Zynq-7000 EPP Software Developers Guide

UG821 (v2.0) April 24 , 2012

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/26/2012	v1.0	Initial Public release.
04/24/2012	v2.0	Added “Partition Data Section Count” to Table 3-1. Removed Chapter 4, Bare-Metal BSP.

XILINX

Table of Contents

XILINX

Chapter 1

Introduction

This document summarizes the software-centric information required for designing with Xilinx® Zynq™-7000 Extensible Processing Platform (EPP) devices. It assumes that the audience is:

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

1.1 Documentation Structure and Additional Information

This document is intended to be an introduction to developing embedded applications for Zynq-7000 EPP devices, and not a general reference. It is recommended that you read this document at your leisure, and refer to the additional listed documents for details. See Appendix A, Additional Resources for a full list of published document resources. Documents that have no links are available in the Zynq Beta Lounge:

- *Zynq-7000 Extensible Processing Platform Overview (DS190)*
- *Zynq-7000 EPP DC and AC Switching Characteristics (DS187)*
- *Zynq-7000 EPP Technical Reference Manual (UG585)*
- *Zynq EDK Concepts, Tools, and Techniques Guide (UG873)*
- SDK Online Help
- *Embedded Systems Tools Reference Manual (UG111)*
- *OS and Libraries Document Collection (UG643)*
- *USB Cable Installation Guide (UG344)*
- *Platform Cable USB II Data Sheet (DS593)*
- *Platform Specification Format Reference Manual (UG642)*

This document has the following content and structure:

Architectural Decisions, page 3, describes the necessary architectural decisions that you must make prior to starting an EPP design.

Operating System (OS) Considerations, page 4 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 EPP devices or in a logic simulation environment while applications execute on a Zynq-7000 EPP processor on a physical board or an emulator. See Software Tools Overview, page 6.

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 EPP 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 EPP devices. It details the three possible boot modes, then documents the two 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, 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 Concepts, Tools, and Techniques Guide (UG873)*.

1.2 Architectural Decisions

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

Because the Zynq-7000 EPP devices have dual-core ARM Cortex™-A9 processors, developers 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.

1.2.1 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 more easily 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 4, 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

1.3 Operating System (OS) Considerations

1.3.1 Bare-Metal System

Bare-metal refers to a software system without an operating system. This software system typically does not need a lot of 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.

1.3.2 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.

1.3.3 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.

XILINX®

Chapter 2

Software Application Development Flows

The Zynq™-7000 EPP 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 EPP 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.

2.1 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 EPP devices or in a logic simulation environment while applications execute on a Zynq-7000 EPP processor on a physical board or an emulator.

Xilinx provides design tools for developing and debugging software applications for Zynq-7000 EPP 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 operating system, and applications for the open source Linux operating system. Xilinx hardware design tools such as the Xilinx Platform Studio (XPS) capture information about the PS and peripherals, including configuration settings, the register memory map, and the bitstream for PL initialization.

XPS captures hardware platform information in XML format along with other data files that 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.

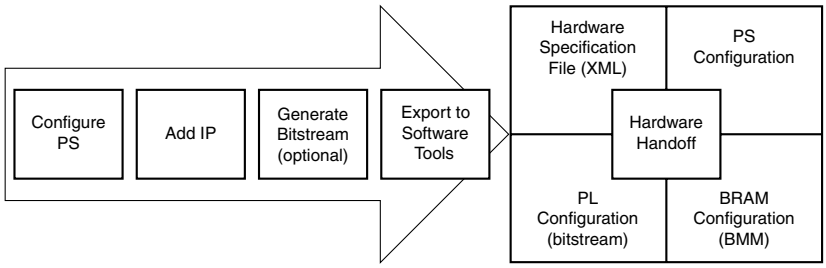

Figure 2-1: **Hardware Design Tool Handoff to Software Tools**

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 EPP 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 software development tools. Tools are available on 32- and 64-bit Windows and x86 Linux host computing platforms.

2.1.1 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, bare-metal software, 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 is available from the Xilinx ISE® Design Suite installation package, the Xilinx Embedded Development Kit (EDK), or as a standalone installation. SDK also includes an application template for creating a First Stage Bootloader (FSBL), as well as a graphical interface for building a boot image.

2.1.2 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.

2.1.3 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) startup support and default linker scripts for the Xilinx Cortex-A9 processors
- Vector Floating Point (VFP) and NEON™ optimized libraries

2.1.4 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 tool. Events (changes) in the custom logic can trigger breakpoints in the software debugger, halting a program running the processor, and vice versa. A variety of co-debugging flows are supported using SDK, ChipScope Pro Analyzer, and other Xilinx tools.

2.1.5 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.

2.1.6 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.

2.2 Bare-Metal Device Driver Architecture

The bare-metal device drivers are designed with a layered architecture as shown in Figure 2-2, page 10. 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.

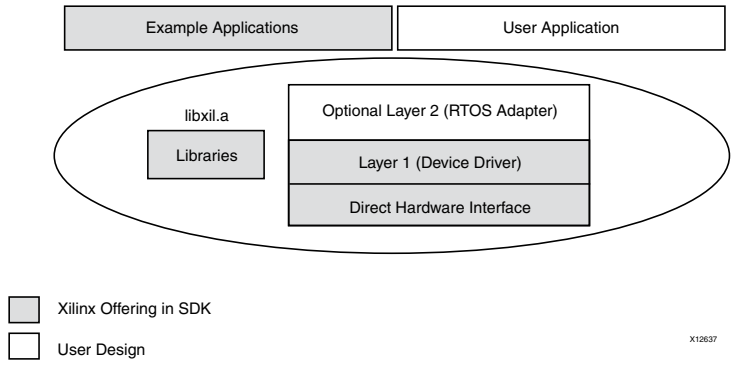

Figure 2-2: **Bare-Metal Drivers Architecture**

2.2.1 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 on the RTOS interface and requirements for the device driver

2.2.2 Layer 1 (Device Driver)

Layer 1 is an abstract device driver interface that shields the user from potential changes to the underlying hardware. It is implemented with macros and functions and designed to allow a developer to utilize 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 the user an "out-of-the-box" solution. The abstract API helps isolate the user from hardware changes.
- No 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
- Comprehensive support of device features
- 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

2.2.3 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 designed so a developer 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, and simple macros that give the user access to the hardware registers
- A small memory footprint
- Little to no error checking
- Minimal abstraction such that 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

2.3 Bare-Metal Application Development

Xilinx software design tools facilitate the development of embedded software applications for a variety of 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 EPP 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 2-3 is an overview flow chart for bare-metal application development.

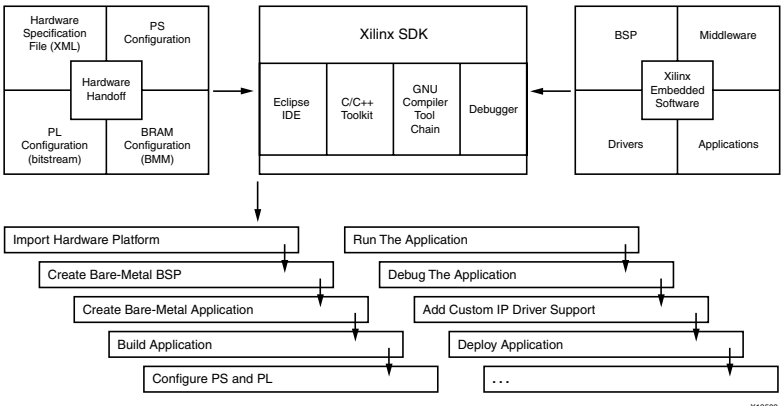

Figure 2-3: **Bare-Metal Application Development Overview**

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)* for more details and examples of SDK tool usage.

2.3.1 Importing Hardware Platform Information

XPS creates hardware platform data that SDK imports 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

2.3.2 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 (v3.05.a) (UG652)* that is included in the *OS and Librarires Document Collection (UG643)*. *Appendix A, Additional Resources* contains a link to the document.

2.3.3 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.

2.3.4 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 bootloader 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.

2.3.5 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.

2.3.6 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 EPP 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`.

2.3.7 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.

2.3.8 Adding Custom IP Driver Support

The hardware platform data that XPS creates captures the Xilinx IP blocks used in the PL area; 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)*. *Appendix A, Additional Resources,* contains a link to the document.

2.3.9 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.

2.4 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 EPP 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 (v3.05.a) (UG652)* that is included in the *OS and Librarires Document Collection (UG643)* for information about the Bare-Metal Board Support Package. *Appendix A, Additional Resources* contains a link to the document.

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 flow chart in Figure 2-4 provides an overview of the flow for Linux application development.

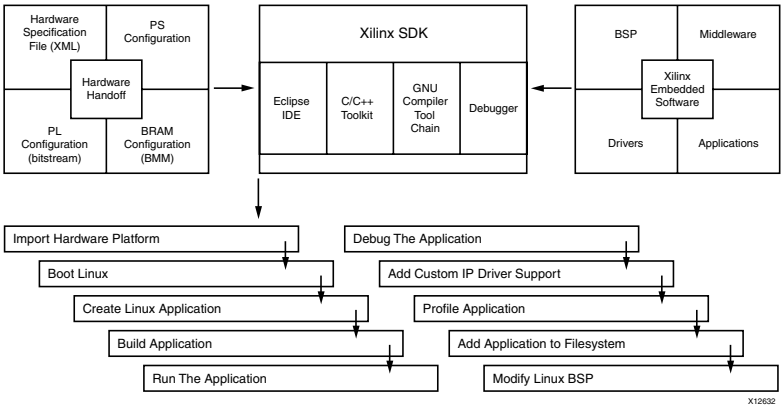

Figure 2-4: **Linux Application Development**

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

2.4.1 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 Linux Kernel
- Use U-Boot to load and run images

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

While flash memory offsets differ for NAND, NOR, Quad-SPI partitions can include FSBL, U-Boot, Linux Kernel, User, Scratch, and `/root` file system.

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.

2.4.2 Creating an Application Project

SDK provides a template-based application generator for included sample programs, from a basic "Hello World" or Dhrystone 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.

2.4.3 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.

2.4.4 Running the Application

An SDK run configuration can be created to copy the compiled application to the file system and run the application. With Linux running on the Zynq-7000 EPP, 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

2.4.5 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.

2.4.6 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.

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.

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

2.4.7 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.

2.4.8 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 EPP, 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.

2.4.9 Modifying the Linux BSP (Kernel or File System)

See the *Standalone BSP (v3.05.a) (UG652)* that is included in the *OS and Librarires Document Collection (UG643)* for information about the Bare-Metal Board Support Package. *Appendix A, Additional Resources* contains a link to the document.

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

2.5 Additional Information

This section provides an overview of Xilinx software solutions, broadly covering bare-metal and Linux application development, but touching only briefly on other tools and flows. For additional information related to topics mentioned in this chapter, consult the references listed in the introduction. For further reading, review the following sections of the *Zynq EDK Concept, Tools, and Techniques Guide, (UG873)*:

- "EDK Design Flow"
- "Adding Sysgen IP"
- "Simulation (System Level - Hardware in the Loop)"

XILINX®

Chapter 3

Boot and Configuration

3.1 Overview

You can boot or configure Zynq™-7000 EPP 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 details of these boot modes are described in the *Zynq-7000 EPP Technical Reference Manual (UG585)*.

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 when the remaining 64 kilobyte is no longer reserved.

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 3.3.2 First Stage Bootloader for details. The boot image can contain a bitstream for the programmable logic (PL).

The PL is not required to be configured at this stage, as the PS is fully operational even with an unconfigured PL. 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.

Figure 3-1, page 20 shows the overall layout of the Zynq-7000 EPP Processor System (PS) from the boot perspective.

Note: DDR and SCU are not enabled by the BootROM. See the *Zynq-7000 EPP Technical Reference Manual (UG585)* for details.

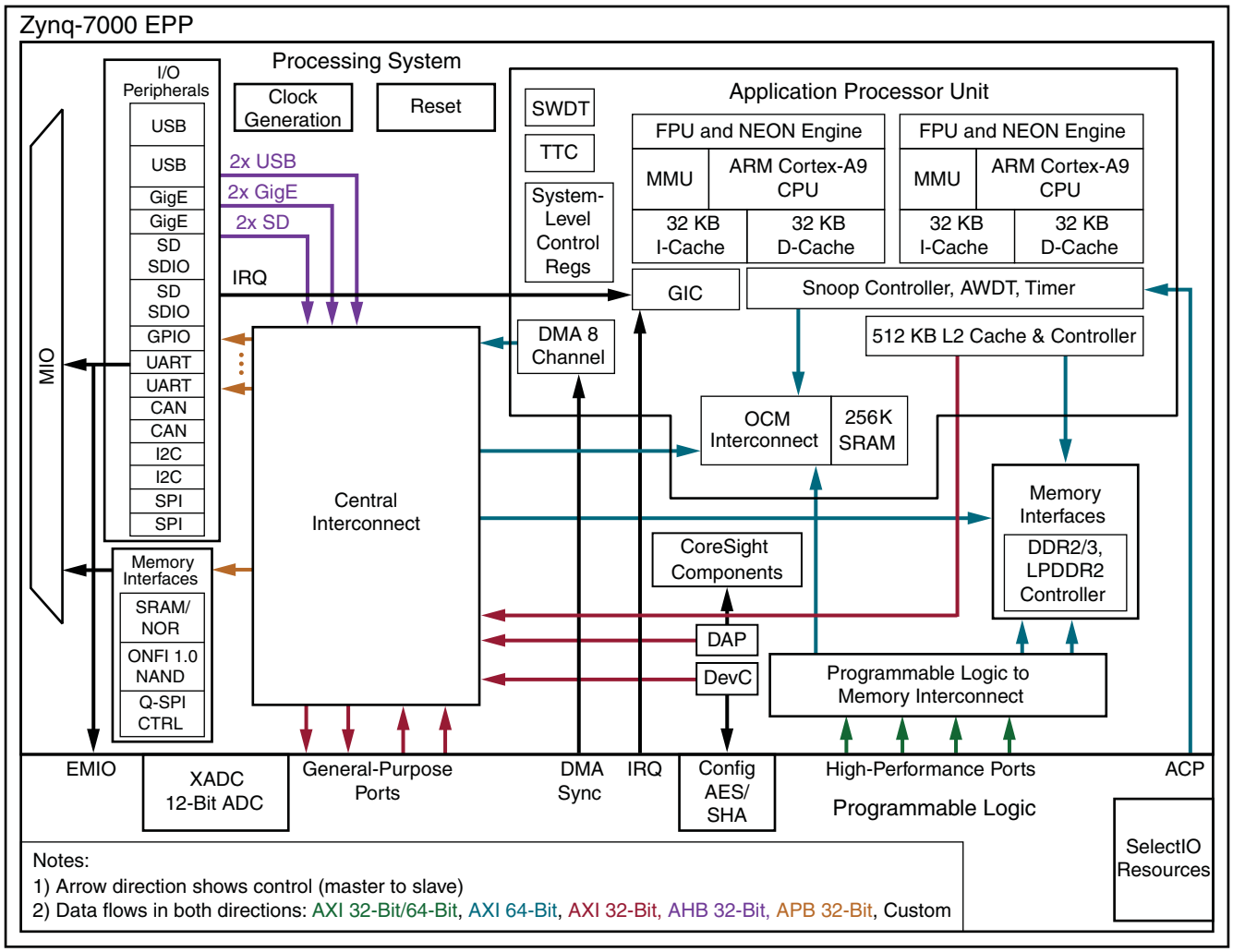

Figure 3-1: **Zynq-7000 EPP Processor System High-Level Diagram**

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, refer to "Device Configuration and the Device Configuration Unit" in the *Zynq-7000 EPP Technical Reference Manual (UG585)*.

3.3 Boot Stages

Zynq-7000 EPP 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 Chapter 6 of the *Zynq-7000 EPP Technical Reference Manual (UG585),* particularly section 6.3, "BootROM."

3.3.2 First Stage Bootloader

The First Stage Bootloader (FSBL) starts after the boot. It is loaded into the OCM (by the BootROM) or executes in place (XIP) unencrypted from memory-mapped flash (NOR or Quad-SPI) depending on the BootROM header description.

The FSBL is responsible for:

- Initialization using the PS configuration data provided by XPS (see Zynq PS Configuration)
- Programming the PL using a bitstream
- Loading second stage bootloader or bare-metal application code into memory
- Starting execution of the second stage bootloader or bare-metal application

Note: The FSBL disables the MMU before jumping to the second stage bootloader or bare-metal application, because Linux (and perhaps other operating systems) assume it is disabled upon start.

See the FSBL code provided with SDK for details on how the FSBL initializes the CPU and peripherals used by the FSBL, and how it uses a simple C runtime library.

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 3.4.1 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 3.4.2 Partition Header Table for details.

See 3.5 Boot Image Creation for details on how the boot image containing these partitions is constructed.

Figure 3-2, page 22 shows the flow of FSBL loading in OCM by the BootROM code.

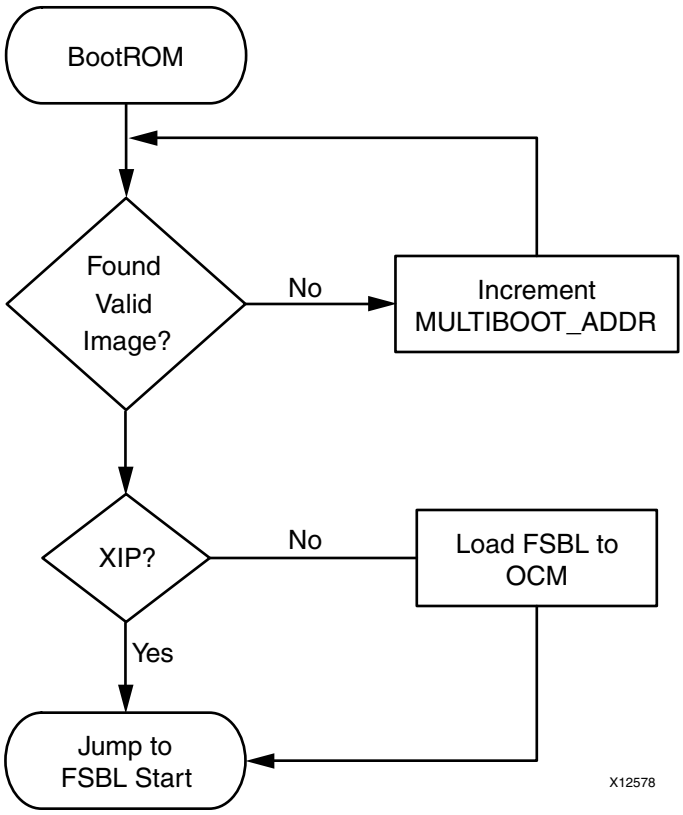

Figure 3-2: **FSBL Flow**

Zynq PS Configuration

Using the Zynq configuration UI, XPS generates code for initialization of the MIO and SLCR registers. In the XPS 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

IMPORTANT: The location and format of the PS initialization data could change in future releases.

Note: XPS maintains 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 4.3 U-Boot for an example second stage bootloader.

3.4 Image Formats

3.4.1 Boot Image Format

The Boot image format consists of:

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

Figure 3-2 shows the layout of the boot image format.

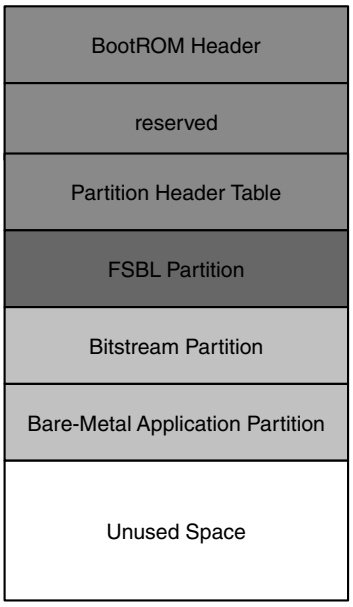

Figure 3-2: **Zynq-7000 EPP Boot Image Format**

Figure 3-3 shows an example of the Zynq-7000 EPP Linux boot image format.

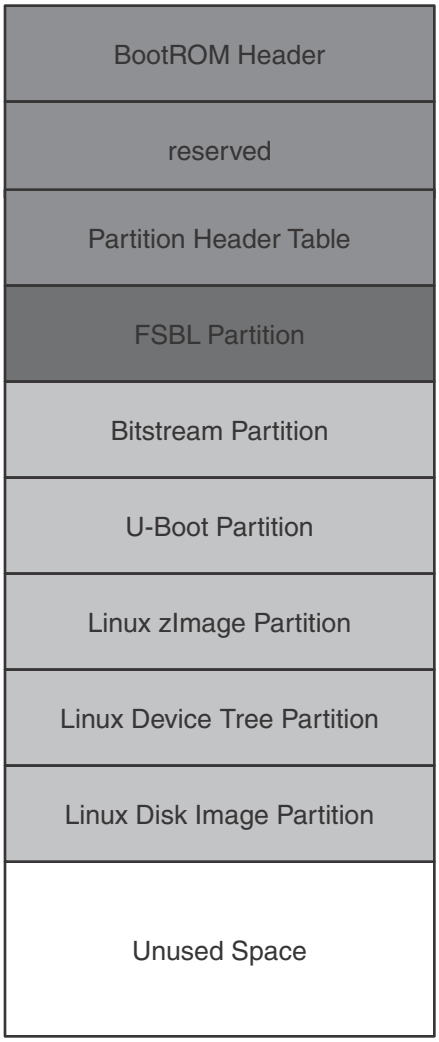

Figure 3-3: **Zynq-7000 EPP Example Linux Boot Image Format**

See Chapter 6 of the *Zynq-7000 EPP Technical Reference Manual (UG585),* particularly sections 6.3.2, "BootROM Header Format."

3.4.2 Partition Header Table

The partition header table is an array of structures containing the data described in Table 3-1, page 25. 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 3-1: **Partition Header Table**

Offset	Name	Description
0x00	Partition Data Size	Length of partition data, in words.
0x04	Written Data Length	Length of decrypted partition data, in words. This value is the same as the partition data size for unencrypted data.
0x08	Partition Total Size	The size of the boot image partition corresponding to this partition header table, including possible unused space (to provide room for expansion). See Figure 3-4.
0x0C	Load Address	Address to load the image into memory. This is the address to which the PS data is to be written. (This field is not applicable to PL bitstream partitions.)
0x10	Execution Address	The address where execution needs to start. (This field is not applicable to PL bitstream partitions.)
0x14	Partition Offset	Number of words from beginning of boot image where the partition data corresponding to this partition header table entry begins.
0x18	Attributes	Bit 5 is set for bitstream partitions. Bit 4 is set for PS partitions (that are copied to memory). All other bits are reserved.
0x1C	Partition Data Section Count	The number of independently loaded data sections within the Partition data. Only applies to memory loading Partition data such as ELF files
0x20 to 0x3B	Reserved	
0x3C	Checksum	Used by FSBL to check the validity of the header. The checksum is bitwise NOT of the sum of all previous values in this structure.

Figure 3-4 illustrates the length fields.

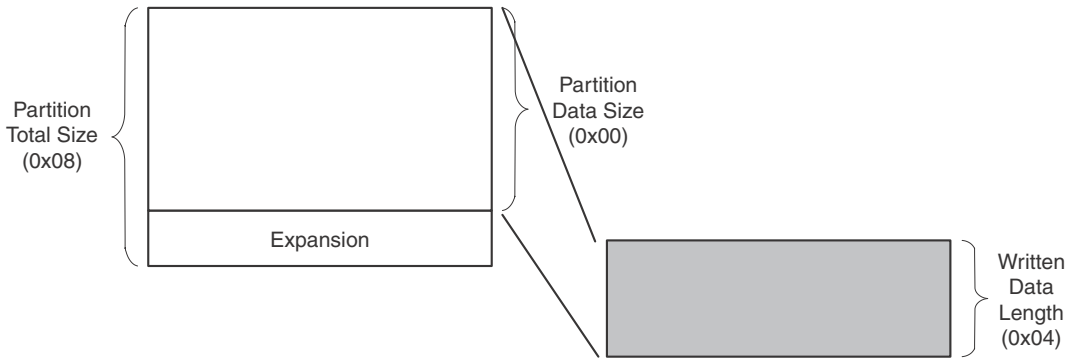

Figure 3-4: **Length Fields in Partition Header**

3.5 Boot Image Creation

Bootgen is a tool for constructing boot images for Zynq configuration. Bootgen merges BIT and ELF files into a single boot image, with a format defined in the Boot Image Format (BIF) file to be flashed into Zynq flash devices. Unlike previous usage with tools such as PROMGen, you are not required to calculate any flash partitioning or image word offsets. Bootgen automatically combines the data into a single boot image. After the boot image is programmed to flash storage, it is then consumed by the Zynq BootROM and FSBL, which automatically extract the user data images and configure the Zynq device.

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`. See the Bootgen online help (displayed by using the -h command-line option).

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 filename 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 filename 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 can be on their own line. White space is ignored, and can be added for readability. You can use C style block comments of /*...*/, or C++ line comments of //...

The following code snippet is an example:

```
// A simple BIF file example.

the_ROM_image:
   {
   [init]init_data.int
   [bootloader]myDesign.elf
   Partition1.bit
   Partition1.rbt
   Partition2.elf
   Partition3.elf
   }
```

There are two attributes:

- `bootloader`—Identifies an ELF data file as the FSBL. Only ELF files can have these attributes, and 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.

ELF files include user-compiled processor code. Bootgen strips the ELF file of any unnecessary data sections, and adds the compiled code as a data partition.

BIT and RBT files are bitstream files produced from Bitgen. Bootgen strips the bitstream of the BIT file header, encrypts the file if requested, and adds the bitstream as a data partition.

INT files are data files with the [init] attribute, and are an optional Bootgen feature used to initialize registers at boot time before the FSBL is loaded and run. It can set up values, such as baud rates or clock rates.

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`.

These initialization pairs are set with a text file with a default `.int` file extension, but can have any file extension. This file is identified as the `INIT` file in the `.bif` with the `[init]` file attribute preceding the file name.

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 also supports a fully C/C++ compatible preprocessor and all of the directives. This includes:

- `#ifdef`
- `#ifndef`
- `#define`
- `#undef`
- `#include`
- `#if`
- `#else`

- #endif
- #elif
- #error
- #pragma

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.

Linux

Xilinx® Zynq™ 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 on http://forums.xilinx.com. 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.

More information about Xilinx Zynq Linux and other Xilinx open source projects is available on the Xilinx Open Source Wiki site: http://wiki.xilinx.com. See http://wiki.xilinx.com/zynq-linux for 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. This Git server is designed to allow 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 IPs are supported.

4.1 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 users 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

gitk is a tool that 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.

See Figure 4-1, page 30 for a screen shot of the tool.

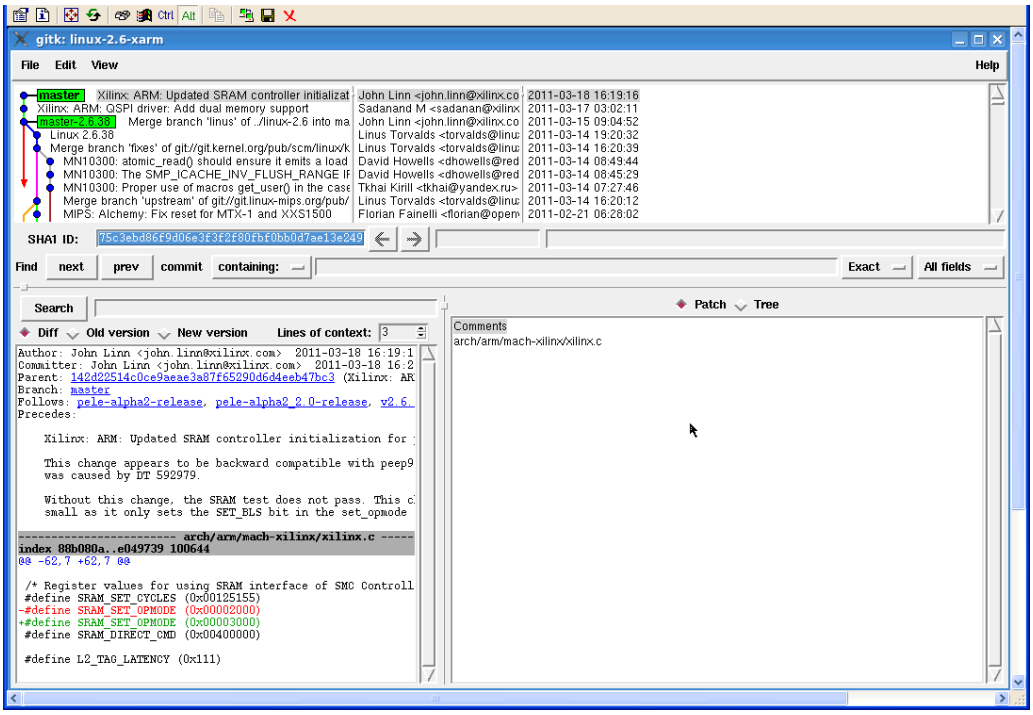

Figure 4-1: **Gitk**

4.2 Linux BSP Contents

4.2.1 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 31 shows a high order Linux Kernel diagram to help visualize how the different functions relate to the different layers.

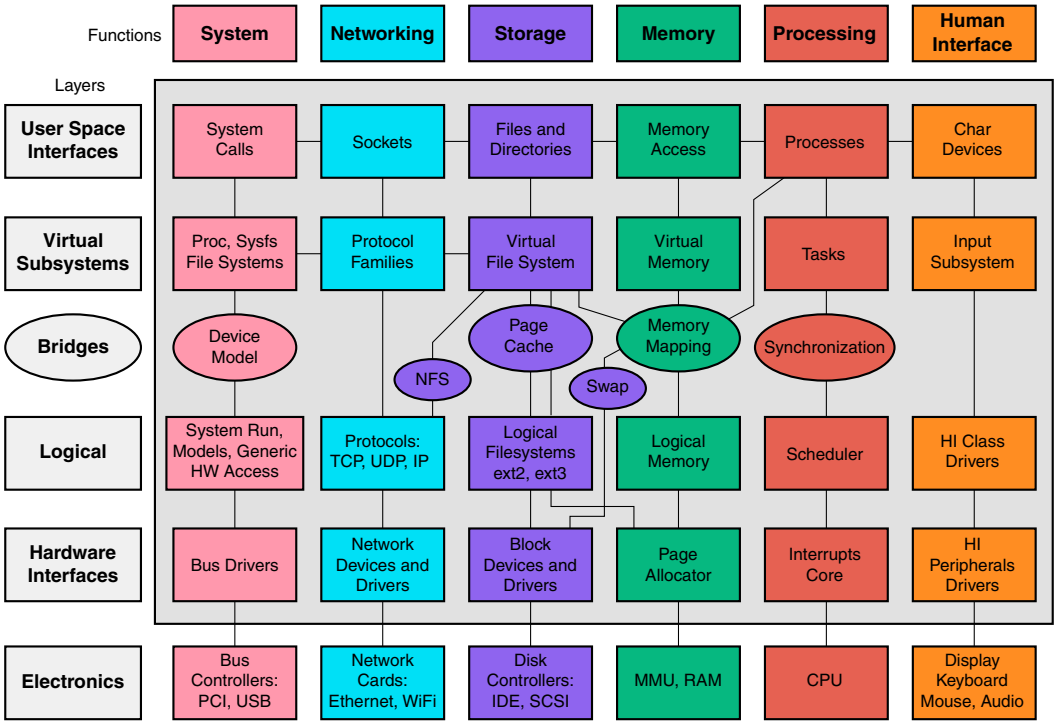

Figure 4-2: **Linux Kernel**

4.2.2 Drivers

Refer to Xilinx SDK online driver documentation.

4.3 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 PowerPC® processors and the MicroBlaze™ 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 a variety of useful 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

Additional Resources

A.1 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
- ***Xilinx Design Tools: Installation and Licensing Guide,*** *(UG798)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/iil.pdf
- ***Xilinx Design Tools: Release Notes Guide,*** *(UG631)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/irn.pdf
- **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

A.2 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.

A.3 References

A.3.1 Zynq-7000 EPP Documents

Refer to the following Zynq-7000 EPP documents for further reference:

- ***Zynq-7000 EPP Product Brief*** *(CS692)*
- ***Zynq-7000 EPP Product Overview*** *(DS190)*
- ***Zynq-7000 EPP Data Sheet*** *(DS187)*
- ***Zynq-7000 EPP Packaging and Pinout Specifications*** *(UG865)*
- ***Zynq-7000 EPP Errata Sheet*** *(EN191)*

A.3.2 PL Documents - Device and Boards

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

- ***Xilinx LogiCORE IP 7 Series FPGAs Integrated Block for PCI Express Product Specification***, *(DS821)*
- ***Xilinx 7 Series FPGAs SelectIO Resources User Guide,*** *(UG471)*
- ***Xilinx 7 Series FPGAs Clocking Resources User Guide,*** *(UG472)*
- ***Xilinx 7 Series FPGAs Memory Resources User Guide***, *(UG473)*
- ***Xilinx 7 Series FPGAs Configurable Logic Block User Guide,*** *(UG474)*
- ***Xilinx 7 Series FPGAs GTX Transceiver User Guide***, *(UG476)*
- ***Xilinx 7 Series FPGAs Integrated Block v1.3 for PCI Express User Guide,*** *(UG477)*
- ***Xilinx 7 Series FPGAs DSP48E1 User Guide,*** *(UG479)*
- ***Xilinx 7 Series FPGAs XADC User Guide***, *(UG480)*

These user guides and additional relevant information can be found on the Xilinx website:

http://www.xilinx.com/support/documentation/7_series.htm

A.3.3 Software Documentation

- ***Zynq EDK Concepts, Tools, and Techniques Guide***, *(UG873):*
  http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug873-zynq-ctt.pdf

  The source drivers for stand alone 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, operation systems, and so forth. See the Zynq-7000 landing page at http://www.xilinx.com/products/silicon-devices/epp/zynq-7000 for the latest information.

A.3.4 git Information

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

A.3.5 Design Tools Documents

- ***Xilinx Command Line Tools User Guide*** *(UG628)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/devref.pdf
- **Xilinx ISE Manuals**: http://www.xilinx.com/support/documentation/dt_ise.htm
- **Xilinx XPS/EDK Supported IP website**: http://www.xilinx.com/ise/embedded/edk_ip.htm
- ***ChipScope Pro Software and Cores User Guide*** *(UG029)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/chipscope_pro_sw_cores_ug029.pdf
- ***PlanAhead Tutorial: Debugging with ChipScope*** *(UG677)*: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/PlanAhead_Tutorial_Debugging_w_ChipScope.pdf
- **Xilinx Problem Solvers**: http://www.xilinx.com/support/troubleshoot.htm

EDK Documentation

You can access the full EDK documentation set online at: http://www.xilinx.com/support/documentation/dt_edk_edk14-1.htm

Individual documents are linked below.

- SDK Online Help
- ***Embedded System Tools Reference Manual*** *(UG111)*: http://www.xilinx.com/support/documentation/xilinx14_1/est_rm.pdf
- ***OS and Librarires Document Collection*** *(UG643)* http://www.xilinx.com/support/documentation/xilinx14_1/oslib_rm.pdf
- ***Platform Specification Format Reference Manual*** *(UG642)*: http://www.xilinx.com/support/documentation/xilinx14_1/psf_rm.pdf
- **EDK Tutorials website**: http://www.xilinx.com/support/documentation/dt_edk_edk14-1_tutorials.htm
- **Platform Studio and EDK website**: http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm

A.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