



XAPP1260 (v1.0) July 8, 2015

# eFUSE Programming on a Device Programmer

Author: Randal Kuramoto

## Summary

This application note provides guidelines for programming the 7 series FPGA eFUSE bits on a device programmer. Device programmers are effective for:

- High-volume production programming.
- Preprogramming devices with sensitive data into the eFUSE before they are sent for assembly at a board manufacturing site.
- Cataloging the DNA (unique serial numbers) of programmed devices.



---

**IMPORTANT:** *The 7 series FPGA eFUSE bits are one-time programmable and cannot be changed once they are programmed.*

---

Specifying the exact settings and a verification method is important to ensure the correct programming of the eFUSE bits. This application note provides guidelines for the eFUSE bit settings. A reference design is provided for creating an encrypted verification bitstream for a device programmer.

The [reference design files](#) for this application note can be downloaded from the Xilinx website. For detailed information about the design files, see [Reference Design](#).

# Introduction to Programmable eFUSE

Figure 1 shows the inputs to a device programmer for programming and verifying the eFUSE bits, as well as the uses of the eFUSE bits in the 7 series FPGA.

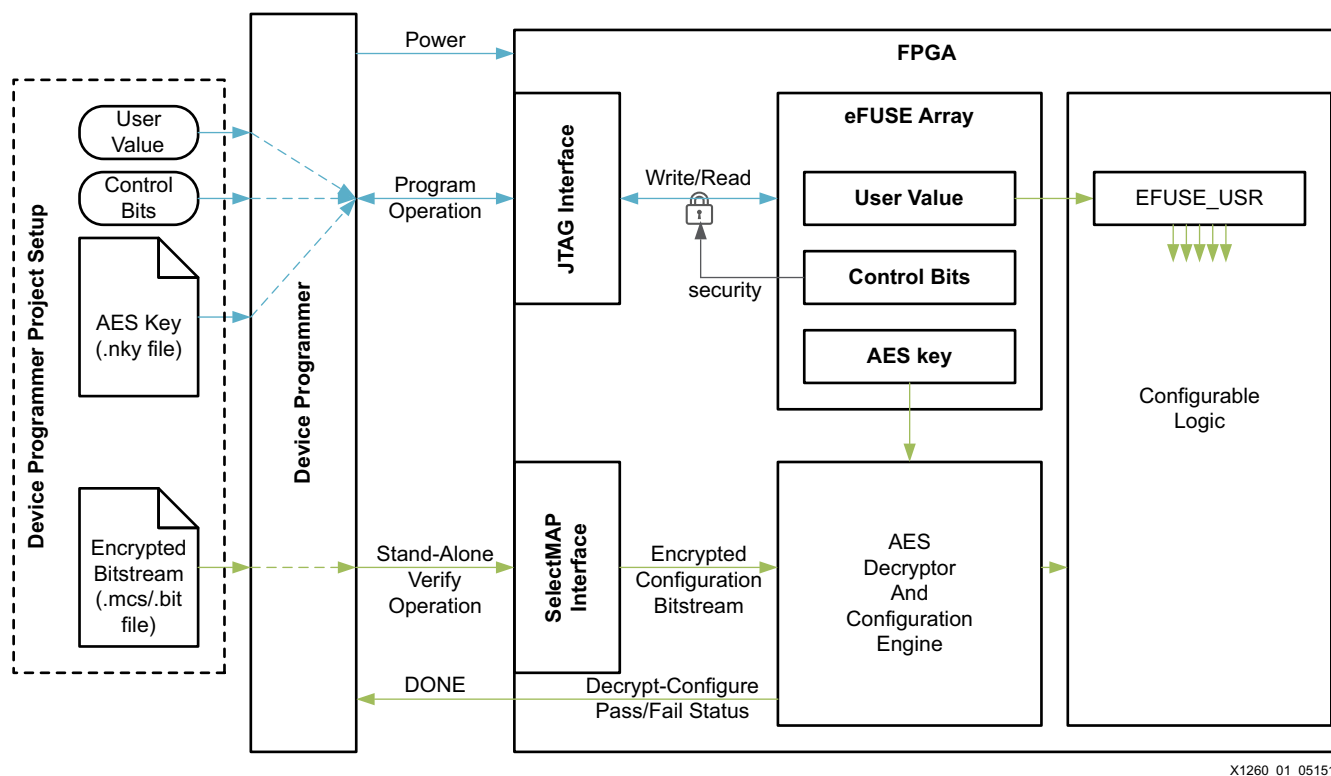


Figure 1: Block Diagram of eFUSE Array and Associated Functions

For the 7 series FPGAs, the bitstream can be encrypted to prevent unauthorized use of a copied design. The 7 series FPGAs use the advanced encryption standard (AES) that is issued in the Federal Information Processing Standards Publication 197 (FIPS PUB 197) by the National Institute of Standards and Technology (NIST). The AES is a symmetric key algorithm, which means that the same secret key is used for encrypting and decrypting a bitstream. The Vivado® design tools use the key to encrypt the bitstream when writing the bitstream file. The FPGA uses the key to decrypt a bitstream during configuration. For the FPGA AES decryptor, the AES key can reside in battery-backed RAM or in nonvolatile, one-time programmable eFUSE bits within the FPGA. Only the eFUSE bits can be programmed on a device programmer. The battery-backed RAM cannot be effectively programmed on a device programmer because the RAM loses the settings when the device is removed from the programmer.

The advantages of using the nonvolatile eFUSE bits to store the key include:

- No battery is required to preserve the key value when power is removed from the FPGA.
- Lower bill of materials cost (than with a battery).
- The eFUSE can be preprogrammed into the device at a secure location on a device programmer before assembly at a board manufacturing site.

The nonvolatile 7 series FPGA eFUSE bits can also store a user defined 32-bit "user value" that the FPGA configurable logic design can read from the EFUSE\_USR primitive. Also, external devices can read this value through the FPGA JTAG port.

Additional eFUSE bits are used to control FPGA security settings, such as read-security or write-security of the eFUSE against further reading or writing.



---

**IMPORTANT:** *The secret AES key must always be read-secured during eFUSE programming.*

---

All user-programmable eFUSE bits are a "0" value when shipped from Xilinx. Selected eFUSE bits are programmed to "1" through the FPGA JTAG port. The eFUSE programming procedure includes verifying the programmed eFUSE bits and setting the selected security options. After completion of the eFUSE programming procedure, the eFUSE bits are read-secured and cannot be directly inspected.



---

**RECOMMENDED:** *For device programming, Xilinx recommends using a sample bitstream designed for the device programmer and encrypted with a matching key to validate the correct eFUSE key programming on a device programmer.*

---

This sample bitstream can also be used to identify a device that has been programmed with a matching key in situations when the device programmer is used in stand-alone verification mode to recheck an inventory of programmed devices. The sample bitstream is sent to the 7 series FPGA SelectMAP configuration interface during the stand-alone verify operation.

See the *7 Series FPGAs Configuration User Guide* (UG470) [Ref 1] for more information on the 7 series FPGAs eFUSE features and device configuration.

---

## Using the Device Programmer for eFUSE Programming

The device programmer can be used to:

- Program the eFUSE bits in the FPGA and record the FPGA DNA value.
- Stand-alone verify that an FPGA has been previously programmed with a specific AES key.

### Programming the eFUSE

This section provides the steps for programming the eFUSE in a 7 series FPGA on a device programmer.

1. The programming manager creates a device programmer project that requires:
  - a. Specification of input files, including:
    - NKY file, which contains the AES key
    - Encrypted verification bitstream file
  - b. Definition of settings for programmable options, including:

- User-defined 32-bit value
  - eFUSE control bits for read-protection, write-protection, and other security features
2. The device programmer operator programs devices on the device programmer, and performs these operations:
    - a. Identifies the device and records the device DNA.
    - b. Programs and verifies the eFUSE bits in the device.
    - c. Secondary verification by downloading the encrypted verification bitstream to the device.

After eFUSE programming, the eFUSE within the device is typically secured, which means the AES key is secured and cannot be read from the device.

At a later time, the device programmer can perform a stand-alone operation to identify and verify that a previously programmed 7 series FPGA contains the correct AES key in its eFUSE, even though the eFUSE is secured in the device. For this stand-alone verify operation, the device programmer downloads an encrypted verification bitstream to the device. If the device successfully configures from the encrypted verification bitstream, then the device is verified to contain the matching key.

---

## Preparing for a Device Programmer

The required preparations for programming on a device programmer are:

- Establish the device programmer site or service.
- Locate the NKY file that contains your AES key.
- Decide on the security setting for each eFUSE control bit option.
- Build the encrypted verification bitstream from the provided reference design using your NKY file.
- Deliver device samples and device programmer settings and files to the device programming site.

### Establish the Device Programmer Site or Service

Contact the local authorized Avnet distributor for device programming services or contact an FAE for in-house programmer options.



---

**IMPORTANT:** *It is critical to establish the device programmer site or service as early as possible. It can take many weeks to develop a device programmer socket for a device-package that has not been previously established.*

---

## Define the Device Programmer Settings and Input Files

A complete and clear list of settings with required files must be provided for the device programmer. Use [Table 1](#) as a checklist and definition sheet for the device programmer settings and input files. Supply the completed [Table 1](#) with the required files to the device programming site with device samples for the first few pilot devices.

Table 1: Device Programmer Settings


Description of Function When Programmed	Setting Type	Device Programmer Input File or Setting (Define your settings and file names here)
<p><b>256-bit AES key</b> This key can decrypt an encrypted bitstream for FPGA configuration.</p>	<p>NKY file name The NKY file contains the 256-bit AES key value.</p>	
<p><b>32-bit user-defined value</b> Optional user-defined device identifier value accessible from the EFUSE_USR primitive in the FPGA design or from JTAG (if R_EN_B_User is not programmed). If not used, then leave blank.</p>	<p>8-character hexadecimal value [31:0]</p>	
<p><b>CFG_AES_Only</b> (FUSE_CNTL[0]) Forces the use of the AES key stored in eFUSE, and prevents bitstreams without a matching key from configuring the FPGA. (If this bit is not programmed, the FPGA might configure using an unencrypted bitstream, or a bitstream encrypted with a key value stored in battery-backed RAM.)</p> <hr/> <p> <b>CAUTION!</b> Return material authorization (RMA) returns cannot be accepted if this bit is programmed because it prevents Xilinx test bitstreams from being loaded into the device for failure analysis.</p>	<p>Yes=program to '1', No=do not program ('0')</p> <p>Typical setting=No</p>	

Table 1: Device Programmer Settings (Cont'd)




Description of Function When Programmed	Setting Type	Device Programmer Input File or Setting (Define your settings and file names here)
<p><b>AES_Exclusive</b> (FUSE_CNTL[1])</p> <p>Disables partial reconfiguration from external configuration interfaces. However, partial reconfiguration is allowed via the ICAPE2.</p>  <hr/> <p><b>CAUTION!</b> <i>If the AES_Exclusive bit is programmed, return material authorization (RMA) returns are limited in device analysis and debug. The recommended alternative, that achieves the same security function, is to set the BITSTREAM.READBACK.SECURITY property to Level2 in the Vivado design tools before generating the bitstream.</i></p>	<p>Yes=program to '1', No=do not program ('0')</p> <p>Typical setting=No</p>	
<p><b>W_EN_B_Key_User</b> (FUSE_CNTL[2])</p> <p>Disables the programming of the AES key and FUSE_USER.</p>	<p>Yes=program to '1', No=do not program ('0')</p> <p>Typical setting=Yes</p>	
<p><b>R_EN_B_Key</b> (FUSE_CNTL[3])</p> <p>Disables reading of AES key as well as programming of AES key and 32-bit user-defined value.</p>  <hr/> <p><b>IMPORTANT:</b> <i>The R_EN_B_Key bit must be programmed to prevent the secret AES key from being read through the JTAG port.</i></p>	<p>Yes=program to '1', No=do not program ('0')</p> <p>REQUIRED setting=Yes, if the AES key is programmed</p>	<p>Yes</p> <p>Do not change unless you are not programming an AES key.</p>
<p><b>R_EN_B_User</b> (FUSE_CNTL[4])</p> <p>Disables reading of 32-bit user-defined value through JTAG as well as programming of AES key and user-defined value. This does not disable reading the user-defined value through the EFUSE_USR component, although it disables reading the user-defined value through the JTAG port.</p>	<p>Yes=program to '1', No=do not program ('0')</p> <p>Typical setting=No</p>	

Table 1: Device Programmer Settings (Cont'd)

Description of Function When Programmed	Setting Type	Device Programmer Input File or Setting (Define your settings and file names here)
<b>W_EN_B_Cntl</b> (FUSE_CNTL[5]) Disables programming of control bits.	Yes=program to '1', No=do not program ('0')  Typical setting=Yes	
<b>Encrypted verification bitstream</b> The bitstream is encrypted using the AES key from the NKY file and is supplied in MCS format.   <b>CAUTION!</b> Do NOT use your actual design. See <a href="#">Encrypted Verification Bitstream</a> for bitstream design constraints.	MCS file name	

## AES Key and NKY File

Typically, the 256-bit AES key is defined and provided to the Vivado design tools for encrypting the bitstream. The `write_bitstream` command is run to generate the encrypted bitstream. The Vivado design tools save the AES key value in an NKY file. Keep the NKY file as an input to the device programmer for programming the AES key value into the FPGA eFUSE, and for using the same key to encrypt the bitstream in successive runs of the `write_bitstream` command, including a run of `write_bitstream` to encrypt the verification bitstream.

See "Generating an Encrypted Bitstream" in the *Vivado Design Suite Programming and Debugging User Guide* (UG908) [Ref 2] and *Using Encryption to Secure a 7 Series FPGA Bitstream* (XAPP1239) [Ref 3] for instructions. Ensure that the selected location of the encryption key is "eFUSE" in the Vivado design tools encrypted bitstream flow.

Example NKY file:

```
Device xc7a200tfbg676;
Key 0 0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef;
Key StartCBC 9966d617cc42731291374e58239a823e;
Key HMAC cd552f296733ac7abec573bf72e1b2d91d8c5ee603857a62f743ab8229fcbb25;
```

The device programmer uses only the 64-character hexadecimal value that follows the "Key 0" and the bit order of the key 0 value is KEY[0:255]. The StartCBC and HMAC values are used during bitstream generation, and are not programmed into the eFUSE.

## User-Defined 32-bit Value

A value can optionally be specified as input to the device programmer for programming the 32-bit user-defined value field in the eFUSE.

The 32-bit value can be accessed from:

- JTAG FUSE\_USER instruction (if the R\_EN\_B\_User [read-security] eFUSE bit is not programmed)
- FPGA design EFUSE\_USR primitive

The value is typically specified to the device programmer as an 8-character hexadecimal value[31:0].

**Note:** Due to the physical assignment of the user-defined value bits in the eFUSE array, some programmer solutions, such as the Vivado hardware manager, divide the user value into the user value[31:8] and user value[7:0] segments. A programmer operator should be able to make this division from the given complete value[31:0].

## eFUSE Control (FUSE\_CNTL) Bits

The eFUSE control bits are located in the FUSE\_CNTL eFUSE register, and each bit permanently enables a specific security setting. In general, the selected eFUSE control settings are dependent on the requirements for your application. See [Table 1](#) for descriptions of each FUSE\_CNTL bit and sample FUSE\_CNTL settings for a typical application. The typical application is one that secures the bitstream contents and its use only to FPGAs that have been programmed with the matching AES key.

Additional security for application requirements beyond the typical application can be achieved through selection of additional eFUSE control bits to program. See the caution statements in [Table 1](#) on how programming the eFUSE control bits beyond the typical settings can affect RMA analysis.

## Encrypted Verification Bitstream

A specially designed and encrypted bitstream can be provided to the device programmer for the verification procedures described in the [Using the Device Programmer for eFUSE Programming](#).

This section provides the reference design and instructions for building an encrypted verification bitstream for use on a device programmer.



---

**CAUTION!** Due to the device programmer socket pinout and limitations, do NOT use your actual design bitstream for verification on a device programmer.

---



## Reference Design

Due to constraints imposed by the device programmer, which are described in [Encrypted Verification Bitstream Design Modification and Limitations](#), the reference design is a mostly empty design.

The reference design files for this application note can be downloaded from:

<https://secure.xilinx.com/webreg/clickthrough.do?cid=393635>

Table 2 shows the reference design matrix.

Table 2: Reference Design Matrix

Parameter	Description
<b>General</b>	
Developer name	Randal Kuramoto
Target devices	7 series FPGAs
Source code provided	Yes
Source code format	VHDL/Verilog
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	Vivado 2014.4 synthesis
Implementation software tools/versions used	Vivado 2014.4 implementation
Static timing analysis performed	Yes
<b>Hardware Verification</b>	
Hardware verified	Yes
Hardware platform used for verification	AC701 evaluation kit

## Building the Encrypted Verification Bitstream Reference Design

After downloading the reference design files, use these instructions to build your encrypted verification bitstream:

1. Unzip the reference design directory tree and files to your computer.
2. Copy your NKY file to the `verilog` or `vhdl` directory within the reference design directory tree.
3. Edit the `XAPP1260_make_encrypted_verification_bitstream.tcl` file within the `verilog` or `vhdl` directory, and follow the usage instructions within the Tcl script file to customize the script for your device and NKY file. The corresponding usage instructions specify edits in the Tcl script:
  - a. Specify the device by replacing `xc7a200tfbg676-1` on this line in the Tcl script:

```
set part xc7a200tfbg676-1
```

- b. Specify the NKY file by replacing `XAPP1260_example.nky` on this line in the Tcl script:

```
set nkyFileName XAPP1260_example.nky
```

4. Start the Vivado design tools, change (cd) to the `verilog` or `vhdl` directory, and source the `XAPP1260_make_encrypted_verification_bitstream.tcl` file or use this command line:

```
vivado -mode batch -source XAPP1260_make_encrypted_verification_bitstream.tcl
```

5. Verify that the summary of `write_bitstream` options in the log includes the settings shown in [Encrypted Verification Bitstream Checklist](#).

The Tcl script builds the encrypted verification bitstream and formats the bitstream in a MCS file. The MCS file can be used on the device programmer. The resulting file is named:

```
XAPP1260_<part>_encrypted_verification_bitstream.mcs
```

where `<part>` is the part name specified in step 3.a. in [Building the Encrypted Verification Bitstream Reference Design](#).

For design validation or debug purposes, the Tcl script also creates an unencrypted version of the bitstream for checking that the design works in the FPGA without encryption. The unencrypted bitstream file name is:

```
XAPP1260_<part>_unencrypted_bitstream.bit
```

where `<part>` is the part name specified in step 3.a. in [Building the Encrypted Verification Bitstream Reference Design](#).

**Note:** The `XAPP1260_example.nky` file contains an all zero key value that matches the key of an FPGA with unprogrammed eFUSE. An encrypted test bitstream built using the `XAPP1260_example.nky` file can be correctly decrypted and used to successfully configure an FPGA whose eFUSE have not been programmed. This is true because the default all zero eFUSE key matches the all zero encryption key. This method can be used to validate the Vivado design tools encryption flow without having to program the eFUSE in an FPGA.

### ***In Case of a Build Error***

Check the `vivado.log` file from the Vivado design tools run of the Tcl script. The Tcl script can report an error when an issue with the part name or NKY file is encountered. If this error occurs, check the log files in the project subdirectories for an indication of an error:

```
ERROR: [Common 17-69] Command failed: Run 'impl_1' has not been launched.
```

Specifically, check the

`XAPP1260_encrypted_verification_bitstream.runs\synth_1\runme.log` file for errors during synthesis.

## ***Encrypted Verification Bitstream Design Modification and Limitations***

The reference design can be modified to alter the encrypted contents from the base reference design as needed. However, the resulting FPGA design must still comply with these limitations:

- A verification bitstream must not consume more than the minimum power-on current. Typically, a device programmer is power supply limited. It should be understood that the device programmer only has sufficient power to power on the device and program the eFUSE.
- Do not use any user-configurable I/O. A device programmer has fixed pin connections. It should be understood that the device programmer only properly connects to the dedicated JTAG interface pins. The connection or power for all user-configurable pins is undefined.
- Do not instantiate or enable the transceivers. It should be understood that the device programmer socket is designed with the assumption that the transceivers are not used or enabled.

## ***Encrypted Verification Bitstream Checklist***

These properties must be accurately specified for the encrypted verification bitstream:

```
BITSTREAM.ENCRYPTION.ENCRYPT = YES
BITSTREAM.ENCRYPTION.ENCRYPTKEYSELECT = EFUSE
BITSTREAM.ENCRYPTION.KEYFILE = <NKY file name that contains your AES key>
```

This property is recommended to minimize the verification time on the device programmer:

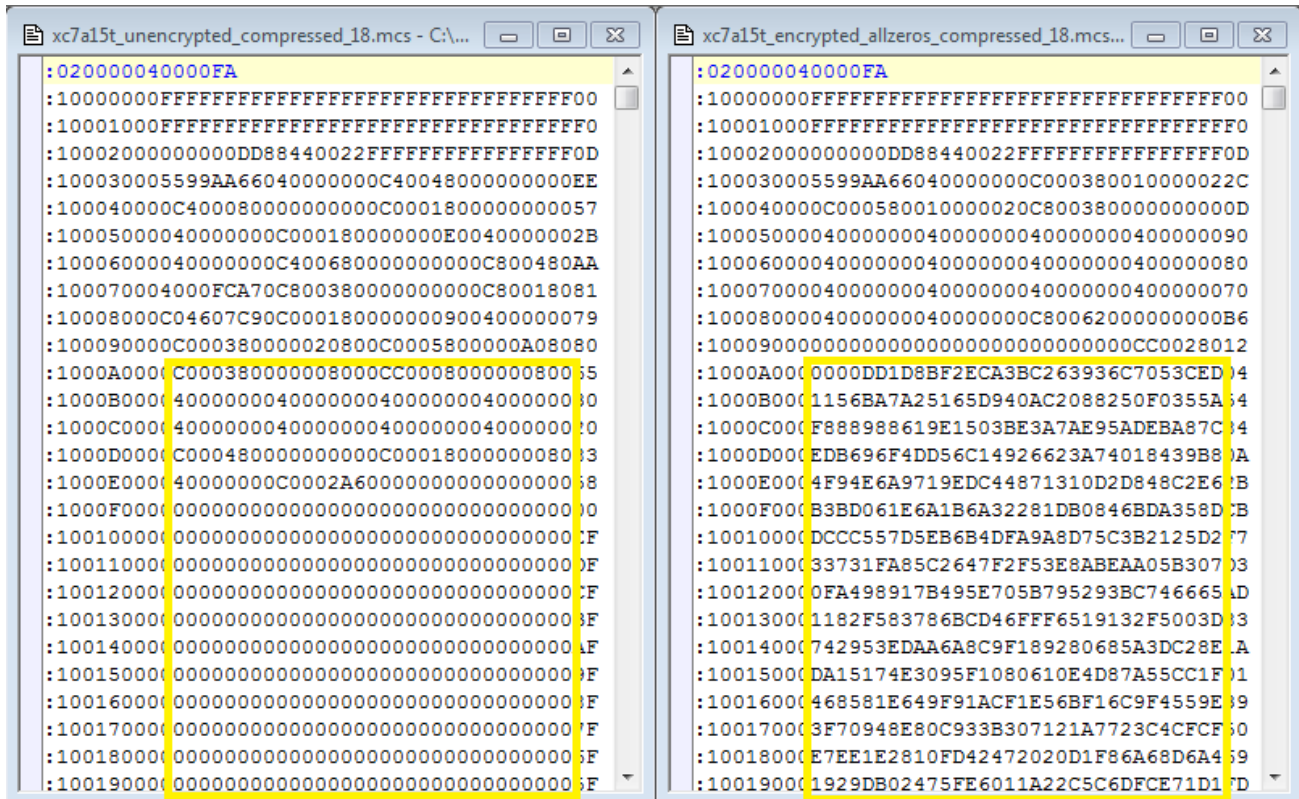
```
BITSTREAM.GENERAL.COMPRESS = TRUE
```

The settings for the properties listed can be verified by running the `write_bitstream` command with the `-verbose` option. The reference design Tcl script includes the `-verbose` option with its `write_bitstream` command. When `write_bitstream` is run with the `-verbose` option, a summary of the property settings when writing the bitstream is printed to the log. An extracted subset of the important fields within the summary is shown here:

```
Summary of write_bitstream Options:
+-----+-----+
| Option Name           | Current Setting       |
+-----+-----+
...
+-----+-----+
| KEYFILE                | your_key_file_name.nky |
+-----+-----+
...
+-----+-----+
| ENCRYPTKEYSELECT       | EFUSE                 |
+-----+-----+
...
+-----+-----+
| ENCRYPT                 | YES                    |
+-----+-----+
...
+-----+-----+
| COMPRESS               | TRUE                   |
+-----+-----+
...
```

The property settings previously listed are important because the bitstream dictates the decryption (or not) and the source of the AES decryption key. A bitstream generated without the correct settings can successfully configure the FPGA without having tested the AES key value stored in eFUSE. For example, the typical FPGA eFUSE settings allow configuration from an unencrypted bitstream. Consequently, an unencrypted bitstream can successfully configure the typical FPGA without having tested the stored AES key.

The data beyond the bitstream header (i.e., beyond the first approximately 12 lines) in the resulting encrypted verification bitstream MCS file can be visually inspected for encryption. The unencrypted bitstream data has mostly zero bit values (because most of the resources in a given FPGA design are not used). The encrypted bitstream should appear to be more like a random data pattern. See [Figure 2](#) for a comparison of an unencrypted versus encrypted bitstream data pattern. Bitstream data is highlighted in the yellow boxes.



X1260\_02\_051515

Figure 2: Unencrypted (On Left) Versus Encrypted (On Right) Bitstreams

# Guidelines for Validating Device Programmer Setup and Operation

The previous sections provide instructions for defining the device programmer settings and building the device programmer input files.

This section provides guidelines for:

- Checking the intended device programmer settings and input files before they are sent to the device programmer
- Checking the device programmer settings and input files
- Checking that the first device programmer programmed device works for the eFUSE settings on an assembled board

Use these guidelines as a checklist to ensure successful 7 series FPGA eFUSE programming on a device programmer.

## Precheck for Device Programmer Files

The guidelines in this section are for checking the intended device programmer settings and input files before they are sent to the device programmer.

1. Validate that the base reference design works for your device without encryption on a prototype board:

Use the Vivado hardware manager to download the XAPP1260\_<part>\_unencrypted\_bitstream.bit file to the FPGA on your prototype board that has not been programmed with an AES key in its eFUSE. The FPGA should successfully be configured.

2. Validate the security of an encrypted bitstream on a prototype board:

Use the Vivado hardware manager to download the encrypted verification bitstream to the FPGA on your prototype board that has not been programmed with an AES key in its eFUSE. The FPGA should fail the configuration attempt.

3. Check that the encrypted verification bitstream and actual encrypted design bitstream work in the device on your prototype board:

- a. Use the Vivado hardware manager to program the eFUSE with the settings from your completed [Table 1](#). See *Using Encryption to Secure a 7 Series FPGA Bitstream* (XAPP1239) [[Ref 3](#)] for instructions on using the Vivado tools to access the device eFUSE.
  - Use the Vivado hardware manager to view the device **Properties** -> **Register** -> **eFUSE** -> **FUSE\_CNTL** value. Save a copy of the FUSE\_CNTL value that results from the Vivado hardware manager programming of the eFUSE. This can be used later to verify that the device programmer results in the same FUSE\_CNTL value.

- b. Use the Vivado hardware manager to download the encrypted verification bitstream to the device on your prototype board. The FPGA should successfully be configured.
- c. Use the Vivado hardware manager to download the actual encrypted design bitstream to the device on your prototype board. The FPGA should successfully be configured.

## Device Programmer Setup Verification

The guidelines in this section are for checking the device programmer settings and input files.

4. Check that the device programmer standalone verify operation reports an error when the device is NOT programmed with the matching key:
  - a. Put a factory new device in the device programmer socket.
  - b. Run the standalone verify operation with the encrypted verification bitstream.
  - c. The device programmer should report an error.

If the standalone verify passes without error, then:

- Recheck that the device is blank.
  - Recheck the encrypted verification bitstream. Use the Vivado hardware manager to configure a device on a board, which does not have the eFUSE programmed, with the encrypted verification bitstream. The Vivado hardware manager should report a configuration error.
5. Program and verify one or two test devices with the settings, and send the programmed test devices for testing on an assembled board.

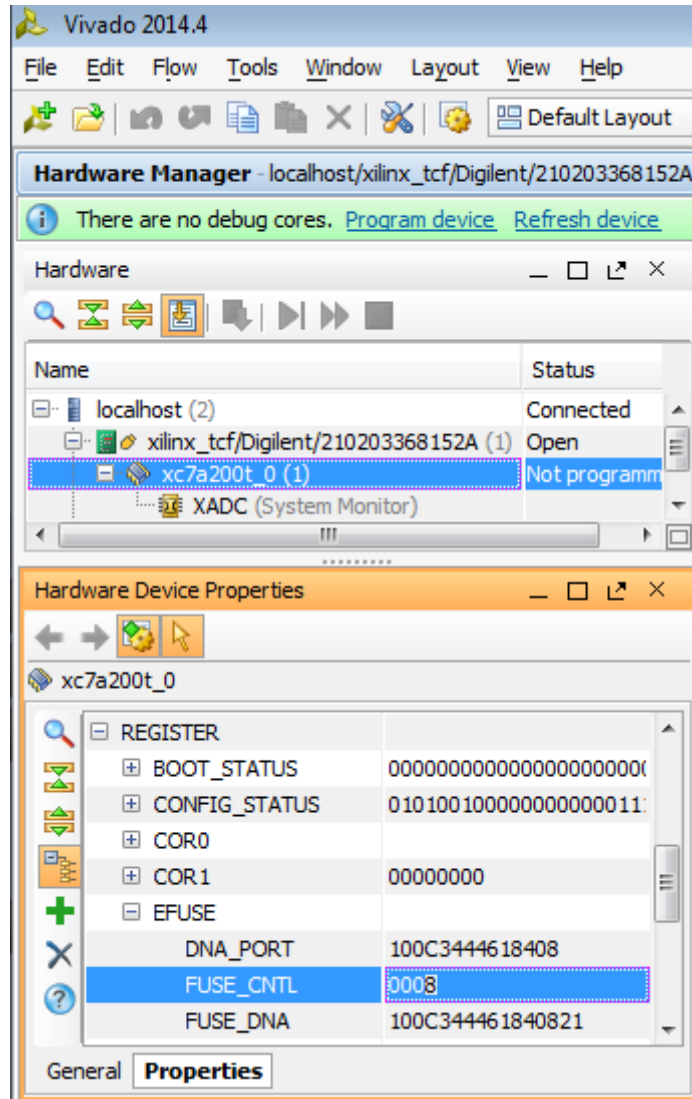
If the verify operation fails, then recheck the encrypted verification bitstream using the guidelines from the [Precheck for Device Programmer Files](#) section.

## First Programmed Device Verification on Assembled Board

The guidelines in this section are for checking that the first device programmer programmed device works correctly for the eFUSE settings on an assembled board.

6. Check the FUSE\_CNTL register for read-security of the AES key and other security settings:
  - a. Use the Vivado hardware manager to connect to the device on a board through JTAG.
  - b. View the device **Properties** -> **Register** -> **eFUSE** -> **FUSE\_CNTL** value. If you saved the FUSE\_CNTL value that resulted from the Vivado hardware manager programming of the eFUSE in a prototype board, then compare the FUSE\_CNTL values from the Vivado hardware manager programming to the device programmer programmed device.
  - c. Check that at least the R\_EN\_B\_Key (FUSE\_CNTL[3]) bit is programmed to a "1". For example, see [Figure 3](#), which shows the hexadecimal value 0008 (binary value 00000000001000). The example correctly has the FUSE\_CNTL[3] bit programmed to a "1".

- If the FUSE\_CNTL[3] bit is not programmed to "1" or if the FUSE\_CNTL register value from the device programmer programmed device is not the same as the Vivado hardware manager programmed device, then recheck all device programmer settings.
7. Load your encrypted bitstream into the FPGA.
- a. If configuration fails, recheck that your design and the device programmer settings used the exact same NKY file (AES key) to encrypt the bitstream.



X1260\_03\_051515

Figure 3: Vivado Hardware Manager FUSE\_CNTL Value



## References

1. *7 Series FPGAs Configuration User Guide* ([UG470](#))
2. *Vivado Design Suite Programming and Debugging User Guide* ([UG908](#))
3. *Using Encryption to Secure a 7 Series FPGA Bitstream* ([XAPP1239](#))

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/08/2015	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 2015 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.