XAPP1175 (v1.0) September 12, 2013

Secure Boot of Zynq-7000 All Programmable SoC

Author: Lester Sanders

Summary

The Zynq®-7000 All Programmable SoC (AP SoC) integrates a system on chip (SoC) and programmable logic (PL). The boot mechanism, unlike previous Xilinx devices, is processor driven.This application note shows how to boot the Zynq device securely using QSPI and SD modes. The optimal use of RSA authentication and AES encryption for different security requirements is described. A method of handling RSA keys securely is provided. Multiboot examples show how to boot a golden image if the boot of an image fails. Examples show how to generate and program keys. Applications using Zynq security features are discussed.

Included Systems

The reference systems listed in this section are available in the following link:

https://secure.xilinx.com/webreg/clickthrough.do?cid=339774

- zc702_u-boot
- zc702_linux_trd
- zc702_secure_key_driver
- zc702_secure_key (includes xil_rsa_sign)
- zc702_multiboot
- zc702_jtag_en
- zc702_data
- zc702_udf

Introduction

Because of the value of intellectual property (IP), and because the incremental effort and cost to boot securely is small, secure boot should be used to boot Zynq devices. Secure boot of Zynq devices uses public and private cryptographic algorithms. This application note provides the concepts, tools, and methods to implement a secure boot. It shows how to create a secure embedded system, and how to generate, program, and manage the public and private cryptographic keys.

To build and boot a secure embedded Linux system quickly, skip to the section Booting the TRD Securely, and use the zc702_linux_trd system. Secure boot does not require programmable logic resources which are otherwise available to the user. The boot time of a secure Linux system is approximately the same as a non-secure system.

How to Read this Document

Figure 1 shows the secure boot topics discussed.

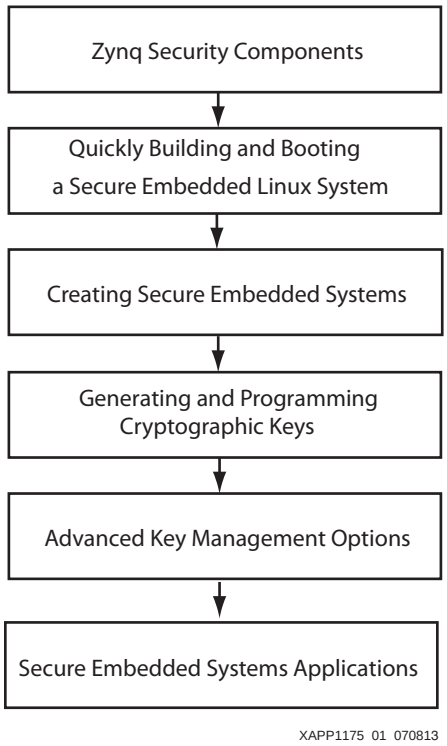


Figure 1: **Topics in Secure Boot**

The Boot Architecture, Boot Process, AES Encryption and RSA Authentication, Security in Embedded Devices, and Secure System Development sections provide background information on Zynq secure boot. Users familiar with booting Zynq devices on the zc702 board can skip to the Booting the TRD Securely section and quickly boot the zc702_linux_trd system.

The Building and Booting a Secure System section shows new users how to build and boot a secure system using the Xilinx graphical user interface (GUI). The system built is not used in other sections, so readers experienced with Xilinx tools can skip this section. RSA authentication is not available using the 14.6 Bootgen GUI. The Bootgen GUI that supports all security features is scheduled to be provided in the 14.7 release.

The Creating a Secure Boot Image section shows how to build custom secure embedded systems. A wide variety of use cases are supported.

The Generating and Programming Keys section shows how to create AES and RSA keys, and how to program the control functions and keys into Zynq devices for a secure embedded system.

The Advanced Key Management Options section shows how to protect RSA keys.

The Secure Embedded Systems Applications provides examples using the included reference systems. The Multiboot section develops systems which combine security and multiboot. The zc702_data system shows how to load data into Zynq devices securely. The zc702_jtag_en system discusses the use of JTAG after a secure boot.

Hardware and Software Requirements

The hardware requirements for the reference systems are as follows:

- ZC702 Evaluation Board with Revision C xc7020 silicon
- AC Power Adaptor (12 VDC)
- USB Type-A to USB Mini-B Cable (for UART communication)
- Xilinx Platform Cable or Digilent USB Type-A to USB Micro B cable for programming and debugging using JTAG
- Secure Digital Multimedia Card (SD) flash card
- Ethernet cable to connect the target board with host machine (optional)
- Xilinx Platform Studio 14.6
- Xilinx Integrated Software Environment (ISE) 14.6
- Xilinx Vivado® 2013.2

Note: The Xilinx Bootgen software, along with the First Stage Boot Loader, are the principle tools used in secure boot. Bootgen is provided in the Xilinx Software Development Kit (SDK) tool. SDK is independently downloadable from the Xilinx website. In the SDK 14.6 release, the GUI provided by Bootgen does not support all of the advanced functions of Bootgen. For the 14.6 release, the advanced functions must be implemented using the command line interface. The secure boot functions are provided independent of whether the system is developed using ISE or Vivado.

Boot Architecture

This section provides an overview of the hardware and software components used in the boot process.

Hardware Components Used in Boot

The two functional blocks in Zynq devices are the processing system (PS) and programmable logic (PL). The PS contains the ARM Cortex A9 MPCore and ten (x2) hard IP peripherals. The PL is the FPGA.

The hardware components used to boot are the CPU, system level control register (SLCR), non-volatile memory (NVM), secure storage, JTAG, AES/HMAC, On Chip Memory (OCM), Dual Data Rate Random Access Memory (DDR), and BootROM. Figure 2 is a diagram of the hardware components used in boot. The NVM and DDR memory are off chip. Booting typically uses only one NVM type.

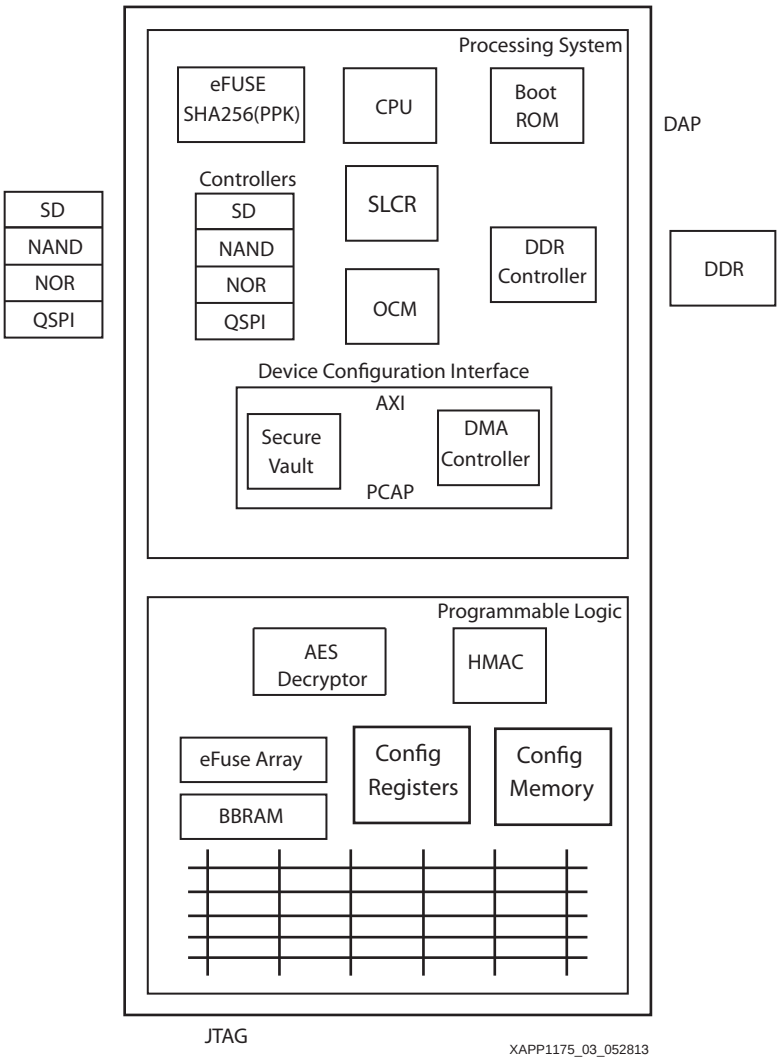


Figure 2: **Zynq-7000 AP SoC Hardware Components Used in Boot**

Central Processing Unit

The ARM Cortex-A9 MPcore contains two central processing units (CPUs). CPU0 is used for boot. The CPU controls boot and other operations by writing/reading registers in the Device Configuration (DEVCFG) and other System Level Control Registers.

System Level Control Register

The System Level Control Register (SLCR) consists of approximately 150 registers. The registers used in boot are the Boot Mode, PS Reset Control, FPGA Reset, Reboot Status, Reset Reason, PS Level Shifter, Control Register, Miscellaneous Register, Reboot Status Register, Lock, Configuration, and Interrupt registers. The registers for Direct Memory Access Controller (DMAC), NVM, and DDR controllers used in boot are also in the SLCR, but these generally do not require modification for boot.

Device Configuration Interface

The Device Configuration Interface contains the direct memory access controller DMAC used in boot. The DMAC transfers partitions from one memory, usually NVM, to another memory, usually DDR, at a high transfer rate. The DMAC interfaces to the PS using the AXI bus, and to the PL using the PCAP interface.

Secure Storage

Secure storage is on chip memory which is inaccessible to an adversary. The memory resides within the security perimeter of Zynq devices. At build time, the designer controls input/outputs (IOs) and internal switches to restrict access to Zynq device internal components. The OCM, L1 and L2 cache, AXI block RAM, PL configuration memory, BBRAM, and eFUSE array are secure storage in Zynq devices.

Nonvolatile Memory

The types of NVM used to boot Zynq devices are Secure Digital (SD), Quad Serial Peripheral Interface (QSPI), NAND, and NOR. The ZC702 and ZC706 Evaluation Boards support SD and QSPI, but not NAND and NOR NVM.

BootROM

The BootROM is 128K mask programmed boot Read Only Memory (BootROM) which contains the BootROM code. The BootROM is not visible to the user or writable. The BootROM code reads the Boot Mode Register, and initializes essential clocks and NVM at startup or power on reset. For all boot modes except JTAG, the BootROM code uses the memory controller to copy the FSBL partition from the specified NVM to the OCM.

On Chip Memory

The OCM is 256K random access memory (RAM). The initial function of the OCM is to store the first stage boot loader (FSBL) when the Zynq device is booted. The maximum allowable size of the FSBL is 192K. Since the OCM has no address or data lines at Zynq device pins, OCM is secure storage. The OCM can be used as secure storage for sensitive software after boot. OCM is very fast memory. After boot, the full 256K OCM is available.

AXI Block RAM

The AXI BRAM is PL RAM. It is not used in boot. It provides secure storage for sensitive software or data. AXI BRAM is used by both the ARM and MicroBlaze CPUs.

eFUSE Array

The PL eFUSE array is on chip one time programmable (OTP) NVM. The eFUSE array stores the 256-bit AES key. It is also used to control security functions, including enabling/disabling the JTAG port. The PS eFUSEs store the RSA_Enable bit and the hash of the Primary Public Key (PPK) used in RSA authentication.

Battery Backed Up RAM

The Battery Backed RAM (BBRAM) is an on chip alternative to eFUSE for nonvolatile AES key storage. BBRAM is reprogrammable and zeroizable NVM. BBRAM is NVM when an off-chip battery is connected to the Zynq device. The ZC702 board provides the battery, the Zed board does not. The BBRAM can be used to store the AES key when a battery is not attached, but it is volatile.

AES/HMAC

The Advanced Encryption Standard (AES) is used for private key encryption/decryption. The Hashed Message Authentication Code (HMAC) provides private key authentication using the SHA-256 hash function.

AES cryptography is used by Zynq devices to provide confidentiality. The Zynq device contains a hardened AES decryption engine which is coupled to the HMAC engine. The AES decryption/HMAC authentication cannot be decoupled. The SDK Bootgen tool encrypts the software in the software development process, at the manufacturing end. Decryption is done in the fielded embedded device. The AES decryption uses a private key programmed into either eFUSE or BBRAM.

JTAG/DAP Boundary Scan Chain

The JTAG chain is a boundary scan chain used by the PL. The DAP is a boundary scan chain used by the PS. The two chains can be cascaded or used independently. The JTAG chain is used to load PS and PL code, program the keys in eFUSE and BBRAM, and for debugging.

Software Components Used in Boot

The Xilinx ISE Design Suite is used for system development. The ISE Design Suite includes PlanAhead, Xilinx Platform Studio (XPS), Software Development Kit (SDK), Bitgen, Bootgen, and iMPACT. The Integrated System Environment (ISE) and Vivado design suites are used to implement VHDL/Verilog code. SDK is used to compile C code, generate a boot image, load the boot image, and debug the software and hardware.

SDK is used to create software projects, and download and debug the projects. SDK runs on a PC under Windows or Linux. The software programs which run on Zynq devices and are used in boot are the BootROM code, FSBL, ps7_init, U-Boot, and the DEVCFG code.

Boot Header

The Boot Header defines characteristics of the FSBL partition. The image ID and Header Checksum fields in the Boot Header allow the BootROM code to run integrity checks. The Encryption Status field specifies whether the FSBL is non-secure or secure, and if secure, whether the key source is eFUSE or BBRAM. The Boot Header format is provided in the *Zynq-7000 AP SoC Technical Reference Manual* (UG585) [Ref 1]. For additional information see *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 2].

Bitgen

Bitgen generates an unencrypted bit file for the bitstream partition. Bitgen is not used for encryption in Zynq devices.

Bootgen

Bootgen is a SDK tool which generates the image for booting. Bootgen generates the image which is loaded into NVM. Bootgen accepts a user generated Bootgen Image Format (BIF) file that lists the partitions which are to be included in the image. Bootgen outputs a single image file in MCS or BIN format. Bootgen encrypts and signs partitions, using AES and RSA algorithms respectively.

Note: The 14.6 Bootgen GUI in SDK provides limited support of security functions. In SDK 14.6, Bootgen is used at the command line for RSA authentication and/or mixed encrypted/unencrypted partitions. In a later SDK release, the Bootgen GUI will support RSA authentication and mixed encrypted/unencrypted partitions.

Secure Key Driver

The Secure Key Driver software programs the PS and PL eFUSE array. The Secure Key Driver runs on Zynq devices. If RSA authentication is used, the Secure Key Driver must be used to program the PS eFUSE array. If AES Encryption is used, th eFUSE Driver is an alternative to using iMPACT to program the AES key. The steps to use the Secure Key Driver are given in the Secure Key Driver section.

First Stage Boot Loader

The FSBL is the partition loaded into OCM by the BootROM code. The FSBL loads partitions (software programs, the bitstream) in the image, which is stored in NVM, to the partitions destination. The destination of software partitions is usually DDR, OCM, or AXI BBRAM. The destination of the bitstream is the PL configuration memory. Using the AES/HMAC engine and the RSA libraries, the FSBL controls the decryption and authentication process. Although the functionality of the FSBL meets most user load requirements, the source code is editable if there are custom requirements.

iMPACT

iMPACT is used to program FPGAs, including the PL, principally in development. The iMPACT tool programs the PL eFUSE array or BBRAM, including control parameters and the 256-bit AES key.

U-Boot

U-Boot is open source software that runs on Zynq devices. It is commonly used to load Linux. Other U-Boot functions include reading DDR memory, and erasing, reading, writing NVM. U-Boot is loaded by the FSBL or XMD. It is used in both non-secure boot, but is not required for either.

BootROM Code

BootROM code is masked programmed ROM code which runs at power-up and in some cases in a multiboot. The BootROM code determines the boot mode, initializes the memory controllers used in boot, and if in a boot mode other than JTAG, loads FSBL into the OCM.

Chain of Trust

Booting a device securely starts with the BootROM code loading the FSBL, and continues serially with the FSBL loading the bitstream and software. With a secure boot foundation established by the boot ROM code, the chain of trust is created by the successive authentication of all software loaded into the device. This prevents an adversary from tampering with software or the bitstream file.

Device Configuration (devcfg)

The devcfg is the Xilinx device configuration driver which uses the direct memory access controller (DMAC) to load the bitstream and software. Typical uses of the devcfg software are to load the bitstream from non-volatile memory (NVM) to random access or configuration memory.

Image

An image is a file which contains the bitstream and software which define Zynq's functionality. Typically, an image is loaded into NVM first. At power up, the image is copied from NVM into RAM and/or configuration memory as part of the boot process. An image consists of one or more, typically more, partitions. In addition to the bitstream and software partitions, the image contains header (boot, partition) information used to define the characteristics of the partitions and image.

Partition

Partitions are the individual PL bitstream and PS software (ELF, BIN) that comprise an image. Example partitions are `system.bit`, `fsbl.elf`, `hello_world.elf`, `u-boot.elf`, `uImage.bin`, `devicetree.dtb`, and `uramdisk.image.gz`.

Boot Image Format (BIF)

The BIF is the input file into Bootgen that lists the partitions (bitstream, software) which Bootgen is to include in the image. The BIF also includes attributes for the partitions. Partition attributes allow the user to specify if the partition is to be encrypted and/or authenticated.

ps7_init

The ps7_init command is an alternative to using an FSBL in a non-secure boot. The ps7_init command provides a simple method to initialize boot components during development when XMD is used.

Xilinx Microprocessor Debugger (XMD)

XMD is a Software Development Kit (SDK) software tool commonly used to load PL and PS partitions in development. In addition to loading partitions. XMD is used to quickly test device functionality. XMD uses the JTAG port, so it cannot be used in secure boot.

RSA

RSA is a public key algorithm used to authenticate software, including ELF, BIN, and BIT partitions. Authentication verifies that software has not been modified. In Zynq devices, each software partition can be individually authenticated. RSA uses a public/private key pair. The private key is used by Bootgen in signing the partition at the manufacturing facility. The public key is used in verifying the partition in the fielded Zynq device. In Zynq devices, the public/private key pair can be changed as often as desired, even on different partitions in the same image.

Software Development Kit

SDK is Eclipse based software which is downloadable from the Xilinx website. In addition to software development, SDK supports creating images, downloading software and the bitstream into the Zynq device, writing the image into QSPI, and debugging software programs.

Boot Process

This section provides an overview of boot modes, boot steps, boot flows, and maintaining security after boot. Following this, the software used in boot, including the BootROM code, FSBL, and U-Boot, is discussed.

Boot Modes

The boot modes are PS Master Non-secure, PS Master Secure, and JTAG. The master modes use QSPI, SD, NAND, or NOR NVM.

Secure Boot Steps

Figure 3 shows the steps to develop a secure embedded system.

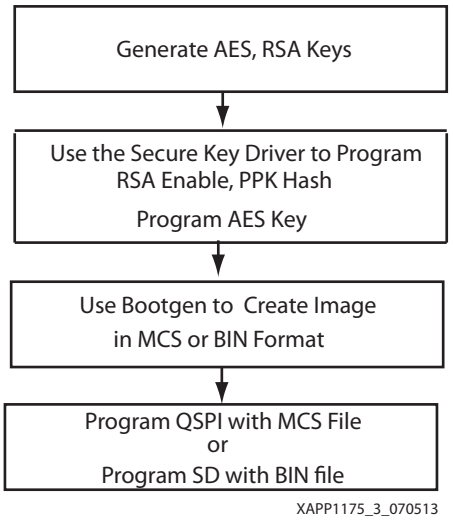


Figure 3: **Steps in Developing a Secure System**

Boot Flows

To boot the Zynq device, software (FSBL, U-Boot) is developed on a PC using SDK, and the image is created by Bootgen. Running on Zynq devices, the FSBL loads the software used by Zynq devices.

Two distinct load operations are required: loading an image into NVM, and copying partition(s) from NVM to DDR (or OCM). SDK's Flash Programmer, zynq_flash or U-Boot load an image into QSPI.

If SD is used, a BIN image is written to the SD card. This uses a SD card reader/writer which is connected to the PC with a USB cable. The SD card goes into the ZC702 Evaluation Board SDIO slot (J64).Copying the image from the SD card or QSPI is done by FSBL or U-Boot. In loading NVM, all partitions are typically loaded into flash or SD. Booting Zynq is commonly a two step process, with the FSBL loading the bitstream file and U-Boot partitions, and U-Boot loading the remaining partitions. The remaining partitions are usually Linux partitions, including Linux applications.

While it is common for U-Boot to load Linux and Linux applications, in most of the use cases in this application note, the FSBL loads U-Boot, Linux, and the Linux applications. U-Boot is still loaded because it is used for functions other than loading Linux. Using the FSBL to load Linux partitions allows the user to specify whether each partition is encrypted or authenticated. This capability will be provided in a future release of U-Boot.

If the boot mode pins specify JTAG, the BootROM code enables the JTAG port. XMD is used to load and run software. In the JTAG boot mode, the FSBL displays a message that JTAG boot mode is used. The FSBL does not load partitions when JTAG mode is used. In a non-secure JTAG boot, either the FSBL or ps7_init initializes boot components.

BootROM code

The BootROM code does the initial setup at boot. If the boot is a Master Secure or Master Non-Secure boot, the BootROM code initializes the NVM controller specified by the boot mode register, parses the boot header, and copies the FSBL to OCM.

Figure 4 shows the BootROM code flow. The BootROM code reads the boot mode register to determine if a master or slave boot mode is used, and if master, the type of NVM used. The BootROM code reads the Boot Header to determine whether the boot is non-secure or secure, and if secure, whether the key source is BBRAM or eFUSE. If there is a key mismatch between the key source specified in the PL eFUSE array and the key source specified in the boot header, the BootROM code transitions the Zynq device to a secure lockdown state. If the BootROM code determines that the device is in an illegal boot mode based on its state, the BootROM code transitions the Zynq device to a secure lockdown state. An example of an invalid state is a Boot Header in which the Encryption status field specifies encryption using BBRAM and the PL eFUSE array specifies an *eFUSE only* key source. In a secure boot, the BootROM code executes proprietary tests to ensure security before it authenticates the FSBL.

In the eXecute In Place (XIP) mode, the CPU runs code directly from NVM rather than DDR. The XIP mode cannot be used in secure boot.

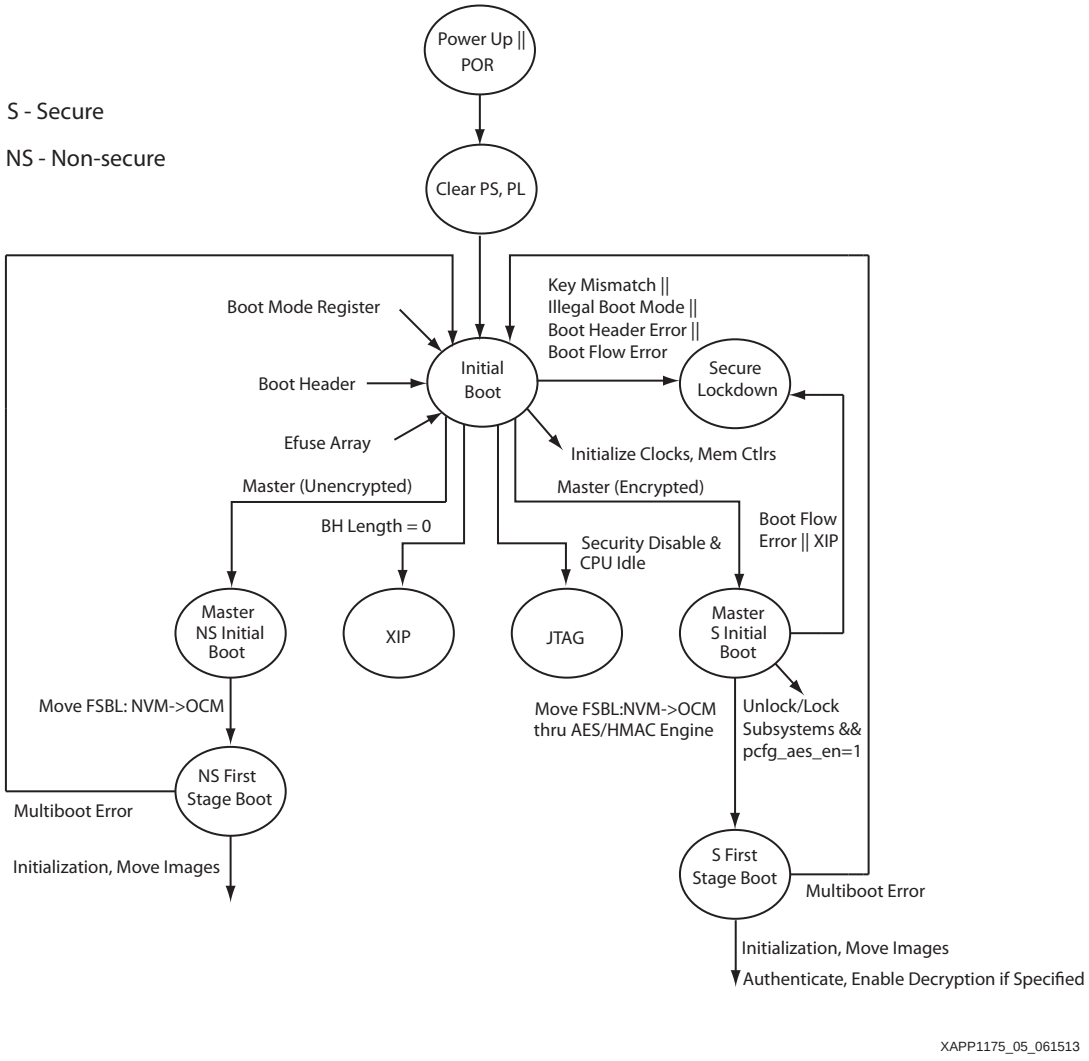


Figure 4: **BootRom Code Flow diagram**

First Stage Boot Loader

The first stage boot loader (FSBL) is loaded into OCM by the BootROM code. The FSBL is closely aligned with Bootgen in that it reads the partitions in the image created by Bootgen. The principle function of the FSBL is to copy partitions from NVM to DDR and PL configuration memory. If the partition is encrypted, the partition is routed to the AES/HMAC engine for decryption before it is loaded in DDR or other destination address. If the system.bit is in the image, the FSBL transfers the system.bit into the PL configuration memory. It then transfers the secondary storage boot loader (SSBL) or application partition(s) to their destination address, typically DDR. The FSBL can load multiple ELF files.

A second method of loading partitions is for the FSBL to load u-boot.elf, and U-Boot loads software partitions.

Prior to loading partitions, the FSBL completes the initialization of the device started by the BootROM code. The MIO, clocks, and DDR controller are initialized.

The FSBL supports most user's software load requirements. In some cases, users need to edit the FSBL source code to meet additional load or functional requirements. As an example, the User Defined Field in the Authentication Certificate can be used for a function such as defining the software version being loaded. To support this, the FSBL code would require edits which

check that the correct software version is loaded. The FSBL source code is in the src directory of the zynq_fsbl_0 software project.

Note: The FSBL code most likely to be edited is in `main.c` and `image_mover.c`

FSBL hooks provide a framework to plug in user defined functions. An example use of the FSBL hooks is to initialize PL cores after a bitstream is downloaded. The FSBL hook functions in fsbl_hook.c are:

- FsblHookBeforeBitstreamDload: Provides a region for FSBL edits before bitstream download
- FsblHookAfterBitstreamDload: Provides a region for FSBL edits after bitstream download
- FsblHookBeforeHandoff: Provides a region for FSBL edits before the FSBL hands off to the SSBL or an application.

The Targeted Reference Design (TRD) system in zc702_linux_trd provides an example of FSBL edits in initializing an I2C. In XAPP1078 Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors, the FSBL searches for additional partitions to load.

Figure 5 shows a flow chart of the FSBL.

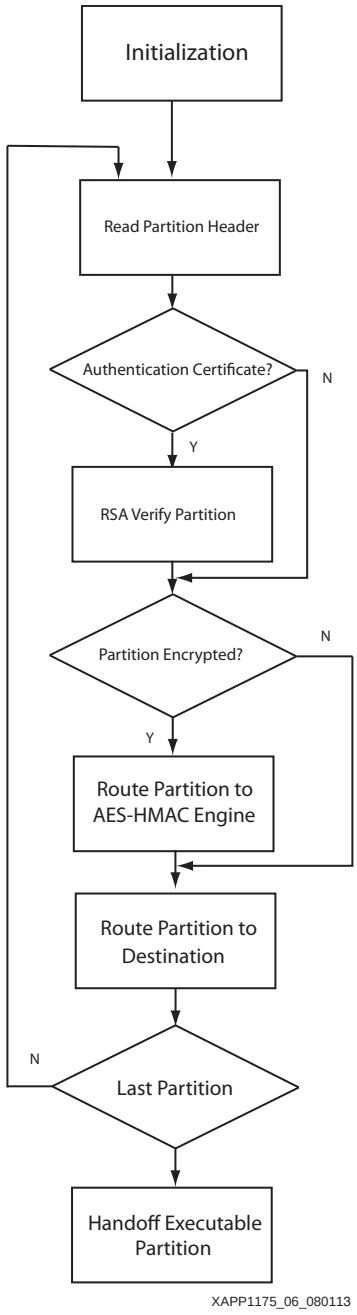


Figure 5: **FSBL Flow**

The FSBL parses the boot image to determine if the image is to be RSA verified and/or AES decrypted. If the partition is AES encrypted, the FSBL routes the partition to the AES/HMAC engine, and then to its final destination. If the partition is RSA authenticated, the FSBL reads the authentication certificate (AC) to verify the partition. The AC contains the public key and the signature.

The FSBL uses the sha256 and rsa2048_ext library functions to verify the partition. The sha256 and rsa2048_ext compiled functions reside in `$XILINX_EDK/lib`. The FSBL parses the image in NVM, executing these steps:

- Verify the Secondary Public Key (SPK) using RSA
- Verify the partition using RSA

The RSA functions called in the FSBL code are conditionally executed based on the existence of partition authentication certificates in the image.

U-Boot

U-Boot is an open source bootloader commonly used in embedded systems. U-Boot performs similar functions to the FSBL. U-Boot has additional functions, such as reading and writing NVM and DDR. The zc702_uboot reference system provides a system with U-Boot. U-Boot typically runs in DDR, not OCM. The wiki.xilinx.com site provides information on configuring and building U-Boot. In addition to loading Linux from NVM to DDR, U-Boot is used to read DDR, and to erase, write, and read NVM. The U-Boot erase, read, and write operations on QSPI are an alternative to the SDK Flash Writer, which runs on a PC. U-Boot runs on Zynq devices.

U-Boot can run interactively, providing a zynq-uboot prompt, or it can run automatically at power up. The `zynq_common.h` file in the include/configs directory contains options which set U-Boot functionality. After configuration edits, U-Boot must be re-compiled as described in wiki.xilinx.com. For development, configure U-Boot with a 5 second delay. For production, particularly for secure boot, re-configure with a 0 delay, and rebuild U-Boot.

U-Boot support is an active area of development at Xilinx, including adding the authentication/decryption features currently in the FSBL to U-Boot, and adding support for additional NVM configurations.

AES Encryption and RSA Authentication

Bootgen and FSBL software support AES encryption, HMAC authentication, and RSA authentication. RSA is effective for authentication. AES is more efficient than public key cryptography in encryption. Private keys are used in AES encryption and HMAC authentication, and private/public key pairs are used in RSA authentication. For RSA authentication, Bootgen signs partitions and the BootROM code and the FSBL verifies partitions.

The private/public key pair used in RSA authentication have significant security advantages over cryptography which only uses private keys. In RSA, the private key is used at the manufacturing facility which usually has physical security. The public key is loaded into the embedded device. If an adversary steals the embedded device and extracts the public key, the damage is limited. The RSA key pairs can be changed as often as needed. Changing the key reduces the risk that the key is compromised, and reduces the vulnerability of the IP the key is protecting.

Zynq devices provides a silicon based AES/HMAC engine which decrypts/authenticates at 100 MB/s. The AES/HMAC engine does not encrypt. AES-256 is used for encryption/decryption and HMAC is used for private key authentication. AES encryption is done by Bootgen. The AES and HMAC functions in the AES/HMAC engine cannot be used independently to decrypt / authenticate partitions. AES is a symmetric cryptographic algorithm which uses a private 256 bit key. The HMAC key is a 256 bit private key.

The RSA asymmetric cryptographic algorithm in Zynq devices uses a 2,048 bit modular. Modular is the generally accepted description of the key length. The BootROM code authenticates the FSBL partition, and the FSBL authenticates the partitions it loads. The BootROM code and FSBL use the identical RSA algorithm.

Since Bootgen signs partition(s) and the BootROM code and the FSBL verify the partitions, Bootgen, BootROM code, and FSBL software must agree on the image format. For each partition authenticated, an authentication certificate (AC) field in the image is used for RSA authentication.

Figure 6 shows the interaction between Bootgen and the FSBL. Bootgen runs on a PC, and the FSBL runs on Zynq devices. For each partition, Bootgen executes the cryptographic functions in the order shown. Similarly, in loading each partition, the FSBL executes the cryptographic functions in the order shown.

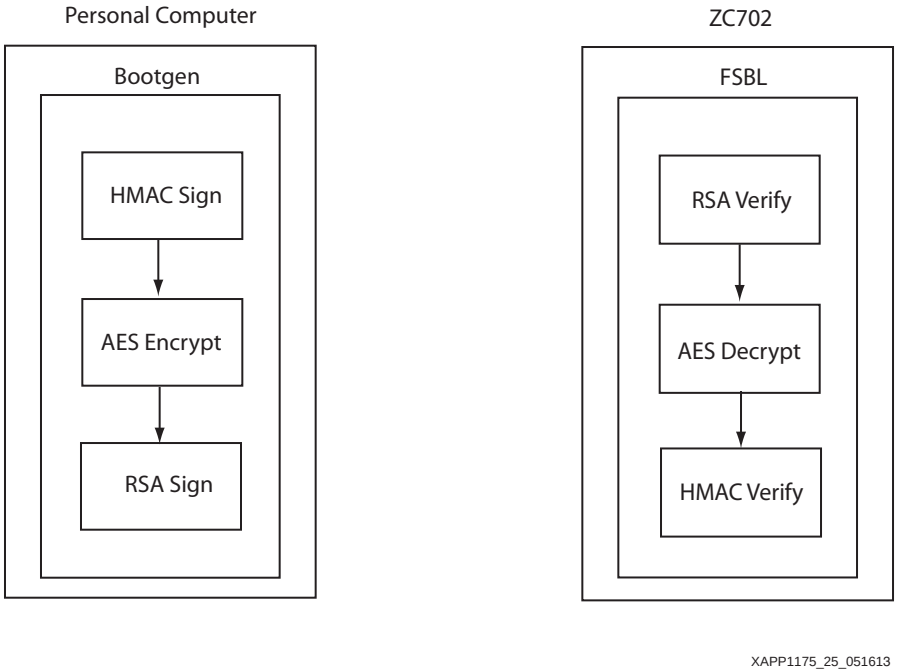


Figure 6: **FSBL - Bootgen Interaction**

In Bootgen, the HMAC is generated first, followed by AES encryption, followed by RSA signing. In Zynq devices, these steps are reversed: the partition is RSA authenticated, AES decrypted, and then HMAC authenticated.

In RSA authentication, the partition is not signed. Instead, a hash of the partition is generated, using a SHA256 function. The SHA256 hash is a one way function which produces the same size output, independent of whether the partition size is 1,000 or 1 MB. The hash is signed using the private key. For each partition which is RSA authenticated, Bootgen writes an Authentication Certificate (AC) which contains the keys and the signatures for the partition.

In the ISE 14.7 release, U-Boot will add capability to RSA verify partitions using the same RSA library used by the FSBL.

Security in Embedded Devices

Security should be considered at the beginning of embedded device development, starting with identifying potential threats. Potential threats to an embedded device are provided in the following list:

- Theft of the Embedded Device
- Privacy of the data in the Embedded Device or System
- Cloning of the Embedded Device
- Denial of Service
- Insertion of malware to change the behavior of the Embedded Device
- Insider Providing Key to Adversary

Figure 7 is a boot flow in which authentication is used to load partitions in a chain of trust. The BootROM code loads the FSBL. The FSBL and U-Boot bootloaders load the hardware and software partitions. The principle objectives in a secure boot are to prevent an adversary from loading a modified partition, and to keep proprietary partitions confidential.

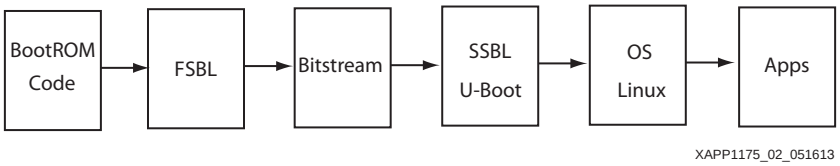


Figure 7: **Chain of Trust in Secure Boot**

In Bootgen, users define which partitions are encrypted, and which partitions are authenticated using a RSA private/public key pair. Authenticating all partitions in a chain of trust ensures that only partitions which have not been tampered with are loaded.

An important part of embedded device security is key security. An advantage of RSA is that the private key is not loaded into the device. A second advantage is that different RSA keys can be specified for each partition, and the RSA key can be changed when partitions are updated. Changing the key limits the time an adversary has to attack the key, and limits the information that is vulnerable.

To facilitate key security, Bootgen provides the ability to handle RSA keys securely, limiting access to an Infosec staff. Since only the Infosec staff has access to keys used in the final embedded product, this reduces the threat of an insider attack.

Zynq devices also provide security by integrating a large amount of software and hardware IP within its security perimeter. A combination of build options and software allows the protection of IP within the security perimeter. Additionally, Zynq devices have a relatively large amount of secure storage available for sensitive program and data storage. Since this storage is not large enough to hold the Linux OS, system partitioning of sensitive and non-sensitive functions is required. Linker script and BIF attributes allow open source code to run from DDR and sensitive applications to run from on chip secure storage.The Secure Key Driver section provides a linker script which locates code from On-Chip Memory (OCM).

Embedded systems are commonly attacked after a secure boot (i.e., during operation). The loaded software, such as the operating system, should not allow an adversary access to Zynq's hardware or software resources.

In a secure boot, all partitions are loaded in a chain of trust. After the transition from the boot stage to an operational stage (i.e. when Linux applications have been loaded), the OS must maintain the system as secure. In a non-secure boot, U-Boot and Linux applications can load the bitstream. After Linux is loaded, an application can use the devcfg driver to load a bitstream.

Restated, the OS should not allow access to the devcfg driver to non-trusted users/applications. The most direct method to do this is to keep the module out of the kernel build. Linux has supervisor/user modes. If the devcfg driver is included in the Linux build, the supervisor needs to restrict access, requiring passwords for users and limiting the devcfg file permissions to only the supervisor.

The PS-PL architecture provides the user with the ability to provide redundancy in recovering from operational failure in either the PS or PL. The PS can monitor the PL for a tamper event triggered by single event upset (SEU) activity. The PL can monitor the PS using a security monitor.

In addition to secure boot, embedded device security requirements may include anti-tamper and information assurance. Xilinx sells a high-end, fully tested security monitor IP which executes from the PL. Alternatively, a user can develop a security monitor which does not consume PL logic resources. *Using the Zynq-7000 Processing System to Xilinx Analog to Digital Converter Dedicated Interface to Implement System Monitoring and External Channel Measurements* (XAPP1172) [Ref 3] is recommended as a good start in developing a lite security monitor.

Anti-tamper (AT) is discussed in XAPP1084 Developing Tamper Resistant Designs with Xilinx Virtex Series 6 and 7 FPGAs. Information assurance (IA) is discussed in *Solving Today's Design Security Concerns* (WP365) [Ref 4].

Secure System Development

The steps in Creating a Project Using Xilinx Platform Studio are used to initially test a basic secure system. Zynq devices have many security options not discussed in this section. Figure 8 shows a typical secure development process which allows users to incrementally learn to use Zynq device security features. This approach is used because once the eFUSE key and *eFUSE only* control bit are programmed, there is no returning to using the BBRAM key if the eFUSE key is lost. Also, once the RSA Enable eFUSE is programmed, a board cannot be used without at least FSBL authentication in the master mode.

The process described in this section is not required, and is presented because Zynq devices provide many security options with RSA and AES/HMAC. After starting with a non-secure design, a subsequent step is a secure design using a BBRAM AES key. The BBRAM AES key is reprogrammable. The next step is to enable RSA authentication. The RSA Enable eFUSE control bit and the hash of the PPK are programmed into the PS eFUSE area using the Secure Key Driver. BBRAM can still be used as the source of the key for AES decryption.

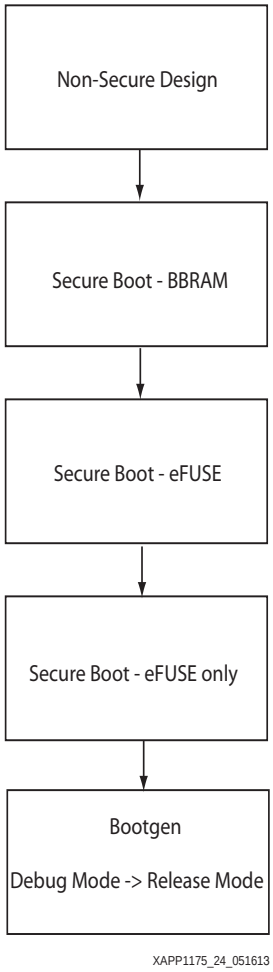


Figure 8: **Using Zynq Security Options**

The next step is to use the eFUSE key for AES decryption. At this stage, a non-secure boot, RSA authentication, secure boot with BBRAM AES key, and a secure boot with an eFUSE key are possible. The board must be powered down to change the AES key source. If the RSA Enable bit is programmed, the FSBL must be authenticated. An RSA enabled "non-secure" boot differs from the un-authenticated non-secure boot.

Key Swapping eFUSE and BBRAM keys

Keys can be programmed in either eFUSE or BBRAM NVM. The advantages of BBRAM is that it can be reprogrammed, and it can be erased if there is a tamper event. The eFUSE array control bits **eFUSE Secure Boot** and **BBRAM Key Disable** prohibit swapping between the eFUSE and BBRAM key. If these bits are not programmed, either key source can be used after a power down. If **BBRAM Key Disable** is programmed but **eFUSE Secure Boot** is not programmed, a non-secure boot or a secure boot using the eFUSE key can be done. To use only the eFUSE as the AES key source, the **eFUSE Secure Boot Only** bit is programmed in the PL eFUSE control array. The PL eFUSEs can be programmed either with iMPACT or the Secure Key Driver. The Secure Key Driver section shows how to program this functionality.

The last option in the figure, Bootgen Release mode, is used by the Infosec staff for the production release of the secure embedded device. In this stage, Bootgen Release mode can be used to increase the security of the RSA private key. This is discussed in the Advanced Key Management Options section.

Booting the TRD Securely

The source files for booting the ZC702 Base System TRD quickly are provided in the zc702_linux_trd reference system. This section provides an example, the Target Reference Design, which shows that the creation of a secure boot image is straightforward. This section uses pre-existing keys in the reference design systems. There is a section on Generating Keys later in this document.

Note: Using the following step, the PS eFUSEs are blown. After this, in all subsequent tests using the ZC702, at least one partition must be authenticated.

Use the following steps to boot the TRD securely.

1. Setup the ZC702 board. See the Setup the ZC702 Evaluation Board section. Set the Boot mode switches to JTAG boot mode.
2. Invoke the Teraterm communication terminal and set Baud Rate = 115200, Data = 8 Bits, Parity = None, Stop Bit = 1, Flow Control = None.
3. Change to the `xapp1175/zc702_secure_key_driver/ready_for_download` directory. This directory contains the `ps_secure_key_read.elf` and `ps_secure_key_write.elf` files. If the eFUSEs are programmed, run the `ps_secure_key_read.elf` in step 4. If the eFUSEs need to be programmed, and it is acceptable to use the Xilinx provided keys for the evaluation board, run the `ps_secure_key_write.elf` in step 4.
4. Run **xmd** at a command prompt or from SDK, and run the following:

   ```
   connect arm hw
   source ps7_init.tcl
   ps7_init
   ```

   then either:

   `dow ps_secure_key_write.elf` (if programming the eFUSEs)

   or

   `dow ps_secure_key_read.elf` (if reading the eFUSEs)

   `con`
5. Optionally, verify that the hash of the PPK displayed in the communication terminal matches the values in `xapp1175/zc702_efuse_driver/ready_for_download/hash_ppk.txt`.
6. Change to the xapp1175/zc702_linux_trd directory.
7. Run bootgen `-image zc702_linux_trd.bif -o zc702_linux_trd.mcs -encrypt bbram`.

Note: In the SDK 14.6 release, this command must be run on Linux. Windows support will be provided in SDK 14.7.

8. Invoke SDK. Select the workspace as xapp1175/zc702_linux_trd/SDK.
9. In SDK, enter **Xilinx Tools > Program Flash**.
10. Specify the image by browsing to `xapp1175/zc702_linux_trd/zc702_linux_trd.mcs`. Set the offset to **0x0**. Set the flash type to **QSPI Single**. Click **Program**.
11. On the ZC702 Evaluation Board, change the boot mode switch to **QSPI boot mode** by moving the J25 jumper to 1 (or 4 if the evaluation board uses the single switch).
12. Power cycle. Verify that Zynq boots to the Petalinux prompt. To login, use **root** for the user name and **root** for the password.
13. To compare non-secure and secure boot time, repeat step 7-step 12. When re-running step 7 for a non-secure BIF, eliminate the -encrypt bbram argument, but program QSPI instead with `zc702_linux_trd_ns.mcs`.

Building and Booting a Secure System

This section provides the steps to develop a Zynq system using the GUIs provided by Xilinx Platform Studio (XPS) and SDK. The steps for AES key generation and creating non-secure and secure images are provided. RSA authentication is not supported with the GUI in 14.6. The steps for setting up the ZC702 evaluation board are given. The zc702_uboot system is booted using JTAG mode. This is followed by non-secure and secure boots using ZC702 SD and QSPI memory.

The required tasks for a secure boot are:

- Create the Zynq hardware and software using Xilinx software
- Use Bootgen Advanced Tab to generate a secure image
- Use Bootgen to generate the AES key
- Use iMPACT or the Secure Key Driver to program the AES key to either BBRAM or eFUSE
- Load the SD Card or Program QSPI flash on ZC702 Evaluation Board

This section provides an introduction to developing a system and using the Bootgen GUI. The ISE Design Suite 14.6 Bootgen GUI does not support RSA authentication or mixed encrypted/unencrypted partitions. Bootgen supports mixed encrypted/unencrypted partitions, user selectable security on partitions, and RSA authentication when run at the command line.

Note: The steps in this section only work on a board on which the RSA Enable has *not* been programmed. This design files for this section are meant to be created by the reader. The design files in the zc702_uboot system are for a board in which RSA_Enable has been programmed.

Details on advanced security functionality are provided in later sections. The later sections do not require the files produced in this section. Starting with the 14.6 Release, Xilinx recommends that users transition to Vivado for design entry. Because boot uses SDK, any method of developing the software and bitstream partitions is acceptable.

Creating a Project Using Xilinx Platform Studio

In this section, Base System Builder (BSB) is used to create the zc702_uboot project. The system includes the UART1 and GPIO IP in the PS and the GPIO switch, GPIO LEDs, AXI BRAM, and AXI Timer IP in the PL.

The zc702_uboot system contains the MHS, XMP, and UCF files. This allows systems to be built with new releases of Xilinx software. If using this section to create a new project, create a new project directory.

1. Invoke XPS, by entering **ISE Design Suite 14.6 > EDK > Xilinx Platform Studio** if using Windows, or entering **xps &** if using Linux. In the Getting Started window, click **Create New Project Using Base System Builder**. Browse to the zc702_uboot directory. Verify that the **File Name** box contains `system.xmp`. Click **Save**. Figure 9 shows the creation of a BSB project. Enter a project name in the **Project File** dialog box and click **OK**.

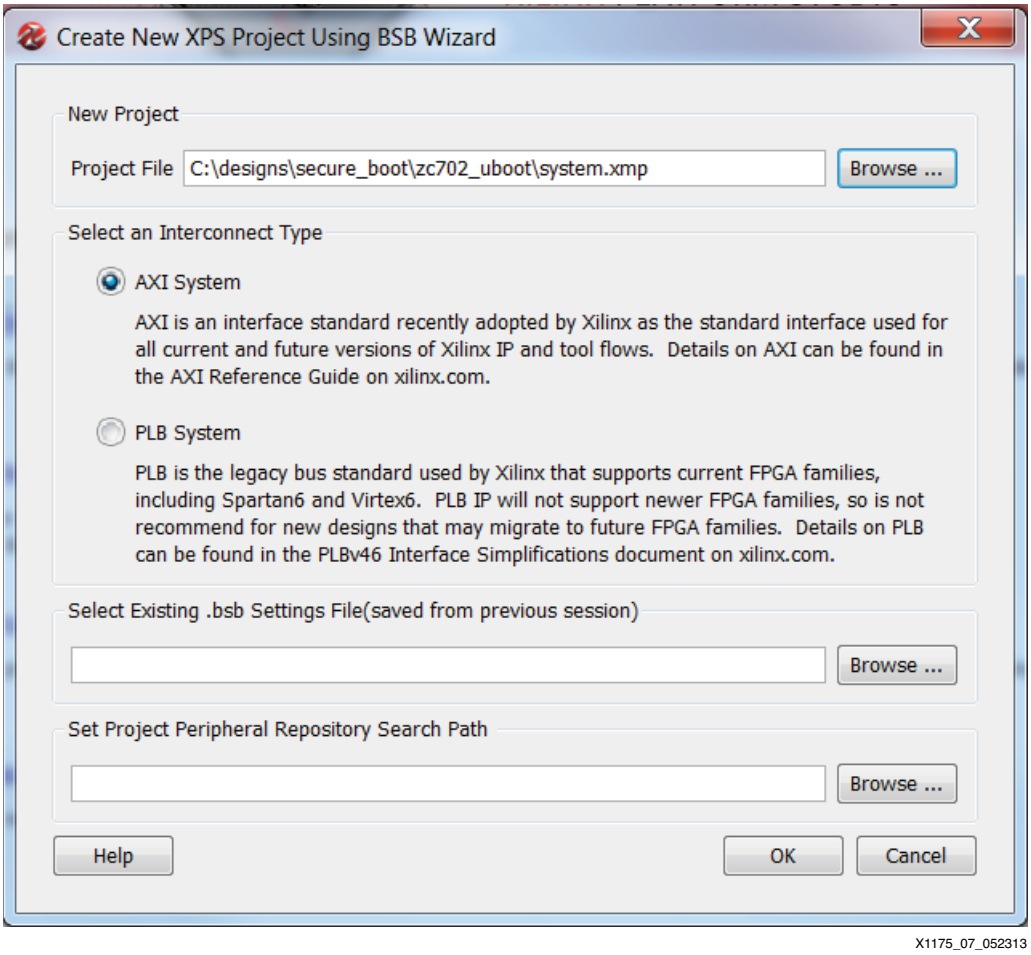
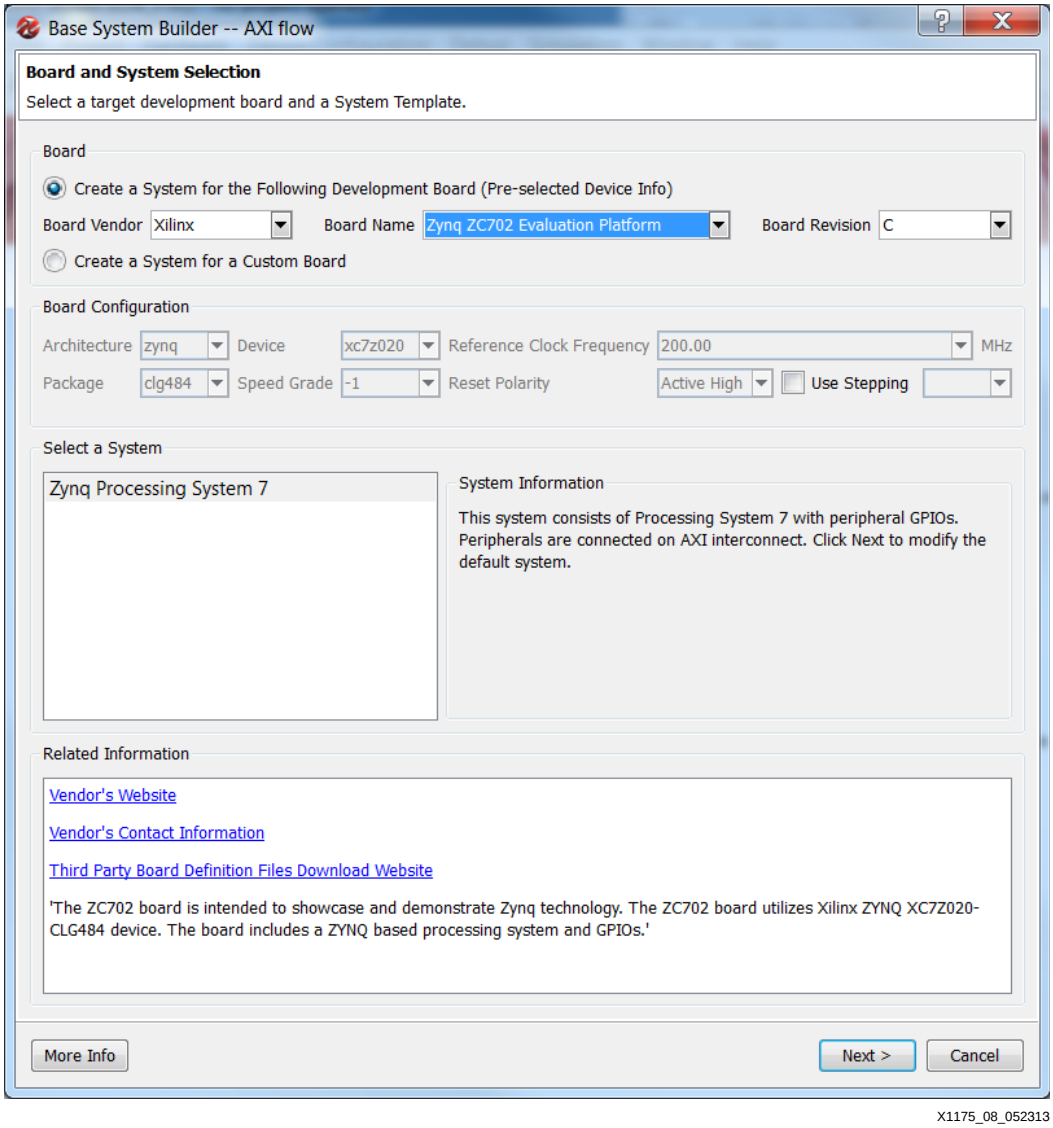


Figure 9: **Creating the zc702_uboot Base System Builder Project**

2. As shown in Figure 10, specify **Zynq ZC702 Evaluation Board** in the **Board Name** field. Click **Next**.



X1175_08_052313

Figure 10: **Selecting the ZC702 Evaluation Platform**

3. As shown in Figure 11, highlight `axi_bram_ctl` in the Available Peripherals pane and click **Add** to add the AXI BRAM intellectual property (IP). The AXI BRAM is PL IP. Select the 64K size. Repeat this Add Peripheral step by selecting axi_timer and clicking **Add**. Check **Use Interrupt for the AXI Timer**. Click **Finish**.

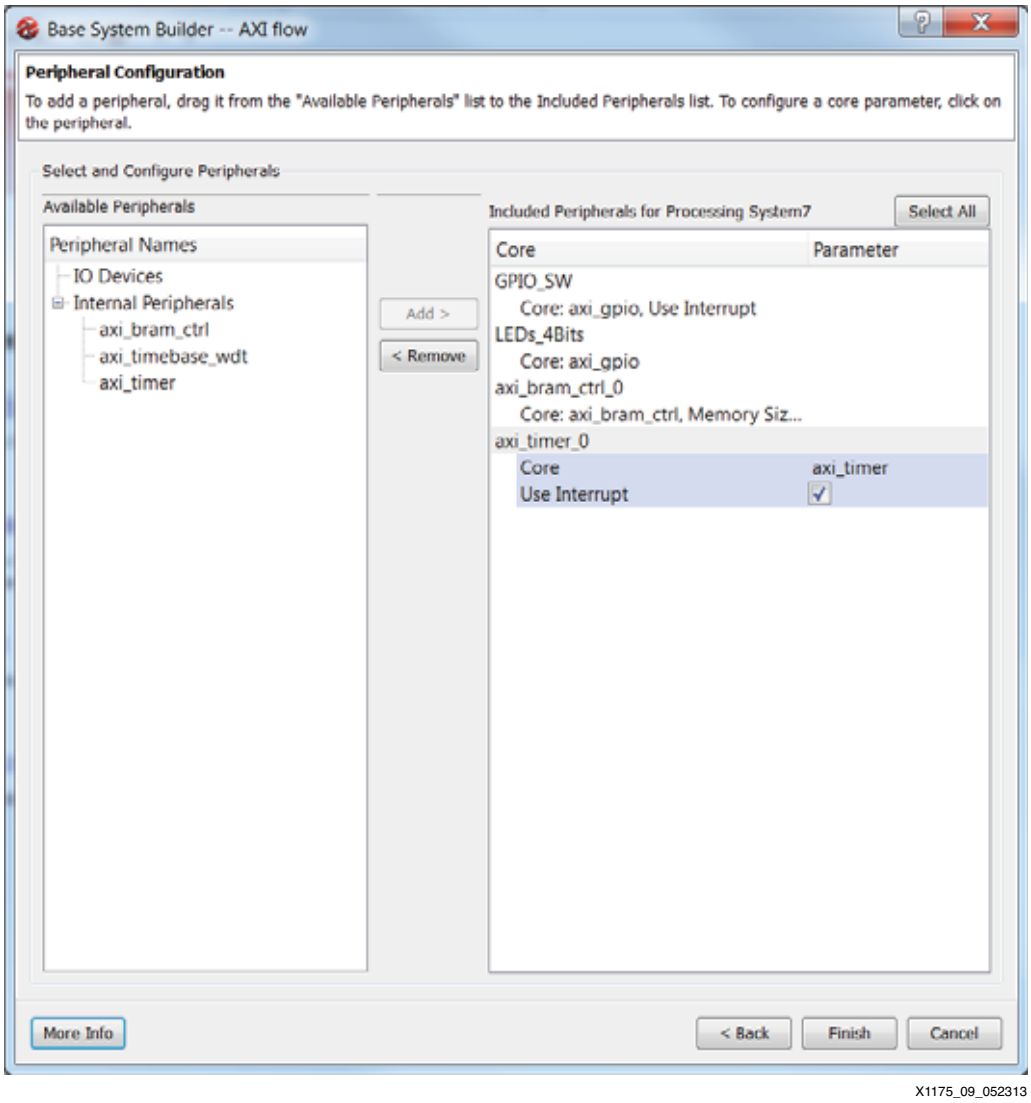


Figure 11: **Adding AXI BRAM Intellectual Property to PL**

4. As shown in Figure 12, enter **Project > Export Hardware Design to SDK** to invoke the project and launch Software Development Kit (SDK). Select "**`Include bitstream and BMM file`**." If run with the default option to create the bitstream, this step can take 10 minutes. A status icon is displayed in the lower-right corner of XPS during this time.

   When the hardware is exported and SDK is invoked, the Workspace Launcher dialog box prompts for a workspace name. Set the workspace to the **zc702_uboot/SDK** directory. Click **OK**.

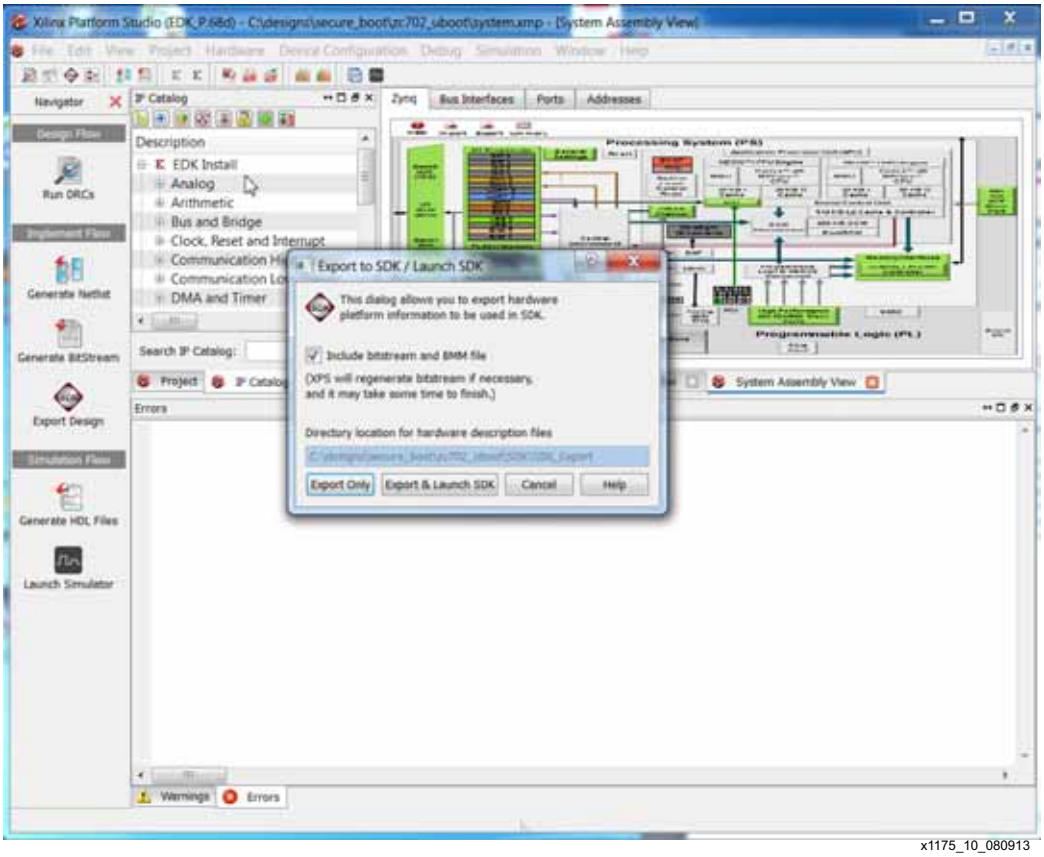
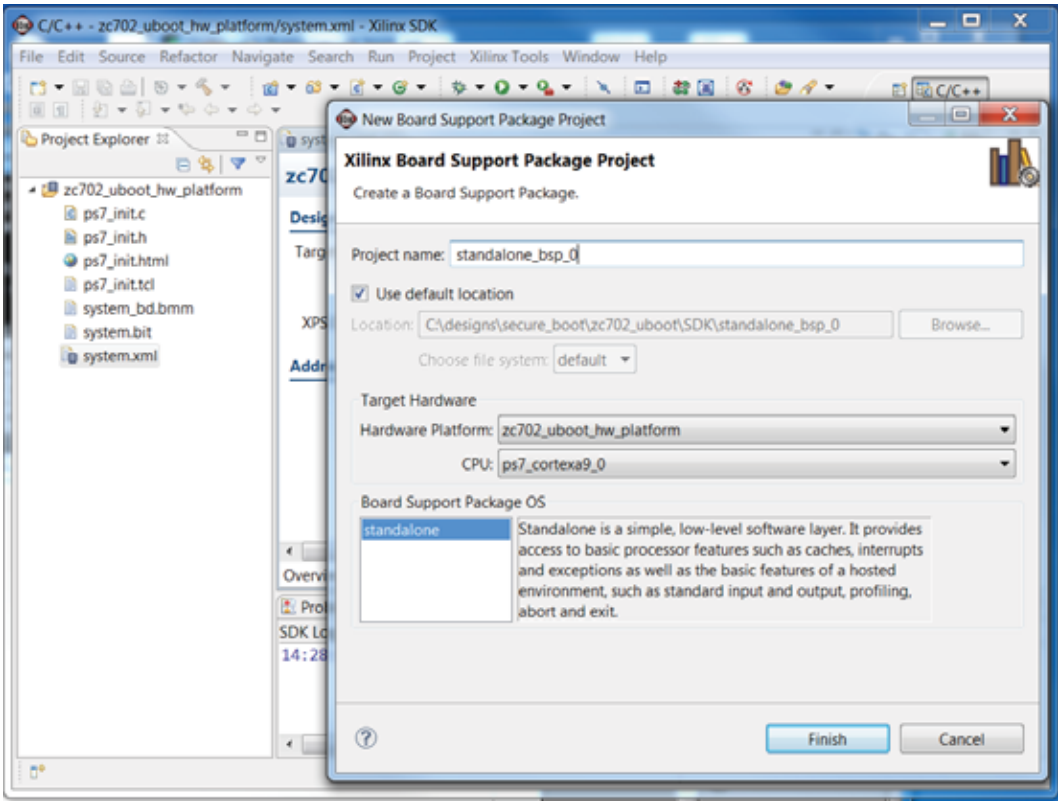


Figure 12: **Exporting Hardware Design to SDK**

5. As shown in Figure 13, select **File > New > Board Support Package**. Select **Standalone** and use default options to create a Board Support Package (BSP). Click **Finish**. When the **Board Support Package Settings** dialog box is displayed, click **OK** without selecting any libraries. SDK builds the board support package.



X1175_11_053113

Figure 13: **Creating a Board Support Package**

6. Create the FSBL project. Select **File > New > Application Project**. The **Application Project** dialog box is displayed. Enter **fsbl** as the project name. Select **standalone_bsp_0** from the **Use existing** Board Support Package option. Click **Next**. The **New Project Templates** dialog box is displayed. Figure 14 shows the creation of the FSBL project. Select **Zynq FSBL** and click **Finish**.

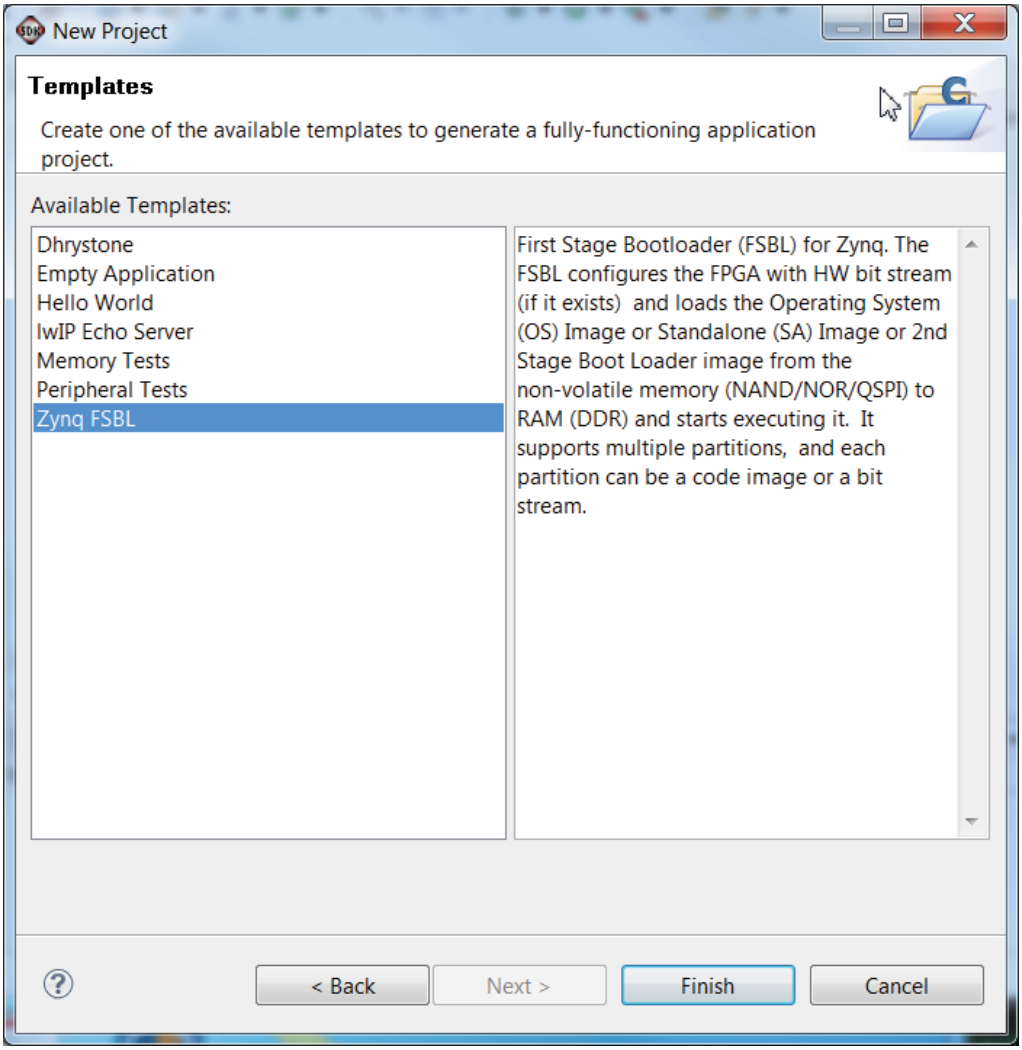
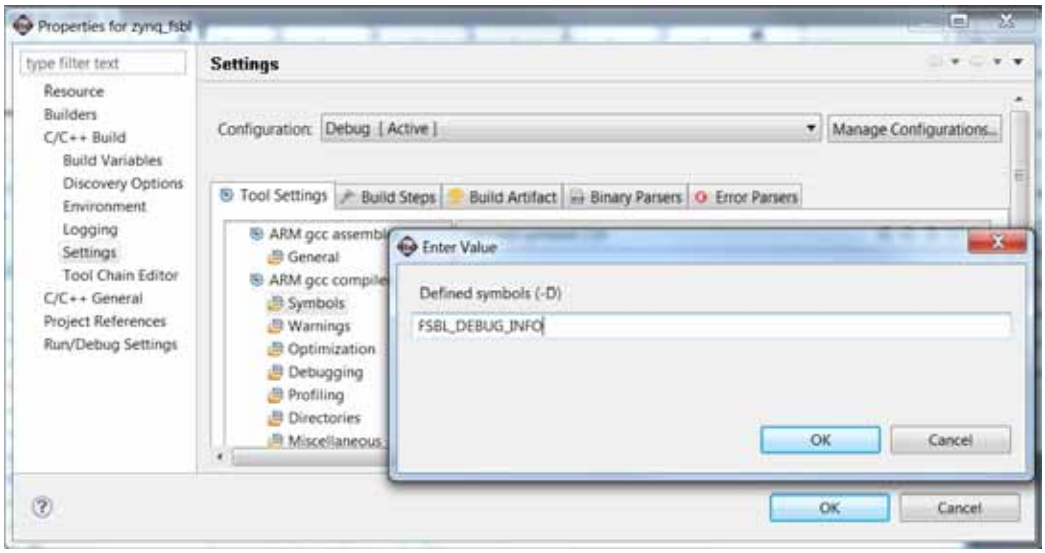


Figure 14: **Creating the First Stage Boot Loader (FSBL) Software Application**

7. Right-click on the FSBL in the **`Project Explorer`** pane and select **`Properties`**. As shown in Figure 15, edit FSBL compilation options so that debug information is displayed in the SDK or communication terminal window. With the Debug options used in compilation, FSBL provides useful information on the partitions in the image. If the SD or QSPI boot modes are used, the debug information is useful. If JTAG boot mode is used the FSBL does not copy partitions, and therefore information is not provided.

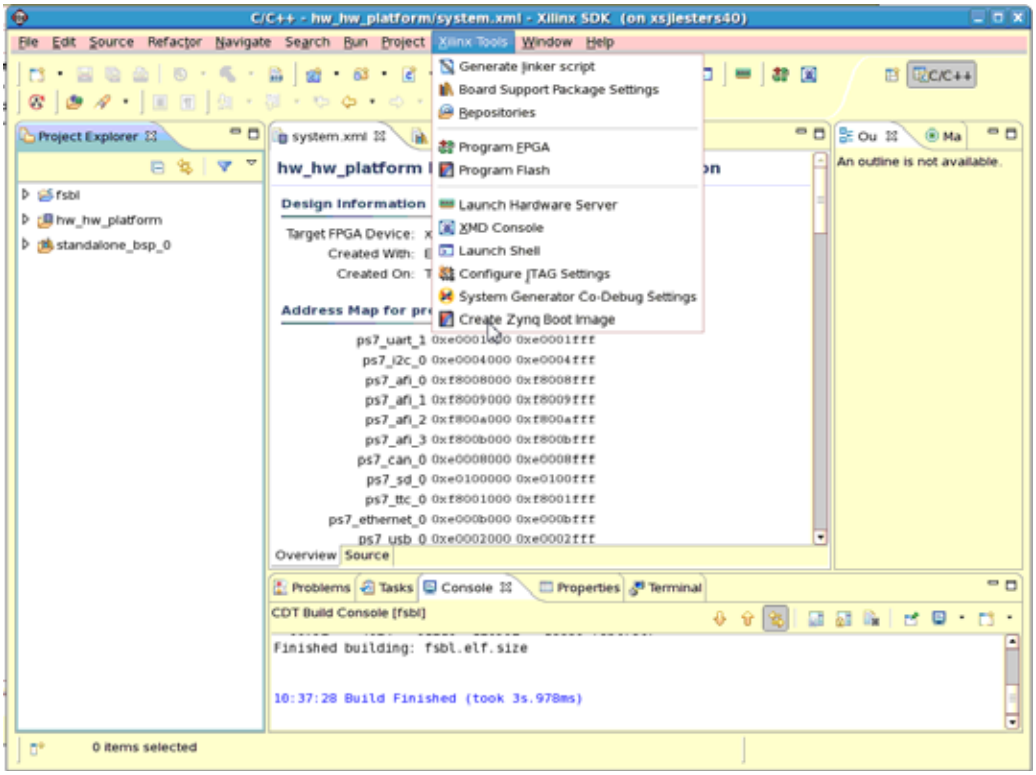
   To view details of the boot process in a FSBL debug log file, select **C/C++ Build > Settings > ARM gcc compiler > Symbols** and compile using **DEBUG**, **FSBL_DEBUG_GENERAL**, and **FSBL_DEBUG_INFO** symbols. In the **Defined Symbols** pane, click the "**+**" icon to iteratively select the symbols. The figure shows the entry of **FSBL_DEBUG_INFO** in the **Enter Value** dialog box. Perform this step three times, once for each symbol.
   Click **Apply,** then click **OK**. The FSBL software project is compiled. The ELF (`fsbl.elf`) is in the **fsbl/Debug** directory.



X1175_13_052313

Figure 15: **Using Symbol Compile Options for the FSBL Software Application**

8. In SDK, select **Xilinx Tools > Create Zynq Boot Image** to invoke Bootgen. Figure 16 shows the invocation of the Bootgen GUI.



X1175_14_052313

Figure 16: **Invoking Bootgen**

9. Use the Basic tab in the **Create Zynq Boot Image** dialog box to specify the directory and name of the BIF file. Use the Browse button to select the `fsbl.elf file` from the fsbl/Debug directory, where fsbl was specified as the project name. Add `system.bit` to the list of partitions. If used, the bitstream partition must follow the FSBL partition. Add u-boot.elf to the list of partitions. As shown in Figure 17, click **Create Image** to generate the BIF and non-secure BIN and MCS files. The file is the input file into Bootgen that lists the partitions to include in the image. The MCS formatted image is used in QSPI boot mode. The BIN formatted image is used in SD boot mode. Create a directory ready_for_download_unencrypted in the project. Use the **Browse** button to select the **ready_for_download** directory as the Output folder.

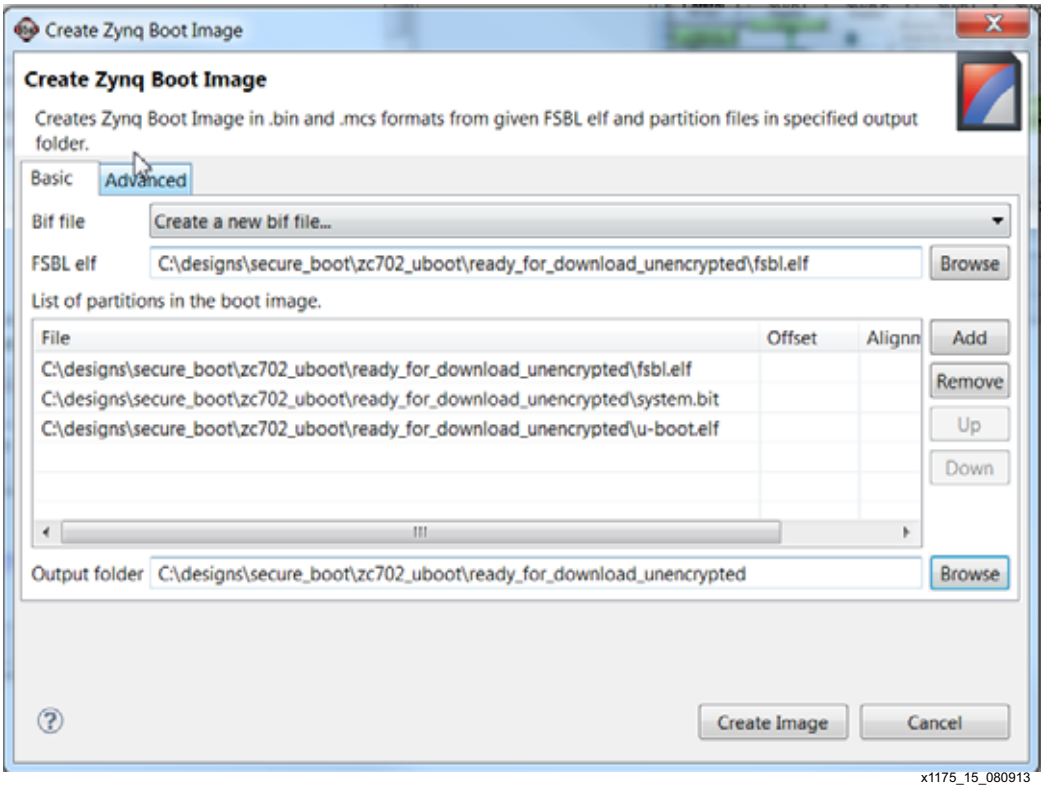


Figure 17: **Creating a Non-secure BIF Using the Bootgen GUI**

10. When encryption is selected, SDK 14.6 Bootgen GUI generates a secure image in which all partitions in the image are encrypted. The AES/HMAC engine requires a 256-bit AES key and a 256-bit HMAC key. The AES key can be generated using the Xilinx Bootgen tool or an external tool. To generate a development AES key using the Xilinx Bootgen software, create a `generate_aeskey.bif` file with the following content:

    generate_aeskey_image:

    ```
    {
    [aeskeyfile] bbram.nky
    [bootloader, encryption=aes] fsbl.elf
    }
    ```

    Use the following Bootgen command to generate an AES key:

    bootgen -image generate_aeskey.bif -o temp.mcs -encrypt efuse | bbram

    If the specified AES key does not exist, Bootgen generates the key with the name in the `generate_aeskey.bif` file (`bbram.nky` in this case).

    Note: The value to the -encrypt argument is either efuse or bbram. This command does not work on Windows in 14.6. It will be supported in 14.7.

11. To create a secure image, specify partitions in the Basic tab in the Create Zynq Boot Image dialog box using the same method used for the non-secure image. Create an output directory ready_for_download_bbram. Specify this directory in the Output folder. Click on the **Advanced** tab in the **Create Zynq Boot Image** dialog box, click **Enable encryption**, and browse to the key generated in step 10. As shown in Figure 18, click **Create Image** to create the secure boot image. Bootgen writes the image in either MCS or BIN format.

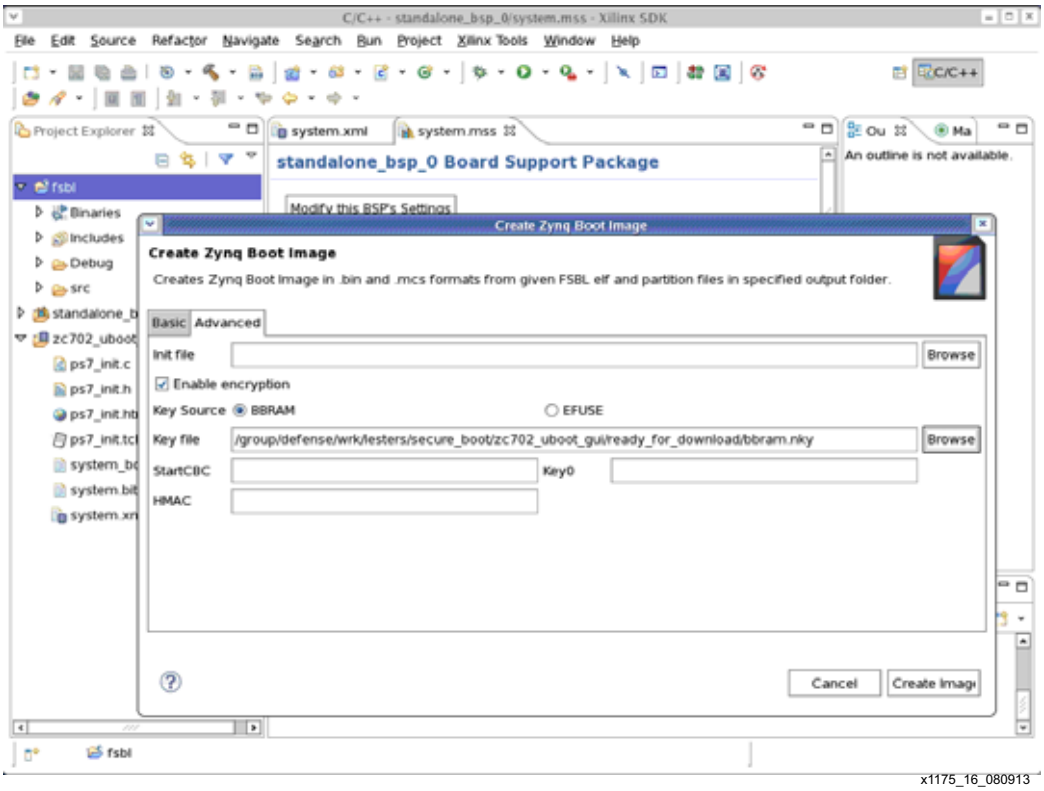


Figure 18: **Creating an Encrypted Image**

Setup the ZC702 Evaluation Board

1. Connect the power cable to the 12V J60 Connector.
2. Connect the USB cable from the PC to the USB UART J17 Connector.
3. Connect the Platform USB Cable II to JTAG Connector J2.

Table 1 provides the function of mode select switch on the ZC702. Some ZC702 boards use SW16, while some ZC702 boards use the J25, J22, and J20 jumpers.

Table 1: **ZC702 Boot Mode Selection**

	MIO[5] - J25	MIO[4] - J22	MIO[3] - J20
JTAG	0	0	0
NOR	0	0	1
NAND	0	1	0
Quad-SPI	1	0	0
Secure Digital	1	1	0

After the board is setup, use the following steps to program the AES key, load the boot image, and boot an encrypted image.

1. As shown in Figure 19, invoke iMPACT with **impact &** if using Linux or **ISE Design Suite 14.6 > ISE Design Tools > 64-Bit Tools > iMPACT** if using Windows. Click **No** when prompted to save the project file. Click **Cancel** when the New iMPACT Project dialog box is displayed. Double-click **Boundary Scan** in the **iMPACT Flows pane**. Right-click and select **Initialize Chain**. Initialize the JTAG chain. If only one of the ARM® core or xc7z020 FPGA is displayed on the JTAG chain, change the Boot Mode selection switch to **JTAG** and re-initialize the chain.

   Click **Yes** when asked to assign a configuration file. With the DAP (far-left device) displayed in green, click **Bypass**. With the **xc7z020** selected, enter **`xapp1175/xc702_uboot/bbram.nky`** in the **Assign New Configuration File** dialog box. Click **Cancel** when the **Device Programming Properties** dialog box is displayed.

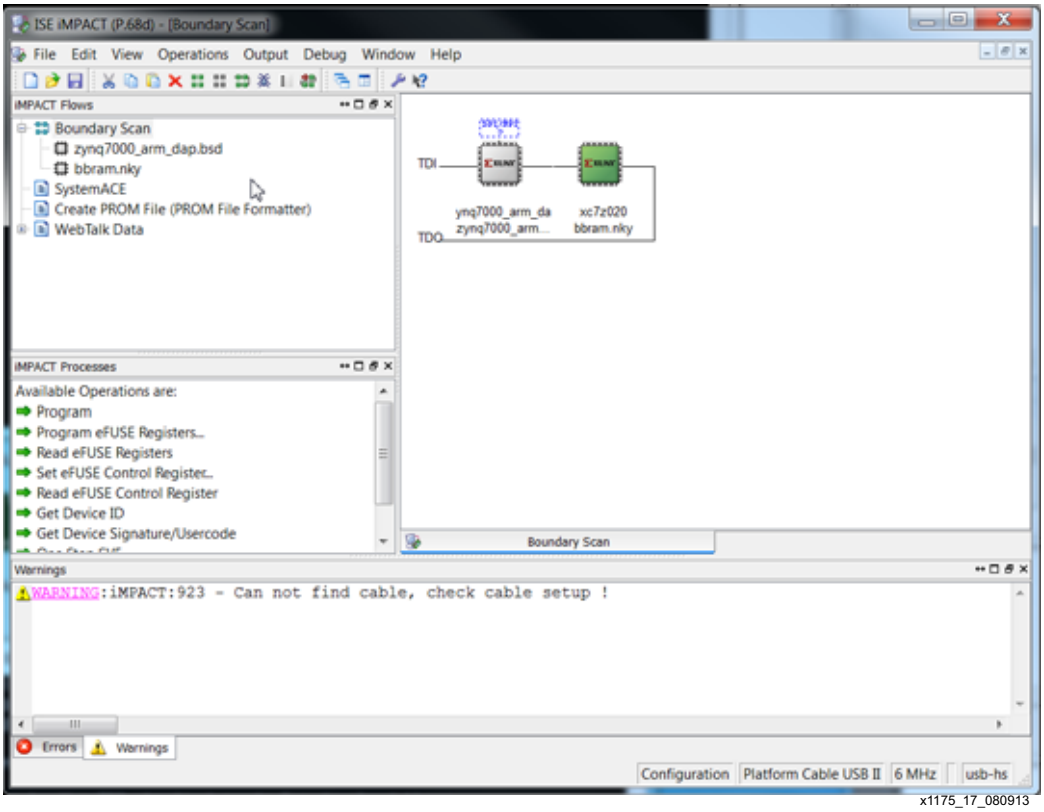
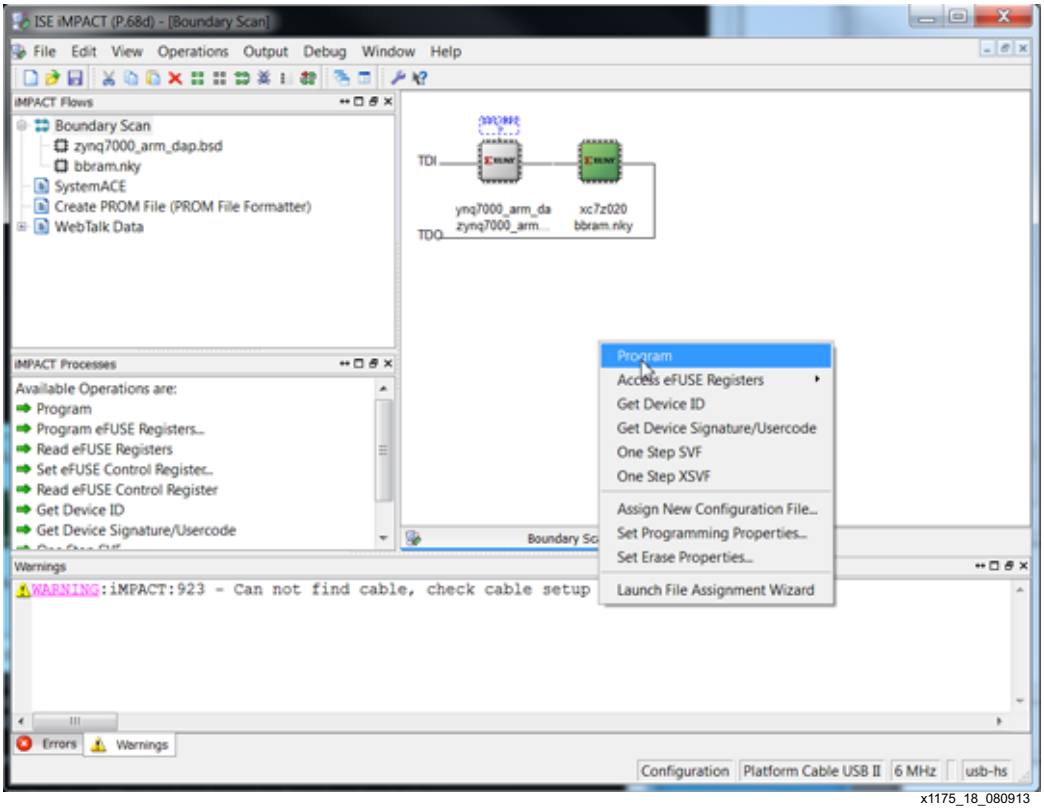


Figure 19: **iMPACT - Detecting Concatenated JTAG Chain**

2. As shown in Figure 20, use iMPACT to program BBRAM key. Right-click on **xc7z020** and click **Program**. Select **Cancel** when the Device Programming Properties is displayed. iMPACT can also program the eFUSE key. If eFUSE and BBRAM keys are programmed, the Bootgen -encrypt efuse | bbram argument specifies which key is used. Bootgen writes the key source to the Boot Header region of the image. At power up, the BootROM code reads the Boot Header to determine which key source to use. The eFUSE control register is also programmed with iMPACT.

   An alternative to using the iMPACT GUI is to run iMPACT from the command line as shown in Figure 20. Edit loadkey.cmd to use the correct key file.

```
cd xapp1175/zc702_uboot
impact -batch loadkey.cmd
```



x1175_18_080913

Figure 20: **Using iMPACT to Program BBRAM Key**

3. Invoke SDK by running **xsdk &** at the Linux prompt or **Xilinx Design Tools > ISE Design Suite 14.6 > EDK > Xilinx Software Development Kit** from the Program Start Menu. Set the workspace at zc702_uboot/SDK. Click the **Terminal** tab to setup a terminal window. Figure 21 shows how to set up the SDK communication terminal to use a 115200 baud rate. Alternatively, minicom, TeraTerm or Hyperterminal can be used as the communication terminal.

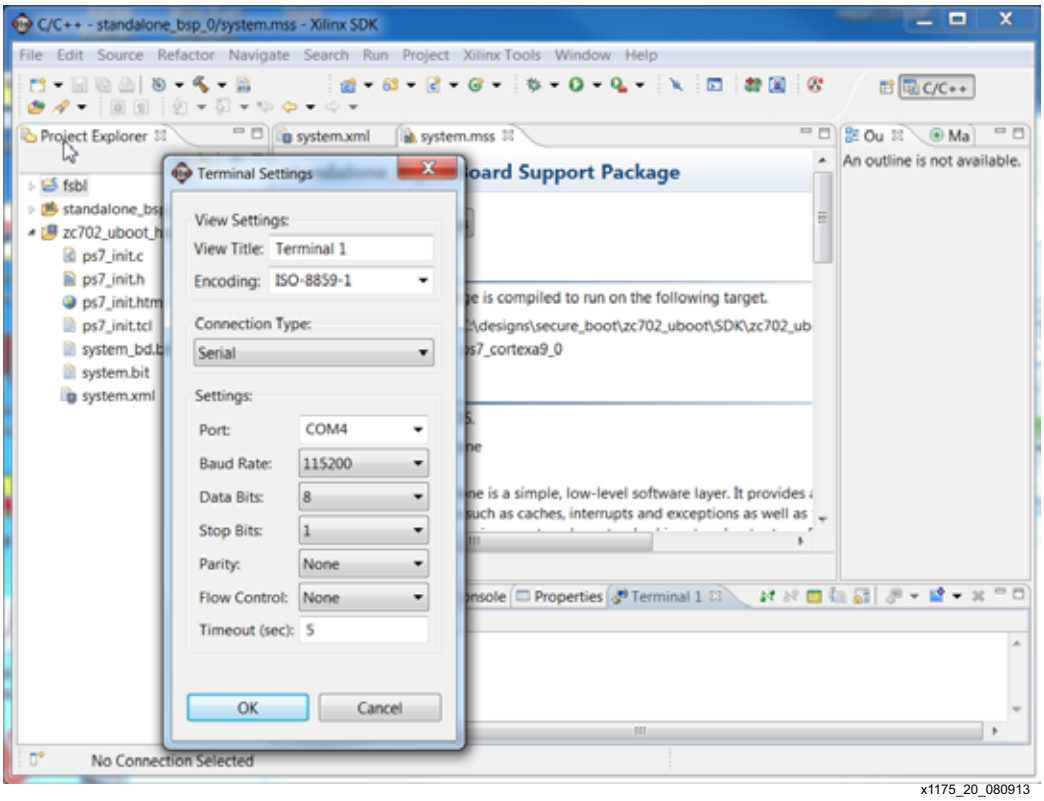


Figure 21: **SDK Communication Terminal**

4. From SDK, click **Xilinx Tools** > **Program Flash**. Optionally, click **Verify after Flash**, and Click **Program**. Figure 22 shows programming the NVM QSPI flash memory.

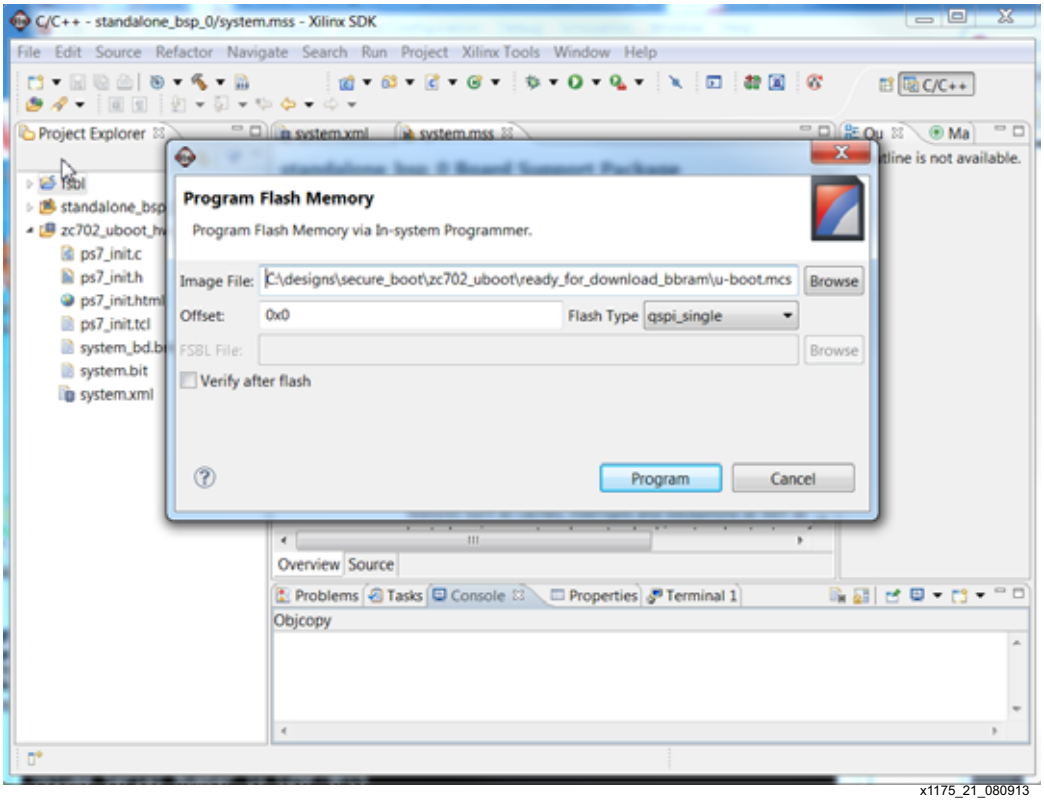


Figure 22: **Programming QSPI NVM**

To program QSPI at the command prompt, enter the following:

zynq_flash -f u-boot.mcs -offset 0x0

Note: This assumes the u-boot.`mcs` is in the current directory. Use the full path if it is in a different directory.

5. Set the J20/22/25 switches to select the QSPI boot mode. Power cycle the board. Figure 23 shows the communication terminal output with the U-Boot prompt.

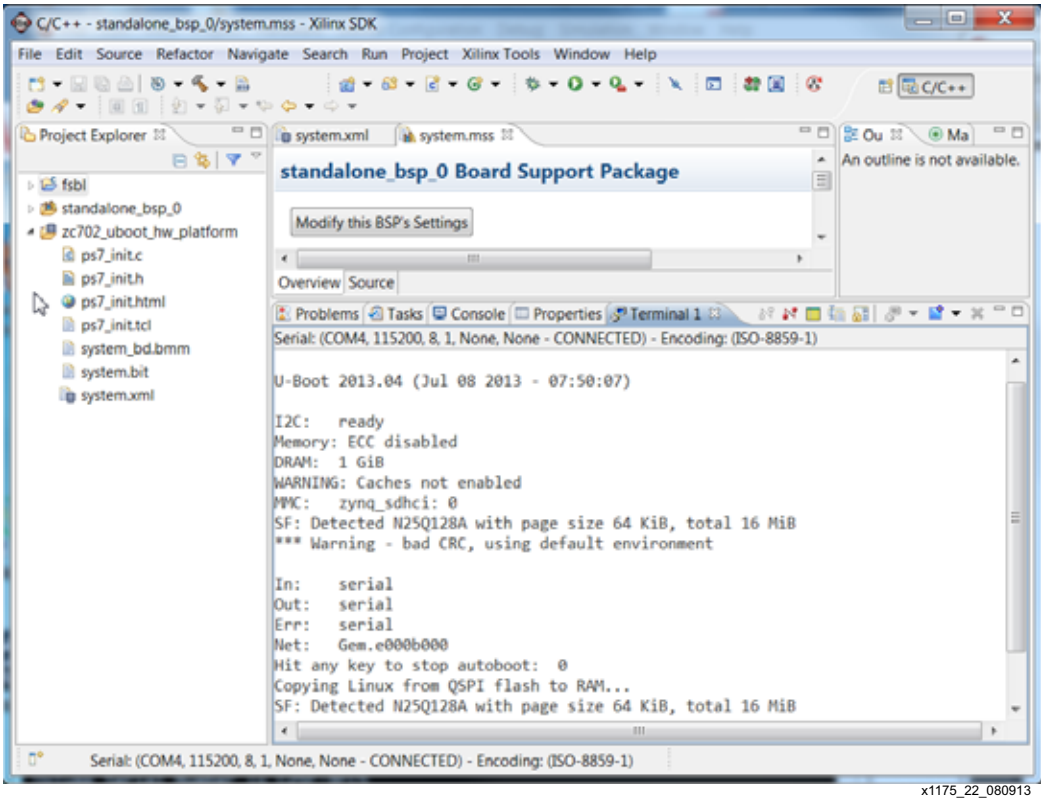


Figure 23: **Communication Terminal Output after Running U-Boot Application**

6. To boot using the SD card, copy the **<design>.bin** created by Bootgen to **BOOT.bin**. Connect a SD/MMC Card Reader/Writer to a PC using a USB cable. Copy **BOOT.BIN** to the SD card. Insert the card into the SD MMC slot. Set the Boot Mode settings to **SD (011)**. Power cycle. Verify that the same output is displayed in the communication terminal as when QSPI boot mode is used.

Debugging QSPI Boot Failure

If there is a functional failure after booting from QSPI mode, verify that the function works as expected after the partitions are loaded using JTAG boot mode. Select **Xilinx Tools > XMD Console**, and enter the following XMD commands:

```
xmd
fpga -debugdevice devicenr 2 -f system.bit
connect arm hw
rst -processor
source ps7_init.tcl
ps7_init
ps7_post_config
dow u-boot.elf
con
exit
```

Note: If the files are not in the current directory, use full paths to the files.

Failure to boot is usually due to incorrect setup of the clocks or memory controllers. The clock frequencies may change across releases of different versions of silicon and software.

If the JTAG boot works, the next step to debug the QSPI failure is to create a version of the FSBL which provides debug information. This is defined in the Creating a Project Using Xilinx Platform Studio section. Create the new FSBL, and see the steps to include the debug FSBL in the BIF. Re-run the QSPI boot, and read the FSBL debug log file to locate the boot error.

Creating a Secure Boot Image

The system architect has two considerations in creating a secure boot image. One is to specify which partitions are encrypted and which partitions are authenticated. The other is to architect the system so that sensitive programs and sensitive data are located in secure storage, within the security perimeter of Zynq.

A typical software image consists of relatively large open source U-Boot and Linux partitions, and proprietary partitions which contain software and PLIP. In most cases, only the proprietary or sensitive partitions are encrypted. When large partitions are unencrypted, the key is exposed less. Conventionally, open source software such as U-Boot and Linux should be authenticated to ensure that the partitions are not modified, but not encrypted. If proprietary changes are made to U-Boot and/or Linux, these partitions can be encrypted. Encryption is not a factor in booting with a chain of trust. Encryption ensures that the partition is not readable and is confidential. The Bootgen Image Format (BIF) file is a Bootgen input file used to specify encryption/authentication on a partition basis.

When decryption is used, the AES/HMAC engine runs at a lower clock frequency than other circuitry. This means that boot time can be slower than when encryption is not used. The option to leave open source partitions unencrypted may decrease boot time. In tests of QSPI on the ZC702 board, the boot time for non-secure and secure boot are the same. The boot time can differ for different NVM configurations and speed grades.

The first step in creating a secure system is to write the BIF file. Appendix B defines the security requirements for sample embedded systems. The use cases provided are secure U-Boot, Linux, and multiboot systems. Use cases are also provided for data partitions. The BIFs for the use cases in Appendix B are given in Appendix C. The use cases are for boot at power-up, so the use cases include the FSBL partition. Bootgen does not require the FSBL partition to be included in the BIF. Bootgen can AES encrypt and RSA authenticate a single software or data partition. This supports a post-boot load operation without the use of the FSBL.

The Zynq device's secure storage and security perimeter are useful in maintaining security after handoff to Linux applications. The PL partition, which is encrypted in NVM, is decrypted and stored in the PL's configuration memory. The PS partitions can be encrypted in NVM and decrypted by the AES/HMAC engine. The software in DDR is usually unencrypted, outside of Zynq's security perimeter.

To protect sensitive software and data, the destination address needs to be within the Zynq device's security perimeter, typically OCM or AXI BRAM. The Building and Booting a Secure System section shows how to use AXI BRAM with the ARM CPUs. The eFUSE Driver section shows how to locate code/data in OCM. Since the amount of secure storage is limited, this requires architecting the software memory map. Open source code can be stored in plaintext in DDR. Sensitive software should be stored in secure storage. If the code in a software partition does not need to be protected but the data is sensitive, the sensitive data can be loaded into OCM using the linker script.

Bootgen can load data only partitions. If only the data is sensitive, data partitions are an effective use of secure storage. Bootgen allows the data partitions to be encrypted and/or authenticated.

In addition to authenticating software and the bitstream, RSA authentication and AES encryption can be used on software updates, and partial bitstreams.

Bootgen

Bootgen is PC based software which generates the image loaded into NVM. The BIF file lists the partitions and specifies authentication/encryption requirements for each partition. Bootgen outputs a single image file in binary or MCS format. Attributes in the BIF file are used to specify load addresses on a partition basis, and whether a partition is encrypted and/or authenticated. In the 14.6 release, much of the Bootgen security functionality is only available when Bootgen is run from the command line.

Bootgen operates in a Debug mode or a Release mode. Debug mode is easier to use, and meets many users' security requirements. Release mode provides improved security for RSA keys. Debug mode is easier to use because RSA signatures for the partitions do not need to be provided. In Debug mode, the user provides private RSA keys in the BIF, and Bootgen generates the hashes and signatures. Bootgen Release mode uses public keys and signatures in the BIF.

Bootgen is available in SDK and as a standalone tool.

Figure 24 shows the Bootgen flow.

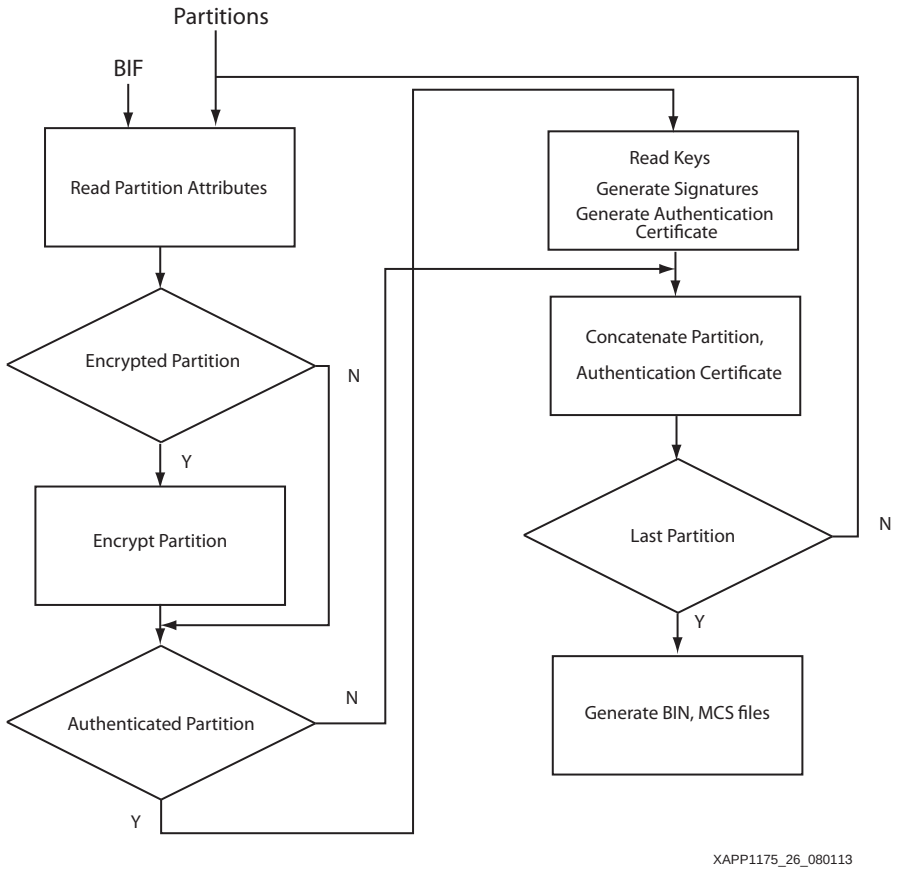


Figure 24: **Bootgen Flow Diagram**

To use Bootgen, create a Bootgen Image File (BIF) such as `bootimage.bif`, and run Bootgen at the command line as follows:

```
bootgen -image bootimage.bif -o <design>.mcs -encrypt bbram
```

If the -encrypt <efuse | bbram> argument is used and an AES key file is not included, Bootgen generates an AES (`design.nky`) key file with the prefix name of the [aeskeyfile] attribute argument. This key filename must be used as the argument to the [aeskeyfile=<design>.nky] attribute in the BIF file.

The following BIF is an example of a Linux image in which all partitions are authenticated, and the FSBL and Linux applications are encrypted:

```
image: {
[aeskeyfile] bbram.nky
[pskfile] psk.pk1
[sskfile] ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[encryption=aes, authentication=rsa] system.bit
[authentication=rsa] u-boot.elf
[authentication=rsa,load=0x3000000,offset=0x100000] uImage.bin
[authentication=rsa,load=0x2A00000,offset=0x600000] devicetree.dtb
[authentication=rsa,load=0x2000000,offset=0x620000] uramdisk.image.gz
[authentication=rsa, encryption=aes] sobel_cmd.elf
}
```

In this BIF, the `pskfile` attribute is for the primary secret key (`psk.pk1`), and the `sskfile` attribute is for the secondary secret key (`ssk.pk1`). The [aeskeyfile] attribute specifies the AES key. All partitions are authenticated. The `fsbl.elf, system.bit,` and `sobel_cmd.elf`

are encrypted. The load attribute causes the FSBL to copy the partition to the specified address.

Using these attributes, the FSBL copies the partition from the [offset] address in NVM to the [load] address in DDR. When these attributes are used in the FSBL, U-Boot must be configured and built such that U-Boot does not load the partitions. To define this U-Boot configuration, edit:

`u-boot-xlnx/include/configs/zynq_common.h` as follows:

Remove the `sf read` operations under the following line.

```
#define CONFIG_EXTRA_ENV_SETTINGS
qspi=echo Copying Linux from QSPI flash to RAM
```

Do not change the following command.

```
bootm 0x3000000 0x2000000 0x2A00000
```

When this BIF is used, the image is too large for the QSPI on the ZC702 board. One option is to remove the `system.bit` partition.

Generating and Programming Keys

This section discusses the keys used, key format, key generation, and how keys are used by Bootgen and the FSBL code. The zc702_secure_key_driver system programs keys and control information. The use of iMPACT to program PL keys and control information was shown in the Building and Booting a Secure System section.

The cryptographic keys used by Zynq are:

- AES 256 bit key
- HMAC key
- RSA Primary Secret Key (PSK)
- RSA Primary Public Key (PPK)
- RSA Secondary Secret Key (SSK)
- RSA Secondary Public Key (SPK)

As noted, RSA authentication in Zynq uses primary and secondary keys. The primary keys authenticate the secondary keys. The secondary key(s) authenticate partition(s). In Bootgen Debug mode, the user only provides private keys in the BIF.

Generating Keys

A developer's key for AES/HMAC engine is generated using Bootgen. The AES key is programmed into either eFUSE or BBRAM using iMPACT or the Secure Key Driver.

OpenSSL is used to generate RSA keys in this application note. There are other methods of generating RSA keys. The primary and secondary secret RSA keys are generated using the following OpenSSL command:

```
openssl genrsa -out psk.pem 2048
openssl genrsa -out ssk.pem 2048
```

Note: Openssl is in Linux distributions. Windows users can use Cygwin openssl or download openssl from http://slproweb.com/products/Win32OpenSSL.html.

The format of these Privacy Enhanced Mail (PEM) key files is a Base64 encoding of a binary DER file. These files can be recognized by their first line, which is "-----BEGIN RSA PRIVATE KEY-----". In RSA, the public key is contained in the private key. The openssl command to extract the public key from the private key is:

```
openssl rsa -pubout -in psk.pem -out ppk.pub
```

```
openssl rsa -pubout -in ssk.pem -out spk.pub
```

In the naming convention for these files, PEM files are private (secret) and PUB files are public. Both files are PEM format, but the extensions are different.

Note: There is nothing that enforces this convention, and the file extensions are ignored by Bootgen.

The PEM and PUB files can be used directly with Bootgen. Other tools may require these files to be converted to a simple text format that displays the N, E, and D coefficients directly. This can be performed with the `convert_key.pl` Perl script, which uses OpenSSL to extract the coefficients.

```
convert_key.pl psk.pem psk.pk1
convert_key.pl ssk.pem ssk.pk1
```

Note: It may be necessary to precede `convert_key.pl` with xilperl, Active State Perl, or /usr/local/bin/perl.

The `.pk1` files contain N, E, D, P, and Q fields. The N and E coefficients represent the public component, while the N, E, and D fields represent the private component. The *.pk1 files can also be used with Bootgen, instead of the PEM/PUB files. Bootgen automatically determines the format of the key file.

The AES key can be stored in a *.nky file, and referenced in the BIF using the [aeskeyfile] attribute. While storing a key in a file is usually necessary, it increases the exposure of the key. To avoid using a file containing a private key, specify the AES and HMAC keys on the command line as:

```
bootgen -image zc702_u-boot.bif -o zc702_u-boot.mcs -w on -encrypt efuse
key=<aeskey> StartCBC=<initialization_vector> hmac=<hmac_key>
```

Note: This command is not supported on Windows in the 14.6 release.

eFUSE / BBRAM in Zynq Security

For RSA authentication, the hash of the PPK is stored in the PS eFUSE array. For AES decryption, the key is stored in either the eFUSE or BBRAM in the PL. The PL eFUSE control bits for eFUSE Secure Boot, BBRAM Key Disable, and JTAG Chain Disable are programmed using iMPACT or the Secure Key Driver. The eFUSEs are OTP. A power on reset (POR) is required after programming the eFUSEs.

Secure Key Driver

The Secure Key Driver programs the eFUSEs in the PS and the PL.

To compile and run the driver to program eFUSEs, the following tasks may be required.

- Generate the AES key
- Generate the hash of the PPK
- Edit the xilskey_input.h with the generated AES and hash (PPK) values
- Define XSK_EFUSEPS_DRIVER and XSK_EFUSEPL_DRIVER in xilskey_input.h as required
- Use SDK to build the Secure Key Driver software project
- Run the Secure Key Driver using XMD or load the driver in NVM and power cycle

The Secure Key Driver can program the PS and/or PL eFUSEs. To program the PL eFUSEs, four MIO outputs must be routed out of the MIO outputs into the JTAG pins. These are external connections which must be made on the printed circuit board. Table 2 provides a sample pinout which matches the connections used in the driver for the zc702 board. If other MIO pins are used, change the locations specified in the XilSkey driver.

Table 2: **eFuse Connections**

MIO	JTAG
17	TDI
21	TDO
19	TCK
20	TMS

For development, it is usually simpler to use iMPACT to program the AES key.

Generate Hash of PPK

After generating the RSA keys using openssl, use Bootgen to generate the hash of the PPK.

Create a `gen_hash_ppk.bif` file with the following content:

gen_hash_ppk:

```
{
[pskfile] psk.pk1
[sskfile] ssk.pk1
[bootloader, authentication=rsa] fsbl.elf
}
```

Run

```
bootgen -image gen_hash_ppk.bif -efuseppkbits hash_ppk.txt
```

This bootgen command produces the `hask_ppk.txt` file, which contains the hash of the PPK. Using eFUSEs for the hash of the PPK is an efficient use of silicon.

Generate the ELF for the Secure Key Driver

To use SDK to create a Secure Key Driver application, change to the zc702_secure_key_driver directory. The functionality of the Secure Key Driver is controlled by editing the xilskey_efuse_example.c and xilskey_input.h files. If the PL eFUSEs are not programmed with the driver, comment the

#define XSK_EFUSEPL_DRIVER

line. To only read the PS eFUSE, comment the write function in xilskey_efuse_example.c.

The `xilskey_input.h` file can be edited to program PS eFUSEs, PL eFUSEs, or both. Edit `xilskey_input.h` using the following steps:

1. If used, define the XSK_EFUSEPS_DRIVER and XSK_EFUSEPL_DRIVER.
2. Copy the PPK hash from `hash_ppk.txt` to XSK_EFUSEPS_RSA_KEY_HASH_VALUE.
3. Set XSK_EFUSEPS_ENABLE_RSA_KEY_HASH to TRUE to program RSA PPK hash.
4. Set XSK_EFUSEPS_ENABLE_RSA_AUTH to TRUE to enable RSA authentication.
5. If programming the PL, copy the AES key to XSK_EFUSEPL_AES_KEY.
6. Set XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY to TRUE to program the AES key.

Compile the zc702_Secure Key Driver. As shown in Figure 25, run **xsdk &** and change to the **zc702_secure_key_driver SDK** workspace.

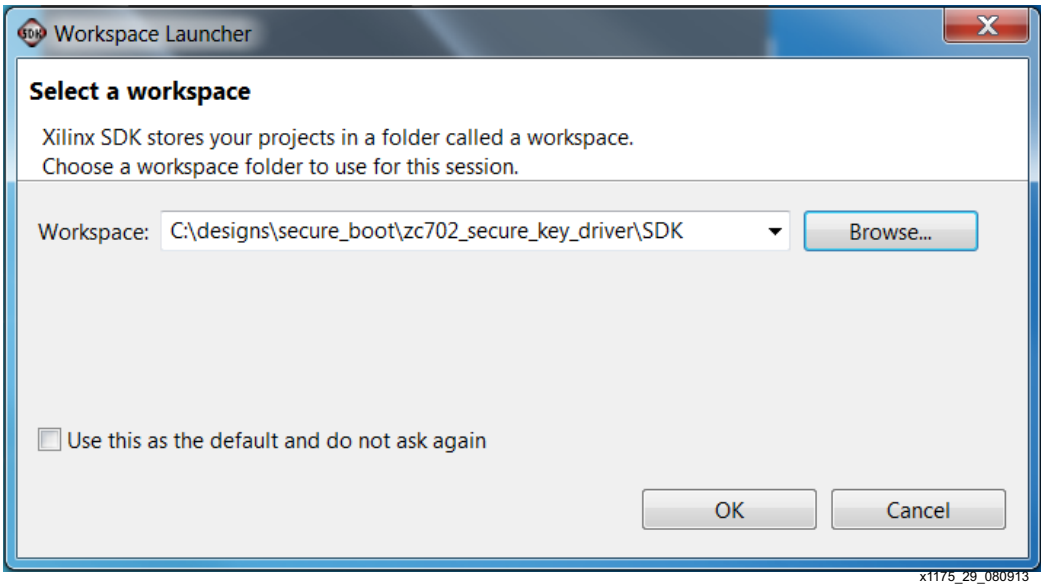


Figure 25: **Create zc702_secure_key_driver SDK Workspace**

As shown in Figure 26, enter **File > New > Application Project** and define an application project as `secure_key_driver`. Click **Next**.

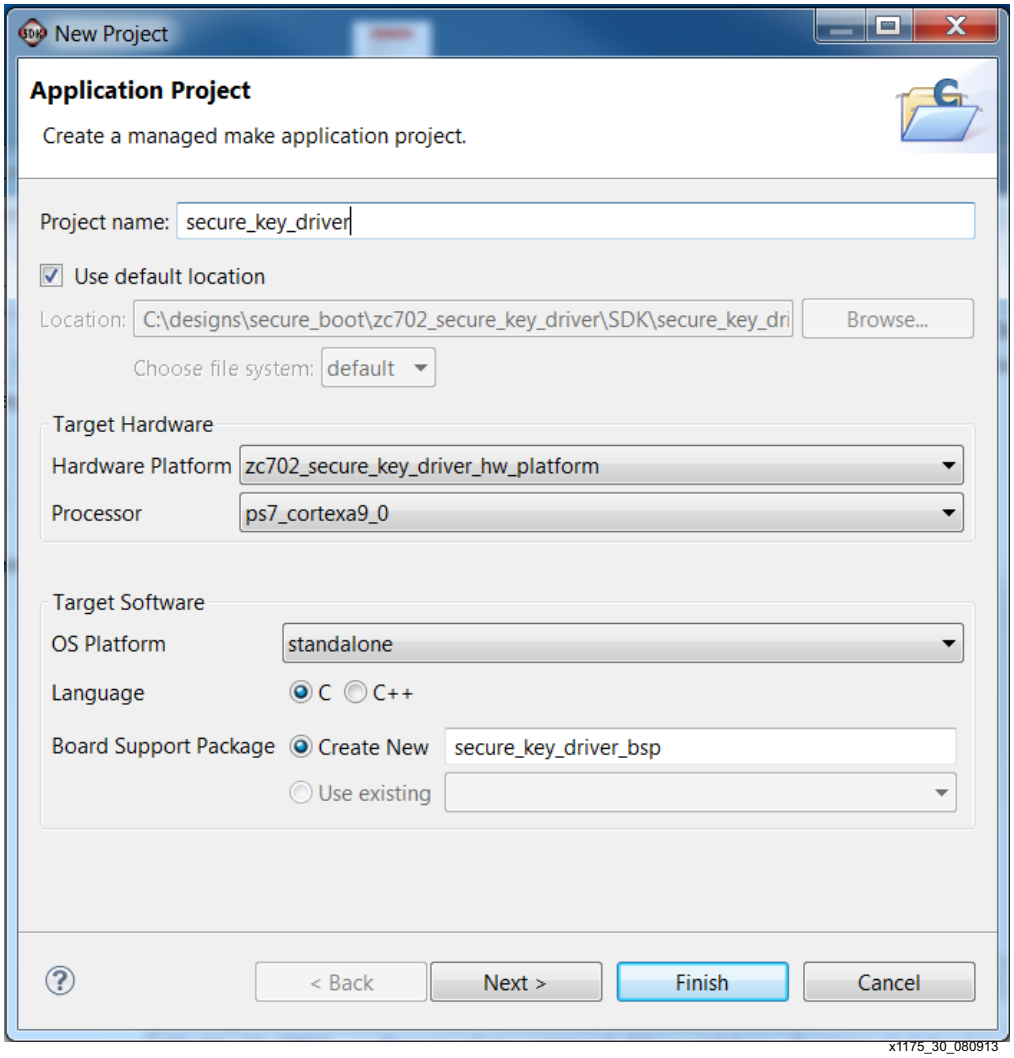


Figure 26: **Define secure_key_driver Application Project**

As shown in Figure 27, select **Empty Application**, and when the dialog box is displayed, name the project **secure_key_driver**. Click **Finish**.

Note: It may be necessary to close a Welcome screen which hides the SDK display.

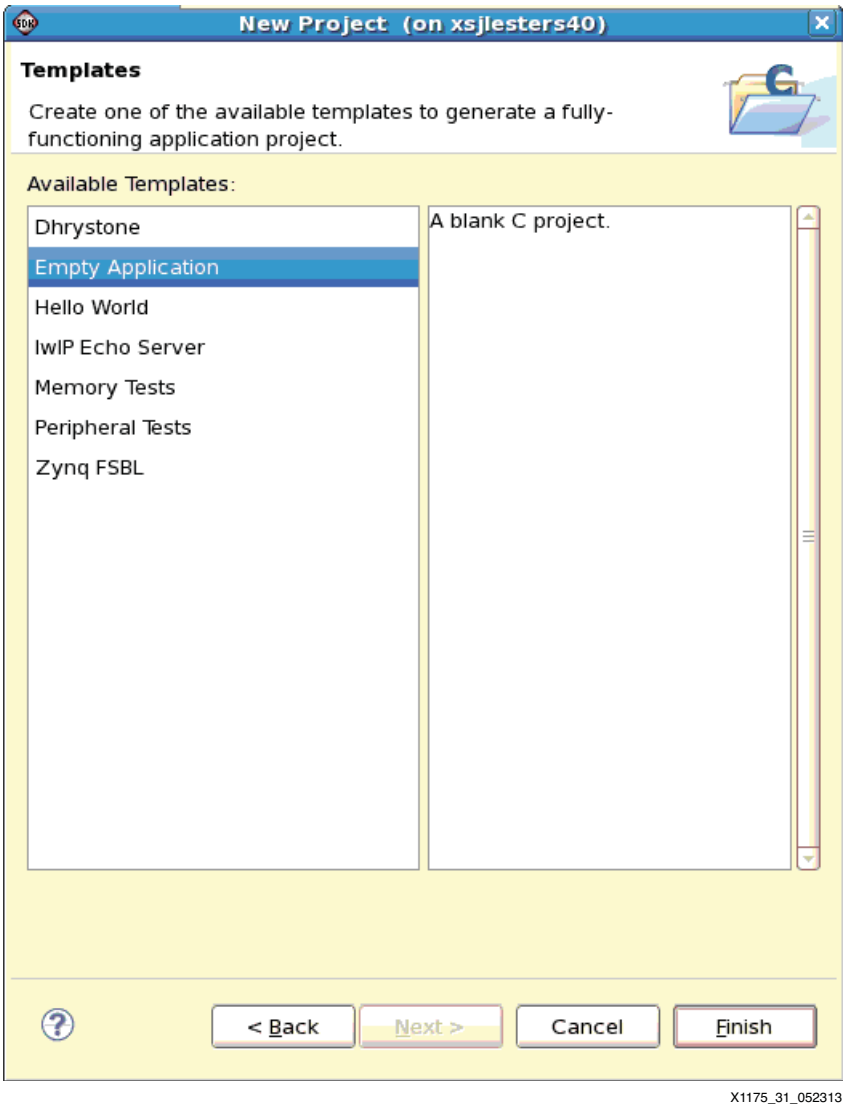


Figure 27: **Create Empty Application**

As shown in Figure 28, right-click on the **secure_key_driver_bsp** board support package, and click on **Board Support Package Settings**.

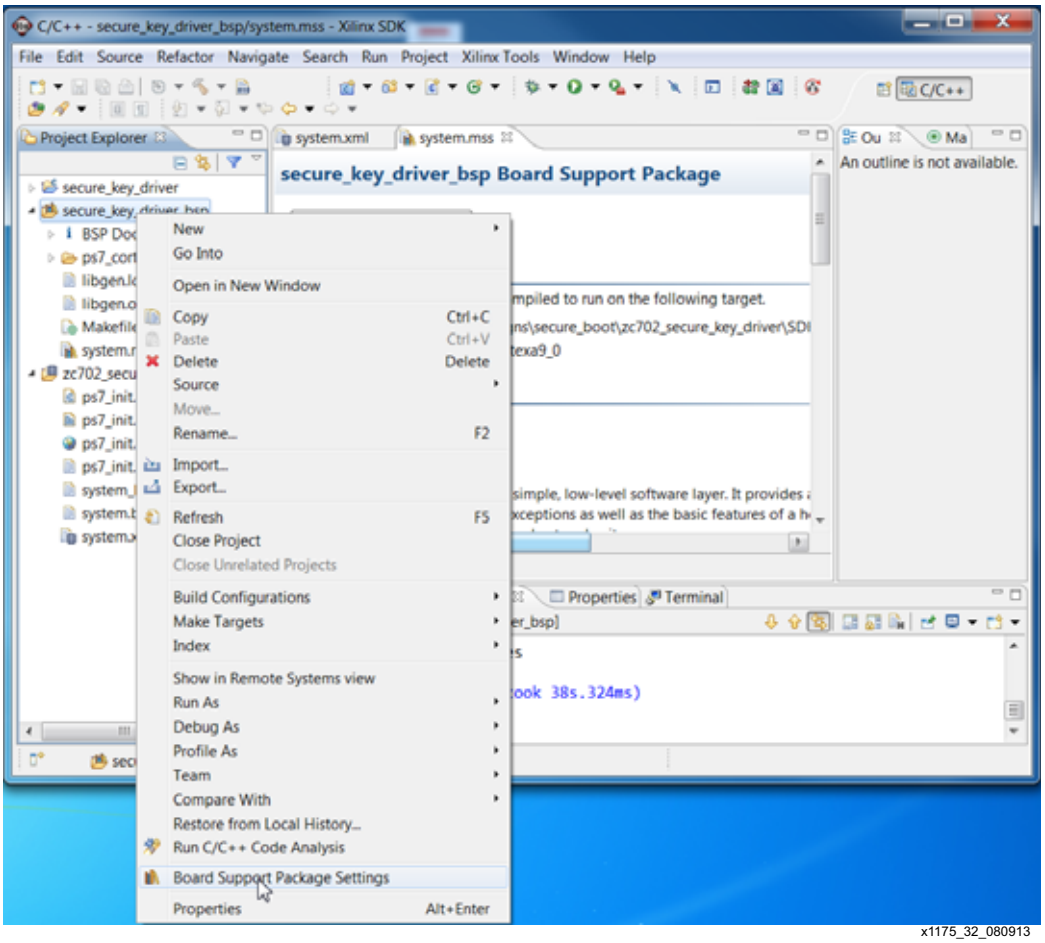


Figure 28: **Specifying BSP Settings**

As shown in Figure 29, select the **xilskey** library and rebuild the BSP. Click **OK**.

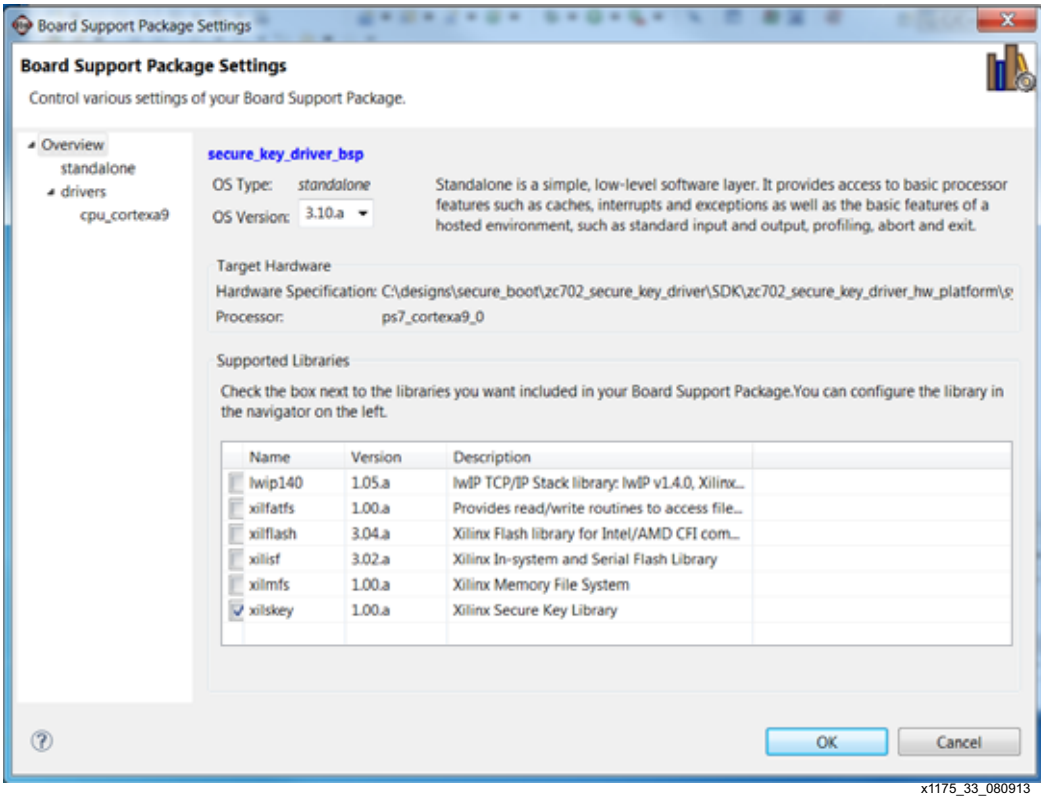


Figure 29: **Selecting the Secure Key Library**

Figure 30 shows the compiled **`xilskey`** library.

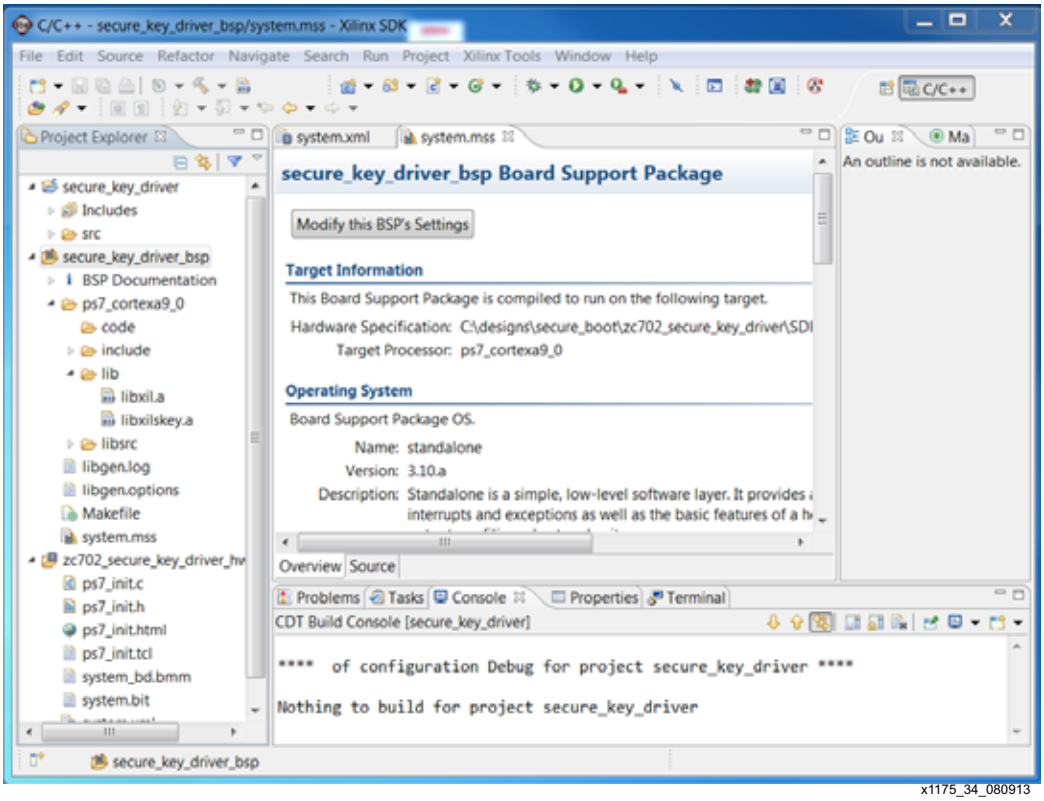


Figure 30: **Compiled xilskey Library**

With efuse_driver selected, enter **File > Import > General File System**, click **Next**, and browse to the src directory containing and import **`xilskey_efuse_example.c`** and **`xilskey_input.h`**. Select the two files and click **Finish**. Select the **secure_key_driver** project, and run **Project > Build Project**. Figure 31 shows a compiled efuse_driver software project.

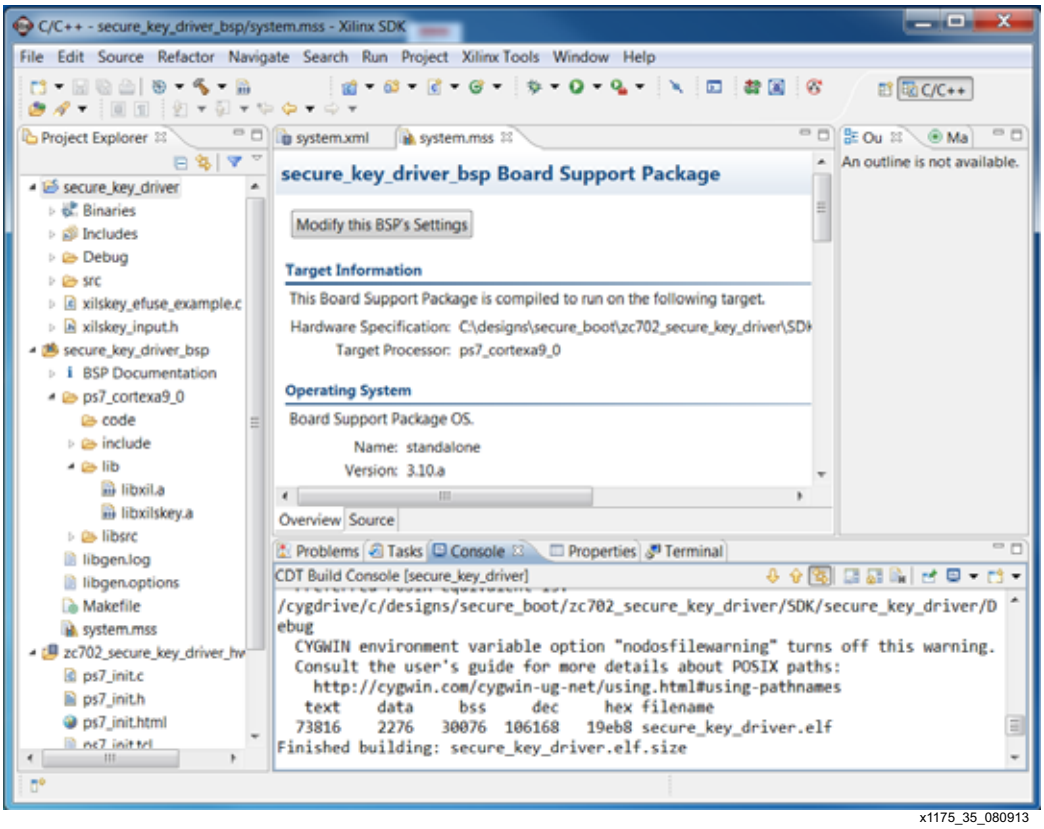


Figure 31: **Compiled Secure Key Driver**

To run the Secure Key Driver from OCM rather than DDR, edit the linker script `lscript.ld` as shown in Figure 32. The linker script in SDK is opened by selecting `efuse_driver/src/lscript.ld` in Project Explorer, right-clicking `lscript.ld`, and selecting **Open**.

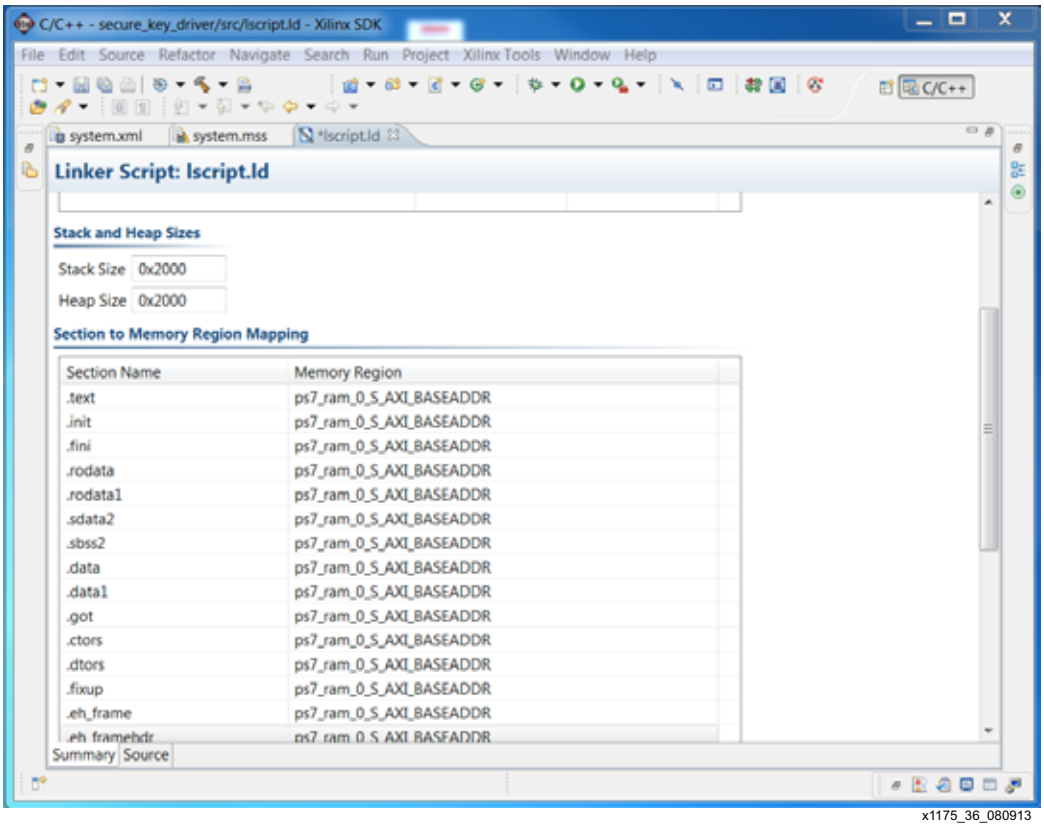


Figure 32: **Loading the Secure Key Driver in OCM**

Change the location of the sections to **ps7_ram_0_S_AXI_BASE_ADDR**. Add the `ps7_init.c` and `ps7_init.h` files from the hw_platform directory to the source files. Uncomment the ps7_init() call. Rebuild the efuse_driver software application.

Run the Secure Key Driver

The eFUSE driver can be run using any boot mode. The simplest is to use XMD in the JTAG boot mode. The xapp1175/zc702_secure_key_driver/ready_for_download directory contains the `ps_secure_key_write.elf` and `ps_secure_key_read.elf` files. If the PS eFUSES have been programmed, use `ps_secure_key_read.elf`.

Run the Secure Key Driver in XMD to write eFUSEs using the following steps:

```
xmd
connect arm hw
source ps7_init.tcl
ps7_init
stop
dow ps_secure_key_write.elf (if writing)
dow ps_secure_key_read.elf (if reading)
con
stop
```

The output of the eFUSE driver can be viewed in a communication terminal such as Tera Term or the SDK terminal. Creating a communication terminal is shown in the Booting the TRD Securely section. An alternative to running the Secure Key Driver in XMD is to load it into QSPI or SD. To do this, create a BIF containing the FSBL and Secure Key Driver.

secure_key_driver:

```
{
[bootloader] fsbl.elf
ps_secure_key_write.elf
}
```

The FSBL is not included in the BIF when the Secure Key Driver is executed from OCM. The FSBL is included in the BIF when the driver is executed from DDR. When executed from OCM, the base address of the OCM must be used.

Use Bootgen to create a MCS or BIN file. For additional information, see UG996 *LibXilSkey for Zynq-7000 AP SoC Devices*, which is located in the *OS and Libraries Document Collection* (UG643) [Ref 5].

If using QSPI, run **SDK > Program Flash** to program the QSPI. If using the SD card, load BOOT.bin on the SD card and insert the card into the zc702 card slot (J64 SDIO). Set the boot mode pins to SD and power cycle.

A second method of using the Secure Key Driver is to create a SVF and use iMPACT to play the SVF. The steps for this flow are given in Appendix E.

Additional information is on the secure key driver is available in *OS and Libraries Document Collection* (UG643) [Ref 5].

Advanced Key Management Options

The Bootgen Release mode increases key handling security since the BIF attributes use public rather than private RSA keys. In some organizations, an Infosec staff is responsible for the production release of a secure embedded product. The Infosec staff's key handling responsibilities differ from those in the product development organization.The Infosec staff may use a Hardware Security Module (HSM) for digital signatures and a separate secure server for encryption. The HSM and secure server typically reside in a secure area. The HSM is a secure key/signature generation device which generates private keys, encrypts partitions using the private key, and provides the public part of the RSA key to Bootgen. The private keys do not leave the HSM. The BIF for Bootgen Release mode uses public keys and signatures generated by the HSM. The public keys associated with the private keys are `ppk.pk1` and `spk.pk1`.

The HSM accepts hash values of partitions generated by Bootgen and returns a signature block based on the hash and the secret key. To emulate a HSM, the xil_rsa_sign software is provided in the xapp1175/xc702_secure_key/xil_rsa_sign_src directory, with instructions on compiling the executable with a make file. Analogous to the HSM, xil_rsa_sign signs the hashes provided by Bootgen. To build the xil_rsa_sign executable, change to the xil_sign_rsa_src directory, copy makefile_linux(makefile_xp) to Makefile, and run:

```
make xil_rsa_sign
```

Note: If make fails when Windows is used, edit the source of the GCC in makefile_xp. There are many embedded processor specific `gcc.exe` in $XILINX_EDK. Do not use these `gcc.exe` executables. Change the GCC to a full path as c:/cygwin/bin/gcc.exe or $XILINX/gnu/MinGW/5.0.0/nt/bin/gcc.exe.

To use xil_rsa_sign, add <path>/xapp1175/zc702_secure_key/xil_rsa_sign_src to $PATH.

Figure 33 shows the flow for Bootgen Release mode. The stages labeled "xil_rsa_sign" can be performed either by the HSM (Standard mode) or by xil_rsa_sign.

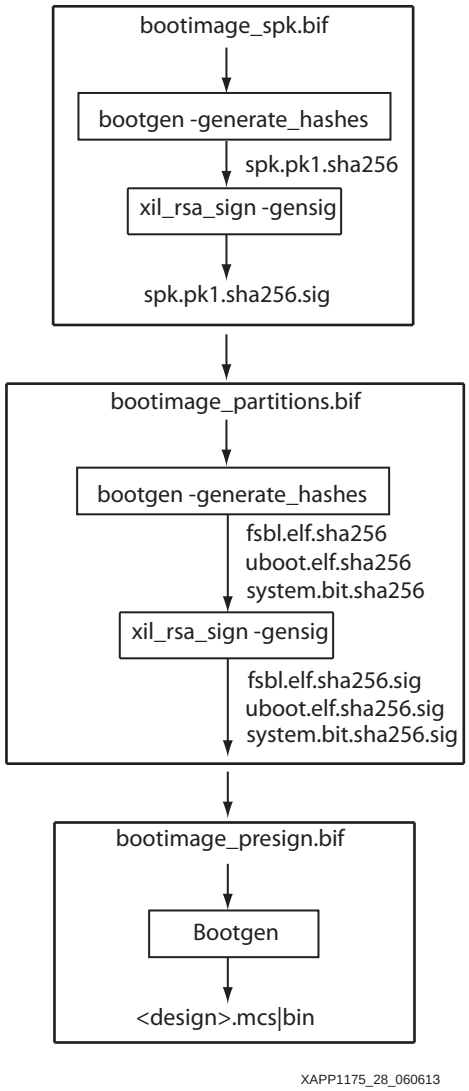


Figure 33: **Bootgen Release Mode Flow**

In this section, Bootgen is run using the Debug mode and the Release mode. The output image files, `zc702_uboot_dm.mcs` and `zc702_uboot_rm.mcs`, are shown to be identical. To run Bootgen in Debug and Release modes, change to the xapp1175/zc702_secure_key directory.

Bootgen Debug Mode Step

Run Bootgen as follows:

```
bootgen -image bootimage_dm.bif -o zc702_uboot_dm.mcs -encrypt efuse
```

In which `bootimage_dm.bif` is:

```
bootimage_dm:
{
[aeskeyfile] efuse.nky
[pskfile] psk.pk1
[sskfile] ssk.pk1
[bootloader, encryption=aes, authentication=rsa] fsbl.elf
[encryption=aes, authentication=rsa] system.bit
[authentication=rsa] u-boot.elf
}
```

Note: This command is not supported on Windows in 14.6.

Bootgen Release Mode Steps

The xapp1175/zc702_secure_key directory provides an example of running Bootgen in Release mode. The steps in generating the keys and signatures using the Release mode are:

1. Use the following command to create the SPK hash file.

   ```
   bootgen -image spk.bif -generate_hashes
   ```

   in which `spk.bif` is

   ```
   spk:
   {
   [spkfile] spk.pk1
   }
   ```

   Bootgen generates the SPK hash spk.pk1.sha256.

2. Run xil_rsa_sign to generate the SPK signature.

   ```
   xil_rsa_sign -gensig -sk psk.pk1 -data spk.pk1.sha256 -out
   spk.pk1.sha256.sig
   ```

3. Generate the partition hashes using the following bootgen command.

   ```
   bootgen -image bootimage_partitions.bif -o temp.txt -encrypt efuse
   -generate_hashes
   ```

   where the BIF is

   bootimage_partitions:

   ```
   {
   [ppkfile] ppk.pk1
   [spkfile] spk.pk1
   [spksignature] spk.pk1.sha256.sig
   [bootloader, encryption=aes, authentication=rsa] fsbl.bin
   [encryption=aes, authentication=rsa] system.bit
   [authentication=rsa] u-boot.elf
   }
   ```

   Bootgen generates `fsbl.elf.0.sha256`, `u-boot.elf.0.sha256`, `u-boot.elf.1.sha256`, and `system.bit.0.sha256`. Since some ELF files consists of two partitions (separate program text and MMU blocks), there may be two SHA256 files for some partitions. The image is not built yet. Hashes are generated.

4. Use xil_rsa_sign to generate the signatures of the hashes just created.

   ```
   xil_rsa_sign -gensig -sk ssk.pk1 -data fsbl.elf.0.sha256 -out
   fsbl_debug.elf.0.sha256.sig
   xil_rsa_sign -gensig -sk ssk.pk1 -data system.bit.0.sha256 -out
   system.bit.0.sha256.sig
   xil_rsa_sign -gensig -sk ssk.pk1 -data u-boot.elf.0.sha256 -out
   u-boot.elf.0.sha256.sig
   ```

These operations generate signatures for the SSK and partitions. Create an image with the following bootgen command.

```
bootgen -image bootimage_presign.bif -o zc702_u-boot.mcs -encrypt efuse
```

in which the BIF is

bootimage_presign:

```
{
[ppkfile] ppk.pk1
[spkfile] spk.pk1
```

```
[spksignature] spk.pk1.sha256.sig
[bootloader, encryption=aes, authentication=rsa,
presign=fsbl.elf.0.sha256.sig] fsbl.elf
[encryption=aes, authentication=rsa, presign=system.bit.0.sha256.sig]
system.bit
[encryption=aes, authentication=rsa, presign=u-boot.elf.0.sha256.sig]
u-boot.elf
}
```

Bootgen Debug and Release modes generate an identical MCS / BIN image. To verify this, run

```
diff zc702_uboot_dm.mcs zc702_uboot_rm.mcs
```

In xil_rsa_sign, a make file is provided to run these steps. Edit xil_rsa_sign and bin to use the correct paths. Then run

```
make clean
make all
```

The default is to create zc702_*.bin files. To create MCS files, change bin to mcs in the Makefile.

Secure Embedded Systems Applications

This section provides applications which use the security features in Zynq. A number of multiboot examples are provided that combine security and high reliability. The GDB debugger is used to analyze the RSA authentication and the multiboot flow using the ps_autherr multiboot reference design. Two methods of testing Linux boot time using QSPI are described. The use of Bootgen to create a user defined field in the authentication certificate is provided. The zc702_data reference system shows how to load a data file into the Zynq device's secure storage. The use of hierarchical control of the JTAG debug port for different security requirements is discussed.

Multiboot

Multiboot is used to ensure that the device boots a golden image in the event of a failure to boot the original update boot image. Examples of multiboot are given in the zc702_multiboot reference system. Alternative multiboot implementations are possible. In this implementation, the update and golden image are identical, with the intent that the golden image is a backup if there is a problem loading the update image. The term "fallback" is sometimes used for this functionality.

A different multiboot requirement provides a software update. If the software update image fails to load, the FSBL loads the original image. For device security, it is critical that the device boot to a known working state. The multiboot methodology provided can be used for either of these multiboot requirements.

The multiboot systems are intended to show how to create testable multiboot systems. To be testable, an intentional failure must be introduced into the system. For the zc702_multiboot systems, Figure 34 shows the layout of QSPI for multiboot using three images. The first image is the FSBL image, located at 0x0. The second image is the update image, located at 0x400000. The third image is the golden image, located at 0xA00000. All images use identical FSBLs. The start addresses of the second and third images can be changed based on the image size.

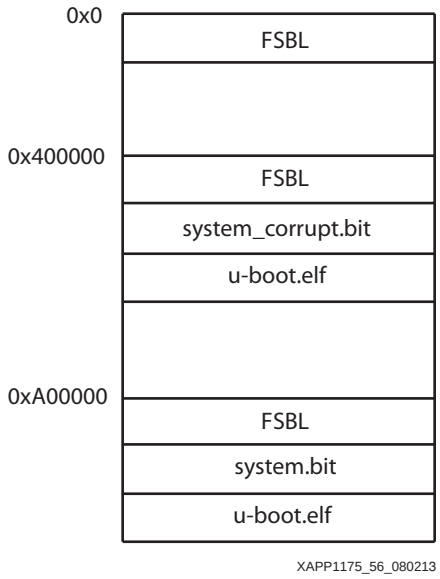


Figure 34: **zc702_multiboot Flash Layout**

The BIFs for the creating the three images are the `bootimage_fsbl.bif`, `bootimage_update.bif`, and `bootimage_golden.bif` files. Create MCS files for the three images using the following Bootgen commands:

```
bootgen -image bootimage_fsbl.bif -o fsbl.mcs -encrypt efuse
bootgen -image bootimage_update.bif -o update.mcs -encrypt efuse
bootgen -image bootimage_golden.bif -o golden.mcs -encrypt efuse
```

Note: In the xapp1175/zc702_multiboot directory, there a five multiboot systems. This section provides generic instructions to any of the examples. Change to the multiboot system directory of interest (e.g. ps_autherr) and follow the steps listed in this section.

For the zc702_multiboot examples, the update image is intentionally corrupted. Note that the image output of Bootgen is corrupted, not the partition input into Bootgen. In multiboot, the corrupted image is detected in the boot process, followed by a load of the uncorrupted golden image.

As an example, in the xapp1175/zc702_multiboot/ps_autherr system, the intent is to induce a RSA error in the software (PS). After creating `update.mcs` as shown previously in this section, use

```
cp update.mcs update.mcs$
```

so that an original version is kept. The diff instruction can be used to verify that the corruption was done. The corruption to induce an RSA authentication error is done in line 15900 of `update.mcs`. The `update.mcs$` is the original update MCS file, which in this case is the same as the golden MCS image.

Figure 35 shows the use of the **SDK > Program Flash** to write the FSBL image to QSPI location 0x0. Repeat this process, programming `update.mcs` to location 0x400000, and `golden.mcs` to location 0xA00000.

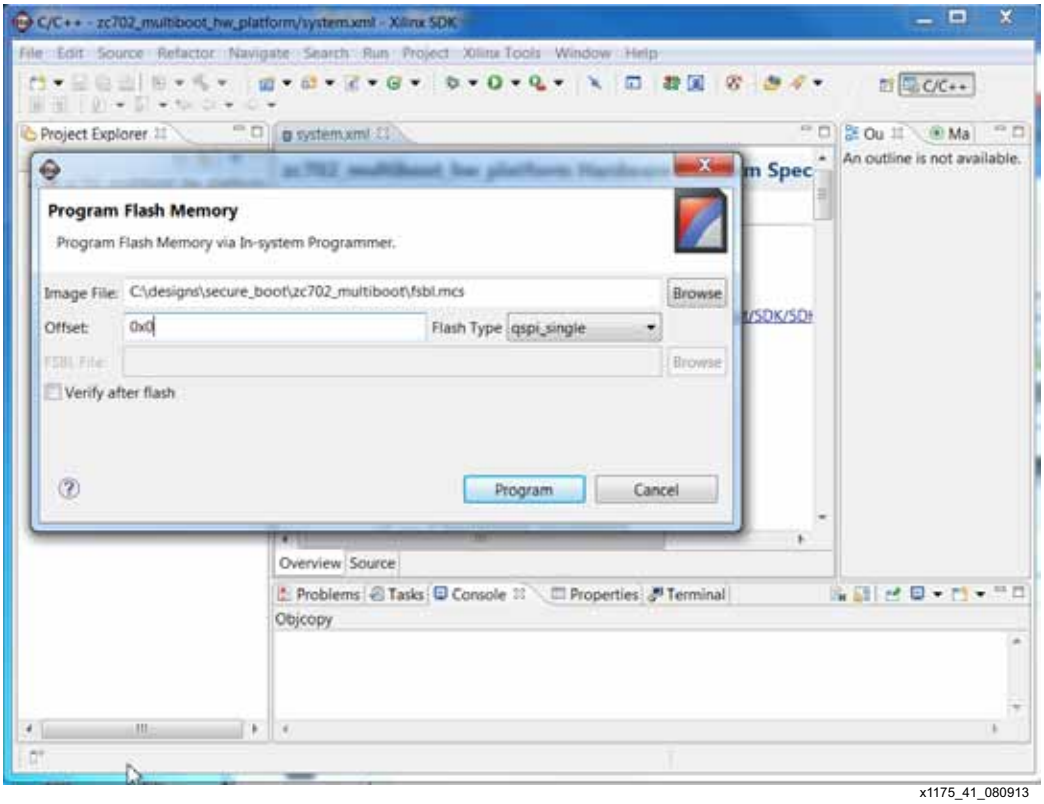


Figure 35: **Programming FSBL Image**

Note: In software releases prior to 14.7 the SDK Program Flash may not program multiple images into QSPI correctly.

A second method to load the three images for the zc702_multiboot examples is to use U-Boot. Use the following steps to use U-Boot to load the images.

1. Create the images with the BIN output format using the following bootgen commands:

   ```
   bootgen -image bootimage_fsbl.bif -o fsbl.bin -encrypt efuse
   bootgen -image bootimage_update.bif -o update.bin -encrypt efuse
   bootgen -image bootimage_golden.bif -o golden.bin -encrypt efuse
   ```

2. Corrupt update.bin as follows:

   ```
   cp update.bin update.bin$
   ```

   Use a hex editor (such as gvim or hd) as in the following example:

   ```
   gvim update.bin
   ```

   The bitstream starts at 0x194C0.

   Change a character in a line such as line 19570.

   Save the update.bin file.

3. Verify the change with:

   ```
   diff update.bin update.bin$
   ```

4. In the zc702_multiboot/ready_for_download directory, the `BOOT.bin` file includes the FSBL and U-Boot partitions. When this `BOOT.bin` is booted from the SD card, U-Boot is

run on Zynq. Copy `BOOT.bin`, `fsbl.bin`, `update.bin` and golden.bin to the SD card. Set the Boot Mode to **SD**. Open a communication terminal and power cycle.

Enter the commands in steps 5-12 at the U-Boot prompt.

5. mmcinfo
6. fatload mmc `0 0x100000 fsbl.bin`
7. sf probe `0 0 0`
8. sf write `0x100000 0 0x20000`
9. fatload mmc `0 0x100000 update.bin`
10. sf write `0x100000 0x400000` ${filesize}
11. fatload mmc `0 0x100000 golden.bin`
12. sf write `0x100000 0xA00000` ${filesize}
13. Power down. Change from SD to QSPI boot mode. Power up.

Figure 36 shows the programming of QSPI using U-Boot for multiboot operation. This is a display of the commands provided in steps 5-12 of this procedure.

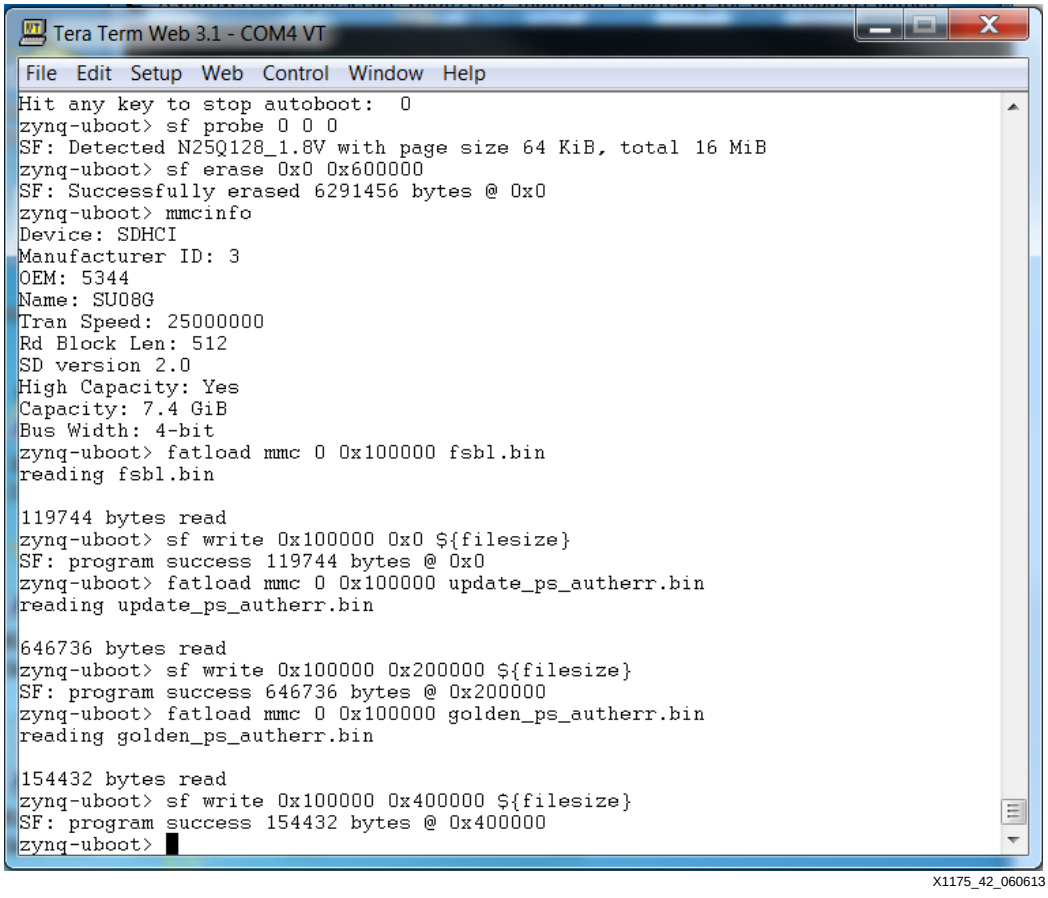


Figure 36: **Programming with U-Boot**

View the log output displayed in the communication terminal to verify that multiboot functions as expected. If the multiboot successfully loads the golden image, the U-Boot prompt is displayed in the communication terminal window, and DS19 blinks after the successful load of system.bit. This indicates that the multiboot procedure successfully loads the golden image after the load of the update image failed.

The FSBL debug log provides definitive verification that the multiboot operation functions as designed. The FSBL debug log indicates the number of partitions, and for each partition, the

load address, length, whether the partition is encrypted and/or RSA signed. All DEVC register values are displayed for each partition. These registers are defined in the *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [Ref 6]. While DEVC register values make the debug log relatively long, much of the information is repetitive, so the debug log is easy to read. A portion of the FSBL debug log file is shown in Figure 37.

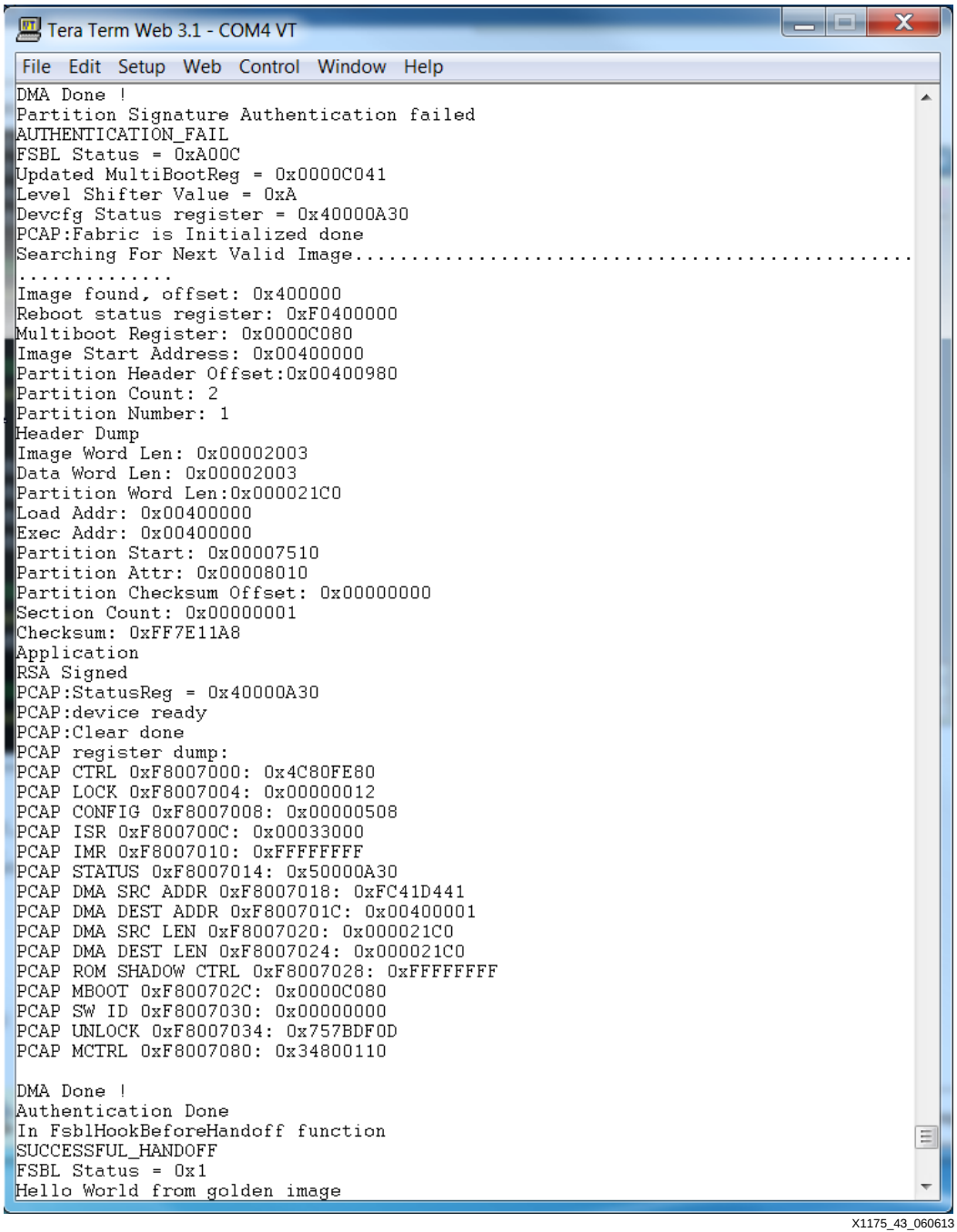


Figure 37: **FSBL Debug Log Output**

In the communication terminal, save the log file as shown in Figure 38. This allows the complete debug log to be reviewed easily in a text editor.

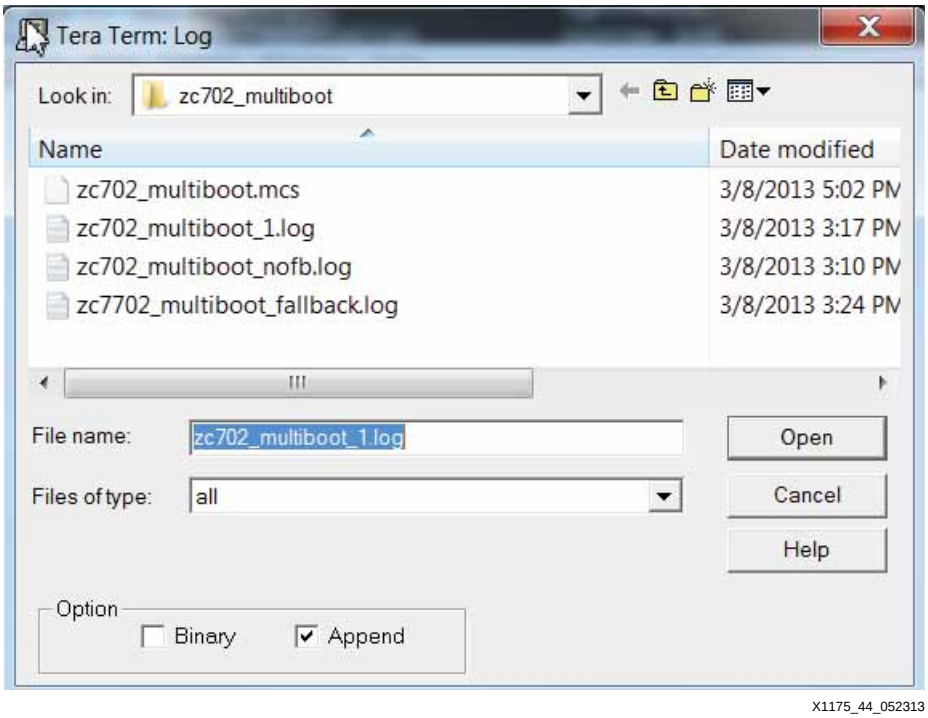


Figure 38: **Capturing Multiboot Log**

The multiboot reference design provides several multiboot systems. To test that multiboot functions correctly in response to a contrived error, the systems require the development of a working image and an image which fails in a specific manner. Since the failure mode is not always the expected failure mode, the user must examine the log file to verify that the cause of the multiboot is the one intended.

In some multiboot systems, the error is generated correctly, but the error is not the cause of the multiboot. The BIF needs to be constructed so that the expected failure mode occurs. Using the zc702_pl_encerr system as an example, an authentication error can mask an encryption error. To prevent this, omit the [authentication=rsa] attribute on the partition in which an encryption error is intended.

Table 3 lists zc702_multiboot systems. In the examples, independent hello_update and hello_golden ELFs are used. The C print statements in the hello partitions indicate which partition is running. In the zc702_ps_autherr system, the golden image is booted after an authentication error in the software (PS).

Table 3: **Multiboot Examples**

Project	BIF	Error	Log
zc702_ps_autherr	ps_autherr.bif	update_ps_autherr.bin	ps_autherr.log
zc702_pl_autherr	pl_autherr.bif	update_pl_autherr.bin	pl_autherr.log
zc702_pl_encerr	pl_encerr.bif	update_pl_encerr.bin	pl_encerr.log
zc702_ps_checksum	ps_checksum.bif	update_s_checksum.bin	ps_checksum.log
zc702_pl_checksum	pl_checksum.bif	update_pl_checksum.bin	pl_checksum.log

The steps to create and run the zc702_ps_autherr project are:

1. Use the following BIF, `bootimage_update.bif`, for the update image.

```
the_image
{
[pskfile] psk.pk1
[sskfile] ssk.pk1
[aeskeyfile] efuse.nky
[bootloader, encryption=aes, authentication=rsa] fsbl.elf
[authentication=rsa] hello_update.elf
}
```

2. Use Bootgen to create the `update.bin` file:

```
bootgen -image bootimage_update.bif -o update.bin -encrypt efuse
Use the gvim text editor (or similar) to insert an error:
gvim update.bin
```

3. As shown in Figure 39, run:

```
Tools -> Convert to Hex
```

to view update.bin file in hex format.

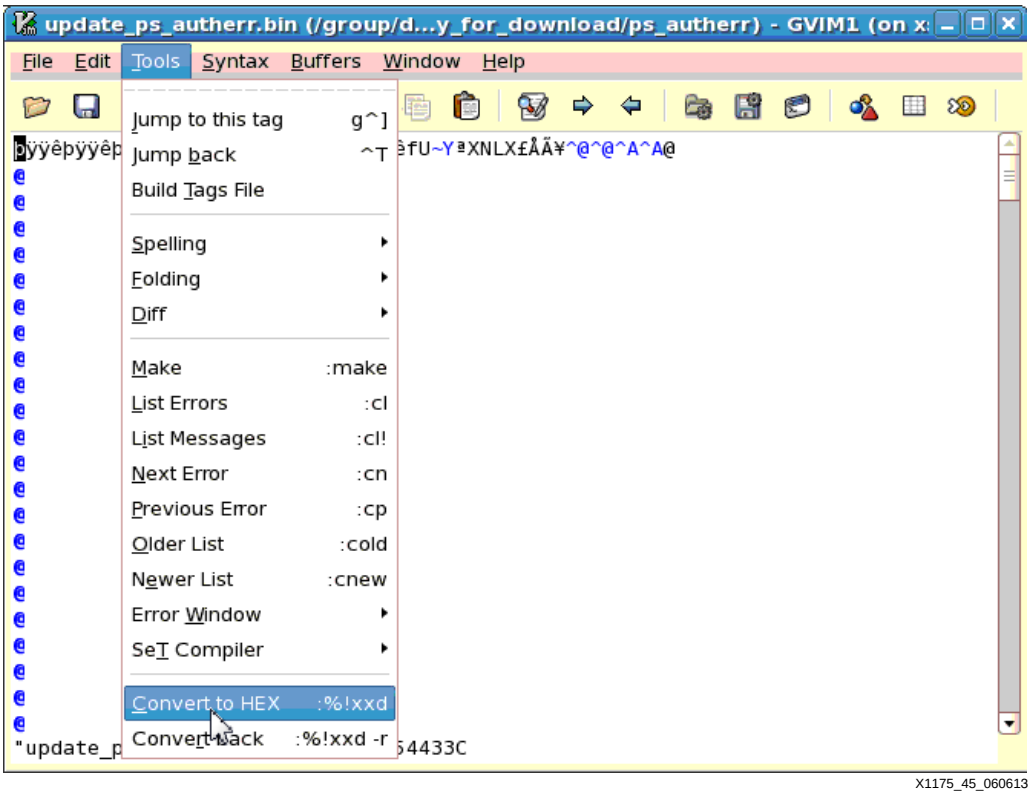
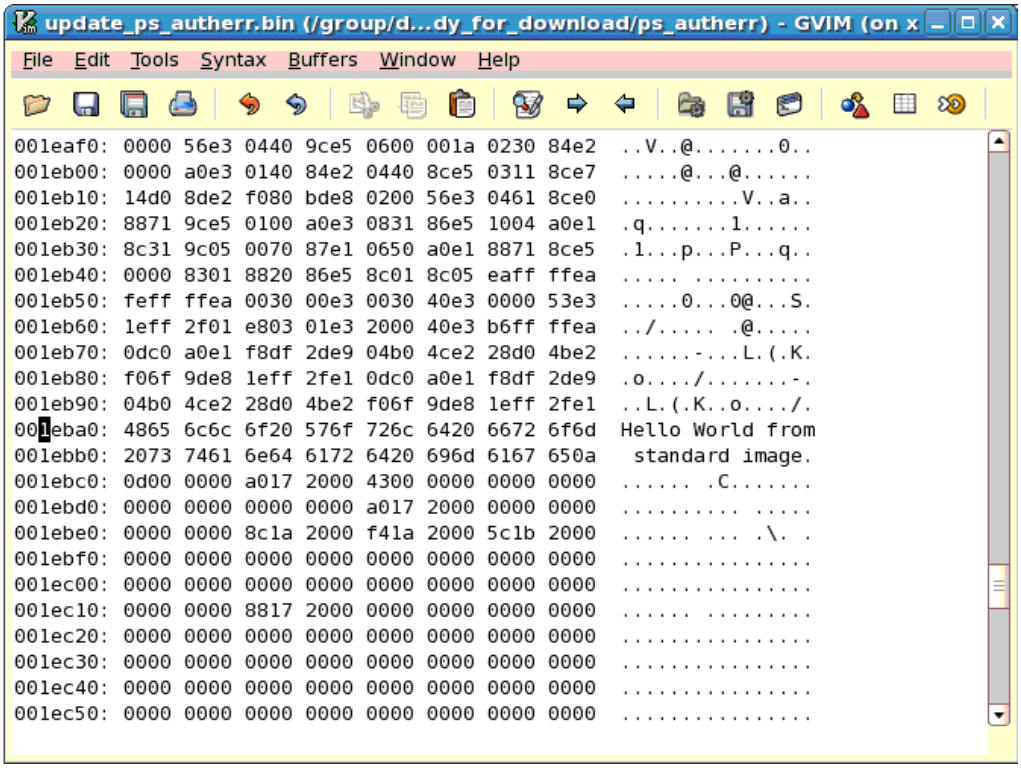


Figure 39: **Using gvim to Generate An Error**

4. As shown in Figure 40, search for `Hello`, which occurs on line 01eba0.



X1175_46_060613

Figure 40: **Inducing an Authentication Error**

5. As shown in Figure 41, change the first nibble on line 1eba0 from 4865 to 5865. Notice that Hello changes to Xello. Enter

   ```
   Tools > Convert Back
   ```

   and save `update.bin`.

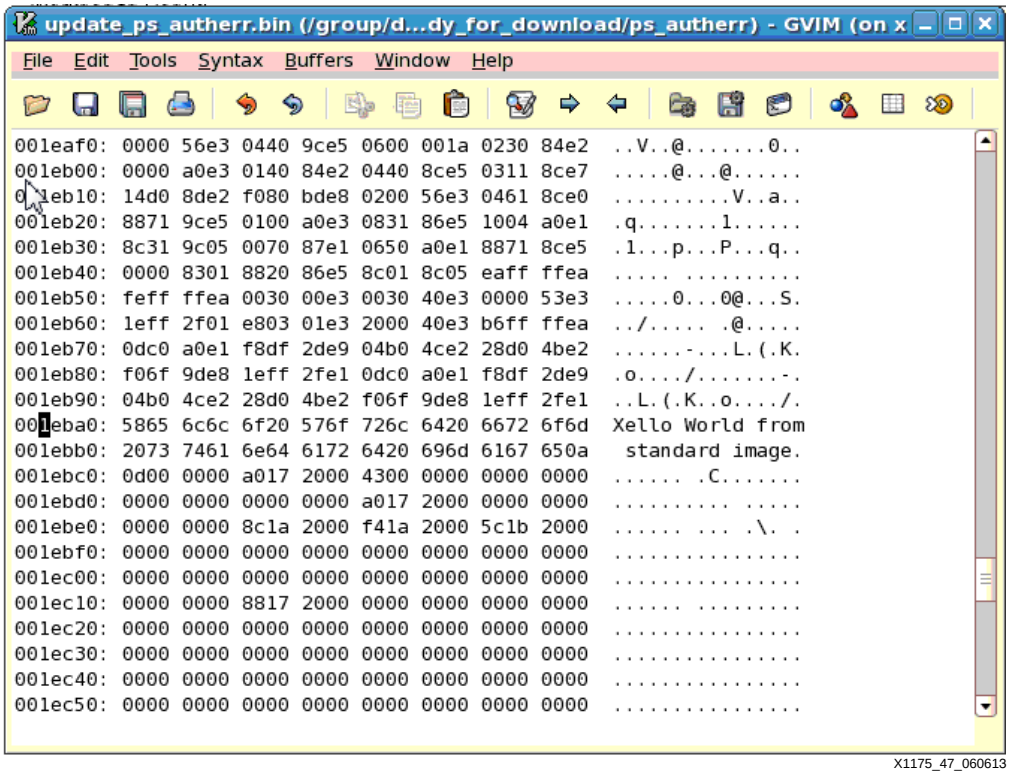


Figure 41: **Update File with PS Authentication Error**

As shown earlier in this section, run steps 5-13 at the U-Boot prompt to write QSPI and boot the system. This process can be used for all of the multiboot examples. This procedure can be used to construct custom multiboot systems.

FSBL Debugging

In the previous section, the intent is to show how to develop a multiboot solution rather than provide an end solution. The FSBL has a central role in multiboot. The FSBL debug log is effective at understanding the behavior of the FSBL during the multiboot process. A second approach to analyze the multiboot process is to use GDB to step through the FSBL. Analyzing the FSBL in GDB is useful in understanding the behavior of the FSBL in loading partitions, multiboot, and RSA authentication.

Figure 42 shows the setup of a debug session in SDK. Click

Run > Debug Configuration

and select the `fsbl.elf` to debug.

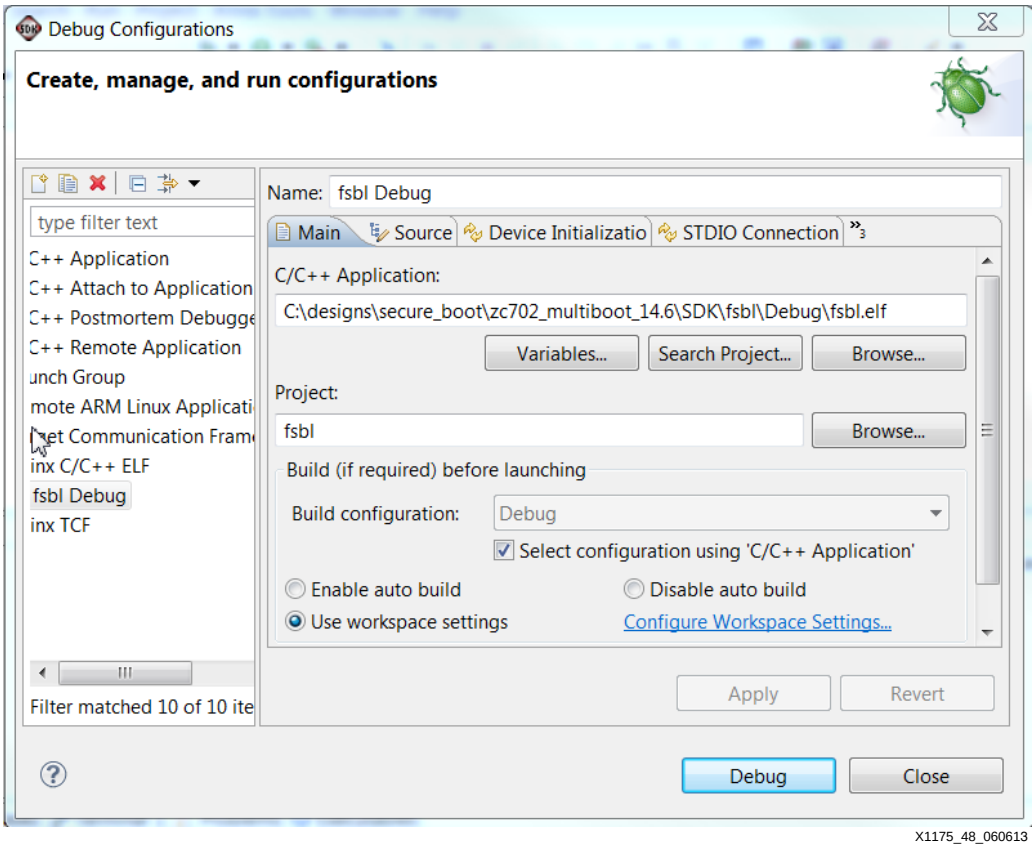


Figure 42: **Starting a Debug Session in SDK**

Figure 43 shows the GDB debugger in SDK. This is the ps_autherr system in the multiboot reference designs. In this system, RSA authentication fails in the hello_update partition. The debugger location is the code pointing to the Authentication Certificate.

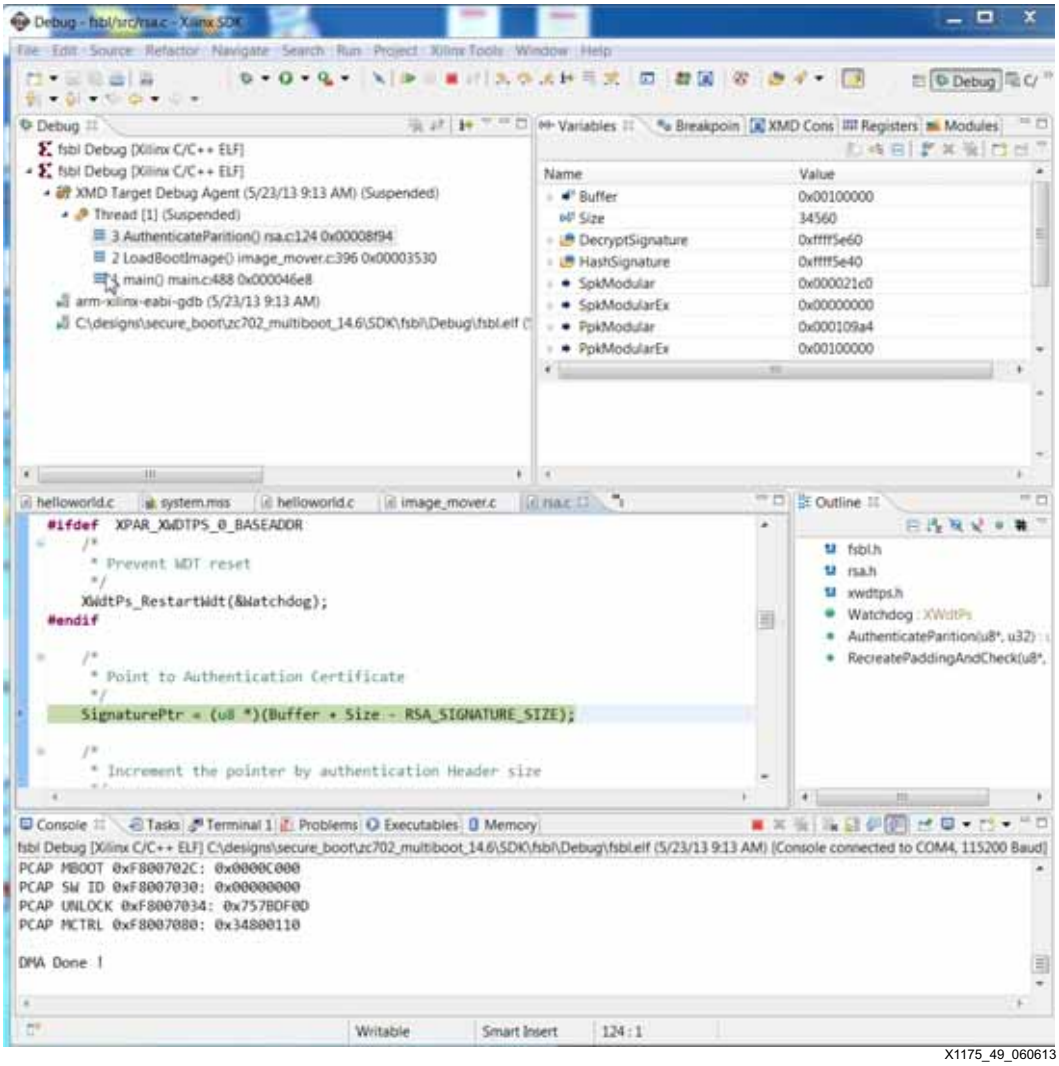
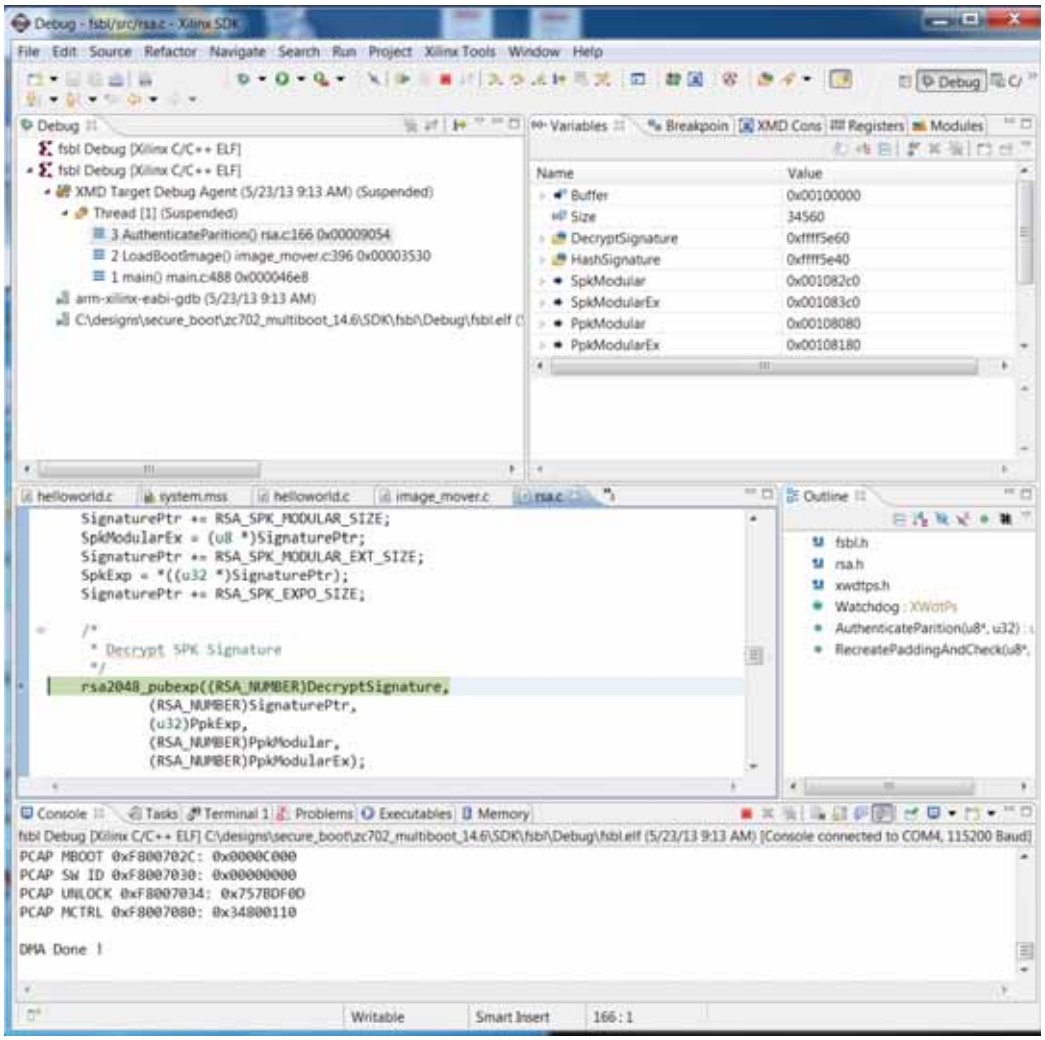


Figure 43: **Running the GDB Debugger on the FSBL in SDK**

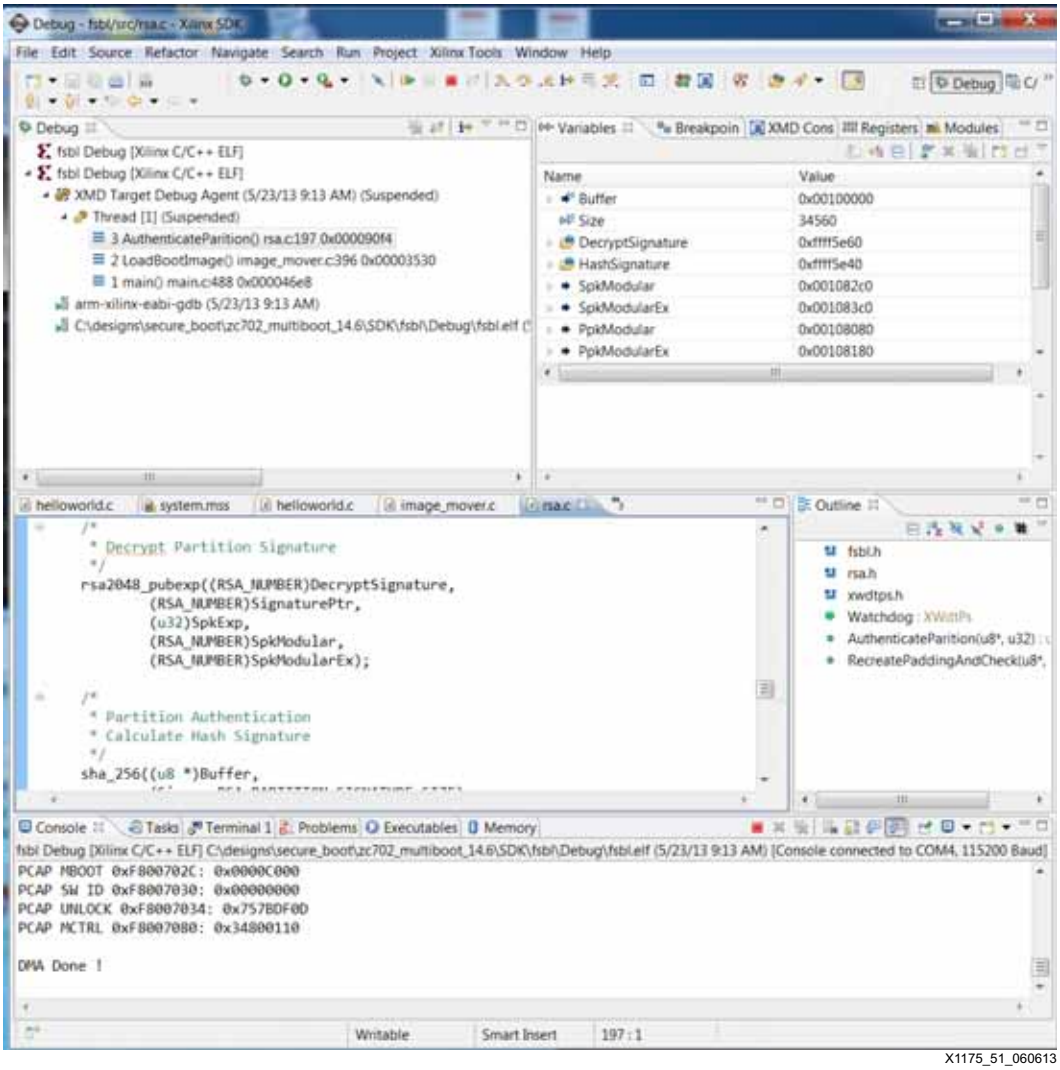
Figure 44 shows the GDB debugger after stepping to the location which verifies the SPK signature.



X1175_50_060613

Figure 44: **RSA Verification of SPK**

Figure 45 shows GDB at the RSA code which verifies the partition.



X1175_51_060613

Figure 45: **RSA Verification of Partition**

Measuring Boot Time

Boot times are a function of the embedded device boot mode, the NVM configuration and speed, and the size of the partitions loaded. In most cases, the configuration of NVM is different than that of the ZC702 Evaluation Board. Contact a Xilinx field application engineer (FAE) for boot time estimates for non-secure and secure boot modes. For information on flash devices supported in the Zynq-7000 AP SoC tools, refer to AR50991.

The zc702_linux_trd image can be used to determine an approximate order of magnitude of the QSPI boot time. When the ZC702 board is used, it is relatively easy to compare non-secure and secure boot times by creating BIFs which include and exclude the security functionality. There is not a measurable difference in the non-secure and secure boot times for the Petalinux builds.

A second method of investigating boot time is to enable the FSBL performance measurement as shown in Figure 46.

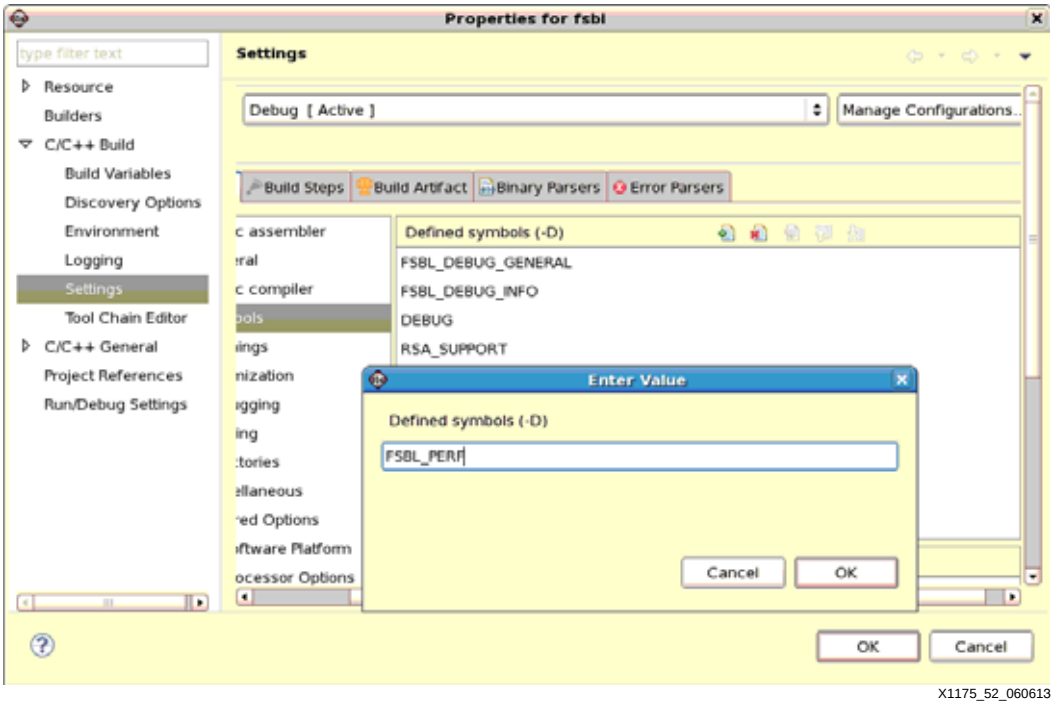


Figure 46: **Setting the FSBL_PERF Option in SDK**

User Defined Field in Authentication Certificate

The BIF for use case 15 provides a User Defined Field (UDF) in the Authentication Certificate. The following are potential uses of the 56 Byte UDF at offset 0x8.

- Software Versioning
- Software Provided Certificate
- Time Stamp
- Partition Identifier/Version

The UDF is written using Bootgen with the flow given in this section. In the device, processing the information in the UDF is typically done in the FSBL. The FSBL code must be written by the user.

To generate the user defined field, create the `uboot_v10.hex` file and a BIF as follows.

the_image:

```
{
[pskfile] psk.pk1
[sskfile] ssk.pk1
[aeskeyfile]efuse.nky
[bootloader, authentication=rsa, encryption=aes]fsbl.elf
[authentication=rsa, udf_data=uboot_v10.hex] u-boot.elf
}
```

The scope of the udf_data attribute is limited to the partition for which it is specified, `u-boot.elf` in the previous BIF. The hex file can contain up to 56 bytes. If the hex file contains less than 56 bytes, Bootgen pads the user defined field (UDF) with 0s to extend the UDF to 56 bytes. An error occurs if the udf_data attribute is specified for a partition in which authentication=none, the hex file does not exist or is not readable, it contains more than 56 bytes, or uses a format other than hex.

As an example, suppose `zc702_udf.hex` contains

1234 5678 9ABC DEF0 9456 f232 227b dd70.

Figure 47 shows the UDF in the authentication certificate for this zc702_udf.hex.

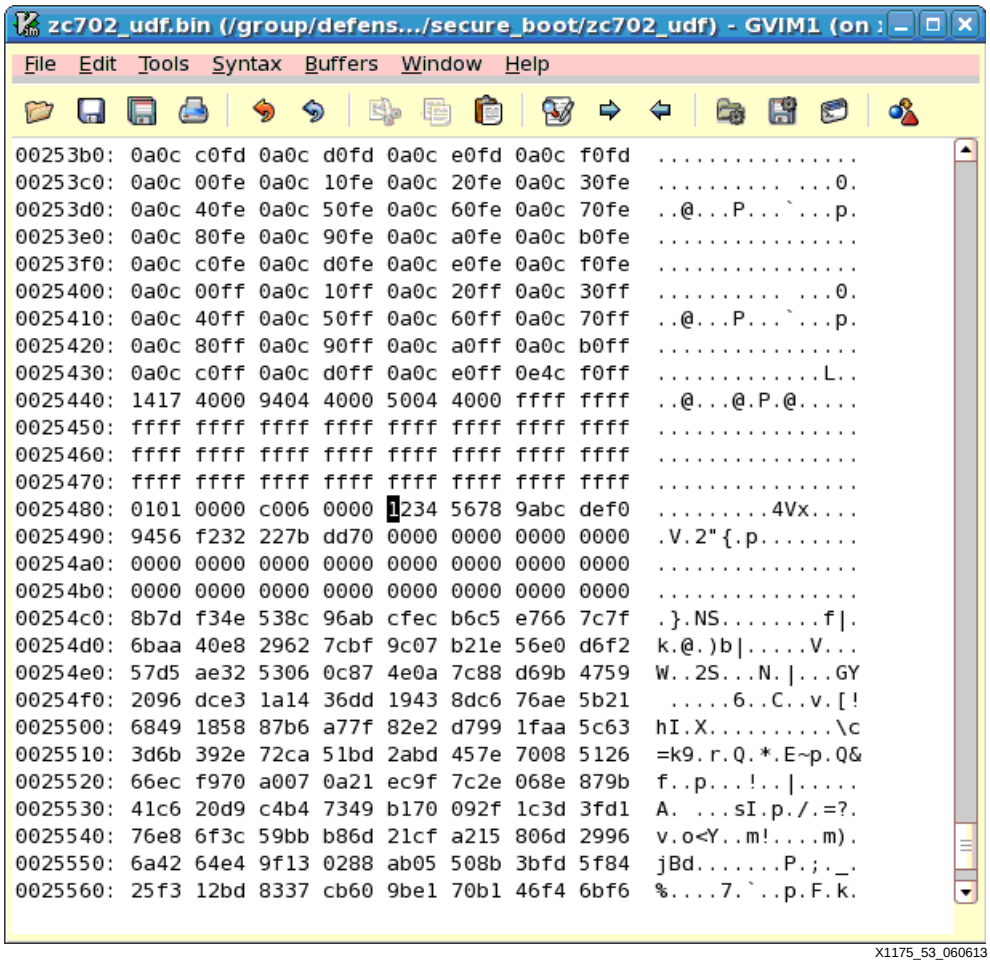


Figure 47: **User Defined Field in the Authentication Certificate**

Loading Data Partitions

In addition to ELF and BIT partitions, Bootgen can load data partitions. Typical application requirements for data partitions are DSP coefficients or health and financial records. Bootgen attributes allow the data partition to be encrypted and/or authenticated. An example BIF is:

```
the_image
    {
    [bootloader, encryption=aes, authentication=rsa] fsbl.elf
    [encryption=aes, authentication=rsa] hello.elf
    [encryption=aes, authentication=rsa, load=0xFFFFC000] coefficients.bin
    }
```

An example data file, coefficients.bin, contains 0101010111001100.

To verify that the data file is loaded into OCM, disable the JTAG port if necessary, and run a XMD mrd 0xFFFFC000 8 command.

Using the DEVCFG and SLCR Registers for Boot Options

The DEVCFG registers used in boot are the Control, Lock, CFG, and MCTRL registers, located at offsets from 0xF8007000. There are also general lock registers in the SLCR located at

0xF8000000. The TRM provides register definitions. Since the CTRL register is used often, Table 4 defines this register for reference. Some control bits are triplicated for enhanced safety and security.

Table 4: **Control Register at 0xF8007000**

Name	Bits	Description
FORCE_RST	31	Forces PS into secure lockdown
PCFG_PROG_B	30	Resets the PL
PCFG_POR_CNT_4K	29	Controls the POR timer
Reserved	28	
PCAP_PR	27	Selects between PCAP and ICAP for PL reconfiguration
PCAP_MODE	26	Enables PCAP interface
PCAP_RATE_EN	25	Selects data rate
MULTIBOOT_EN	24	Enable multiboot out of reset. Cleared by PS_POR_B.
JTAG_CHAIN_DIS	23	Disables JTAG Chain
Reserved	22:16	
User Mode	15	0 indicates CPU is executing BootROM code
Reserved	14:13	
PCFG_AES_FUSE	12	0 - BBRAM, 1 - EFUSE
PCFG_AES_EN	11:9	000 - Disable AES; 111 - Enable AES; Others - Lockdown
SEU_EN	8	0 - Ignore SEU signal from PL; 1 - Lockdown if SEU received.
SEC_EN	7	0 - PS was not booted securely; 1- PS was booted securely
SPNIDEN	6	0 - Disable Non-invasive Debug
SPIDEN	5	0 - Diable Secure Invasive Debug
NIDEN	4	0 - Disable Non-invasive Debug
DBGEN	3	0 - Disable Invasive Debug
DAP_EN	2:0	111 - Enable ARM DAP

JTAG Debug

In Zynq-7000 AP SoC, the JTAG port is used to load software and the bitstream, load the AES key, control information, and for debug. If not disabled, JTAG ports can be used by an adversary to insert malware, and read configuration memory and registers. The JTAG ports must be disabled whenever it is not used in a legitimate debug operation.

The device can be debugged using a DAP controller and/or a JTAG controller. The DAP JTAG chain and PL JTAG chain can be concatenated or used independently. When used independently, the full SoC/FPGA does not need to be exposed to an adversary. For example, if debug only requires access to the PL, the user can select that only PL JTAG chain is used. This prevents access to the PS.

Figure 48 shows the independent and cascaded JTAG chains.

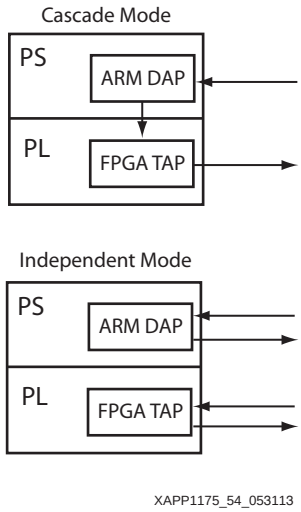


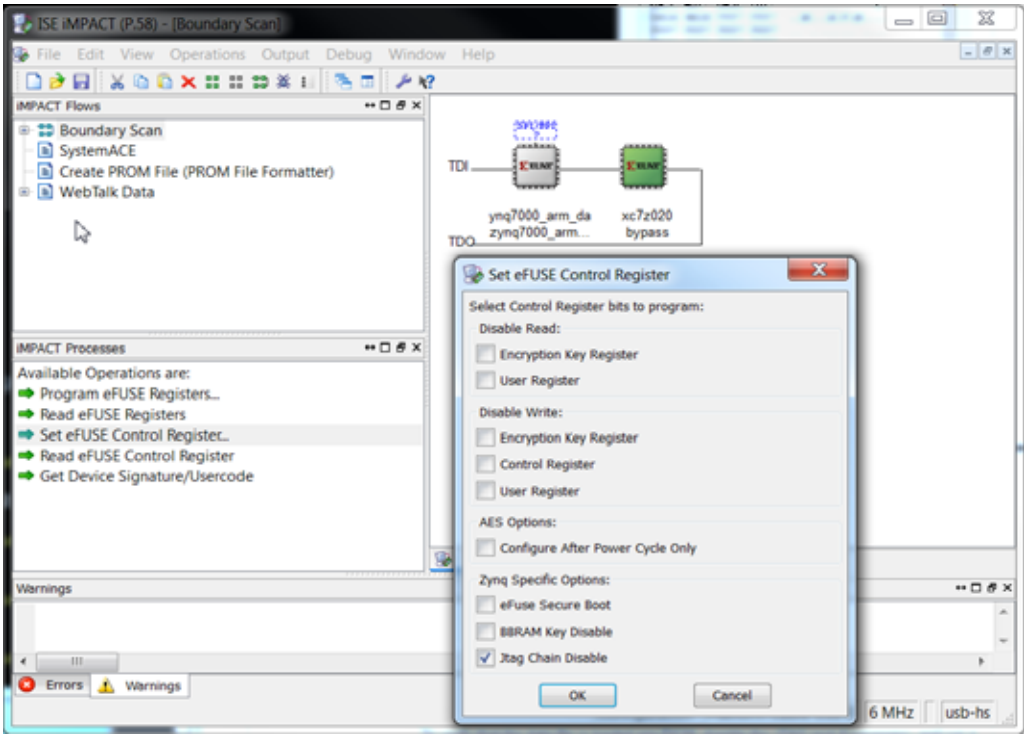
Figure 48: **JTAG Chains**

Zynq provides hierarchical control of the JTAG port. This allows different methods to control access to the debug ports based on security requirements. Security requirements may change over the life cycle the embedded device. There are three methods to disable the JTAG debug ports. Prior to fielding an embedded device with Zynq, a one-time programmable eFUSE bit can permanently disable access to the debug ports. Programming this eFUSE bit is irreversible, and debug ports remain disabled after powering down and recycling power.

The second method, which can be used only if the debug port disable eFUSE is not blown, disables/enables debug access port using the JTAG_CHAIN_DISABLE, DAP_EN, SPINDEN, SPIDEN, NIDEN, DBGEN bits in the Control register at 0xF8007000 (see Table 4). The debug access control is provided independently for the two JTAG chains or the concatenated chain.

In the third method, a lock register provides semi-permanent disabling of access to the debug ports. In this method, the debug port access disable remains in effect until a power cycle.

In a secure boot, the JTAG port is disabled early by the BootROM code. Users who will not use the debug port after product release can disable the JTAG port permanently by writing the eFUSE Disable JTAG register. The disable is done using iMPACT or the Secure Key Driver. Figure 49 shows using iMPACT to write the eFUSE which disables the JTAG port.



X1175_55_060613

Figure 49: **Disabling the JTAG Port Using iMPACT**

To use the Secure Key Driver, change the following line in xilskey_input.h.

#define XSK_EFUSEPL_DISABLE_JTAG_CHAIN TRUE

Use the steps provided in the Secure Key Driver section to compile and run the driver.

Disabling JTAG Using the DEVCFG CTRL Register

If the JTAG_CHAIN_DISBLE eFUSE is not blown, the CPU can enable the JTAG port by writing to CTRL(23)= 0x0. After a secure boot, enabling the JTAG port is necessary to debug a Zynq device.

The zc702_jtag_en system shows how to enable JTAG after a secure boot. The BIF creates a secure system, with authentication and encryption. In this project, the FSBL is modified so that the JTAG port is unlocked at the end of FSBL execution. The following code is added to the FsblHookBeforeHandoff function in `fsbl_hook.c`.

```
ctrl_reg = Xil_In32(0xF8007000);
        fsbl_printf(DEBUG_INFO,"Before 0xF8007000 = 0x%08x\r\n");
        ctrl_reg = ctrl_reg & (~(1<<23));
        ctrl_reg = ctrl_reg | 0x7F;

Xil_Out32(0xF8007000,ctrl_reg);

ctrl_reg = Xil_In32(0xF8007000);
fsbl_printf(DEBUG_INFO,"After 0xF8007000 = 0x%08x\r\n");
```

Without this added FSBL code, the JTAG port is locked, and cannot be accessed with iMPACT or XMD. With this code, JTAG access is enabled at the end of FSBL execution. Secure systems require a more sophisticated method of providing access to the JTAG port. After debug, the JTAG port must be disabled. This can be done using a basic modification of the code previously provided in this section to enable the JTAG port. Suppose a user or technician needs to debug an embedded device which has booted securely. A GPIO interrupt tied to a pushbutton on the

board is generated, requesting JTAG access. The interrupt handler verifies a password before enabling the JTAG port. The user or technician indicates that the debug session is done by pressing another button tied to a second GPIO interrupt. The interrupt handler then disables the JTAG port.

Conclusion

Secure boot is easy to implement in Zynq-7000. Since Zynq-7000 provides the functionality without using the resources of the PL, the incremental cost to boot securely is minimal. Secure boot protects the embedded system against a number of malicious attacks. Zynq provides security options to support different security requirements.

Appendix A

Glossary - Acronyms

The following terms are used in this application note. In most cases, the terms are defined in the Boot Architecture section.

- Advanced Encryption Standard/ Hashed Message Authentication Code (AES/HMAC)
- Authentication Certificate (AC)
- Bitgen
- Bootgen
- Boot Header (BH)
- Boot Image Format (BIF)
- Battery Backed RAM (BBRAM)
- Device Configuration Interface (DevC)
- eFUSE array
- Secure Key Driver
- First Stage Boot Loader (FSBL)
- ISE Design Suite
- Image
- iMPACT
- Partition
- Partition Header
- Programmable Logic (PL)
- Processor System (PS)
- Primary Secret Key (PSK)
- Primary Public Key (PPK)
- Rivest, Shamir, Adleman (RSA)
- Secondary Secret Key (SSK)
- Secondary Public Key (SPK)
- Software Development Kit (SDK)
- Secure Hash Algorithm (SHA)
- Secure Storage
- U-Boot
- Xilinx Platform Studio (XPS)

Appendix B

Use Cases for User Selectable Security Functionality

Using BIF file attributes, users specify, on a partition basis, if a partition is to be RSA authenticated and if it is to be AES/HMAC encrypted/authenticated. Table 5 provides sample use cases of images in which AES/HMAC encryption and RSA authentication are specified on a partition basis. Other use cases are possible. Most of the use cases in Table 5 contain the same partitions used in the TRD. Use cases 11-14 show a single data partition in the BIF. Like software and bitstream partitions, data partitions can be included in the initial boot image (i.e. with an FSBL partition).

Table 5: **Use Cases for Specifying Security**

	BootROM RSA	RSA	AES/HMAC
Use Case 1 - NonSecure Boot			
fsbl.elf			
system.bit			
u-boot.elf			
uImage.bin			
devicetree.dtb			
uramdisk.image.gz			
sobel_cmd.elf			
Use Case 2 - Secure Boot, AES/HMAC partitions			
fsbl.elf			x
system.bit			x
u-boot.elf			x
uImage.bin			x
devicetree.dtb			x
uramdisk.image.gz			x
sobel_cmd.elf			x
Use Case 3 - Secure Boot, RSA authenticate FSBL, AES/HMAC partitions			
fsbl.elf	x		x
system.bit			x
u-boot.elf			x
uImage.bin			x
devicetree.dtb			x
uramdisk.image.gz			x
sobel_cmd.elf			x
Use Case 4 - Secure Boot, RSA authenticate all files			
fsbl.elf	x		x
system.bit		x	
u-boot.elf		x	
uImage.bin		x	
devicetree.dtb		x	

Table 5: **Use Cases for Specifying Security *(Cont'd)***

	BootROM RSA	RSA	AES/HMAC
uramdisk.image.gz		x	
sobel_cmd.elf		x	
Use Case 5 - Secure Boot, RSA authenticate FSBL			
fsbl.elf	x		x
system.bit			
u-boot.elf			
uImage.bin			
devicetree.dtb			
uramdisk.image.gz			
sobel_cmd.elf			
Use Case 6 - Secure Boot RSA authenticate and AES/HMAC all partitions			
fsbl.elf	x		x
system.bit		x	x
u-boot.elf		x	x
uImage.bin		x	x
devicetree.dtb		x	x
uramdisk.image.gz		x	x
sobel_cmd.elf		x	x
Use Case 7 - Secure Boot, RSA authenticate code, AES/HMAC PL bitstream			
fsbl.elf	x		x
system.bit		x	x
u-boot.elf		x	
uImage.gz		x	
devicetree.dtb		x	
uramdisk.image.gz		x	
sobel_cmd.elf		x	x
Use Case 8 - Secure Boot, AES/HMAC FSBL, PL bitstream, Application			
fsbl.elf			x
system.bit			x
u-boot.elf			
uImage.bin			
devicetree.dtb			
uramdisk.image.gz			
sobel_cmd.elf			x
Use Case 9 - Non-secure Multiboot - 3 Images shown			
Image 1			
fsbl.elf			

Table 5: **Use Cases for Specifying Security *(Cont'd)***

	BootROM RSA	RSA	AES/HMAC
Image 2			
fsbl.elf - standard			
system.bit - standard			
u-boot.elf - standard			
Image 3			
fsbl.elf - golden			
u-boot.elf - golden			
PL Bitstream - golden			
Use Case 10 - Secure Multiboot - 3 Images shown			
Image 1			
fsbl.elf	x		x
Image 2			
fsbl.elf - update			
system.bit - standard		x	x
u-boot.elf - standard		x	
Image 3			
fsbl.elf - golden	x		x
system.bit - golden		x	x
u-boot.elf - golden		x	
Use Case 11 - Non-secure Binary Data File - As Is, Not Bootgen source			
datafile.bin			
Use Case 12 - Authenticated Binary Data File			
datafile.bin		x	
Use Case 13 - AES/HMAC Encrypted Binary Data File			
datafile.bin			x
Use Case 14 - AES/HMAC Encrypted - RSA Authenticated Binary File			
datafile.bin		x	x
Use Case 15 - User Defined Field in Authentication Certificate			
Data File		x	x
Use Case 16 - Release Mode Bootgen - Authenticated U-Boot			
fsbl.elf	x		x
u-boot.elf		x	
Use Case 17 - Release Mode Bootgen - Authenticated TRD			
fsbl.elf	x		x
system.bit		x	x
u-boot.elf		x	
uImage.gz		x	

Table 5: **Use Cases for Specifying Security *(Cont'd)***

	BootROM RSA	RSA	AES/HMAC
devicetree.dtb		x	
uramdisk.image.gz		x	
sobel_cmd.elf		x	x

Appendix C

BIFs for Bootgen Debug Mode

BIFs for systems defined in Images, Partitions, and Authentication Certificates are as follows:

Use Case 1 - Non-secure boot BIF example

```
image: {
[bootloader] zynq_fsbl_0.elf
system.bit
u-boot.elf
ulinux.bin
devicetree.dtb
uramdisk.image.gz
sobel_cmd.elf
}
```

Note: If the RSA Enable eFUSE is programmed, the FSBL must be authenticated, and the BIF in Use Case 1 will not work. For this case, Use Case 5 should be used.

Use Case 2 - All partitions are encrypted

```
image: {
[aeskeyfile] system.nky
[bootloader, encryption=aes] zynq_fsbl_0.elf
[encryption=aes] system.bit
[encryption=aes] u-boot.elf
[encryption=aes] ulinux.bin
[encryption=aes] devicetree.dtb
[encryption=aes] uramdisk.image.gz
[encryption=aes] sobel_cmd.elf
}
```

Use Case 3 - FSBL is RSA authenticated; All partitions are encrypted

```
image: {
[aeskeyfile] system.nky
[pskfile] uc3_psk.pk1
[sskfile] uc3_ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[encryption=aes] system.bit
[encryption=aes] u-boot.elf
[encryption=aes] uImage.bin
[encryption=aes] devicetree.dtb
[encryption=aes] uramdisk.image.gz
[encryption=aes] sobel_cmd.elf
}
```

Use Case 4 - All partitions are RSA authenticated

Note: The FSBL and PL are authenticated using the first specified SPK file, and U-Boot, linux, and hello are authenticated with the linux_ssk.pk1 file.

```
image: {
[aeskeyfile] system.nky
[pskfile] uc4_1_psk.pk1
[sskfile] uk4_1_ssk.pk1
```

```
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[authentication=rsa] system.bit
[sskfile] linux_ssk.pk1
[authentication=rsa] u-boot.elf
[authentication=rsa] uImage.bin
[authentication=rsa] devicetree.dtb
[authentication=rsa] uramdisk.image.gz
[authentication=rsa] sobel_cmd.elf
}
```

Use Case 5 - FSBL is RSA authenticated

```
image: {
[aeskeyfile] system.nky
[pskfile] uc5_psk.pk1
[sskfile] uc5_ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
system.bit
u-boot.elf
uImage.bin
devicetree.dtb
uramdisk.image.gz
sobel_cmd.elf
}
```

Use Case 6 - All partitions are RSA authenticated and AES encrypted

```
image: {
[aeskeyfile] system.nky
[pskfile] uc6_psk.pk1
[sskfile] uc6_ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[encryption=aes, authentication=rsa] system.bit
[encryption=aes, authentication=rsa] u-boot.elf
[encryption=aes, authentication=rsa] uImage.bin
[encryption=aes, authentication=rsa] devicetree.dtb
[encryption=aes, authentication=rsa] uramdisk.image.gz
[encryption=aes, authentication=rsa] linux.image.gz
[encryption=aes, authentication=rsa] sobel_cmd.elf
}
```

Use Case 7- All partitions are RSA authenticated. FSBL, bitstream, and sobel_cmd application are AES encrypted

```
image: {
[aeskeyfile] system.nky
[pskfile] psk.pk1
[sskfile] ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[encryption=aes, authentication=rsa] system.bit
[authentication=rsa] u-boot.elf
[authentication=rsa,load=0x3000000,offset=0x100000] uImage.bin
[authentication=rsa,load=0x2A00000,offset=0x600000] devicetree.dtb
[authentication=rsa,load=0x2000000,offset=0x620000] uramdisk.image.gz
[encryption=aes, authentication=rsa] sobel_cmd.elf
}
```

Use Case 8 - AES Encrypt FSBL, Bitstream, and application

```
image: {
[aeskeyfile] system.nky
[bootloader, encryption=aes] zynq_fsbl_0.elf
[encryption=aes] system.bit
[encryption=none] u-boot.elf
```

```
[encryption=none] uImage.bin
[encryption=none] devicetree.dtb
[encryption=none] uramdisk.image.gz
[encryption=aes] sobel_cmd.elf
}
```

Use Case 9 - Non-secure Multiboot

```
image0: {
[bootloader] zynq_fsbl_0.elf
}

standard_image: {
[bootloader] zynq_fsbl_0.elf
system.bit
u-boot.elf
}

golden_image: {
[bootloader] zynq_fsbl_0.elf
system.bit
u-boot.elf
}
```

Note: The update and golden images must be located at addresses which are multiples of 32K.

Use Case 10 - Secure Multiboot

```
image0: {
[aeskeyfile] uc10.nky
[pskfile] uc10_psk.pk1
[sskfile] uc10_ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
}
standard_image: {
[aeskeyfile] uc10.nky
[pskfile] uc10_psk.pk1
[sskfile] uc10_ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[sskfile] bitstream_ssk.pk1
[encryption=aes, authentication=rsa] system.bit
[sskfile] u-boot_ssk.pk1
[authentication=rsa] u-boot.elf
}

golden_image: {
[aeskeyfile] uc10.nky
[pskfile] uc10_psk.pk1
[sskfile] uc10_ssk.pk1
[bootloader, encryption=aes, authentication=rsa] zynq_fsbl_0.elf
[sskfile] bitstream_ssk.pk1
[encryption=aes, authentication=rsa] system.bit
[sskfile] u-boot_ssk.pk1
[authentication=rsa] u-boot.elf
}
```

Note: The update and golden images must be located at addresses which are multiples of 32K.

Use Case 11 - Non-Secure Binary Data File

```
image:
{
[bootloader] fsbl.elf
hello.elf
datafile.bin
}
```

Use Case 12 - Authenticated Binary Data File

```
image: {
[bootloader, encryption=aes, authentication=rsa] fsbl.elf
[encryption=aes, authentication=rsa] hello.elf
[authentication=rsa] datafile.bin
}
```

Use Case 13 - AES Encrypted Binary Data File

```
image: {
[bootloader, encryption=aes, authentication=rsa] fsbl.elf
[encryption=aes, authentication=rsa] hello.elf
[encryption=aes] datafile.bin
}
```

Use Case 14 - AES Encrypted and RSA Authenticated Binary Data File

```
image: {
[bootloader, encryption=aes, authentication=rsa] fsbl.elf
[encryption=aes, authentication=rsa] hello.elf
[encryption=aes, authentication=rsa] data.bin
}
```

Note: The implicit attribute [encryption=none] is the default for all partitions.

Use Case 15 - User Defined Field in Authentication Certificate

```
image
{
[bootloader, encryption=aes, authentication=rsa] fsbl.elf
[encryption=aes, authentication=rsa, udf_data=hello_v10.hex] hello.elf
}
```

Appendix D

Images, Partitions, and Authentication Certificates

This section describes images, partitions, and authentication certificates. Bootgen generates one image which is loaded into NVM. An image consists of one or more partitions. If a partition in an image is RSA authenticated, an authentication certificate follows the partition in the image.

Table 6 shows a sample image format for Use Case 7 in Images, Partitions, and Authentication Certificates. The fields for the FSBL Authentication Certificate are shown for the FSBL partition. For the other partitions, only the location of the Authentication Certificate is shown. Authentication Certificates for all partitions have the same format.

Table 6: **Sample Image Format for Use Case 7**

Boot Header
Partition Header Table
FSBL Data Segment
512-Bit Alignment - Padding 0x

Table 6: **Sample Image Format for Use Case 7**

Authentication Certificate Header
User Defined Field - 56 bytes
RSA PPK - 2 x 2048 + 512 Bits
RSA SPK - 2 x 2048 + 512 Bits
RSA SPK Signature - 2048 Bits
FSBL Signature - 2048 Bits
PL Bitstream - system.bit
Bitstream Authentication Certificate
U-Boot
U-Boot Authentication Certificate
Linux - uImage.bin
uImage.bin Authentication Certificate
devicetree.dtb
devicetree.dtb Authentication Certificate
uramdisk.image.gz
uramdisk.image.gz Authentication Certificate
Sobel Cmd partition
Sobel Cmd Authentication Certificate

Bootgen generates an image which typically consists of the Boot Header, FSBL, PL bitstream and multiple software partitions.

For the use cases in Table 5, Bootgen generates images in the format shown in the next three tables.

The image format for use cases 1 and 2 is shown in Table 7. In these use cases, RSA authentication is not used.

Table 7: **Image Format for Use Cases 1 and 2**

Boot Header
Image Header Table
Partition Header Table
FSBL
PL Bitstream
U-Boot
Linux
Sobel Cmd Application

The image for use cases 3 and 5 is shown in Table 8. In these use cases, only the FSBL is RSA authenticated. In use case 3, all partitions are routed to the AES/HMAC engine. In use case 5, no partition is routed to the AES/HMAC engine.

Table 8: **Image Format for Use Cases 3 and 5**

Boot Header
Image Header Table
Partition Header Table
FSBL
Authentication Certificate
PL Bitstream
U-Boot
Linux
Sobel Cmd Application

The image format for use cases 4, 6, and 7 is shown in Table 9.

Table 9: **Image Format for Use Cases 4, 6, and 7**

Boot Header
Image Header Table
Partition Header Table
FSBL
FSBL Authentication Certificate
PL Bitstream
PL Bitstream Authentication Certificate
U-Boot
U-Boot Authentication Certificate
Linux
Linux Authentication Certificate
Sobel Cmd Application
Sobel Cmd Authentication Certificate

Partitions

Partitions are composed of two or three sections:

- Partition header, which stores information about the partition layout.
- Physical partition which contains the data and padding, optional expansion space
- Authentication certificate if the partition is authenticated

Table 10 shows the format of the Partition Header.

Table 10: **Partition Header**

Offset	Description
0x0	Partition Data Word Length
0x4	Decrypted Data Word Length
0x8	Total Partition Word Length (includes AC)
0xC	Destination Load Address (PS)
0x10	Destination Execution Address (PS)
0x14	Data Word Offset in the Image
0x18	Attribute Bits - PS - Bit 4; PL - Bit 5
0x1C	Section Count
0x20	Checksum Word Offset
0x24	Image Header Word Offset
0x24 - x38	Unused
0x3C	Header Checksum

Authentication Certificates

An authentication certificate is used with each partition (FSBL, software, and bitstream) specified to be authenticated. The format of the AC is the same for all partitions, and is shown in Table 11.

Table 11: **Authentication Certificate**

Offset	Length	Field	Notes
0x0	0x4	Authentication Certificate Header	
0x4	0x4	Authentication Certificate Length	
0x08	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 Modulus	640 Bytes, Little Endian
0x384	0x100	SPK Modulus Extension	
0x484	0x04	SPK Exponent	
0x488	0x3C	Padding	480 0s
0x4C4	0x100	SPK Signature	(sha256^D_p) mod N_p LE
0x5C4	0x100	Partition Signature	(sha256^D_s) mod N_s LE

The PPK Mod + PPK Modular Extension + PPK exponent are 516 bytes. Padding is 60 bytes or 480 bits of all 0.

For the Authentication Certificate offsets 0x140, 0x380, Bootgen computes a modular extension which is used in Montgomery reduction to decrease code verification time.

RSA authentication proceeds by calculating a SHA256 over the necessary data, which results in a 256 bit (32 byte) integer. This hash integer is padded according to PKCS #1v1.5 to 2048 bits (256 bytes). The signature blocks are calculated as the modular exponentiation of the padded 2048 bit hash, using the secret exponent (D) of the key as the exponent in the calculation. The OpenSSL equivalent function is BN_mod_exp_mont(). The SPK signature uses the primary key (denoted with a P subscript) while the Partition Signature uses the secondary key (denoted with a S subscript). All calculations are done in 2048 bit base, so that the padded hash value and the signature are 2048 bits or 256 bytes. The data is stored in little endian order, with the LSB first and the MSB last.

The padded hash integer defined in Table 12 is used in the native storage of bootgen as well as the *.sha256 files that are created with the -generate_hashes command line option.

Table 12: **PKCS #1v1.5 Padded SHA256 Hash Field and Format of *.sha256 Hash Files**

Bytes	Field	Value
0:31	Raw SHA256 hash value (little endian)	Calculated
32:50	PKCS special values	0x20,0x04,0x00,0x05,0x01,0x02, 0x04, 0x03, 0x65, 0x01, 0x48, 0x86, 0x60, 0x09, 0x06, 0x0D, 0x30, 0x31, 0x30
51	zero	0x00
52:253	padding	0xFFs
254	one	0x01
255	zero	0x00

Table 13: **RSA2048 Signature Block, and Format of *.sig Files**

Bytes	Field	Value
0:255	signature value (little endian)	(sha256 ^D_p) mod N_p LE

When using the [spksignature] or [presign=] attributes to load in an externally calculated signature block, the format must be identical to the final signature block defined in Table 13. There is no processing or reversing of byte order when reading in the signature block from an external file; the data is copied exactly into the AC.

The Authentication Header is defined in the Table 14.

Table 14: **Authentication 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: Version 1.0
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

Appendix E

Programming eFUSEs with the SVF

1. Update the directory path for the ELF field in efuse.opt. The `efuse.opt` file is provided in the zc702_secure_key_driver reference system.
2. Run:

   ```
   xmd -tcl efuse.tcl -opt efuse.opt
   ```

   As shown in Figure 50, the efuse_out.svf file is created.

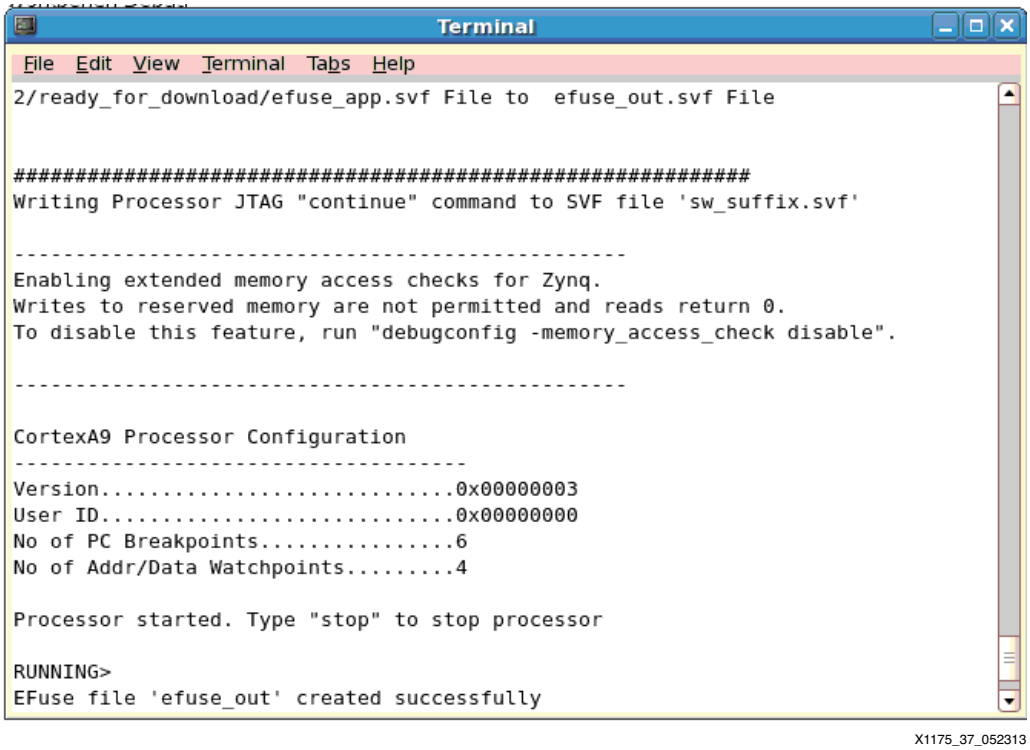
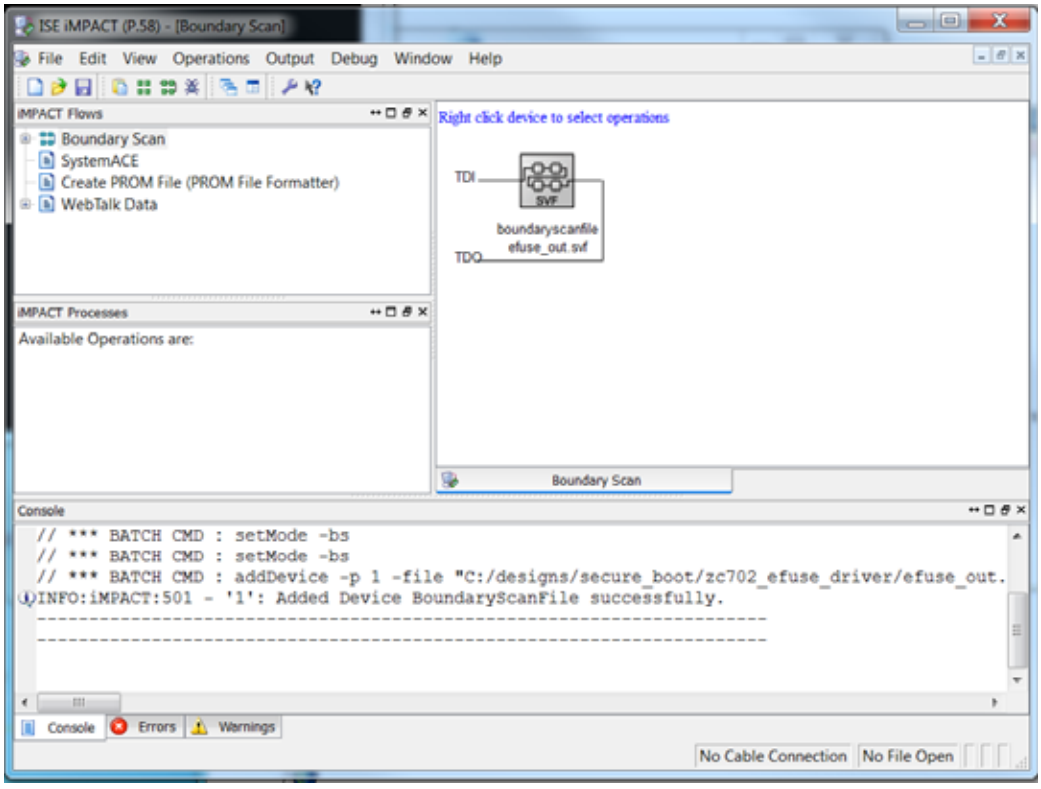


Figure 50: **SVF file efuse_out.svf**

Use iMPACT to Play the SVF file

Set up the zc702 board with either the Digilent or Platform USB cable for the JTAG interface. Set up the cable to the USB UART port. Use iMPACT to download the ELF to program the PPK hash. This is done by playing the SVF.

1. Invoke iMPACT, initialize the chain, and select **Add Xilinx device**. As shown in Figure 51, add `efuse_out.svf` to the scan chain.



X1175_38_052313

Figure 51: **Adding efuse_out.svf**

2. Browse to `efuse_out.svf`.
3. Right click and play `efuse_out.svf`.

As shown in Figure 52, this operation writes (or reads) the PPK hash. It writes the AES key.

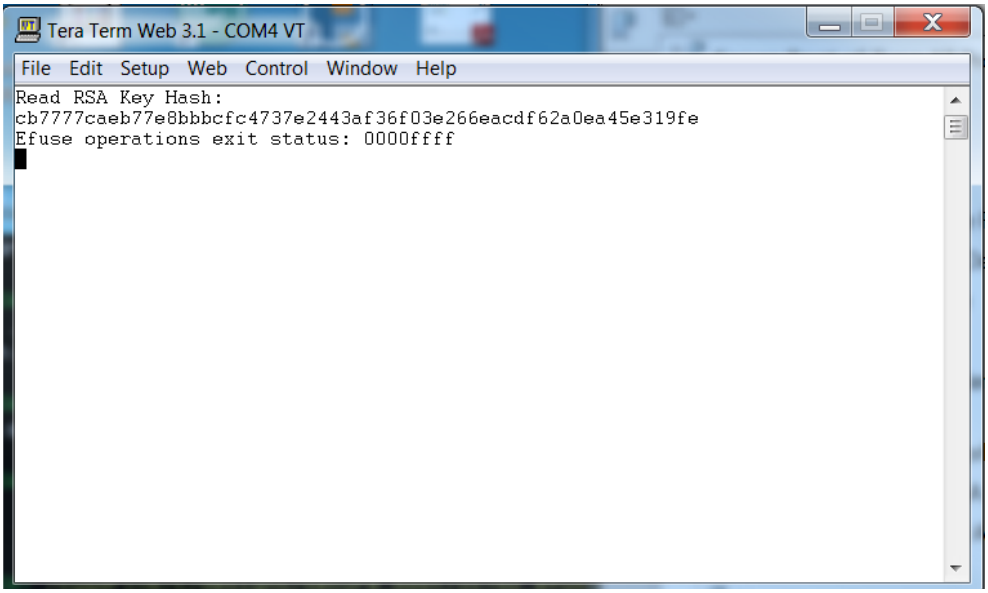


Figure 52: **eFUSE Read Operation Results**

The selections made in xilskey_input.h define the functionality (read, write) of the SVF.

References

1. *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585)
2. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)
3. *Using the Zynq-7000 Processing System to Xilinx Analog to Digital Converter Dedicated Interface to Implement System Monitoring and External Channel Measurements* (XAPP1172)
4. *Solving Today's Design Security Concerns* (WP365)
5. *OS and Libraries Document Collection* (UG643)
6. Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)
7. *OS and Libraries Document Collection* (UG643)
8. *Zynq-7000 All Programmable SoC: Concepts, Tools, and Techniques* (UG873)
9. *Zynq-7000 All Programmable SoC ZC702 Base Targeted Reference Design* (UG925)
10. *EDK Concepts, Tools, and Techniques* (UG683)
11. *7 Series FPGAs Configuration User Guide* (UG470)
12. *Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs* (XAPP1084)
13. *Solving Today's Design Security Concerns* (WP365)
14. The DENX U-Boot and Linux Guide: http://www.denx.de/wiki/DULG/Manual
15. RSA http://en.wikipedia.org/wiki/RSA
16. Montgomery Reduction http://en.wikipedia.org/wiki/Montgomery_reduction
17. OS and Libraries Document Collection (UG643)

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
09/12/13	1.0	Initial Xilinx release.

Notice of Disclaimer

Automotive Applications Disclaimer