on the file type.
- Data files can have an optional set of attributes preceding the data file name with the syntax `["attributes"]`.
- Attributes apply some quality to the data file.
- Multiple attributes can be listed separated with a "," as a separator. The order of multiple attributes is not important. Some attributes are one keyword, some are keyword equates.
- You can also add a filepath to the file name if the file is not in the current directory. How you list the files is free form; either all on one line (separated by any white space, and at least one space), or on separate lines.
- White space is ignored, and can be added for readability.

- You can use C-style block comments of /*...*/, or C++ line comments of //...

BIF File Examples

The following code snippet is an example of a simple BIF file:

```
// A simple BIF file example.

the_ROM_image:
   {
   [init]init_data.int
   [bootloader]myDesign.elf
   Partition1.bit
   Partition1.rbt
   Partition2.elf
   }
```

The following example is of a BIF file where partitions are encrypted and authenticated:

```
image {
[aeskeyfile]secretkey.nky /* this is the key file used for AES */
[pskfile]primarykey.pem /* primary secret key file for authen.*/
[sskfile]secondarykey.pem /* secondary secret key file for authen.*/
[bootloader,authentication=rsa] fsbl.elf /*first stage bootloader */
[authentication=rsa]uboot.elf /* second stage bootloader */
linux.gz /* OS image (compressed)*/
}
```

BIF File Attributes

The BIF has two attribute types:

- `bootloader`: Identifies an ELF data file as the FSBL.
  - Only ELF files can have these attributes
  - Only one file can be designated as the FSBL
- `init`: Identifies an INT - a data file with the `[init]` attribute, as a register initialization file.

The following table lists BIF file attributes and attribute types.

Table A-1: **BIF File Attributes**

Identifier	Description
init	Register Initialization Block
booloader	Partition that contains FSBL
alignment = <value>	Sets byte alignment
offset = <value	Sets absolute offset
checksum =<value>	Specify checksum as `md5`, `sha1`, `sha2`
encryption = <value>	Specify encryption as `none` or `aes`.
authentication = <value>	Reserves a total amount of memory for this partition. The partition is padded to this amount.

Table A-1: **BIF File Attributes *(Cont'd)***

Identifier	Description
pskfile	Primary secret key (PSK) used to sign the partition
psksignature	SPK signature created using the PSK
sskfile	Secondary Secret Key (SSK) file using to sign partitions
ppkfile	Primary Public Key (PPK) file used to authenticate a partition
spkfile	Secondary Public Key (SPK) used to authenticate a partition
aeskeyfile	AES Key File
presign =<filename>	Imports signed partition
udf_data =<filename>	Imports a file containing up to 56 bytes o fdata to be copied to the User Defined Field record of the authentication certificate.

The following table lists the Bootgen supported files.

Table A-2: **Bootgen Supported Files**

Extension	Description	Notes
bin	binary	Raw binary file
.bit/.rbt	bitstream	Strips the BIT file header
.dtb	binary	Raw binary file
image.gz	binary	Raw binary file
.elf	ELF	Symbols and headers removed
.int	Register init	
.nky	AES key	
.pk1	RSA key	

A.3 Initialization Pairs and the INT File Attribute

There are 256 initialization pairs at the end of the fixed portion of the boot image header. Initialization pairs are designated as such because a pair consists of a 32-bit address value and a 32-bit data value. When no initialization is to take place, all of the address values contain `0xFFFFFFFF`, and the data values contain `0x00000000`.

Set initialization pairs with a text file that has a `.int` file extension by default, but can have any file extension.

The `[init]` file attribute precedes the file name to identify it as the INIT file in the BIF file.

The data format consists of an operation directive followed by:

- an address value
- an = character

- a data value

The line is terminated with a semicolon (;). This is one `.set.` operation directive; for example:

```
.set. 0xE0000018 = 0x00000411;   // This is the 9600 uart setting.
```

Bootgen fills the boot header initialization from the INT file up to the 256 pair limit. When the BootROM runs, it looks at the address value. If it is not `0xFFFFFFFF`, the BootROM uses the next 32-bit value following the address value to write the value of address. The BootROM loops through the initialization pairs, setting values, until it encounters a `0xFFFFFFFF` address, or it reaches the 256th initialization pair.

Bootgen supports a fully C/C++ compatible preprocessor and all of the directives. It also supports expansion macros with parameters. Parameters can be passed on the Bootgen command line with the `-D` option that is compatible with GCC, and acts like `#define`.

Bootgen provides a full expression evaluator (including nested parenthesis to enforce precedence) with the following operators:

```
*  = multiply
/  = divide
%  = modulo divide
+  = addition
-  = subtraction
~  = negation
>> = shift right
<< = shift left
&  = binary and
|  = binary or
^  = binary nor
```

The numbers can be hex (`0x`), octal (`0o`), or decimal digits. Number expressions are maintained as 128-bit fixed-point integers. You can add white space around any of the expression operators for readability.

The preprocessor allows parameterization of BIF and INT files, or BIF and INT files that contain multiple configurations to be selectable from the command line. It would be convenient to use an include file with INT files that would allow for symbolic usage instead of naked values.

For example:

```
#include "register_defs.h"

.set. kBAUD_RATE_REG = ( k9600BAUD | kDOUBLE_RATE ) << BAUD_BITS;
```

Values can also be set on the Bootgen command line with the `-D` option. The `-D` option acts just like a `#define` command in an include file. For example:

```
-D kCURRENT_RATE = 10
```

This allows for INT values to set directly from the command line when Bootgen is run from a shell script, or when experimentation of values is needed without requiring the repeated editing of the INT file.

Values can also be passed in to be used in BIT or INT files with `#if`-like directives to select different configurations.

A.4 Encryption Overview

The encryption private key is stored in the eFUSE or block BRAM memory. The BootROM uses the encryption private key to decode the first FSBL partition boot image. The actual decryption is done by the AES/HMAC engine of the Zynq-7000 hardware.

To encrypt a partition:

1. Give the `-encrypt` option on the command line with either `efuse` or `bbram` arguments.
2. List the key file with the `[aeskeyfile]` attribute in the BIF file.
3. Ensure that the `[encryption=aes]` attribute is present for each image file listed in the BIF file that should be encrypted.

The following is an example command line:

```
Bootgen …. -encrypt efuse

Example BIF file:

image: {
   [aeskeyfile]secretkey.nky
   [bootloader,encryption=aes] fsbl.elf
   [encryption=aes]uboot.elf
   linux.gz
}
```

A.5 Authentication Overview

Zynq-7000 RSA authentication uses primary and secondary keys. the primary keys authenticate the secondary keys. The secondary keys authenticate partitions.

The first letter of the acronyms used to describe the keys is either `P` for primary or `S` for secondary. The second letter of the acronym used to describe the keys is either `P` for public or `S` for secret. There are four possible keys:

- PPK = Primary Public Key
- PSK = Primary Secret Key
- SPK = Secondary Public Key
- SSK = Secondary Secret Key

Bootgen can create a authentication certificate in two ways.

- Supply the PSK and SSK. The SPK signature is calculated on-the-fly using these two inputs.
- Supply the PPK and SSK and the SPK signature as inputs. This is used in cases where the PSK is not known.

The primary key is hashed and stored in the eFUSE. This hash is compared to the hash of the primary key stored in the boot image by the FSBL.

The following is an example BIF file:

```
image {
   [aeskeyfile]secretkey.nky
   [pskfile]primarykey.pem
   [sskfile]secondarykey.pem
   [bootloader,authentication=rsa] fsbl.elf
   [authentication=rsa]uboot.elf
   linux.gz
}
```

Using Bootgen on the Command Line Example

```
bootgen -image bootimage.bif -o i my.mcs -efuseppkbits efuseppkbits.txt -encrypt
bbram developer.nky -p xc7z020clg484 -w on
```

A.6 Bootgen Command Options

The following table describes the Bootgen command line options.

Table A-3: **Bootgen Command Line Options**

Argument	Description
-efuseppkbits <filename>	Specifies the name of the efuse file to be written to contain the PPK hash.
-encrypt [bbram | efuse] [StartCBC=<hex_string>] [Key0=<hex_string>] [HMAC=<hex_string>] [<filename>[.nky]]	Specifies how to do encryption. ***Note:*** Arguments in italics are not recommended for new designs, as the key information is now specified in the BIF file.
-f <filename>[.opt]	Name of an option file. If the file extension is missing, an .opt file extension is assumed. These options are identical to the command line options. Newlines are treated as spaces.
-fill [<hex_byte>]	Specifies the byte to use for fill.
-generate_hashes	Specifies to outputting SHA256 hash files.
-h	Prints out a help summary.
-image <filename>[.bif]	Names the input Boot Image File (*bif).
-log <filename>[.log	Outputs human readable text to a file instead of stdout and stderr. If no extension is given, a .log extension is added.

Table A-3: **Bootgen Command Line Options *(Cont'd)***

Argument	Description
-o i <filename>	Specifies the output file. The support output extensions are: • bin • mcs If no extension is given, then bin is appended.
-p <partname>	Specifies the Xilinx part name. This is needed when generating a encryption key, and the name is copied verbatim to the NKY file in the "Device" line. It is otherwise not used by Bootgen.
-q [s|e|w|i	Disable the output of messages. The string following the option indicates which messages types to disable. No spaces can separate the message type characters, but can appear in any order. As many, or as few message type characters can be used at once. The message type string is optional. Leaving the message type blank is equivalent to using `-q s`. An uppercase letter enables the message type. The message-type characters are: `s` = Disable STATUS messages. `i` = Disable INFO messages. `w` = Disable WARNING messages. `e` = Disable ERROR messages.
-spksignature <filename>	Specifies the name of the spk signature file to write. Must be specified with the **pskfile** and **spkfile** options in the BIF file.
-w [on|off]	Specifies what to do if output files exist. `-w on` means to overwrite.

A.7 Image Header Table

Typically the Image Header Table appears immediately after the fixed size Boot Header and Register Initialization Table, so it appears at address 0x000008A0. The Image Header Table consists of a header, followed by a linked list of Image Headers.

The image header table does not need to be contiguous, but it is generated contiguously by bootgen. Each Image header is linked to the next Image header through `NextEntryOffset`.

Note: Offsets are specified in word - not byte offsets. To convert, multiply the word offset by 4 to get the byte offset.

Table A-4: **Image Header Table Header**

Offset	Name	Notes
0x0	Version	0x01010000
0x4	Count of Image Headers	
0x8	Word Offset to the Partition Header	

Table A-4: **Image Header Table Header *(Cont'd)***

Offset	Name	Notes
0xC	Word Offset to first Image Header	
0x10	Padding	Filled with 0xFFFFFFFF to 64 byte boundary

The Image Header Table Header is followed by a sequential list of Image Headers when generated by bootgen, although the specification of the boot image can allow the Image Headers to be scattered throughout the boot image.

A.7.1 Partition Header Table

The partition header table is an array of structures containing the data described in the following table. There is one structure for each partition, including the FSBL partition. The last structure in the table is marked by all NULL values (except the checksum).

Table A-4: **Partition Header Table**

Offset	Name	Description
0x0	Partition Data Word Length	Unencrypted partition data length.
0x4	Extracted Data Word Length	The encrypted data length.
0x8	Total Partition Word Length (Includes Authentication Certificate)	The total encrypted + padding + expansion +authentication length.
0x0C	Destination Load Address	The RAM address into which this partition is to be loaded.
0x10	Destination Execution Address	The executable address of this partition after loading.
0x14	Data Word Offset in Image	The position of the partition data relative to the start of the boot image.
0x18	Attribute Bits	See Table A-5: , page 59.
0x1C	Section Count	The number of sections in a single partition.
0x20	Checksum Word Offset	The location of the checksum word in the boot image.
0x24	Image Header Word Offset	The location of the Image Header in the boot image
0x28	Authentication Certification Word Offset	The location of the Authentication Certification in the boot image.
0x2C	unused	Must be 0x00000000
0x30	unused	Must be 0x00000000
0x34	unused	Must be 0x00000000

Table A-4: ***(Cont'd)*Partition Header Table *(Cont'd)***

Offset	Name	Description
0x38	unused	Must be 0x00000000
0x3C	Header Checksum	A sum of the previous words in the Partition Header.

A.7.2 Partition Attribute Bits

Table A-5: **Partition Attribute Bits**

Bit Field	Description	Notes
31:16	Data Attributes	Not implemented
15	RSA Signature Present	0 – No RSA Authentication Certificate 1 – RSA Authentication Certificate
14:12	Checksum Type	b000 = 0 = No checksum b001 = 1 = RFU (Reserved for Future Use) b010 = 2 = RFU b011 = 3 = RFU b100 = 4 = RFU b101 = 5 = RFU b110 = 6 = RFU b111 = 7 = RFU
11:8	Destination Instance	Not implemented
7:4	Destination Device	0 - None 1 - PS 2 - PL 3 - INT 4-15 - Reserved
3:2	Head Alignment	
1:0	Tail Alignment	

A.8 Image Header

Table A-6: **Image Header**

Offset	Name	Notes
0x0	Word Offset to Next Image Header	Link to next Image Header. 0 if last Image Header.
0x4	Word Offse to First Partition Header	Link to first associated PartitionHeader.
0x8	Partition Count	Always 0
0xC	Image Name Length	Value of the actual partition count.

Table A-6: **Image Header *(Cont'd)***

Offset	Name	Notes
0x10 to N	Image name	Packed in big-endian order. To reconstruct the string, unpack 4 bytes at a time, reverse the order, and concatenated. For example, the string "`FSBL10.ELF`" is packed as 0x10: 'L','B','S','F', 0x14: 'E','.','0','1', 0x18: '\0','\0','F','L' The packed image name is a multiple of 4 bytes.
varies	0x00000000	String terminator.
varies	0xFFFFFFFF	Repeated padding to 64 byte boundary.

XILINX

Appendix B

Additional Resources

6.1 Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

6.2 Xilinx Documentation

- Product Support and Documentation: http://www.xilinx.com/support
- Xilinx Glossary: http://www.xilinx.com/company/terms.htm
- Device User Guides: http://www.xilinx.com/support/documentation/user_guides.htm

1. Xilinx Forums and Wiki Links:
   - http://forums.xilinx.com
   - http://wiki.xilinx.com
   - http://wiki.xilinx.com/zynq-linux
   - http://wiki.xilinx.com/zynq-uboot

- Xilinx git websites: http://git.xilinx.com

2. Xilinx Ecosystem Partners websites: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/ecosystem/index.htm
3. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
4. *Xilinx Design Tools: Installation and Licensing Guide,* (UG798)
5. *Xilinx Design Tools: Release Notes Guide*, (UG631)

6.3 References

- Zynq Landing page: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

6. *Zynq-7000 AP SoC Technical Reference Manual, (*UG585)

6.3.1 PL Documents - Device and Boards

- Xilinx 7 Series Support Page: http://www.xilinx.com/support/documentation/7_series.htm

To learn more about the PL resources, see the 7 Series FPGA User Guides.

6.3.2 Software Documentation

7. *Zynq-7000 All Programmable SoC Concepts, Tools, and Techniques Guide,* (UG873)*:*

The source drivers for standalone and FSBL are provided as part of the Xilinx IDE Design Suite Embedded Edition. The Linux drivers are provided in the Xilinx Open Source Wiki at http://wiki.xilinx.com.

Xilinx Alliance Program partners provide system software solutions for IP, middleware, operating systems, and so forth. See the Zynq-7000 landing page: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

6.3.3 git Information

- http://git-scm.com
- http://git-scm.com/documentation
- http://git-scm.com/download

6.3.4 Design Tools Documents

- *ChipScope Pro Software and Cores User Guide* (UG029):
- EDK Documentation: http://www.xilinx.com/support/documentation/dt_edk_edk14-6.htm

8. *OS and Libraries Document Collection (UG643)*
9. *Platform Specification Format Reference Manual,* (UG642)
10. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
11. *Vivado Design Suite User Guide: Embedded Hardware Design* (UG898)
12. *Vivado Design Suite Tutorial: Embedded Hardware Design* (UG940)
13. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
14. *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995)
15. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
16. *Embedded System Tools Reference Manual* (UG111)
17. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)

B.4 Third Party Documentation

To learn about functional details related to vendor IP cores contained in Zynq-7000 devices or related international interface standards, refer the following documents:

Note: ARM documents can be found at http://infocenter.arm.com/help/index.jsp

- *ARM AMBA Level 2 Cache Controller (L2C-310) Technical Reference Manual* (also called PL310)
- *ARM AMBA Specification Revision 2.0, 1999 (IHI 0011A)*
- *ARM Architecture Reference Manual* (Requires registration with ARM)
- *ARM Cortex-A Series Programmer's Guide*
- *ARM Cortex-A9 Technical Reference Manual*
- *ARM Cortex-A9 MPCore Technical Reference Manual (*DDI0407F): Includes descriptions for Accelerator Coherency Port (ACP), CPU private timers and watchdogs (AWDT), Event Bus, General Interrupt Controller (GIC), Global Timer (GTC), Private Timer and Watchdog Timer (AWDT), and Snoop Control Unit (SCU)
- *ARM Cortex-A9 NEON Media Processing Engine Technical Reference Manual*
- *ARM Cortex-A9 Floating-Point Unit Technical Reference Manual*
- *ARM CoreSight v1.0 Architecture Specification*: Includes descriptions for ATB Bus, and Authentication
- *ARM CoreSight Program Flow Trace Architecture Specification*
- *ARM Debug Interface v5.1Architecture Specification*
- *ARM Debug Interface v5.1 Architecture Specification Supplement*
- *ARM CoreSight Components Technical Reference Manual*: *Includes descriptions for Embedded Cross Trigger (ECT), Embedded Trace Buffer (ETB), Instrumentation Trace Macrocell (ITM), Debug Access Port (DAP), and Trace Port Interface Unit (TPIU)*
- *ARM CoreSight PTM-A9 Technical Reference Manual*
- *ARM CoreSight Trace Memory Controller Technical Reference Manual*
- *ARM Generic Interrupt Controller v1.0 Architecture Specification (IHI 0048B)*
- *ARM Generic Interrupt Controller PL390 Technical Reference Manual (DDI0416B)*
- *ARM PrimeCell DMA Controller (PL330) Technical Reference Manual*
- *ARM Application Note 239: Example programs for CoreLink DMA Controller DMA-330*
- *ARM PrimeCell Static Memory Controller (PL350 series) Technical Reference Manual, Revision r2p1, 12 October 2007 (ARM DDI 0380G)*
- *BOSCH, CAN Specification Version 2.0 PART A and PART B, 1991*
- *Cadence, Watchdog Timer (SWDT) Specification*
- *IEEE 802.3-2008 - IEEE Standard for Information technology-Specific requirements - Part 3:*
- *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, 2008*
- Intel Corp., *Enhanced Host Controller Interface Specification for Universal Serial Bus*, v1.0, 2002

- ISO 11898 Standard USB Association, USB 2.0 Specification
- Multimedia Card Association, MMC-System-Specification-v3.31
- SD Association, Part A2 SD Host Controller Standard Specification Ver2.00 Final 070130
- SD Association, Part E1 SDIO Specification Ver2.00 Final 070130
- SD Group, Part 1 Physical Layer Specification Ver2.00 Final 060509