



XAPP1278 (v1.0) March 23, 2016

# eFUSE Programming on a Device Programmer

Authors: Bryan Penner, Habib El-Khoury, and Randal Kuramoto

## Summary

This application note provides the file preparation instructions for programming the eFUSE in Zynq®-7000 All Programmable (AP) SoC devices on a device programmer. Device programmers are effective for:

- High-volume, off-board device programming.
- Preprogramming and securing sensitive data in the eFUSE before devices are sent for assembly at a board contract manufacturing site.

You can download the [Reference Design Files](#) for this application note from the Xilinx® website. For detailed information about the design files, see [Reference Design](#).

For an in-system solution for programming the Zynq-7000 AP SoC eFUSE, see *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [\[Ref 1\]](#).

Support for the solution described in this application note is available from:

- Device programming service provider:
  - Avnet (<http://www.avnet.com>)
- Device programmer vendor:
  - BPM Microsystems (<http://www.bpmmicro.com>)

**Note:** Contact the programming service provider or the device programmer vendor for the support status of your Zynq-7000 AP SoC device and package.

## Introduction to Programmable eFUSE

In Zynq-7000 AP SoC devices, the eFUSE is one-time programmable (OTP), non-volatile, and includes non-erasable settings. These settings enable device security features, such as boot image authentication and encryption, and enable a user-defined 32-bit value. For additional information on these eFUSE-enabled features, see the documentation listed in [References](#).

In each device, the user-programmable eFUSE is set to "0" when shipped from Xilinx. The eFUSE can be programmed to "1" to enable a setting or to set a user-defined value.

The Zynq-7000 AP SoC device contains two sets of eFUSE:

- Processor system (PS) eFUSE set:
  - Includes settings for Rivest, Shamir, Adleman (RSA) authentication, and a boot ROM CRC check.
- Programmable logic (PL) eFUSE set:
  - Includes settings for Advance Encryption Standard (AES) encrypted boot images, a user-defined 32-bit value, and other security settings.

---

## Using a Device Programmer for eFUSE Programming

The device programmer can be used to:

- Program the eFUSE bits in a device.
- Record the programmed device's DNA value for traceability.
- Stand-alone verify that a device has been previously programmed with expected settings.

For Zynq-7000 AP SoC devices, the device programmer uses the Xilinx eFUSE programming solution described in *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [Ref 1]. For the solution, Xilinx provides a code library for the PS ARM® processor that programs the device's eFUSE. The device programmer loads a prebuilt version of this code into each device and separately sends the user's settings into the device for programming the eFUSE.

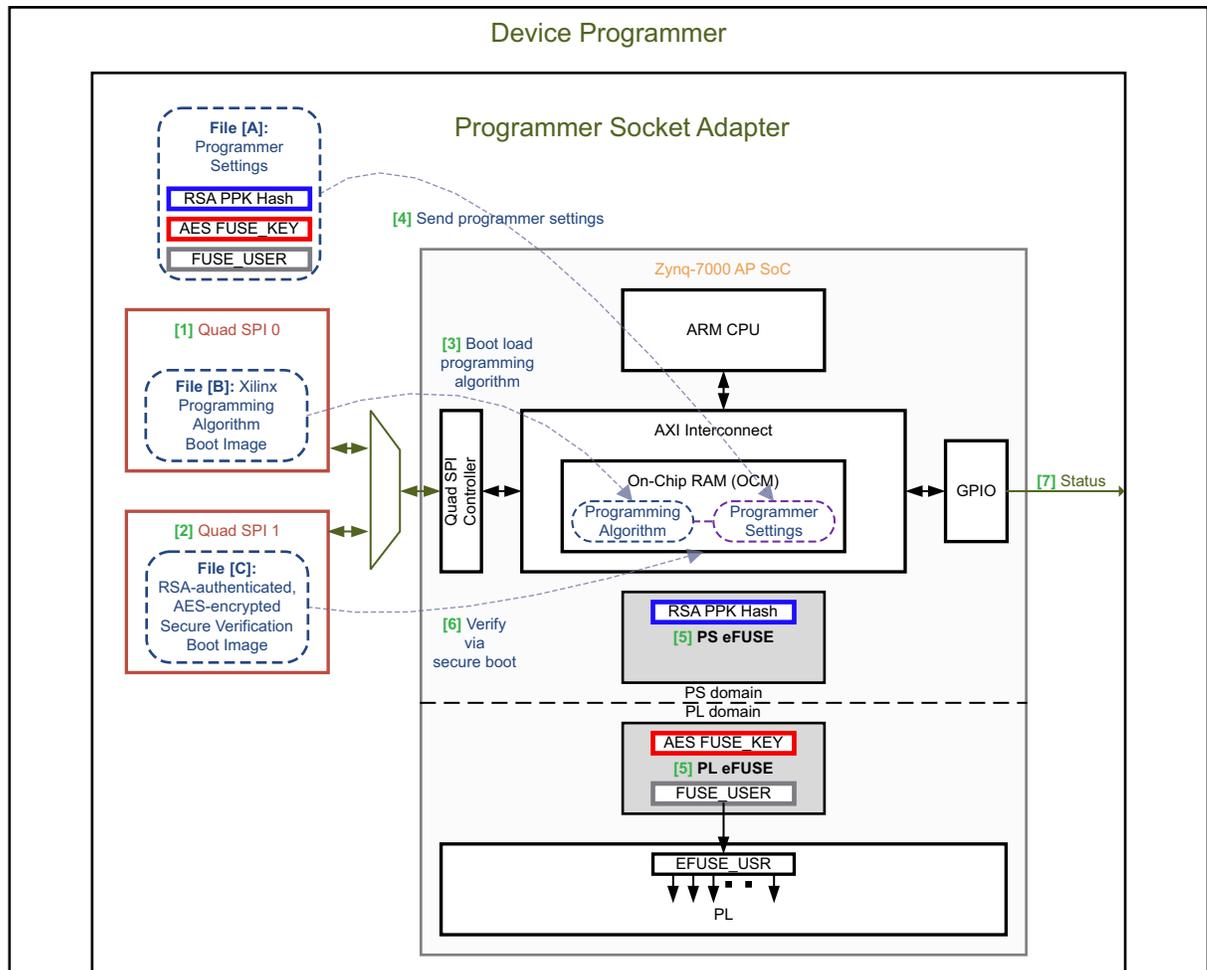
A stand-alone verify operation performs a final sign-off validation of a correctly programmed device and can be used later to revalidate programmed device inventories or RMA returned units.

Because some secure Zynq-7000 AP SoC eFUSE settings cannot be read, a traditional readback-verify method cannot be used for a stand-alone verify. Instead, a secure boot image is provided to the device programmer for the stand-alone verify operation. The device programmer attempts to load the secure boot image into the Zynq-7000 AP SoC device to indirectly validate the eFUSE security settings. If the device contains the correct settings to authenticate and decrypt the secure boot image, then the stand-alone verify is successful. Otherwise, if the device does not contain the expected secure eFUSE settings, the stand-alone verify fails due to a boot security error.

Typically, the device programmer produces a log of the programmed devices that includes device DNA identification, programming results, and stand-alone verify results.

## Overview of the Programmer and Files

Figure 1 shows a representative block diagram of a device programmer socket, files, and the flows for a Zynq-7000 AP SoC.



X15955-021116

Figure 1: Block Diagram of Device Programmer Socket Adapter and Files

For a program operation, the device programmer performs this sequence:

1. If not already programmed, the device programmer programs the Quad Serial Peripheral Interface (SPI) 0 flash memory with the Xilinx programming algorithm boot image file [B].
2. If not already programmed, the device programmer programs the Quad SPI 1 flash memory with the verification boot image file [C].
3. The device programmer boot loads the Xilinx programming algorithm from the Quad SPI 0 flash memory into on-chip memory (OCM).
4. The device programmer sends the programmer settings file [A] to OCM.
5. The device programmer allows the ARM CPU to run the programming algorithm code to program the eFUSE settings.

For a post-programming verify operation or for a stand-alone verify operation, the device programmer applies this sequence:

1. The device programmer attempts to boot the secure verification image from Quad SPI 1 flash memory.
2. If the boot image is successfully authenticated and decrypted, the verification boot image application reports a success status to the device programmer.

---

## Preparing for a Device Programmer

The required preparations for programming on a device programmer are:

1. Establish the device programmer site or service.
2. Define the security keys and eFUSE settings.
3. Create the programmer verification boot image file.
4. Create a programmer settings file.
5. Locate the Xilinx prebuilt programming algorithm image file.
6. Deliver device samples and files to the device programming site for first sample programming.

### Step 1: Establish the Device Programmer Site or Service

Contact an authorized Xilinx distributor for device programming services. Alternatively, contact a device programmer vendor to obtain a device programmer for in-house programming.



---

**IMPORTANT:** *It is critical to establish the device programmer site or service as early as possible. It can take 8-12 weeks (or longer) to develop or obtain a device programmer socket for a device-package that has not been previously established at a programming site.*

---

### Step 2: Define the Security Keys and eFUSE Settings

Define the settings for all of the parameters on the security keys and eFUSE settings sheet shown in [Table 1](#). For information on generating security keys, see *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [[Ref 1](#)]. For descriptions of the eFUSE settings, see *Secure Boot of Zynq-7000 All Programmable SoC* (XAPP1175) [[Ref 1](#)], *Zynq-7000 All Programmable SoC Technical Reference Manual* (UG585) [[Ref 2](#)], and *7 Series FPGAs Configuration User Guide* (UG470) [[Ref 3](#)].

## Security Keys and eFUSE Settings Sheet

Define all the settings in the sheet (Table 1) for your Zynq-7000 AP SoC eFUSE programming project.

**Note:** If using RSA authentication, define the primary and secondary public and private keys. The RSA hash value can be generated later during [Step 3: Create the Programmer Verification Boot Image File](#) from the RSA keys.

Table 1: Security Keys and eFUSE Settings Sheet

Security Keys			
Security Name	Description	Value Type	Setting (Define Settings Here)
RSA primary secret key (psk.pem file)	For RSA authentication. RSA primary secret key. See <i>Secure Boot of Zynq-7000 All Programmable Soc</i> (XAPP1175) [Ref 1] for a method for generating this key/file.	.pem file	
RSA primary public key (ppk.pub file)	For RSA authentication. RSA primary public key. See <i>Secure Boot of Zynq-7000 All Programmable Soc</i> (XAPP1175) [Ref 1] for a method for generating this key/file.	.pub file	
RSA secondary secret key (ssk.pem file)	For RSA authentication. RSA secondary secret key. See <i>Secure Boot of Zynq-7000 All Programmable Soc</i> (XAPP1175) [Ref 1] for a method for generating this key or file.	.pem file	
RSA secondary public key (spk.pub file)	For RSA authentication. RSA secondary public key. See <i>Secure Boot of Zynq-7000 All Programmable Soc</i> (XAPP1175) [Ref 1] for a method for generating this key or file.	.pub file	
AES key	For AES encryption. AES key. User-defined value for image encryption and stored in eFUSE for decryption.	64-digit hex value	
AES StartCBC	For AES encryption. Initialization vector for AES block cipher mode. User-defined random value.	32-digit hex value	
HMAC	For AES encryption. HMAC key for authentication during AES decryption. User-defined random value.	64-digit hex value	

Table 1: Security Keys and eFUSE Settings Sheet (Cont'd)

PS eFUSE Parameters			
PS eFUSE Parameter Name	Description	Setting Option (Default)	Setting (Define Settings Here)
XSK_EFUSEPS_ENABLE_WRITE_PROTECT	TRUE programs the PS eFUSE write protect bits to prevent future changes to the PS eFUSE. This setting takes affect after the device is power-cycled or power-on-reset. FALSE does not modify the write protect bits. See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], eFUSE Write Protection.	TRUE/FALSE (FALSE)	
XSK_EFUSEPS_ENABLE_ROM_128K_CRC	TRUE programs the PS eFUSE bit that enables the calculation and check of the boot ROM 128K CRC prior to load of the FSBL. FALSE does not modify the ROM 128K CRC bit. See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], OCM ROM 128 KB CRC Enable.	TRUE/FALSE (FALSE)	
XSK_EFUSEPS_DISABLE_DFT_JTAG	TRUE programs the PS eFUSE bit that disables the device JTAG functionality. This can limit RMA analysis. FALSE does not modify the disable design-for-test (DFT) JTAG bit. See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], DFT JTAG Disable.	TRUE/FALSE (FALSE)	
XSK_EFUSEPS_DISABLE_DFT_MODE	TRUE programs the PS eFUSE bit that disables the DFT mode. This can limit RMA analysis. FALSE does not modify the disable DFT MODE bit. See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], DFT Mode Disable.	TRUE/FALSE (FALSE)	
XSK_EFUSEPS_ENABLE_RSA_AUTH	TRUE programs the PS RSA authentication enable bit for authenticated boot from NAND flash, NOR flash, Quad SPI flash, and SD card. Before programing this bit, make sure that the RSA HASH contains the valid RSA PPK hash value by setting the XSK_EFUSEPS_ENABLE_RSA_KEY_HASH to TRUE and setting the RSA HASH to a valid RSA PPK hash value. FALSE does not modify the RSA enable bit. See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], RSA Authentication Enable.	TRUE/FALSE (FALSE)	

Table 1: Security Keys and eFUSE Settings Sheet (Cont'd)

<p>XSK_EFUSEPS_RSA_KEY_HASH</p>	<p>TRUE enables the RSA HASH parameter for editing and programs the specified RSA HASH value. FALSE ignores the RSA HASH value and does not modify the RSA HASH eFUSE bits.</p> <p><b>Note:</b> There is no eFUSE bit corresponding to this XSK_EFUSEPS_RSA_KEY_HASH parameter. This parameter enables the RSA HASH field for editing and controls the RSA HASH programming instructions in the Xilinx Programmer Settings File Generator tool.</p>		
<p>RSA HASH</p>	<p>SHA-256 hash value of the RSA primary public key (PPK).</p> <ul style="list-style-type: none"> <li>Set the XSK_EFUSEPS_ENABLE_RSA_KEY_HASH parameter to TRUE to enable editing and programming of this value.</li> <li>Define or obtain the RSA primary and secondary private and public keys. See <a href="#">Step 2: Define the Security Keys and eFUSE Settings</a>.</li> <li>Use the Xilinx Bootgen tool to generate the hash of the public key when creating the verification boot image.</li> <li>Copy the hash value from the output <code>efuse_ppk_hash.txt</code> file produced by the Bootgen <code>-efuseppkbits</code> option and paste into this field. This is the hexadecimal value. This value must be 64 characters long. Valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not program. If an RSA value is given, then the XSK_EFUSEPS_ENABLE_RSA_AUTH parameter should be set to TRUE to enable RSA authentication of a boot image.</li> </ul> <p>See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) <a href="#">[Ref 2]</a>, <i>RSA PPK Hash and Secure Boot of Zynq-7000 All Programmable Soc</i> (XAPP1175) <a href="#">[Ref 1]</a>, Generate Hash of PPK.</p>	<p>64-digit hex value (all zeros)</p>	

Table 1: Security Keys and eFUSE Settings Sheet (Cont'd)

PL eFUSE Parameters			
PL eFUSE Parameter Name	Description	Setting Option (Default)	Setting (Define Settings Here)
XSK_EFUSEPL_FORCE_PCYCLE_RECONFIG	<p>TRUE programs the PL eFUSE bit that disables partial reconfiguration of the PL. This can limit RMA analysis. An alternative to disabling partial reconfiguration by eFUSE is setting the PL bitstream security property to Level2.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2] and <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], AES_Exclusive.</p>	TRUE/FALSE (FALSE)	
XSK_EFUSEPL_BBRAM_KEY_DISABLE	<p>TRUE programs the PL eFUSE bit that disables the BBRAM key functionality.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], BBRAM Key Disable.</p>	TRUE/FALSE (FALSE)	
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	<p>TRUE programs the PL eFUSE bit that disables the Zynq-7000 device JTAG functionality. This can limit RMA analysis. This setting takes affect after the device is power-cycled or power-on-reset.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], JTAG Chain Disable.</p>	TRUE/FALSE (FALSE)	
XSK_EFUSEPL_FORCE_USE_AES_ONLY	<p>TRUE programs the PL eFUSE bit that forces secure boot with eFUSE AES key. RMA analysis cannot be performed.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], eFUSE Secure Boot and <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], CFG_AES_Only.</p>	TRUE/FALSE (FALSE)	
XSK_EFUSEPL_DISABLE_KEY_WRITE	<p>TRUE programs the PL eFUSE bit that disables future writes or changes to the FUSE_KEY and FUSE_USER eFUSE values. This setting takes affect after the device is power-cycled or power-on-reset.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], W_EN_B_Key_User.</p>	TRUE/FALSE (FALSE)	

Table 1: Security Keys and eFUSE Settings Sheet (Cont'd)

<p>XSK_EFUSEPL_DISABLE_AES_KEY_READ</p>	<p>TRUE programs the PL eFUSE bit that disables read access to the FUSE_KEY value. If programming a XSK_EFUSEPL_AES_KEY value, it is critical to set this to TRUE to secure the XSK_EFUSEPL_AES_KEY value from read access.</p> <p><b>Note:</b> This also disables future writes or changes to the FUSE_KEY and FUSE_USER values.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], R_EN_B_Key.</p>	<p>TRUE/FALSE (FALSE)</p>	
<p>XSK_EFUSEPL_DISABLE_USER_KEY_READ</p>	<p>TRUE programs the PL eFUSE bit that disables JTAG read access to the FUSE_USER value.</p> <p><b>Note:</b> This also disables future writes or changes to the FUSE_KEY and FUSE_USER values.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>Regardless of this setting, the FUSE_USER is always accessible to the PL via the EFUSE_USR primitive.</p> <p>See <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], R_EN_B_User.</p>	<p>TRUE/FALSE (FALSE)</p>	
<p>XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE</p>	<p>TRUE programs the PL eFUSE bit that disables future write or changes to the PL FUSE_CTRL eFUSE bits, that include the XSK_EFUSEPL_FORCE_PCYCLE_RECONFIG, XSK_EFUSEPL_DISABLE_KEY_WRITE, XSK_EFUSEPL_DISABLE_AES_KEY_READ, XSK_EFUSEPL_DISABLE_USER_KEY_READ, XSK_EFUSEPL_FORCE_USE_AES_ONLY, XSK_EFUSEPL_DISABLE_JTAG_CHAIN, and XSK_EFUSEPL_BBRAM_KEY_DISABLE settings. This setting takes affect after the device is power-cycled or power-on-reset.</p> <p>FALSE does not modify this eFUSE control bit.</p> <p>See <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], W_EN_B_Cntl.</p>	<p>TRUE/FALSE (FALSE)</p>	
<p>XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW</p>	<p>TRUE enables the AES KEY and USER LOW parameters for editing and programs the specified values.</p> <p>FALSE ignores the AES KEY and USER LOW values and does not modify the AES KEY and USER LOW eFUSE bits.</p> <p><b>Note:</b> The AES KEY and USER LOW values cannot be programmed at separate times.</p> <p><b>Note:</b> There is no eFUSE bit corresponding to the XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW parameter. This parameter enables the AES KEY and USER LOW parameter fields for editing, and controls the AES KEY and USER LOW programming instructions in the Xilinx Programmer Settings File Generator tool.</p>	<p>TRUE/FALSE (FALSE)</p>	

Table 1: Security Keys and eFUSE Settings Sheet (Cont'd)

<p>XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY</p>	<p>TRUE enables the USER HIGH parameter for editing and programs the specified value.                  FALSE ignores the USER HIGH value and does not modify the USER HIGH eFUSE bits.</p> <p><b>Note:</b> There is no eFUSE bit corresponding to this XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY parameter. This parameter enables the USER HIGH parameter field for editing and controls the USER HIGH programming instructions in the Xilinx Programmer Settings File Generator tool.</p>	<p>TRUE/FALSE (FALSE)</p>	
<p>AES KEY</p>	<p>AES key (FUSE_KEY) value for encrypted boot images.</p> <p>Set the XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW parameter to TRUE to enable editing and programming of this value.</p> <p>Copy from the AES KEY setting above or copy the Key 0 value from the aes.nky file and paste into this field.</p> <p>This value must be 64 characters long. Valid characters are 0-9,a-f, and A-F.</p> <p>Any other character is considered an invalid string and will not be programmed.</p> <p>If the AES key value is provided, then XSK_EFUSEPL_DISABLE_AES_KEY_READ must also be set to TRUE to read-secure the key.</p> <p><b>Note:</b> The AES KEY and USER LOW values cannot be programmed at separate times.</p> <p>References: <i>Zynq-7000 All Programmable SoC Technical Reference Manual</i> (UG585) [Ref 2], <i>AES, 7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], <i>FUSE_KEY</i>, and <i>Secure Boot of Zynq-7000 All Programmable Soc</i> (XAPP1175) [Ref 1], generate an AES key.</p>	<p>64-digit hex value for FUSE_KEY [0:255] (all zeros)</p>	

Table 1: Security Keys and eFUSE Settings Sheet (Cont'd)

<p>USER HIGH</p>	<p>Bits [31:8] of the FUSE_USER[31:0] value. Set the XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY parameter to TRUE to enable editing and programming of this value. This value must be six characters long. Valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not be programmed. See <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], FUSE_USER.</p>	<p>6-digit hex value (all zeros)</p>	
<p>USER LOW</p>	<p>Bits [7:0] of the FUSE_USER[31:0] value. Set the XSK_EFUSEPL_PROGRAM_USER_LOW_KEY parameter to TRUE to enable editing and programming of this value. This value must be six characters long. Valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not be programmed. <b>Note:</b> The AES_KEY and USER LOW cannot be programmed at separate times. See <i>7 Series FPGAs Configuration User Guide</i> (UG470) [Ref 3], FUSE_USER.</p>	<p>2-digit hex value (all zeros)</p>	

### Step 3: Create the Programmer Verification Boot Image File

The programmer verification boot image is RSA-authenticated (optionally) or AES-encrypted (optionally) to test the corresponding eFUSE settings. For a stand-alone verify operation, the device programmer attempts to boot the device from this verification boot image. If the Zynq-7000 AP SoC successfully authenticates and decrypts the boot image, then the verification boot image application drives a success status from the multiplexed I/O (MIO) to the programmer. Otherwise, upon a boot failure, the device enters a secure lockdown state and the programmer does not receive the successful boot status value from the device.

This section provides sample instructions for building a secure verification boot image for the device programmer from the verification application in the reference design.

**Note:** If you have already developed an AES encryption and RSA signing flow with your standard boot image, this same flow can be used with the sample `verify.elf` file from the reference design.



**CAUTION!** Complete customer application (`.elf/.bit`) designs cannot be used as verification boot images on a typical device programmer due to the constraints described in [Requirements for Customized Variations of the Verification Boot Image](#).

## ***Prerequisites for Creating the Programmer Verification Boot Image File***

The prerequisites for creating the programmer verification boot image file are:

- Unzip the reference design zip archive to the root of the C:\ drive.
- Install the Xilinx Software Development Kit (SDK).

An SDK Standalone WebInstall Client can be downloaded from the Embedded Development tab at <http://www.xilinx.com/support/download/index.html>

- Get these settings or files from the [Security Keys and eFUSE Settings Sheet](#):
  - If using RSA authentication:
    - RSA primary secret key file: `psk.pem`
    - RSA primary public key file: `ppk.pub`
    - RSA secondary secret key file: `ssk.pem`
    - RSA secondary public key file: `spk.pub`
  - If using AES encryption:
    - AES key value
    - AES StartCBC value
    - HMAC key value

## ***Creating the Programmer Verification Boot Image File***

A Bootgen utility from the Xilinx SDK generates the boot image file. The Bootgen utility obtains the parameters for creating the verification boot image file from the `verify.bif` file. The `verify.bif` file is provided in the reference design and its contents are:

```
the_ROM_image:
{
    [aeskeyfile]aes.nky
    [ppkfile]ppk.pub
    [pskfile]psk.pem
    [spkfile]spk.pub
    [sskfile]ssk.pem
    [bootloader, encryption=aes, authentication=rsa]..\Debug\verify.elf
}
```

Where:

`aes.nky` = AES/HMAC keys input file name in a Xilinx file format

`ppk.pub` = RSA primary public key input file name

`psk.pem` = RSA primary private key input file name

`spk.pub` = RSA secondary public key input file name



Key StartCBC <32-digit hex value> = 128-bit AES block cipher initialization vector

Key HMAC <64-digit hex value> = 256-bit HMAC key value

**Note:** If not using AES encryption, edit the `verify.bif` file to remove the line that refers to the `aes.nky` file and remove "encryption=aes" from the bootloader line.

6. Execute Bootgen with these options to create an RSA authenticated (optionally) and AES encrypted (optionally) verification boot image:

```
bootgen -image verify.bif -efuseppkbits efuse_ppk_hash.txt -encrypt efuse -w on -o  
verify.mcs
```

Where the outputs are:

`verify.mcs` = verification boot image for the device programmer

`efuse_ppk_hash.txt` = contains the RSA hash 64-digit eFUSE setting

**Note:** If applicable, remove these options from the Bootgen command line:

- If no RSA authentication: remove `-efuseppkbits efuse_ppk_hash.txt`
- If no AES encryption: remove `-encrypt efuse`

7. If applicable, save these files for the specified uses:

`verify.mcs` = save for sending to the device programmer

`aes.nky` = save for [Step 4: Create the Programmer Settings File](#)

`efuse_ppk_hash.txt` = save for [Step 4: Create the Programmer Settings File](#)

## Step 4: Create the Programmer Settings File

The programmer settings file contains the eFUSE settings to be programmed by the device programmer. A Tcl-based Xilinx programmer settings file generator tool is provided in the reference design for creating the programmer settings file.

This section provides instructions for creating a programmer settings file using the Xilinx programmer settings file generator tool. Be prepared to apply the settings defined in the Zynq-7000 AP SoC eFUSE settings sheet. If using RSA authentication, the RSA hash value is in the `efuse_ppk_hash.txt` file from [Creating the Programmer Verification Boot Image File](#).

## Prerequisite for the Xilinx Programmer Settings File Generator Tool

The prerequisites for creating the programmer verification boot image file are:

1. Unzip the reference design zip archive to the root of the C:\ drive.
2. Install a Tcl/Tk 8.5 script interpreter. A Tcl/Tk 8.5 script interpreter can be downloaded from <http://www.activestate.com/activetcl/downloads>.



---

**IMPORTANT:** Download the 8.5.\*.\* version. The Xilinx programmer settings file generator tool does not work with Tcl 8.6 or later.

---

3. After installing the Tcl 8.5 script interpreter, start tclsh85 and install the Iwidgets package by entering this command on the tclsh85 command line:

```
tclsh85 install Iwidgets
```

4. Get these files from [Step 3: Create the Programmer Verification Boot Image File](#):
  - If using RSA authentication, get the `efuse_ppk_hash.txt` file for the RSA hash value.
  - If using AES encryption, get the `aes.nky` file for the AES key value.

## Using the Xilinx Programmer Settings File Generator Tool

To create the programmer settings file:

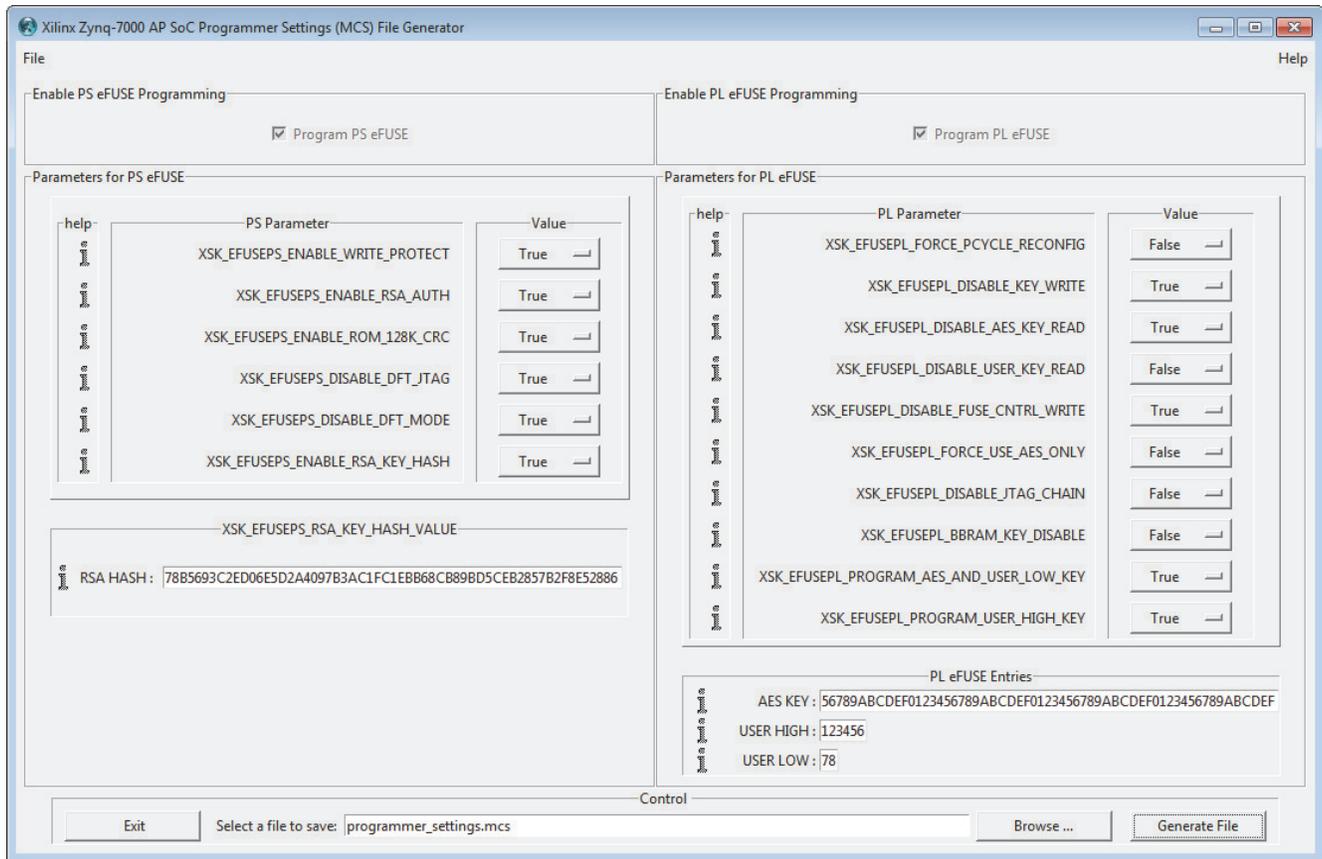
1. Open a Windows command prompt.
2. Change to the `C:\xapp1278\Zynq7000_programmer_settings\` directory of the unzipped reference design.
3. Enter this command to start the Xilinx programmer settings file generator tool:

```
tclsh85 zynq7000_programmer_settings_mcs_file_generator.tcl
```

See [Figure 2](#) for a view of the programmer settings file generator tool interface.

4. Use your definitions from the Zynq-7000 AP SoC eFUSE settings sheet to set the parameters in the programmer settings file generator tool. Alternative sources for exact values are:
  - If using RSA authentication, copy the RSA hash value from the `efuse_ppk_hash.txt` file to the RSA HASH field in the programmer settings file generator tool.
  - If using AES encryption, copy the AES key 0 value from the `aes.nky` file to the AES KEY field in the programmer settings file generator tool.
5. Click **Generate File** to generate the `programmer_settings.mcs` file.

**Note:** Treat the `programmer_settings.mcs` file as a binary file. Do not manually edit the programmer settings file. The device programmer expects a specific format of the file data.



X15923-021116

Figure 2: Sample View of the Programmer Settings File Generator Tool

Tips for using the programmer settings file generator tool interface include:

- Hover the mouse over the 'i' symbol to see the help description for each parameter.
- Click the Value buttons to change settings from False to True, or vice versa.
- Enter RSA HASH, AES KEY, USER HIGH, or USER LOW hex values into the entry boxes for each.

**Note:** These fields have a corresponding parameter that must be set to TRUE to enable editing of the field. For example, the XSK\_EFUSE\_PS\_ENABLE\_RSA\_KEY\_HASH parameter must be set to TRUE to enable editing of the RSA HASH value entry box. If the entry box background is yellow when editing a value, then the value does not yet have enough hex digits or has a non-hex character.



**RECOMMENDED:** Copy and paste the RSA HASH value directly from the `efuse_ppk_hash.txt` file to the entry box to avoid typographical errors in the value.



**RECOMMENDED:** Copy and paste the AES KEY value directly from the Key 0 field of the `aes.nky` file to the entry box to avoid typographical errors in the value.

## Step 5: Locate Xilinx Prebuilt Programming Algorithm Image File

The Xilinx prebuilt programming algorithm image file is a boot image that contains PS ARM processor code for programming the eFUSE. The programming algorithm image file is included in the unzipped reference design at this location:

```
C:\xapp1278\Zynq7000_programming_algorithm_image_file\32MHz_QSPI0_program_01_08_16_slowcpu_Pl_ps.mcs
```

## Step 6: Send Device Samples and Programming Files

When the three files listed in [Table 2](#) are located and created, they should be securely sent to the device programming site with the sample devices to be programmed.

**Table 2: Files for the Device Programmer**

File	Source and Reference File Name (and type)	Description
File [A]	User-created programmer settings file programmer_settings.mcs (binary MCS)	This file contains the user-defined settings to be programmed into the PS eFUSE and PL eFUSE, including the PS RSA PPK hash and AES key. A Xilinx Programmer Settings File Generator tool is provided in the reference design for creating this file. This file must be loaded as a binary file into the programmer buffer. The programmer sends this file through the Zynq-7000 AP SoC UART1 interface to the programming algorithm code in the OCM.
File [B]	Xilinx prebuilt programming algorithm boot image file 32MHz_QSPI0_program_01_08_16_slowcpu_Pl_ps.mcs (Intel hex MCS)	This boot image contains the programming algorithm code for the PS eFUSE and PL eFUSE. Xilinx provides this prebuilt boot image in the reference design. The data content of this Intel hex format MCS file are loaded into the programmer buffer. This code is loaded into OCM and executed within the Zynq-7000 AP SoC PS.
File [C]	User-created secure verification boot image file verify.mcs (Intel hex MCS)	The user creates an (optional) RSA-authenticated and (optional) AES-encrypted version of a Xilinx-provided reference design. The reference design drives the Validation_image_passed (0xA) value to the designated Zynq-7000 AP SoC GPIO status[3:0] pins after a successful boot load. The data content of this Intel hex format MCS file are loaded into the programmer buffer. This secure boot image is used for stand-alone verification that the Zynq-7000 AP SoC device eFUSE has been programmed with a matching RSA PPK hash or AES key.

This concludes the device programmer file preparation process.

## Device Programmer Setup

See the device programmer vendor's guidelines for setup on a specific device programmer. The files listed in [Table 2](#) are required for the device programmer setup. For reference, this section includes a sample setup for a BPM Microsystems 2800 programmer application and a XC7Z010-CLG400 part. The BPM Microsystems programmer requires all file data to be loaded into a linear buffer memory space. For the XC7Z010-CLG400 part, the buffer memory map is shown in [Table 3](#).

**Table 3: BPM Microsystems Buffer Memory Map for an XC7Z010-CLG400 Device**

Start Address (hex)	End Address (hex)	Description
00000	3FFFF	32 MHz_QSPI0_program_01_08_16_slowcpu_P1_ps.mcs: Xilinx prebuilt programming algorithm file for QSPI0 on the BPM Microsystems programmer socket module.
40000	7FFFF	verify.mcs: The user verification boot image file for QSPI1 on the BPM Microsystems programmer socket module.
80000	827FF	programmer_settings.mcs: The user eFUSE settings file to be programmed into the Zynq-7000 AP SoC device.

To load the files into the buffer memory map in the BPM Microsystems programmer application:

1. Select the Xilinx XC7Z010-CLG400 part.
2. Load the `programmer_settings.mcs` file [A] into the programmer buffer memory starting at address 00000h. Clear the buffer when loading the file and load the file as a binary file.
3. Edit the buffer memory and copy the address range 00000h-027FFh to address 80000h.
4. Load the `verify.mcs` file [C] into the programmer buffer memory starting at address 00000h. Do NOT clear the buffer when loading this file and load the file as an Intel hex file.
5. Edit the buffer memory and copy the address range 00000h-3FFFFh to address 40000h.
6. Load the `32MHz_QSPI0_program_01_08_16_slowcpu_P1_ps.mcs` file [B] into the programmer buffer memory starting at address 00000h. Do NOT clear the buffer when loading this file and load the file as an Intel hex file.
7. Save the buffer memory to a file for future device programming.

After the programmer setup is finished, devices can be programmed and stand-alone verified on the device programmer.

For the solution described in this application note, the programmer has two Quad SPI flash devices on the socket adapter. The programmer programs one Quad SPI flash (QSPI\_0) device with the Xilinx programming algorithm image file and programs the second Quad SPI flash (QSPI\_1) device with the verification boot image.

For a program operation, the programmer:

- Boots a target device from the QSPI\_0 flash to load the programming algorithm.
- Sends the programmer settings file to the device to program each device's eFUSE.
- Boots the device from QSPI\_1 as a sign-off verification of the programmed eFUSE settings.

For a verify operation, the programmer:

- Boots a target device from QSPI\_1 to verify the target device contains the expected eFUSE settings that enable the verification boot image to successfully load.



---

**IMPORTANT:** *The BPM Microsystems programmer only programs the Quad SPI flash on its socket adapter during a program operation. Consequently, a program operation must be performed before new stand-alone verify operations. Otherwise, a stand-alone verify operation might attempt to boot the target device from a Quad SPI flash image that belongs to a different programming job.*

---

---

## Checks for Properly Programmed Devices

This section provides recommendations for checking that the programming files, programmer setup, and programmed devices have been done correctly.

### Checks for Device Programmer Files and Setup

This section includes recommendations for checking device programmer files and setup.

Check for a valid verification boot image after completing the device programmer setup by checking for expected passing and expected failing cases:

1. Program one sample device (sample #1).
2. Stand-alone verify sample #1 – this is expected to pass.
3. Stand-alone verify a second sample device (sample #2) that has not been previously programmed – this is expected to fail.

If the results from either step 2 or step 3 are different from expectations, consult with the provider of the files for potential issues.

## Checks for Properly Programmed Device Samples

This section includes recommendations for checking that a sample device has been programmed with the correct settings.

After a sample device has been programmed, assemble the device onto a board and check:

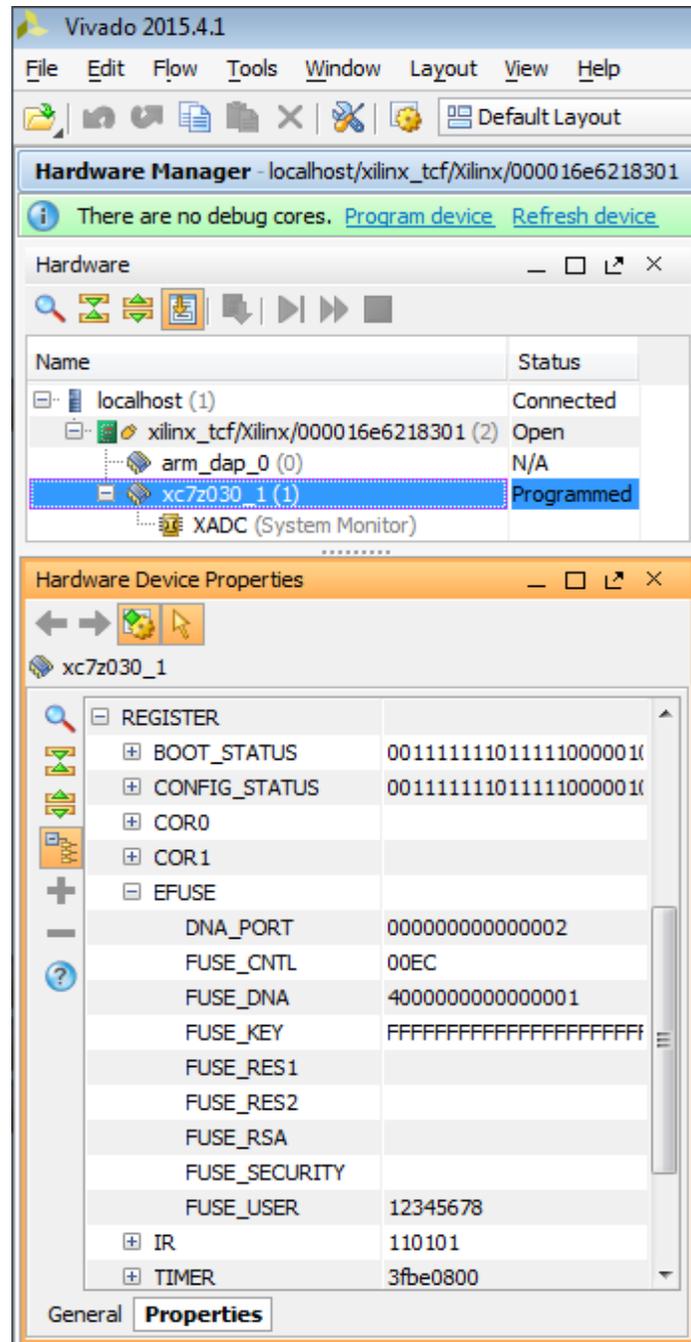
1. Use the Vivado Hardware Manager with a JTAG cable connected to the board with the programmed device to check the PL eFUSE settings. For a Zynq-7000 AP SoC device, view the REGISTER → EFUSE settings in the Hardware Device Properties window. See [Figure 3](#) for a sample view.

- a. If an AES key is programmed, check that the FUSE\_CNTL[3] bit is programmed to a '1'. FUSE\_CNTL[3] is the read-security for the AES key.

**Note:** The FUSE\_KEY value is either not shown or is shown as all 'F' hex values when the FUSE\_KEY is read-secured. In the Vivado Hardware Device Properties window, the FUSE\_CNTL value is a hex value shown with a bit order of [13:0]. See [Table 4](#) for a list of the FUSE\_CNTL bit locations that correspond to PL eFUSE settings.

- b. If a 32-bit user-defined value is programmed, check the FUSE\_USER[31:0] value in the Vivado Hardware Device Properties window.

**Note:** Ignore the FUSE\_RSA property in the Vivado Hardware Device Properties window. The FUSE\_RSA property is NOT the Zynq-7000 AP SoC PS eFUSE RSA value.



X15925-021116

Figure 3: Vivado Hardware Manager FUSE\_CNTL Value

Table 4: PL eFUSE FUSE\_CNTL[13:0] Bit Assignments

FUSE_CNTL (bit index)	PL eFUSE Parameter Name
1	XSK_EFUSEPL_FORCE_PCYCLE_RECONFIG
2	XSK_EFUSEPL_DISABLE_KEY_WRITE
3	XSK_EFUSEPL_DISABLE_AES_KEY_READ
4	XSK_EFUSEPL_DISABLE_USER_KEY_READ
5	XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE
8	XSK_EFUSEPL_FORCE_USE_AES_ONLY
9	XSK_EFUSEPL_DISABLE_JTAG_CHAIN
10	XSK_EFUSEPL_BBRAM_KEY_DISABLE

**Notes:**

Bit locations not listed in the table are reserved for Xilinx use and can be any value.

2. Use the Xilinx microprocessor debugger (XMD) tool from the Xilinx SDK 2015.4 (or earlier) to check some of the PS eFUSE settings. To use the XMD tool:
  - a. Open a Windows command prompt.
  - b. Change to the C:\xapp1278\Zynq7000\_ps\_efuse\_check\_method\ directory in the unzipped reference design.
  - c. To run the XMD tool, enter:

```
xmd
```

If XMD does not start, set the path for Windows to find XMD by entering:

```
call C:\Xilinx\SDK\2015.4\settings64.bat
```

where C:\Xilinx\SDK\2015.4\ is the install location of the Xilinx SDK.

**Note:** If the SDK version being used is not 2015.4, check the C:\Xilinx\SDK\ directory for the appropriate value in the path and adjust the path accordingly.

- d. The XMD tool executes the xmd.ini script in the local directory. A sample of the output from the xmd.ini script is:

```
=====
PS eFUSE Bit Function Name      =Value- Programmed Status
-----
eFUSE Write Protection (2-bits) = 00 - NOT programmed
OCM ROM 128KB CRC Enable       = 0 - NOT programmed
RSA Authentication Enable      = 1 - **PROGRAMMED** <---
DFT JTAG Disable               = 0 - NOT programmed
DFT Mode Disable               = 0 - NOT programmed
=====
```

3. Boot the actual application image to check for a successful boot.

If the eFUSE settings are different than expected or if the application image fails to boot, check the programmer settings.

## Troubleshooting

The programming algorithm can report the status values shown in [Table 5](#) during the programming operation.

*Table 5: Programming Algorithm STATUS[3:0] Values*

STATUS[3:0] Value (binary)	Programming Algorithm Status Name	Description
0000	PS_eFUSE_driver_booted_error	<p>This is the default value for the MIO pins when an image fails to boot. In this case, the MIO pins remain 3-stated, and pull-down termination on the programmer socket keeps the STATUS[3:0] pins Low.</p> <p>Check that the socket is properly installed on the programmer.</p> <p>Check that the device is properly seated in the socket.</p> <p>Check that the programmer has the latest Xilinx programming algorithm boot image file [B].</p> <p>Check that the Xilinx programming algorithm boot image file is properly loaded in the programmer.</p>
0001	PS_eFUSE_driver_booted	<p>This value is reported upon successful boot of the Xilinx programming algorithm boot image file.</p> <p>If the programming operation ends with this status, check that the <code>programmer_settings.mcs</code> file:</p> <ul style="list-style-type: none"> <li>• was properly generated</li> <li>• includes "efuse program" text at the end of the file</li> <li>• was properly loaded into the programmer.</li> </ul>
0010	PS_eFUSE_driver_programming_started	<p>This value is reported when the PS eFUSE programming starts (after the PL eFUSE programming phase).</p>
0011	PS_eFUSE_driver_programming_completed	<p>This value is reported when the PS eFUSE programming phase has completed.</p>
0100	PS_eFUSE_driver_validation_passed	<p>This value is reported at the end of the program operation after the PL and PS eFUSE have been successfully programmed and verified.</p>
0101	PS_eFUSE_driver_validation_failed	<p>This value is reported at the end of the program operation when the programming algorithm has detected an error during programming of the PS eFUSE.</p>
0110	PL_eFUSE_driver_programming_started	<p>This value is reported after the programming algorithm successfully boots, and the algorithm successfully receives the <code>programmer_settings.mcs</code> file data with an "efuse program" command. This value indicates the start of the PL eFUSE programming phase.</p>

Table 5: Programming Algorithm STATUS[3:0] Values (Cont'd)

STATUS[3:0] Value (binary)	Programming Algorithm Status Name	Description
0111	PL_eFUSE_driver_programming_completed	This value is reported when the PL eFUSE programming phase ends.
1000	PL_eFUSE_driver_validation_passed	This value is reported if the PL eFUSE programming phase completed successfully.
1001	PL_eFUSE_driver_validation_failed	This value is reported if the PL eFUSE programming phase encountered an error.
1100	INVALID_CMD	This value is reported if the programming algorithm receives the <code>programmer_settings.mcs</code> file data with an invalid command. Check that the <code>programmer_settings.mcs</code> file is generated correctly and is not modified from its original generated form.
1101	INVALID_CHECKSUM	This value is reported if the programming algorithm receives the <code>programmer_settings.mcs</code> file data, but the checksum for a line in the data is incorrect. Check that the <code>programmer_settings.mcs</code> file is generated correctly and is not modified from its original generated form.

---

## Details for Advanced Device Programmer Files

This section provides details for advanced device programmer applications.

### Requirements for Customized Variations of the Verification Boot Image

The verification boot image can be customized as needed, but the custom boot image must be compatible with the constraints of the device programmer socket hardware.

Typically, the device programmer hardware is built with the bare minimum of the necessities to support eFUSE programming. Consequently, custom verification boot applications must be capable of running with only a minimum of system support functions. The subsequent sections provide guidelines for custom verification boot images.

## Verification Boot STATUS[3:0]

For a passing stand-alone verify operation, the device programmer expects to receive a specific 4-bit STATUS[3:0] value from specific device pins. See [Table 6](#) for the valid STATUS[3:0] values.

**Table 6: Verification Boot STATUS[3:0] Values**

STATUS[3:0] Value (binary)	Verification Result Name (in the Reference Application Code)	Description
1010	Validation_image_passed	The reference verification boot application code outputs this value to indicate a successful boot. All other values result in a failed stand-alone verify operation.
1011	Validation_image_failed	Only use this in custom boot application code where a failure is reported for additional tests in the code.
0000	PS_eFUSE_driver_booted_error	This is the default value for the MIO pins when an image fails to boot. In this case, the MIO pins remain 3-stated and pull-down termination on the programmer socket keeps the STATUS[3:0] pins Low.

The boot application must output the 4-bit STATUS[3:0] value via the PS general purpose I/O (GPIO) to the MIO listed in [Table 7](#).

**Table 7: Verification Boot STATUS[3:0] MIO Assignments**

STATUS Bit	MIO Assignment	I/O Type
STATUS[3]	MIO37	GPIO Output, LVCMOS18
STATUS[2]	MIO36	GPIO Output, LVCMOS18
STATUS[1]	MIO35	GPIO Output, LVCMOS18
STATUS[0]	MIO34	GPIO Output, LVCMOS18

See the Zynq-7000 AP SoC package file for the pin location of each MIO in the selected device and package.

## System Attributes and Constraints

The boot image must be compatible with these hardware and PS attributes:

- PS\_CLK frequency = 32 MHz
- APU clock frequency = 200 MHz (due to power constraints)
- I/O voltages = 1.8V
- No DDR memory – the boot application memory footprint must fit with the PS OCM

**Note:** DDR, transceiver, and SelectIO banks are not used on the device programmer and might not be powered.

### **Boot Flash Constraints**

The Quad SPI boot flash size is 128 Mb. The verification boot image must fit within this flash.

### **Power Constraints**

The device programmer is expected to supply only enough current to meet the minimum power-on current or maximum static current draw, whichever is greater, for each power supply. The device programmer is expected to supply only enough current to run the PS application processor unit (APU) at the minimum clock frequency of 200 MHz.

### **Run-Time Constraints**

The boot image must boot and report a result within two seconds of power-on. Otherwise, the device programmer can time out and report a failed stand-alone verify.

## Reference Design

You can download the [Reference Design Files](#) for this application note from the Xilinx website.

[Table 8](#) shows the reference design matrix.

**Table 8: Reference Design Matrix**

Parameter	Description
<b>General</b>	
Developer Name	Habib El-Khoury, Randal Kuramoto
Target Devices	Zynq-7000 AP SoC family
Source code provided	Yes
Source code format	C
Design uses code and IP from existing Xilinx application note and reference designs or third party	No
<b>Implementation</b>	
Synthesis software tools/versions used	N/A
Implementation software tools/version used	Xilinx SDK 2015.4
Static timing analysis performed	N/A
<b>Hardware Verification</b>	
Hardware verified	Yes
Hardware platform used for verification	BPM Microsystems 2800 programmer with a socket for an XC7Z010-CLG400 device

## References

This application note uses the following references:

1. *Secure Boot of Zynq-7000 All Programmable SoC* ([XAPP1175](#)).
2. *Zynq-7000 All Programmable Technical Reference Manual* ([UG585](#)).
3. *7 Series FPGAs Configuration User Guide* ([UG470](#)).
4. *Vivado Design Suite Programming and Debugging User Guide* ([UG908](#)).

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/23/2016	1.0	Initial Xilinx release.

---

## Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>.

### Automotive Applications Disclaimer

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.

© Copyright 2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners. ARM is a registered trademark of ARM in the EU and other countries.