Advanced Security Encryption Key Programming Guide for ECP Device Family

Technical Note

FPGA-TN-02202-1.8

July 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUNDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations and terms used in this document.

Abbreviations or Terms	Definition
AES	Advanced Encryption Standard.
JTAG	Joint Test Action Group.
OTP	One-Time Programmable.
PCM	Pulse-Code Modulation.
PROM	Programmable Read Only Memory.
SPI	Serial Peripheral Interface.
Authentication	The algorithmic validation process to determine pass/fail results.
Bitstream	A binary file that contains commands and data that can configure FPGA devices.
Configuration	Program (write to the fuses of) a volatile device.
Decipher Key	The key code used for encryption or decryption.
Decrypt	Apply the reverse encryption process on an encrypted file.
Encrypt	The encryption action has taken place.
Encryption	Uses a password (key) and an algorithm to scramble a file.
Hashing	A scrambling algorithm that makes it impossible to restore the original contents after it is applied. For example: Let the seed = 10100101. Let the hashing code = 4 right-shifts. • Hashing: 4 right-shifts of the seed = 00001010 • Reverse hashing: 4 left-shifts of the hashed seed = 10100000
Key Code	The binary key pattern for encryption or decryption.
Key Code Size	The fixed length of the key code in bits. For ECP5, ECP5-5G, LatticeECP3, LatticeECP2MS, and LatticeECP2S devices, it is 128 bits.
Key Lock Fuse	When programmed, the key lock fuse prevents the encryption key code from being read out.
Non-Volatile Fuse	Fuses that keep the fuse state when power is turned off.
OTP Fuse	One-time programmable, non-volatile fuse that can only be programmed once, and cannot be erased.
Over Program	An error caused by blowing a fuse that was supposed to remain un-programmed.
Private Key	The key code that is confined to the Trusted Area.
Program Fuse	An OTP fuse that is blown to produce a logical state 1.
Public Key	The key code that is not confined to the Trusted Area.
Un-program Fuse	An OTP fuse that is not blown (left intact) to produce a logical state 0.
Under Program	An error caused by leaving intact (un-blown) a fuse that was supposed to be programmed.
Unencrypt	No encryption action has taken place.
Trusted Area	The real or virtual space that houses all confidential and high-security material.

1. Introduction

All Lattice FPGAs provide configuration data read security, meaning that a fuse can be set so that when the device is read, all 0s are outputted instead of the actual configuration data. This type of protection is common in the industry and provides good security when the configuration data storage is already programmed into the device. However, if the configuration bitstream comes from an external boot device it is quite easy to read and intercept the configuration data, allowing access to the FPGA design.

For this reason, Lattice offers the 128-bit Advanced Encryption Standard (AES) to protect the bitstream. You select and have total control over the 128-bit key and no special voltages are required to maintain the key within the FPGA. This is available on the *S* (security) versions of LatticeECP2™ and LatticeECP2M™ devices. It is also available in all versions of the LatticeECP3™, ECP5™, and ECP5-5G™ devices.

All volatile FPGAs require non-volatile media, such as a SPI Flash device, to store the bitstream, which configures or boots-up the FPGA. Therefore, SPI Flash memory is also known as the “boot PROM” for volatile FPGA devices.

Before the introduction of encryption, bitstreams are not protected or secure. Anyone could copy the design simply by reading the bitstream out of the boot PROM.

If a bitstream is encrypted it cannot be used, even if it is read out of the boot PROM. An encrypted bitstream only works with an FPGA that contains the same encryption key that was used to encrypt the bitstream.

Encryption offers a mechanism to prevent reverse engineering of intellectual property. It can also be disabled to use the bitstream in an unsanctioned device for the purpose of limiting supply of the device for inventory control purposes.

One of the best techniques to produce a unique encryption key is serialization.

In terms of design security, an encrypted bitstream is known as protected and an unencrypted bitstream is unprotected.

The AES security system is the best method to provide the highest level of protection to the bitstream of volatile FPGA devices. This is demonstrated by the fact that some FPGA vendors started with different encryption schemes and all eventually standardized on AES. A detailed description of AES can be found in the U.S. government publication of the standard.

AES uses the symmetric Decipher Key format (that is, encryption key = decryption key). Therefore, the terms decipher key, encryption key, and decryption key are synonymous. AES defines the decipher key size to be 64, 128, or 256 bits in length.

The LatticeECP3, LatticeECP2MS, and LatticeECP2S FPGA families support:

- The optimal 128-bit AES key
- The reliable one-time programmable (OTP) non-volatile fuses for the AES encryption key
- An unencrypted bitstream can be programmed into the device even if security key is set

The ECP5 and ECP5-5G families support:

- The optimal 128-bit AES key
- The reliable OTP non-volatile fuse for AES encryption key
- Option of unencrypted bitstream can be or cannot be programmed into the device when security key is set.

This document describes the security key programming operations at various stages of the implementation:

- Board prototyping using the Lattice Diamond™ Programmer software
- Manufacturing using the Diamond Programmer software
- Large volume production programming using desktop programmers
- Advanced security key programming using the Diamond Programmer software
    - Decryption key serialization
    - Bitstream encryption with serialized decryption key

The encryption scheme detail implemented in ECP5, ECP5-5G, LatticeECP3, LatticeECP2MS, and LatticeECP2S FPGA families is provided in this document for those interested in the implementation details.

2. Overview

The Diamond Programmer software takes care of all the technical details of programming the device, which saves you from having to understand all the underlying programming complexities.

The technical information about the key code programming provided here is for the benefit of encryption users as a reference. Due to the complexity of programming the key code into the device, only the high-level flow is presented. This document does not provide the details for you to craft your own programming code (driver).

In LatticeECP2S, LatticeECP2MS, LatticeECP3, ECP5, and ECP5-5G devices, the OTP fuses for the key code can be programmed in-system using the Diamond Programmer software in one of the following ways:

- Onboard programming using a Lattice parallel port cable with the 8-pin AMP connector or 10-pin JEDEC connector connected between a PC or Linux system and a board test system
- Onboard programming using a Lattice USB port cable connected between a PC or Linux system and a board test system
- Embedded onboard programming using FTDI single chip USB converter devices connected between a PC or Linux system and an outfitted PCB host
- Off board programming using the Lattice Model 300 Desktop Programmer and Socket Adapter

The OTP fuses require a complicated if-then-else pass/fail decision tree programming flow. The following files cannot support programming of the OTP fuses because of their simple go/no-go only pass/fail format:

- SVF files for IEEE 1149.1 BSCAN tools
- VME files for embedded field upgrades
- ATE vectors for the Agilent HP3070 bed-of-nail board testers
- ISC BSDL files for IEEE 1532 compliant tools

Instead, Lattice provides you with the following methods to program the key code:

- Off board programming using key files for third-party programmer vendors (BPM Microsystems, System General).
- Onboard programming using STAPL files for third-party BSCAN tools

The weakest link of the encryption flow is the transmission path of the key code from the source to the programming area. The following lists two methods for transmitting the key code:

- Verbal
- Written (that is, encrypted key files)

The best deterrence against hackers is to transmit the key code in the form of encrypted key files to the trusted area. The password for opening the encrypted key files can be transmitted when the programming operation begins, and controlled with either time or quantity expiration.

The Diamond Programmer software supports the following methods of key code transmission:

- Entering the key code directly into the GUI
- Importing the key file after entering the password for the key file

To maintain the confidentiality and security of the key code, an industry standard requires all software and programming tools for key code programming to observe the following rule:

It is strongly recommended you do not make a hard copy of the key code in the form of a data record (file) on a non-volatile media, such as hard disks, or as a data image into non-volatile memory such as Flash devices.

Therefore, software and programming tools must only copy the key code into the volatile RAM of the PC during run time to adhere to this rule.

The Diamond software follows the rule except when the key code is entered directly into the Diamond Security Setting Tool GUI or into the Diamond Deployment Tool GUI. The key code is written into the chain configuration (.xcf) file or into a key (.bek) file. You are informed and can make the appropriate arrangement to protect the files.

The AES has two algorithms—encryption and decryption. The U.S. government requires distribution control on all software containing the encryption algorithm. To meet these requirements, the Diamond Programmer software does not support encryption through normal software distribution. Lattice only distributes these capabilities through the licensed-controlled Lattice Diamond design software. The standalone Diamond Programmer software control patches can be obtained from Lattice Sales. The control patch inserts the encryption algorithm into the Diamond software tools to enable encryption.

If the Diamond design software does not have encryption enabled, the following operations cannot be performed:

- Converting an unencrypted bitstream into an encrypted bitstream
- Writing (saving) the password-protected (encrypted) key file

If encryption is not enabled, software cannot generate the key file. Instead, the key code that you entered is written into the chain configuration (.xcf) file.

Note: The .xcf file is not encrypted, thus the key code can be obtained by reading the .xcf file. Therefore, entering the key code directly should be carried out only in a trusted area.

The true security of encryption depends entirely on the security (protection) of the key code. Once the key code is compromised, the corresponding encrypted bitstream is also compromised. This is because the decryption algorithm is available to decrypt the bitstream with a good key.

Note: A good key is the key code that was used to encrypt the bitstream. The bitstream does not work if a different key code is used to decrypt the encrypted bitstream. Therefore, a different key code is a bad key. A working bitstream means the device receiving the bitstream is configured successfully, resulting in both DONE and INITN pins going high. By simply viewing a decrypted bitstream, one cannot tell if it successfully configures a device.

A specialist skilled in the art of encryption can extract the key code if both the unencrypted bitstream and the encrypted version of the same bitstream are available. Thus, for security reasons, it is standard practice in the advanced security industry to demand traceability of the unencrypted bitstream. After the unencrypted bitstream is encrypted, the unencrypted bitstream must be destroyed (deleted) to protect it.

The discussion above focuses on standard practices of the security industry for protecting key codes from leaking out through paperwork control. For hackers, instead of hunting the key code through paperwork leaks, their first line of attack is to read from FPGAs:

- Open the package and inspect OTP fuses on the die visually.
- Open the package and micro-probe the die to read the OTP fuses electronically.

In addition to the standard silicon layout defensive practices (that is, hiding fuses under several layers of metal and scrambling their physical locations), Lattice has implemented the hashing code by the mesh circuit in ECP5, ECP5-5G, LatticeECP3, LatticeECP2MS, and LatticeECP2S devices. This is a reliable defense against the two hacking tactics mentioned above. Even if the hacker succeeds in obtaining a key code, the key code is invalid (useless).

The deployment of the hashing code helps assure that the key code cannot be compromised by hacking the device security. The key code is extremely secure when it is programmed into the device. The only drawback of hashing is that the key code fuse verification (the fuse-by-fuse check for an exact match) is not supported. Instead, verification can only be done by using the encrypted bitstream. Verification by encrypted bitstream can be done before and after the key lock fuse is programmed.

Lattice devices are uniquely designed to accept a regular bitstream whether or not the encryption key is programmed into the device. This allows for system debugging even if the key code is lost or unknown. This is especially handy when each device has a unique key code.

Some designers believe that if a device accepts both an encrypted and unencrypted bitstream, hackers can easily create an unencrypted bitstream for the device to plant the probing agent (hacker probe) to compromise the security of the system, and thus the device would have a security hole. This conclusion results from the assumption that the unencrypted bitstream for the hacker probe can be superimposed on the encrypted bitstream. This cannot happen because prior to configuring the device with an unencrypted bitstream, the Lattice devices perform a mandatory clear-all operation which erases the SRAM fuses prior to configuration. Thus, the hacker probe planted into a cleared device is useless and the key code and the encrypted bitstream remain protected. On top of that, ECP5 and ECP5-5G devices also provide an *Encryption Only* option, which allows you to set the device to accept encrypted bitstream only. However, this option is available with ECP5 and ECP5-5G devices only.

3. Lattice Encryption Feature

The following figure illustrates the logical functional blocks supporting the encryption feature.

The Lattice device families support the encryption feature by using three key rows. Each row has 128 fuses, for a total 384 (3 x 128) fuses forming the 128-bit decryption key.

Theoretically, only one row of 128 fuses should be sufficient to support 128-bit decryption key.

Due to the nature of the OTP fuses, some fuses might not respond to the program command. Thus, one or more additional row(s) of fuses is provided to serve the following purposes:

- Serve as back-up fuses to enhance the encryption key programming yield
- Serve as redundant fuses to enhance encryption key reliability

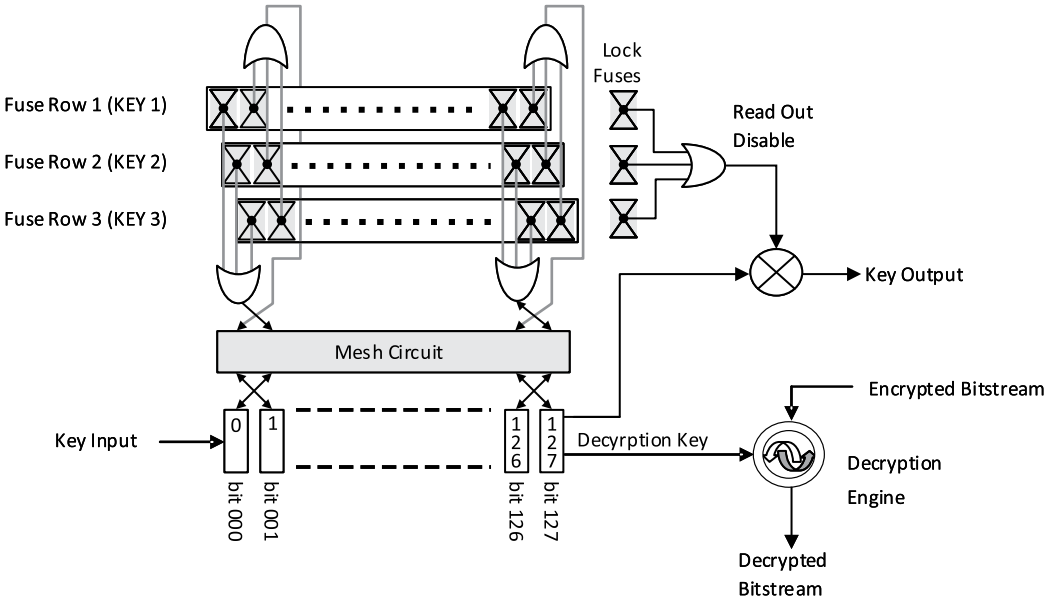

Figure 3.1. Encryption Key Fuse Block Diagram

Because KEY2 and KEY3 rows are redundant fuses, coupling with the fact that the E-fuses are OTP only, the KEY1, KEY2, and KEY3 rows should be programmed to the same key code.

The function of the key lock fuses is to protect the programmed encryption key from being read. Therefore, programming the key lock fuse does not affect the encryption feature. If the device key has been programmed, before shipping the boards out to end-customers, the key lock fuse must be programmed to protect the encryption key from falling into the unauthorized hands.

The purpose of the mesh circuit is to destroy the visual correlation between the physical location of a fuse and its logical function (that is, the decipher key bit number). In other words, the key code is still protected even if the die is open and the fuse pattern is read by visual inspection. This circuit causes the key code shifted in for programming to be different from the one shifted out for verification.

The task of the decryption engine is to decrypt the incoming encrypted bitstream, turning it into an unencrypted bitstream while writing into the SRAM fuses. After the unencrypted bitstream is written into the SRAM fuses, setting the security fuse is the only method that can be used to protect the bitstream. The security fuse is set automatically when encrypting the bitstream.

4. Encryption Key Programming Algorithm

The E-fuses in Lattice devices do not need a super-voltage or high current to be programmed. This allows the encryption key to be programmed in-system. The two additional rows of backup E-fuses ensure the encryption key programming yield is high. However, the nature of E-fuses does not allow for any errors. When an E-fuse that is not supposed to be programmed is programmed, the device is rejected. The following standard precautionary measures are deployed to minimize the possibility of rejects:

- Use a qualification technique to verify first, then program
- Use a double shifting-in technique to screen out double clocking
- Use a double verify technique to protect against data corruption
- Use a redundancy fuses scheme to protect against power disruption during programming

4.1. Encryption Key Programming Flow

Figure 4.1 provides a high-level view of the encryption key programming flow implemented in the Diamond Programmer software. The programming algorithm for the OTP fuses is complex, and it is impractical for users to craft their own programming code. It is also beyond the scope of this document to describe the programming flow in detail.

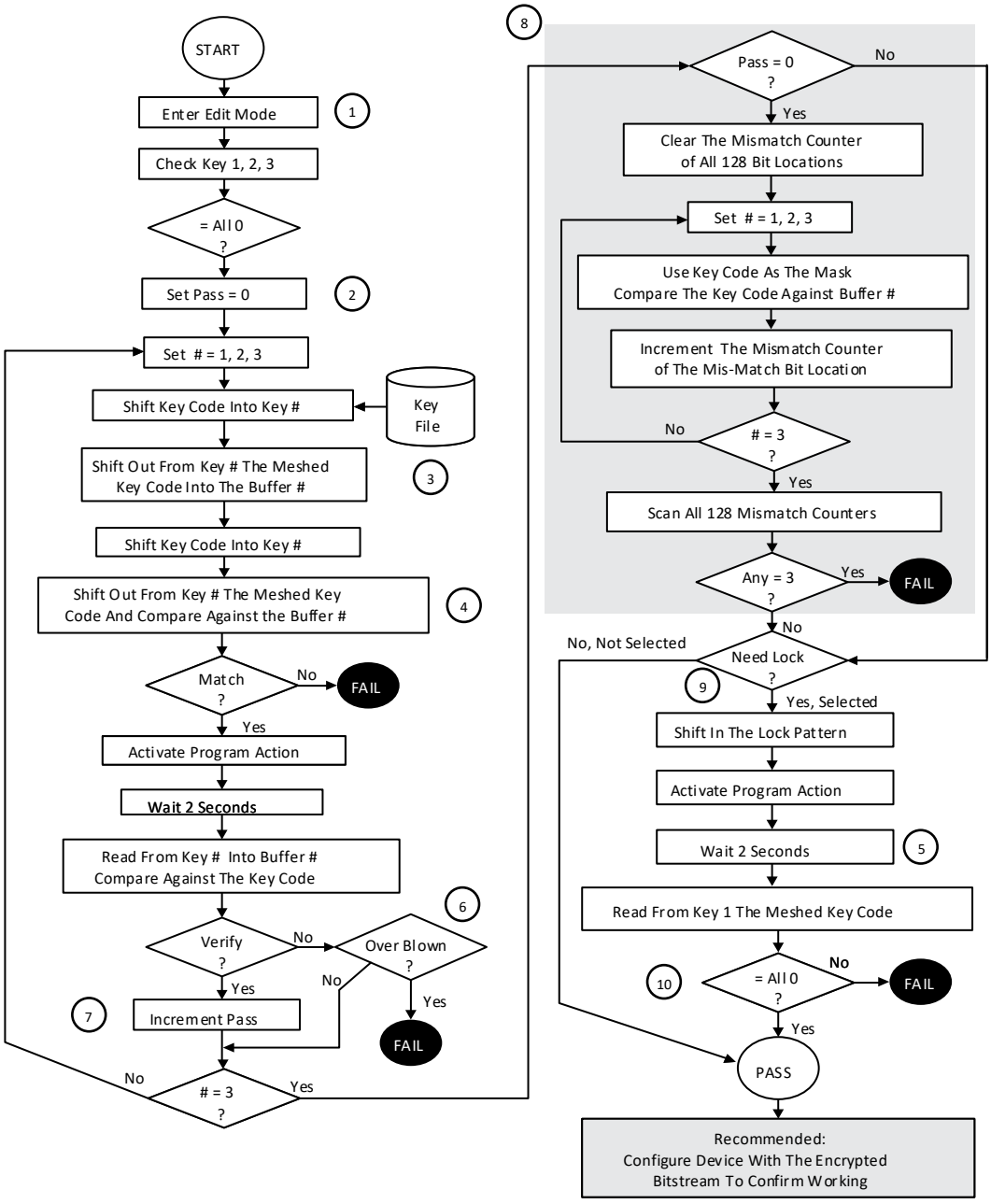

Figure 4.1. Encryption Key Fuse JTAG Programming Flow

Table 4.1. Encryption Key Fuse JTAG Programming Flow

Note	Description
1	Set the device into JTAG programmable mode. The device requires a special password to arm the E-fuse programming engine to prevent accidental and unauthorized access to the E-fuse block.
2	There are three rows of E-fuses that must be programmed. The pass-counter remembers the total number of fuse rows with 100% pass verify.
3	This is the first read of the key code that you entered. The key code is read the first time it is used as the master key code for comparison.
4	This is the second read of the key code. If the comparison with the master key in Note 3 passes, then the key code entered has been shifted in correctly. The key code is authenticated.
5	Programming time or burn time. This is the most critical moment of the programming process. The power supply must not be interrupted. If a power disruption happens (such as a ground surge), the E-fuse state is not undefined.
6	An unprogrammed fuse state = 0. A programmed fuse state = 1. If any un-programmed fuse is found to be programmed, then an over-blow has occurred and the device must be rejected.
7	If the key code read from the fuses matches the master key code from Note 3, the key is 100% verified and the pass counter is incremented to register a perfect match.
8	The pass/fail decision flow changes from a simple go/no-go flow to the more complicated if-then-else flow when there is no perfect match (that is, the value of the pass counter = 0). The if-then-else flow is necessary for all three rows that need to be re-checked to make sure that at least one fuse is programmed in the column(s) where the fuse state is expected to be a 1 (a programmed fuse state). IEEE 1149.1-specific SVF files, IEEE 1532- specific ISC BSDL and ATE vectors are for go/no-go flow only. The JEDEC STAPL file is the only standard file that supports the if-then-else flow. Therefore, for large volume key code programming, Lattice recommends: • Use third-party desktop programmers (BPM Microsystems or System General) • Use BSCAN tools which support STAPL files
9	Once the lock fuse is programmed, the key code cannot be read out to protect it from falling into unauthorized hands. This step, therefore, is mandatory in production programming. It is optional during the board development phase or evaluation phase. Users should select the option accordingly.
10	This step verifies that the lock fuse is programmed. If the lock fuse is indeed programmed, instead of the key code, all 1s, for ECP5 and ECP5-5G devices, or all 0s, for ECP3 and ECP2/MS devices, in the fuse state will be read out.

Due to the complexity of the encryption key programming flow, it cannot be expressed by using the simple SVF format. Instead, you are strongly advised to use the STAPL file.

The Diamond Deployment Tool can generate a STAPL file for you to program the encryption key in-system using a third-party tool that supports SVF files. All third-party tools that support SVF files also support STAPL.

Lattice works closely with third-party programmer vendors to develop off board programmer support. Lattice provides detailed programming algorithm specifications to the Lattice-approved third-party programmer vendors. Lattice engineers perform a full qualification on the programming code developed by these vendors to ensure the fuses are programmed to meet Lattice's quality and reliability requirements.

Lattice also can pre-program a user's encryption key into a device before shipment. Contact Lattice Sales for details.

4.2. ECP5, ECP5-5G, LatticeECP3, LatticeECP2MS, and LatticeECP2S Bitstream Encryption Format

Table 4.2. Encrypted Bitstream Format and Configuration Mode Dependency

	Master Modes	Slave Modes		Description
	SPI[1] (Bit Wide Serial) Bit 0…n[2]	**JTAG, SCM, SSPI (Bit-Wide Serial) Bit 0…n**	**PCM (CPU) (Byte-Wide Parallel) D0..7**	
Comments	(Comment String)			Optional. Can be replaced by >= 128 bits of dummy. See Note 4.
Header	1111…1111			Mandatory if the Comment String above exists.
Encryption Preamble	1011101010110011			Mandatory. 0xBAB3 decryption preamble code.
30,000 Filler Bits	11…1111	11…1111		Mandatory. Provides delay time > 500us for the decryption engine to load the decryption key. The 30K clocks are calculated with Fmax = 60 MHz to yield the 500us delay. See Note 5.
Alignment Preamble	1011110010110011			Mandatory. 0xBCB3 is the alignment code to signify the encrypted bitstream follows.
Delimiter Bit[3]	1			Mandatory. This bit separates and provides a smooth transition between the unencrypted field and the encrypted field.
Encrypted Configuration Data	128 Bits of Encrypted Data	128 Bits of encrypted data		The decryption packet size = 128 bits per packet. Each packet is punctuated by filler equivalent to 64 clocks. SPI mode needs no filler since it is a Master Mode. It provides its own clock, using the technique of clock suspension (stops CCLK for 64 clocks worth of delay). In CPU mode, each clock is for a byte of data, thus 64 clocks = 64 bytes = 64 * 8 bits = 512 bits of filler. See Note 3 for details on filler.
	128 Bits of Encrypted Data	64 bits of 1s as filler	512 bits of 1s as filler	
	- - - - - - - -	- - - - - - - - - - - - - - - -		
	128 Bits of Encrypted Data	128 Bits of encrypted data		
	128 Bits of Encrypted Data	64 bits of 1s as filler	512 bits of 1s as filler	
Encrypted Frame Program Done	32 Bits Encrypted			Mandatory. Terminates the bitstream engine and readies the device for wake-up.
Frame (End)	32-bit Encrypted Terminator			Mandatory. Terminates the decryption engine.
Packet Bound Filler Bits	11…1111			Mandatory. Sizes the last frame to be packet- bound.
207 bits/wake up clocks	111…1111			Mandatory. 207 + the delimiter bit = 208, thus the entire bitstream is byte-bounded.

Notes:

1. Dual Boot with encryption only available with LatticeECP3, ECP5 and ECP5-5G devices. Unsupported with LatticeECP2S and LatticeECP2MS devices. Multiple Boot with encryption is available only with ECP5 and ECP5-5G devices.
2. The direction of shifting the bitstream is from left to right (that is, bit 0, 1, 2…n).

3. The single delimiter bit prepares the device for the encrypted bitstream. This bit offsets the entire bitstream from byte bond until the end of the bitstream where the 207 bits of wake-up clocks are padded. The consequence of this delimiter bit causes the skipping of filler bits to keep the encryption bitstream size small rather difficult.
   The encrypted bitstream in Slave Mode, as compared to Master mode, increases drastically due to the insertion of the filler bits. In SCM and JTAG modes, it would increase the bitstream size by 50%. In PCM mode, it would increase the bitstream size by 400%. The drivers for the SCM and PCM modes can be designed to provide their own filler bits (they are the 64 decryption clocks). The same encrypted bitstream without filler bits for SPI mode can also be used. Embedded applications can only support an encrypted bitstream with the filler bits already inserted. Deployment Tool prompts you to convert the encrypted bitstream to meet the filler bits requirement when users choose the Fast Program operation.
4. Do not select the no-header option when generating the bitstream for SPI mode operation. The header string is used in lieu of dummy bits. The device ignores the first 128 bits of data coming out of the SPI Flash device.
5. When the devices read configuration data out of the SPI Flash devices, the first 128 bits of data are ignored since most SPI Flash devices provide random data on the first eight clocks in a fast read. The header string, which usually stores your entry, is a don't-care for configuration purpose. Deployment Tool uses the header as the filler to meet the 128 bits dummy requirement.

4.3. Creating an Encrypted Bitstream File and Key File

The Encryption feature must be enabled to create a key file and encrypt a Lattice bitstream file. There are two ways to create an encrypted key file and encrypted bitstream file:

1. In the Lattice Diamond design software, set security settings using the Security Setting tool and then generate an encrypted bitstream from within the Process pane.
2. In the Deployment tool, specify security settings and convert an unencrypted bitstream file to an encrypted bitstream file.

4.4. Setting Security and Encryption for FPGA Devices

There is an important difference between using the Deployment Tool and using the Security Setting tool and the Diamond Process flow. When the Security Setting tool is not used, an unencrypted bitstream is created in the Diamond Process flow. When using the Security Setting tool to specify encryption settings and the Diamond Process flow to create a bitstream, the Diamond software only creates an encrypted bitstream. When using the Deployment tool, an unencrypted bitstream file is used to create an encrypted bitstream file. Therefore, using the Deployment tool is considered less secure unless the tools and the unencrypted bitstream file are in a trusted secure area. The unencrypted bitstream could be protected by using a security control such as PKZIP along with a password to protect the file.

4.5. Creating the Encryption Key and Encrypted Bitstream Using Diamond Security Setting Tool and Process Flow

To create the Encryption Key and Encrypted Bitstream, follow these steps:

1. In the **Diamond File List** window pane, double-click on the **Strategy** to be used for your design to invoke the **Strategies** option window. In this window, select **Bitstream**. Set the required Bitstream options and click **OK**.

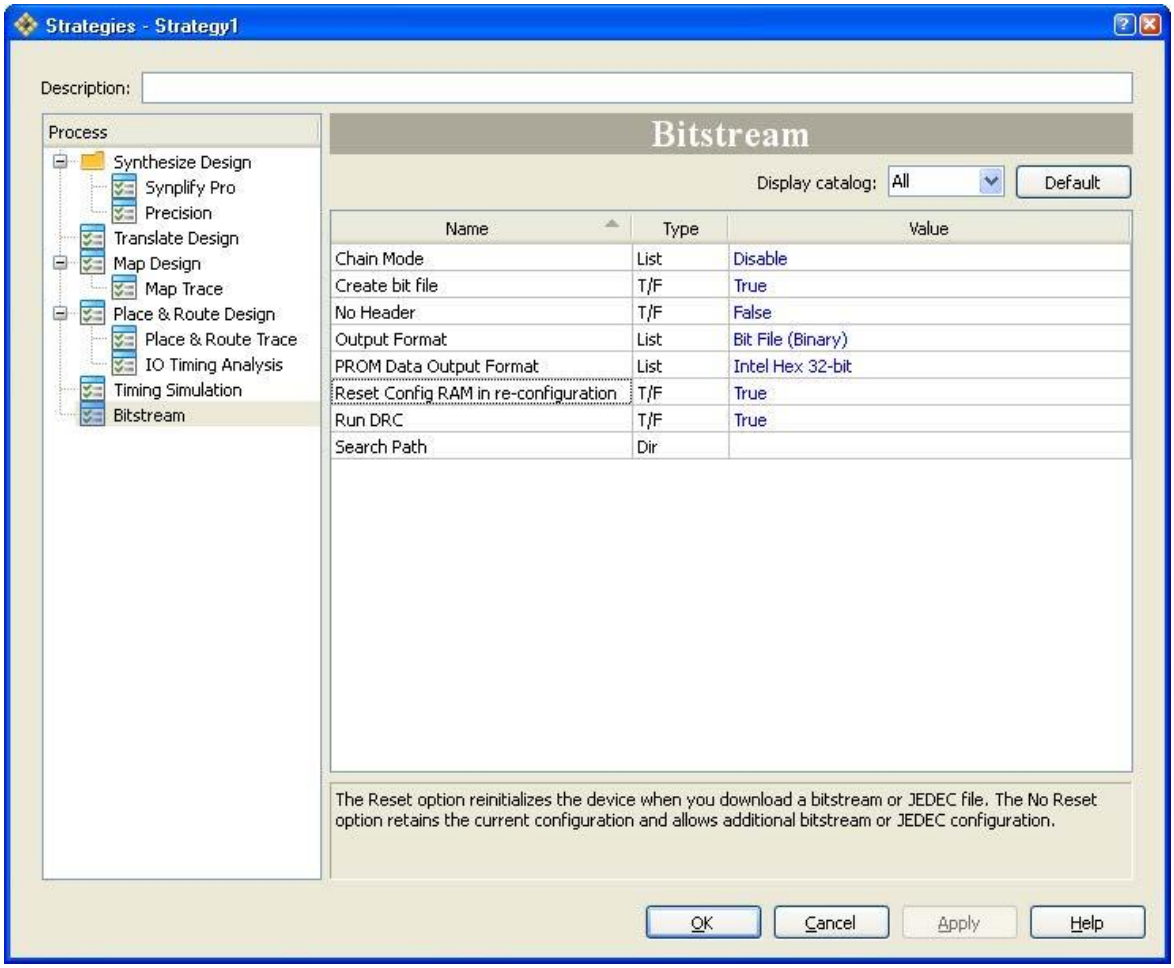

Figure 4.2. Strategies Options Window

2. In the Diamond software, select **Tools > Security Setting**. The **Enter Password** dialog box appears with the default password as LATTICESEMI.

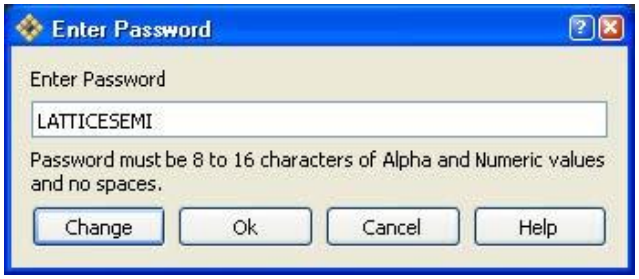

Figure 4.3. Enter Password Dialog Box

3. Select the **Change** button to open the **Change Password** dialog box and create a unique password for the encryption key file. The password must be between 8 and 16 characters. Verify the password and select **OK**.

Figure 4.4. Change Password Dialog Box

4. The **Security Settings** dialog box appears. Select the **Advanced Security Settings** check box to enable entering the Encryption key. Select the **Key Format**. The following lists the choices:
   - **ASCII** – Alphanumeric values, using up to16 characters
   - **Hex** – Values of 0 through F, using up to 32 characters
   - **Binary** – Values of 0 and 1, using up to 128 characters
5. Select **OK** to create the bitstream encryption key file (.bek).

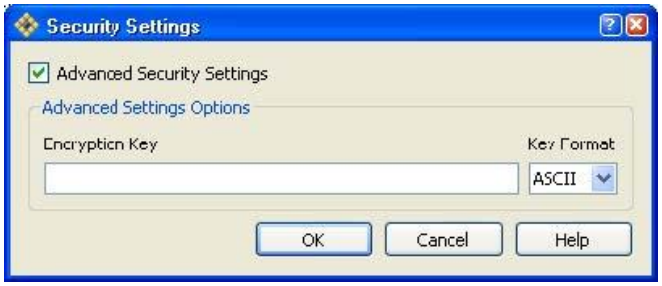

Figure 4.5. Security Settings Dialog Box

6. To create the encrypted bitstream, double-click on **Bitstream File**, under **Export Files**, in the **Process** pane. An encrypted bitstream file (.bit) is created. You can now use this file to program the device using the Diamond Programmer software. The encrypted bit stream generation is dependent on the configuration mode that you selected in the Diamond Spreadsheet view. This step is necessary to pad the correct number of program mode dependent filler bits when encrypting the bitstream. The filler bits are described later in this document.

5. Encrypted Bitstream JTAG Programming

The way Diamond Programmer software supports JTAG programming makes the differences between an encrypted bitstream and an unencrypted (regular) bitstream transparent to you.

The major difference between an encrypted and a regular bitstream is the configuration mode dependency of the encrypted bitstream. Regular bitstream format is mode-independent. Therefore, regular bitstreams do not have the configuration mode note field on the bitstream header, whereas the encrypted bitstreams do.

Programmer uses the note field of the bitstream header to ensure the bitstream is encrypted and the configuration mode is selected, and prompts you accordingly. If the configuration mode selected is SPI or SPIm, you are prompted to convert the configuration mode and save it as a temporary file. If the configuration mode selected is Slave SCM, no conversion is necessary.

Note: When prompted to convert an encrypted bitstream for SPI or SPIm configuration modes, do not overwrite the original file. Instead, save the changed file to a different file. If the encrypted bitstream has the configuration mode changed to Slave SCM then programmed into the SPI Flash, the Lattice devices fail configuration. Only the encrypted bitstream for SPI or SPIm configuration mode can be programmed into the SPI Flash devices.

5.1. Setting Security and Encryption Using the Deployment Tool

The Deployment Tool is a stand-alone tool that allows you to generate files for deployment for single devices or a chain of devices. It is also used to convert data files to other formats. The Deployment tool can be installed as an individual tool or as part of the Diamond software installation.

1. When installed as part of the Diamond software, invoke the Deployment tool from: **Start > Programs > Lattice Diamond > Accessories > Deployment Tool**.

   In Linux, from the **<install_path>/bin/lin** directory, enter the following on a command line: **./deployment**.
2. The **Getting Started** dialog box appears. Choose **Create New Deployment**.

Figure 5.1. Getting Started Dialog Box

3. In the **Function Type** drop-down menu, choose **File Conversion**.
4. In the **Output File Type** drop-down menu, choose **Bitstream.**
5. Click **OK**. A **File Conversion** (step 1) window is displayed.
6. Left-click on the blank **File Name** column. Then click the browse (**…**) button in the **File Name** column. Browse to select the unencrypted .bit file that was created by double-clicking on the Bitstream file in the **Process** window.

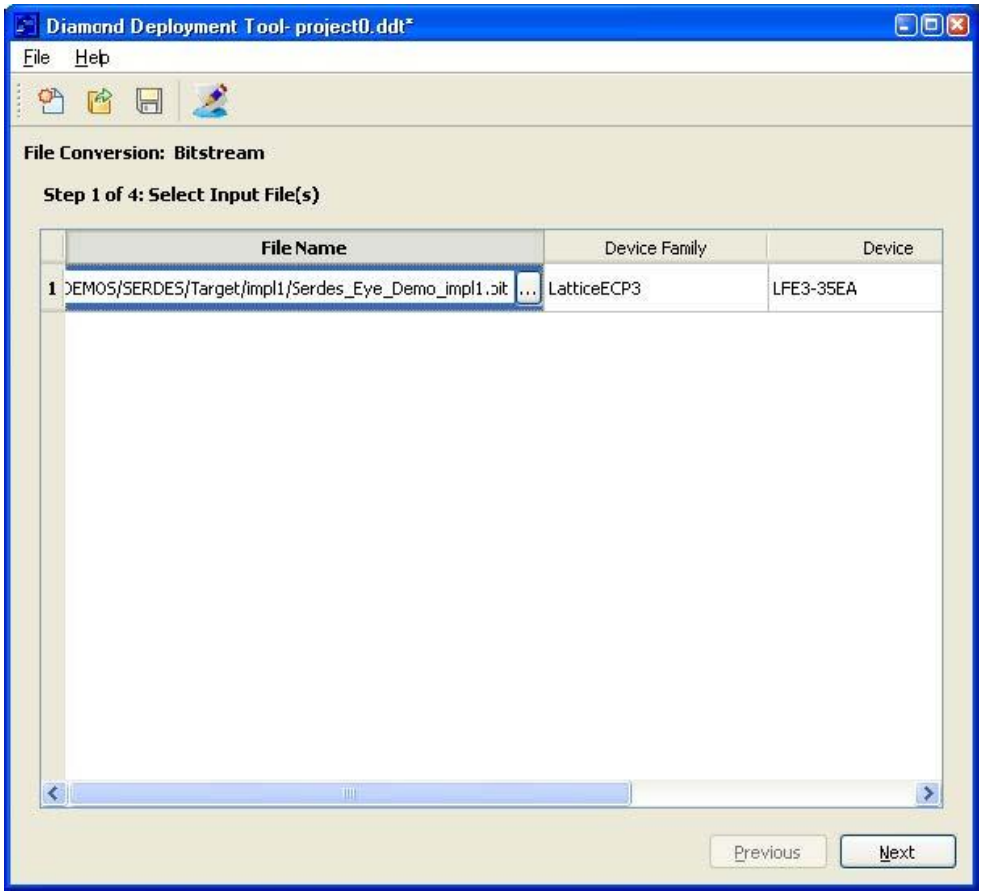

Figure 5.2. File Conversion (Step 1)

7. Select **Next** to go to the step 2 window. Select the **Output Format**.

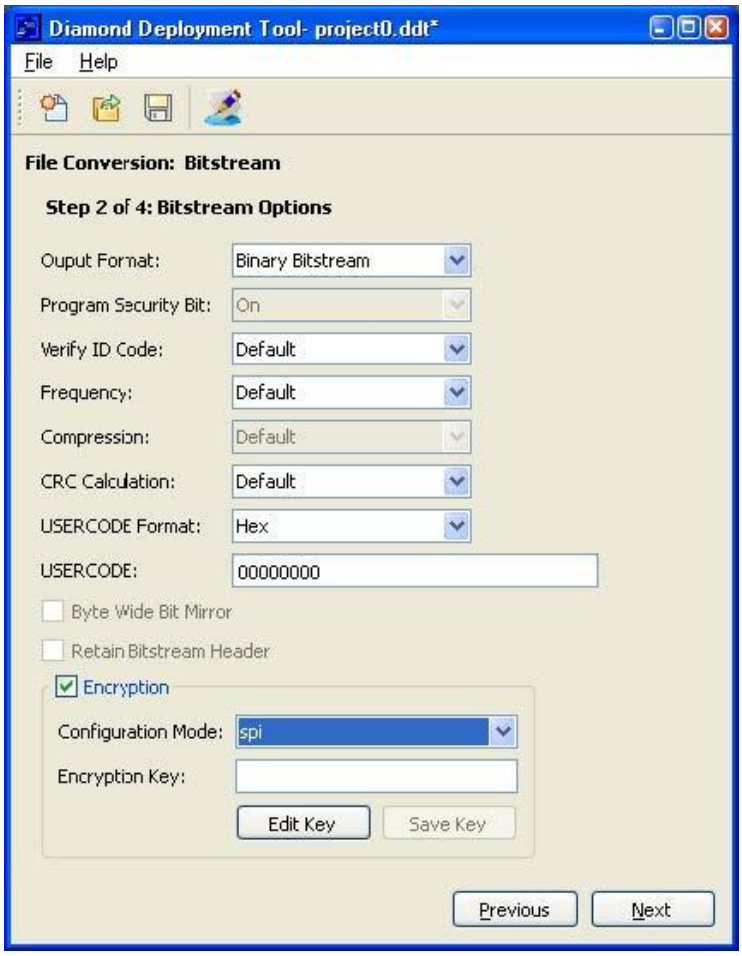

Figure 5.3. File Conversion (Step 2)

8. To burn the security fuses to disable the ability to read back the bitstream from the SRAM in the FPGA, set the Program Security Bit to **On**.
9. To store device data such as firmware version number, manufacturer ID, programming date, programmer make, pattern code, etc., set a USERCODE. Select the **USERCODE Format** and then enter a **USERCODE**:
   - **ASCII** – Alphanumeric values, using up to 16 characters
   - **Hex** – Values of 0 through F, using up to 32 characters
   - **Binary** – Values of 0 and 1, using up to 128 characters
10. To create an encrypted bitstream from an unencrypted bitstream, select the **Encryption** option box. Select the **Encryption Configuration Mode**. This step is necessary to pad the correct number of program mode dependent filler bits when encrypting the bitstream. The following lists the choices of modes and filler bits:
    - **slave_scm** – 64 bits.
    - **slave_pcm** – 64 bytes or 512 bits.
    - **jtag** – jtag burst (also known as JTAG fast program) – 64 bits.
    - **spi** – 0 bits.
    - **spim** – 0 bits. This mode is for setting up the dual-boot feature. The encrypted bitstream must be located only in one pattern, either primary or golden. Selecting the SPI or SPIm mode yields the same bitstream. Both modes require no filler bits. The dual-boot mode is not available in LatticeECP2MS and LatticeECP2S devices.
    - **slave_spi** – 0 bits.

Since the ECP5 and ECP5-5G devices do not have different padding bits for different modes, this option is no longer valid nor necessary. The following lists other file conversion options to choose from:

- **Verify ID Code** – For bitstream debugging, inserts the verify device 32-bit JTAG IDCODE frame into the bitstream. The setting Default means do not override the bitstream. ON (insert) or OFF (omit) overrides the current setting in the bitstream. It is recommended to leave this as default.
- **Frequency** – Used to adjust the master clock configuration frequency in the bitstream for the two master modes, SPI and SPIm mode. The setting has no effect on the Slave modes. The setting of Default means keep the bitstream setting. Selections other than Default overrides the current setting in the bitstream.
- **Compression** – Used to compress the bitstream. Default means do not change the bitstream. ON (compress) or OFF (no compress) overrides the bitstream. It is recommended to not use compression with encryption. Compression not available on LatticeECP3 devices.
- **CRC Calculation** – Disables the frame-by-frame CRC for bitstream debugging. It is recommended to keep the default of the bitstream for maximum configuration reliability.

11. To enter the encryption key, select the **Edit Key** button. The **Encryption Key Setup** dialog box appears:

Figure 5.4. Encryption Key Setup

12. Select the **Encryption Key Format**. The choices are **ASCII** or **Hexadecimal**.
13. Enter the Encryption Key. Select/deselect the **Hide Encryption Key** checkbox to turn off/on the visibility of the encryption key. If you enter less than the full Encryption Key code, the fillers are padded on the left (most significant) position. The user interface blocks the entry of more than the maximum number of characters and truncates the overflow. This step is necessary to protect the .bek file with a password. You can also load an existing Encryption Key Format File by selecting the **Load From File** button and then browse to select the .bek file. Select **OK** when your selections are complete. The Encryption Key is filled into the step 2 window.
14. Select **Save Key** and enter the Encryption Key File Name (.key file). This step is necessary because an encrypted bitstream must have some record of the Encryption Key used. Select **Save**.
15. Enter the **Encryption File Password** (up to 16 characters) into the prompt and select **OK**.
16. To view what is being typed, un-check the **Hide Password** checkbox. The password is restricted to ASCII characters only. If fewer than 16 characters are entered, the fillers are padded on the left position. The user interface blocks the entry of more than the maximum number of characters. This step is necessary to protect the .bek file with a password. Select **OK**. Select **Next** to continue to the step 3 window.
17. Select or enter an Output file name for the encrypted bitstream and select **Next** to proceed to step 4. A Deployment Tool Summary appears. Select the **Generate** button to create the encrypted bitstream. An example of the summary and messages is shown in the following figure.

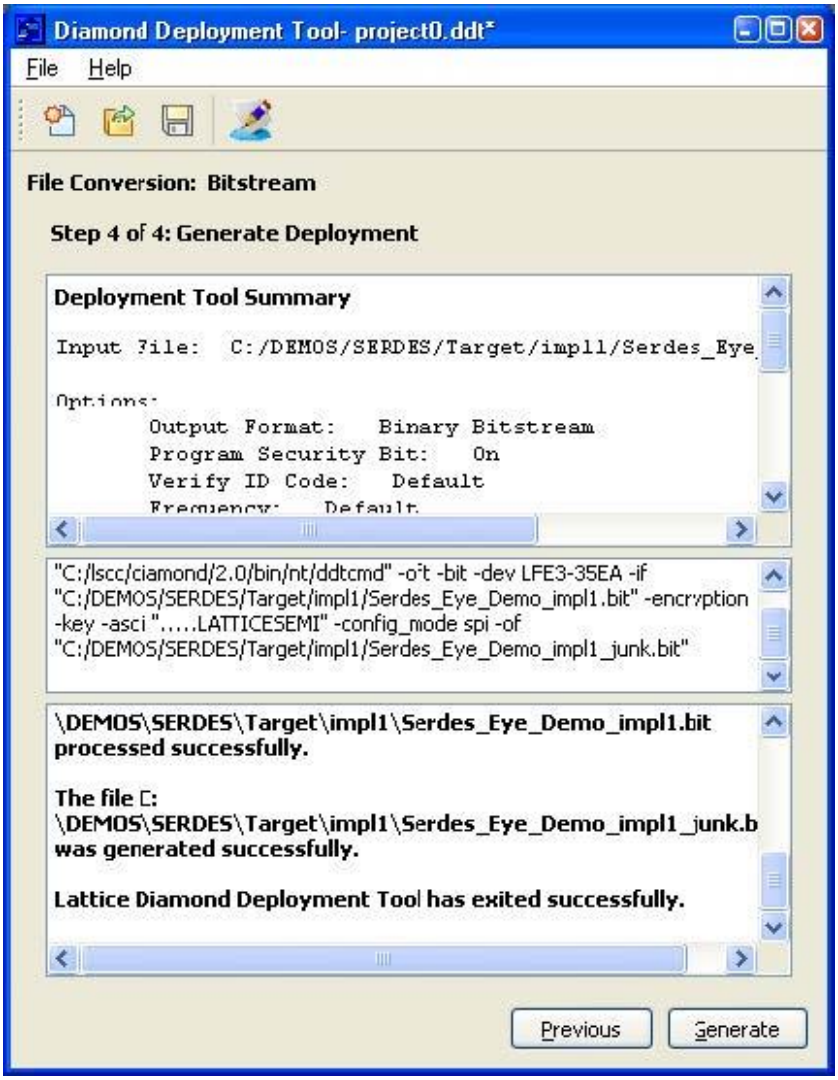

Figure 5.5. File Conversion (Step 4)

You are now ready to download the encrypted bitstream using the Diamond Programmer software.

5.2. Encryption Key Programming

To program the key code onto the OTP fuses in the FPGA, use the Diamond Programmer software. The encryption control pack is required to enable the programming of encryption keys and bitstreams. Only the encryption-enabled version of the Diamond software can save the entered key code into the key (.bek) file. If the encryption-enabled version is not available, then the key code is written to the .xcf file. It is strongly advised to protect the .xcf file by using PKZIP to encrypt it after use.

Because the Programmer is used primarily in the board design prototype development phase, choosing to program the key lock fuse while programming the encryption key code is left to you. The purpose of the key lock fuse is to disable the reading of the fuse state of all the OTP fuses, which provides the key code as the first line of defense.

Even if the key lock fuse is not programmed, verifying the key code (that is, comparing for an exact match) is not possible due to the hashing. Therefore, the only method to verify that the key code is correct is to configure the device with an encrypted bitstream.

Note: Before programming the encryption key, be sure the board with the Lattice FPGA is properly connected to your computer and turned on.

To program the encryption key in the Diamond Programmer software, follow these steps:

1. Open the Diamond Programmer software. The Diamond Programmer software can be invoked in several ways:
   - In the **Lattice Diamond** window, select **Tools > Programmer** or select the **Programmer** icon from the **Diamond** toolbar.

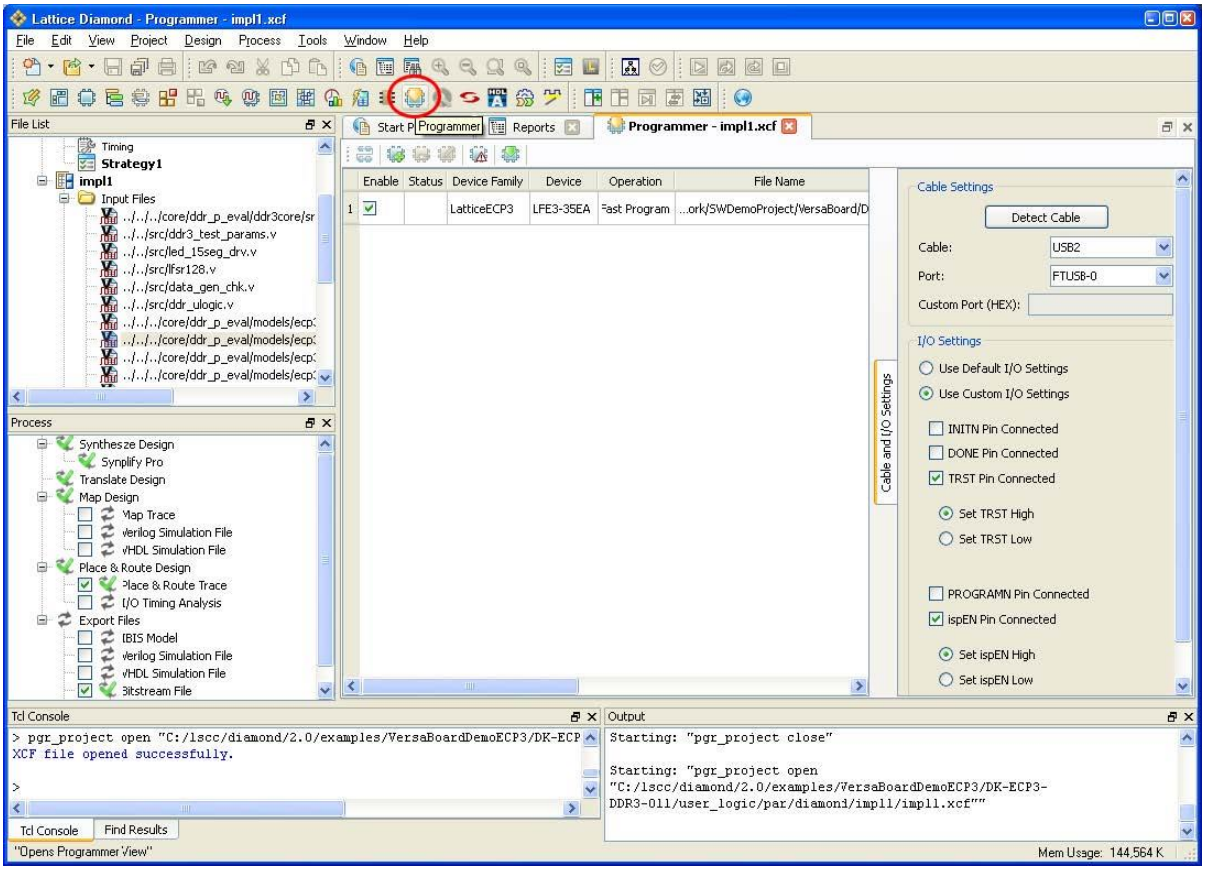

Figure 5.6. Programmer Interface

   - In Windows, go to the Windows Start menu and choose **Programs > Lattice Diamond Programmer > Programmer**.
   - In Linux, from the **<install_path>/bin/lin** directory, enter the following on a command line:

```
<Install_path>/.programmer
```

2. In the **Programmer** window, double-click in the **Operation** column for the FPGA device you want to program. The **Device Properties** dialog box appears:

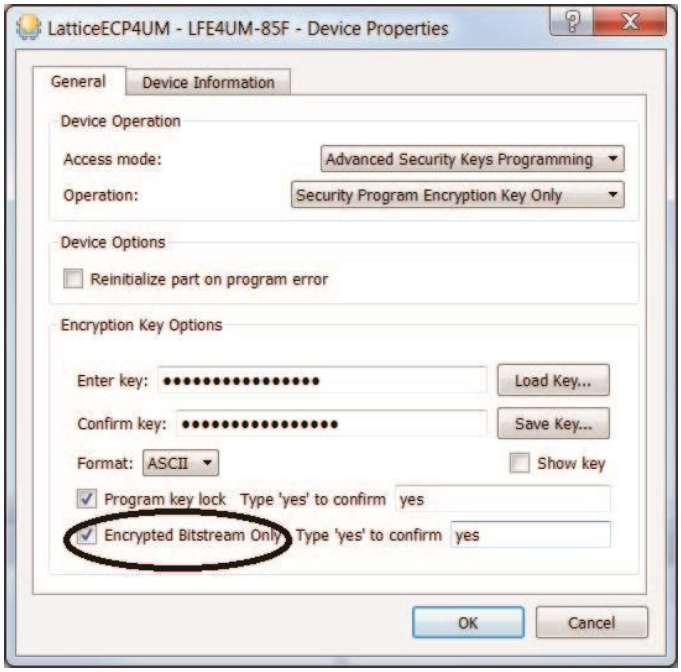

Figure 5.7. Device Properties Dialog Box

Note: The **Encrypted Bitstream Only** is for ECP5 and ECP5-5G devices only.

3. Select **Advanced Security Keys Programming** as **Access Mode**.
4. Select **Security Program Encryption Key** in **Operation**.

   Other options include:
   - **Security Program Encryption Key** – Verify the programmed encryption key and program it into the device. This operation carries out the entire encryption key fuse programming. Whether the key lock fuse is programmed at the same time depends on the setting for the key lock fuse. Since fuses are OTP, each device is allowed to go through this procedure once. Pass or fail, there is no second attempt. Programmer blocks any subsequent programming operations.
   - **Security Verify Encryption Key**
   - **Security Read Encryption Key** – Read back the encryption key from the device. Valid only when the key lock is not programmed. If the lock fuse is not programmed, unprogrammed fuses read as 0s and programmed fuses read as 1s.
   - If the lock fuse is programmed, all the key fuses read back as 1s, for ECP5 and ECP5-5G devices, or as 0s, for ECP3 and ECP2/MS devices. Programmer checks for an all-1 fuse state, for ECP5 and ECP5-5G devices, or an all-0 fuse state, for ECP3 and ECP2/MS devices, from keys 1, 2, and 3 to determine if the lock fuse is programmed.
   - **Security Program Key Lock** – Program the key lock only. This is valid only when the device encryption key is programmed but the key lock is not programmed. This operation is provided so that the key lock can be programmed separately after the encryption key fuse is programmed. This operation can only be done once. Programmer blocks all second attempts when an all-1 fuse state, for ECP5 and ECP5-5G devices, or an all-0 fuse state, for ECP3 and ECP2/MS devices, is read from keys 1, 2, and 3.
5. Select the **Encryption Key Format**, enter the encryption key and confirm the encryption key or select the **Load Key** button to load the encryption key from an existing .bek file. Selecting **Show Key** controls whether the encryption key is visible.
6. Select the **Save Key** button to save the encryption key as a .bek file. Enter the encryption key password and select **OK**. This option is available only if Encryption patch is installed.

5.3. Encrypted Bitstream JTAG Programming Procedures

When performing manufacturing (production) encryption key programming, verifying the key code using the encrypted bitstream and programming the key lock fuse both become mandatory.

The encrypted bitstream can verify the key code because if the key code programmed into the key fuses exactly matches the one that was used to encrypt the bitstream, then the encrypted bitstream configures the device and both INITN and DONE go high. Otherwise, both remain low. The Programmer only needs to check the status register for the DONE bit state = 1 to flag to determine a pass.

The encryption key production programming flow is as follows:

1. Launch the **Device Properties** menu by double-clicking in the **Operation** field in the Programmer window.
2. Select **Advanced Security Production Programming** under **Device Access Options**.
3. Select **Security Fast Program with Encryption Operation** to select the flow as follows:
   a. Program the encryption key.
   b. Use the encrypted bitstream to verify the encryption key is OK.
   c. Program the key lock fuse if selected (strongly recommended).
   d. Program the Encryption only fuse if selected.

   The Key lock fuse is purposely the last to provide some debugging capability by reading out the fuse state in the event that failure happens at step a or b.
4. Select the encrypted bitstream so that the encryption key can be verified correctly. If the encrypted bitstream is not available, use the Programmer to encrypt the regular sample bitstreams first.
   Note: Select JTAG configuration mode when encrypting the bitstream. If the mode is not JTAG, a prompt guides you to convert the file.
5. Launch the Security key entry menu.
6. Select the encryption key format under **Encryption Key Format**. The choices are ASCII or Hexadecimal. You can load the encryption key from an existing .bek file by selecting the **Load From File** button.
7. Enter the encryption key and re-enter to confirm.
8. Check the **Programming Key Lock** box. For production, the key lock fuse is strongly recommended to be programmed at the same time. If it is not, then the key lock fuse should be programmed afterwards.
9. Check the **Encryption Bitstream** Only box for extra security. This option is for ECP5 and ECP5-5G devices only. For LatticeECP2/MS and LatticeECP3, whether or not the encryption key is programmed into the device, the device can always be programmed with non-encrypted bitstream.
10. Select **Apply** and **OK**.
11. Select the **Program** button in the Programmer toolbar.

5.4. Advanced Encryption Key Programming Flow

The Diamond Programmer software supports a command line option for encryption fuse programming, and also supports passing the key code through the command line. The Deployment Tool can also accept the key code through the command line to encrypt an unencrypted bitstream file.

Programmer and Deployment Tool are designed to implement customized programming flows. Customized programming flows are usually defined by using a script file. The script file can be launched by your user interface program or ATE.

The .xcf file is still needed to launch the Programmer.exe command line. Use the Programmer user interface to generate the .xcf file first. The .xcf file contains the following information required by Programmer to run:

- Device name
- Operation
- Encrypted bitstream file name
- Encryption key code

The following examples illustrate the key serialization:

1. Launch the Deployment Tool to encrypt the bitstream with the first key code. Select **Slave SCM Mode for JTAG**.
2. Launch the Deployment Tool with the .xcf file already generated to program the key code into the FPGA device and verify the correct key code using the encrypted bitstream obtained in step 1.

   The -key switch lets the encryption key that follows replace the one in the .xcf file.
3. Repeat steps 1 and 2 after incrementing the key code.

If you wish to program the encrypted bitstream into the SPI Flash device as well, follow this procedure:

1. Launch the Deployment Tool to encrypt the bitstream to Key Code 1. Select **Slave SCM mode for JTAG**.
2. Launch the Deployment Tool to encrypt the bitstream to Key Code 1. Select **SPI mode for SPI Flash**.
3. Launch the Deployment Tool with the encryption key programming .xcf file to program the key code into the FPGA device and verify the correct key code using the encrypted bitstream obtained in step 1.
4. Launch the Deployment Tool with the SPI Flash programming .xcf file to program the SPI Flash device using the encrypted bitstream obtained in step 2.
5. Repeat steps 1 and 2 after the key code has been incremented.

Appendix A. ispVM Applications

A.1. Setting Security and Encryption for FPGA Devices

When encryption is enabled, ispVM System software can also create a key file and encrypt a Lattice bitstream file. The most important difference between ispLEVER or the Diamond software and ispVM System is that ispLEVER or the Diamond software can create a bitstream file without an unencrypted bitstream. ispVM System can only encrypt an existing unencrypted bitstream. Therefore, when using ispVM System, an unencrypted bitstream exists. Thus, using ispVM System is less secure than using ispLEVER or the Diamond software. However, if ispVM System is also installed in the Trusted Area, then the security is the same. If ispVM System is used in the open (Untrusted) area, then it is important to keep the unencrypted version of bitstream in security control, such as using PKZIP and a password to protect it.

Figure A. illustrates the procedures to convert the unencrypted (regular) bitstream to an encrypted bitstream. Figure A.2 illustrates the differences between a regular bitstream and an encrypted bitstream.

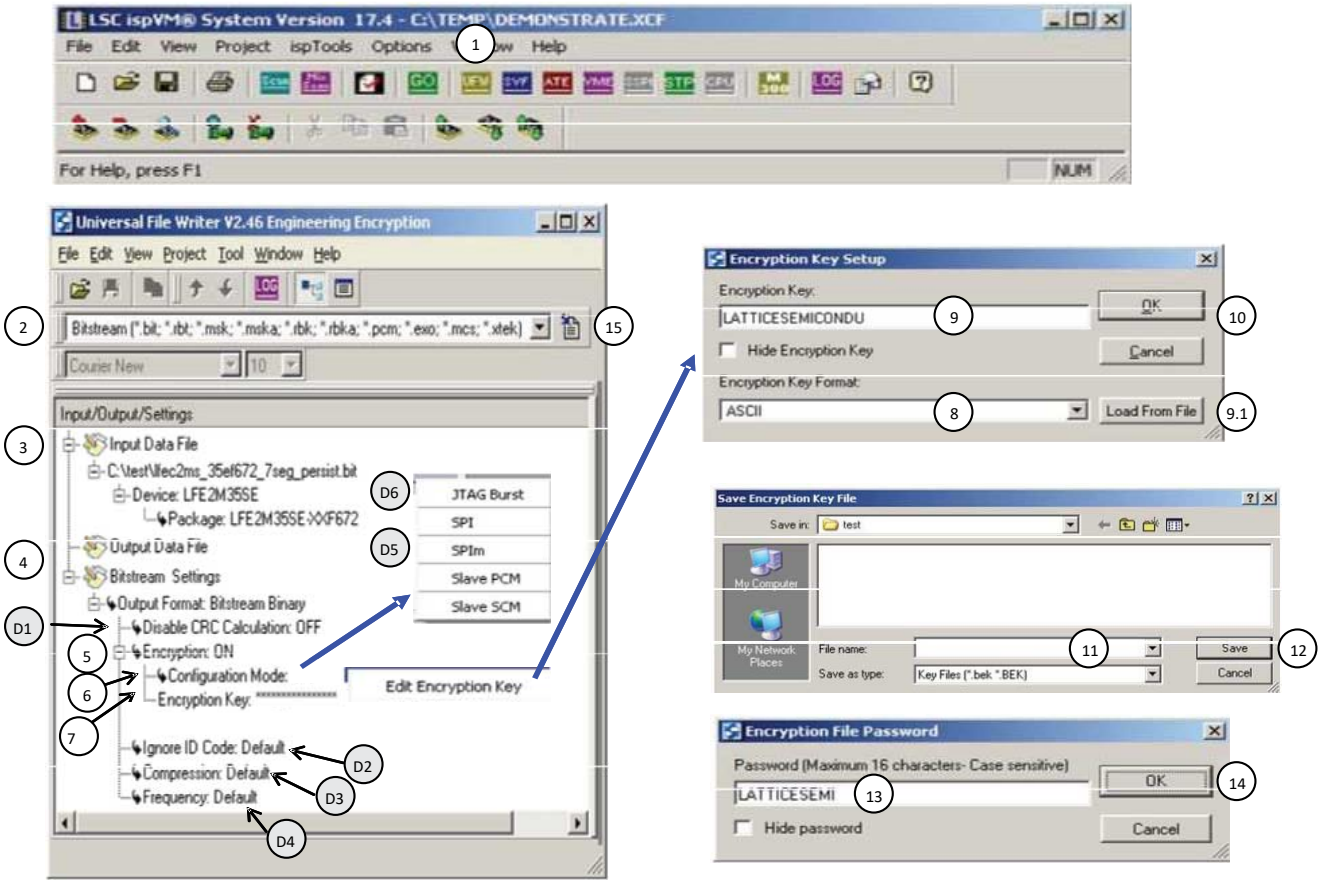

Figure A.1. Procedure to Encrypt a Regular Bitstream

Table A.1. Encrypting a Regular Bitstream

Step	Description
1	Launch the UFW in ispVM.
2	Select **Bitstream File** as the output format.
3	Browse to the input bitstream file. The device name is extracted from the header of the bitstream file automatically. If the ispVM version is encryption-enabled and the device name is a LatticeECP2S, LatticeECP2MS or LatticeECP3 device, the Encryption option shows up automatically for step 5. Otherwise, it does not show up.
4	Browse or enter the output file name.
5	Right-click on the **Encryption** option and set the Encryption to **ON**. **Note:** This option appears only on the encryption version and this option appears only for LatticeECP2S, LatticeECP2MS or LatticeECP3 device families.
6	Right-click on the **Configuration Mode** option and set the **Encryption Configuration Mode**. This step is necessary to pad the correct number of program mode dependent filler bits when encrypting the bitstream. The modes and the filler bits include: JTAG Burst (also known as JTAG Fast Program) = 64 bits SPI = 0 bits SPIm (not recommended, see note D5) = 0 bits Slave PCM = 64 bytes or 512 bits Slave SCM = 64 bits
7	Right-click on the **Encryption Key** option and click **Edit Encryption Key** to launch the Encryption Key setup dialog.
8	Select the Encryption Key format under **Encryption Key Format**. If you want to use the Hexadecimal format, select **Hexadecimal**.
9	Enter the Encryption Key. To view what is being typed, un-check **Hide Encryption Key**. If you enter less than the full Key Code, the fillers are padded on the left (the most significant) position. The user interface blocks the entry of more than the maximum number. The command line truncates the overflow.
9.1	This step is optional. It loads the Encryption key from an existing .bek file by clicking the **Load From File** button.
10	Click the **OK** button. The prompt to save the Encryption file into a .bek file is shown.
11	Enter the .bek file name. This step is mandatory since an encrypted bitstream must have some record of the Encryption Key used.
12	Click the **Save** button and the prompt to enter the password appears.
13	Enter the password. To view what is being typed, un-check **Hide Password**. The password is restricted to ASCII characters only. If fewer than 16 characters are entered, the fillers are padded on the left position. The user interface blocks the entry of more than the maximum number of characters. This step is necessary to protect the .bek file with a password.
14	Click the **OK** button to write the .bek file.
15	Click the **File Generation** button to generate the encrypted bitstream.
Note	**Comments**
D1	This option disables the frame-by-frame CRC for bitstream debugging. Users are not recommended to change the setting to ON. The default setting, OFF, provides the maximum configuration reliability.
D2	This option is provided to help bitstream debugging by inserting the verify device 32-bit JTAG IDCODE frame into the bitstream. The setting default means do not over-ride the bitstream. If change setting to ON (insert) or OFF (omit), then over-ride what is already in the bitstream. It is not recommended to change the default setting.
D3	This option is provided to compress the bitstream. The setting default means do not change the bitstream. If change setting to ON (compress) or OFF (no compress), then over-ride what is already in the bitstream. It is not recommended to deploy compression with encryption. **Note:** Compression is not available for LatticeECP3 devices.
D4	This option is provided to adjust the master clock configuration frequency in the bitstream for the two master modes, SPI and SPIm mode. The setting has no effect on the Slave modes. The setting default means keep the bitstream setting. Right clicking on the item shows the selection table. Selection other than the default overrides the current setting in the bitstream.
D5	SPIm mode is for setting up the dual-boot feature. The encrypted bitstream must be located only in one pattern, either primary or golden. Therefore, this mode is not recommended. As far as bitstream format is concerned, selecting SPI or SPIm mode yields the same bitstream. They both require no filler bits. **Note:** The LatticeECP2/M and LatticeECP2MS devices do not support dual boot with encryption.
D6	JTAG Burst is another name for the Fast Program operation in ispVM. The bitstream format requires the same number of

Step	Description
	filler bits as the Slave SCM (Serial Configuration Mode). If using the ispVM Fast Program operation, ispVM accepts encrypted bitstreams for SPI and Slave SCM modes and prompts you to convert if necessary.

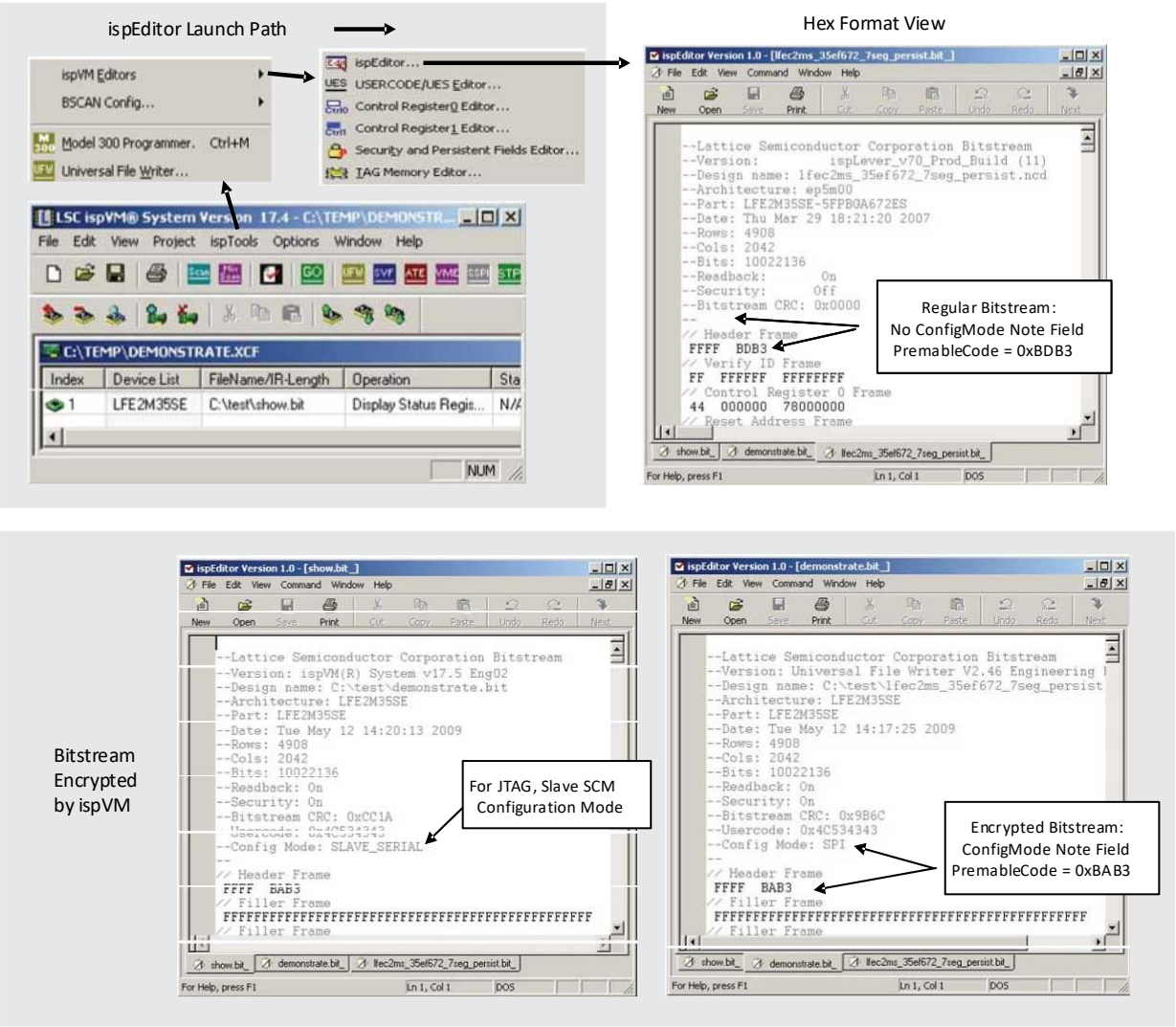

Figure A.2. Configuration Modes and Format of Encrypted Bitstream

A.2. Programming Encryption Key

Figure A.3 illustrates the procedure for using ispVM System to program the key code onto the OTP fuses in the FPGA. The version of ispVM System shown in Figure A.3 is the version with encryption enabled. Only the Encryption enabled version can save the entered key code into the key (.bek) file. If the encryption-enabled version is not available, then the **Save to File** button is grayed out (disabled) and the key code is written into the .xcf file. Users are strongly advised to protect the .xcf file by controlling it by using PKZIP to encrypt it after use.

Because the ispVM System is used primarily in the board design prototype development phase, choosing to program the key lock fuse while programming the encryption key code is left to you. The purpose of the key lock fuse is to disable the reading of the fuse state of all the OTP fuses, which provides the key code the first line of defense.

Even if the key lock fuse is not programmed, verifying the key code (that is, comparing for an exact match) is not possible due to the hashing. Therefore, users are not provided with a Verify Security Key operation in ispVM. The only method to verify that the key code is correct is to configure the device with an encrypted bitstream, as illustrated in Figure A.4.

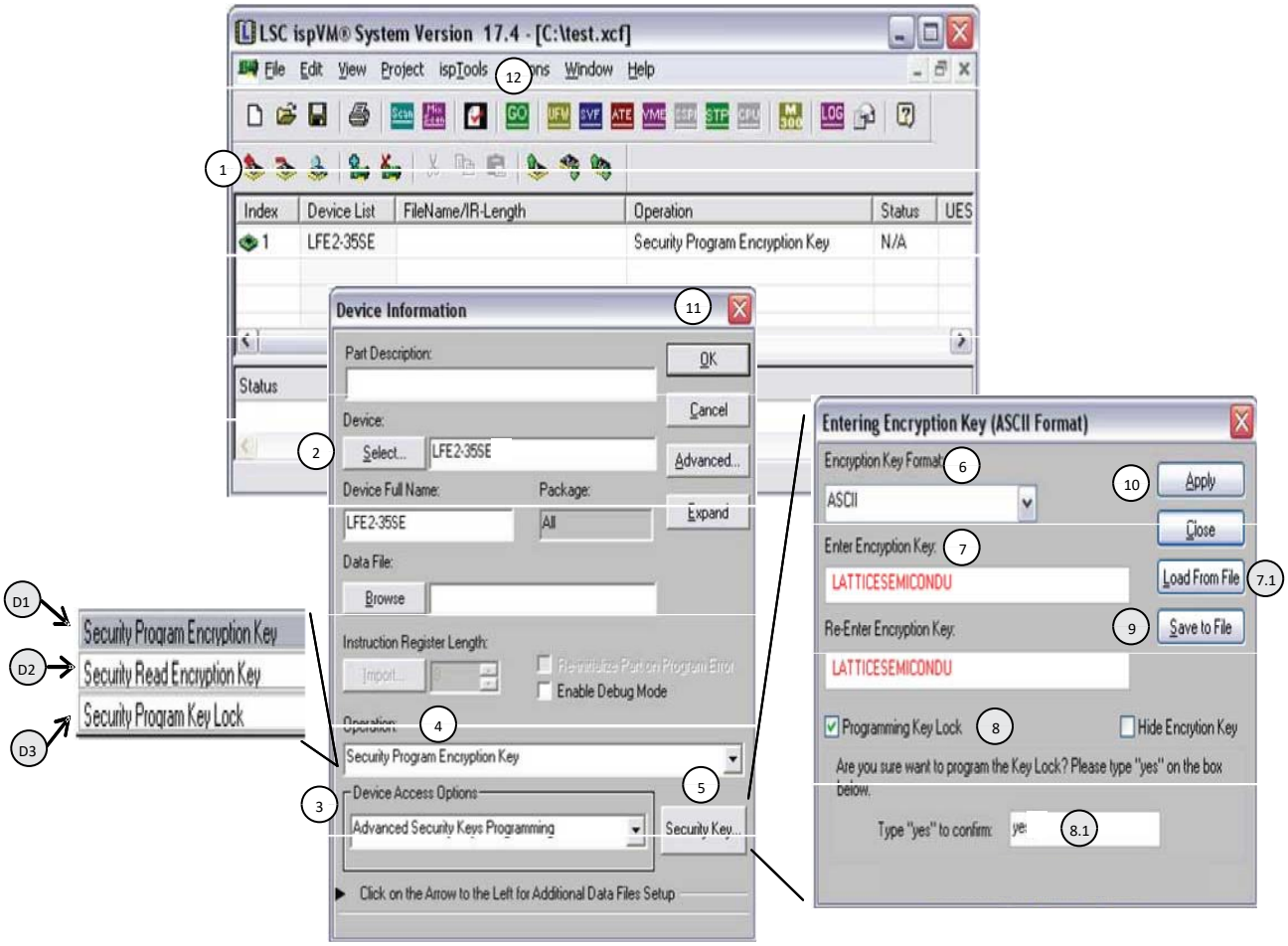

Figure A.3. Procedure for Programming the Encryption Key Only

Table A.2. Programming the Encryption Key Only

Step	Description	
1	Launch the **Device Selection** menu.	
2	Select the FPGA device.	
3	Select **Advanced Security Keys Programming** under **Device Access Options**.	
4	Select **Security Program Encryption Key** under **Operation**.	
	Operation	**Comment**
	Security Program Encryption	Set up the Encryption Key and program it into the device.
	Security Read Encryption Key	Read back the encryption key from the device. Valid only when the key lock is not programmed.
	Security Program Key Lock	Program the key lock only. Valid only when the device encryption key is programmed but the key lock is not programmed.
5	Launch the encryption key dialog by clicking on the **Security Key** button.	
6	Select the encryption key format under **Encryption Key Format**. The default is ASCII with an option for Hexadecimal.	
7	Enter the encryption key.	
	Re-Enter the encryption key.	
7.1	This step is optional. Load the encryption key from an existing .bek file by clicking the **Load From File** button.	
8	This step is optional. Select the **Programming Key Lock** option if you wish to lock the device.	
8.1	Type **Yes** to confirm locking the device.	
9	This step is optional. Save the encryption key as a .bek file by clicking the **Save To File** button.	
10	Click **Apply** to come back to the Device Information Dialog.	
11	Click **OK** to return to the ispVM windows. ispVM generates a chain configuration file, which can be saved as an .xcf file. It contains the encryption key.	
12	Select the **Go** button to program the Encryption Key into the device.	
Notes	**Comments**	
D1	This operation carries out the entire encryption key fuse programming flow shown in Figure A.2. Whether the key lock fuse is programmed at the same time depends on the setting for the key lock fuse. Since the fuses are OTP, each device is allowed to go through this procedure once. Pass or fail, there is no second attempt. ispVM blocks any subsequent programming operations.	
D2	Only the Read operation is provided. The Verify (compare for exact match) operation is not possible due to the deployment of hashing and each key code bit is formed by the three-input (fuses) OR gate. If the lock fuse is not programmed, un-programmed fuses read as 0s and programmed fuses read as 1s. If the lock fuse is programmed, all the key fuses will read back as all 1s, for ECP5 and ECP5-5G devices, or as all 0s, for ECP3 and ECP2/MS devices. ispVM checks for an all-1 fuse state, for ECP5 and ECP5-5G devices, or an all-0 fuse state, for ECP3 and ECP2/MS devices, from keys 1, 2, and 3, and the PES rows to determine if the lock fuse is programmed. **Note:** The PES (Private Electronics Signature) row is for Lattice factory use only.	
D3	This operation is provided so that the key lock can be programmed separately after the encryption key fuse is programmed. As discussed in Note D1, this operation can only be carried out once. ispVM blocks all second attempts when an all-1 fuse state, for ECP5 and ECP5-5G devices, or an all-0 fuse state, for ECP3 and ECP2/MS devices, is read from keys 1, 2, and 3 and the PES rows. **Note:** The PES (Private Electronics Signature) row is for Lattice factory use only.	

A.3. Encrypted Bitstream JTAG Programming

The way ispVM System supports JTAG programming makes the differences between an encrypted bitstream and an unencrypted (regular) bitstream transparent to you.

The major difference between an encrypted and a regular bitstream is the configuration mode dependency of the encrypted bitstream. Regular bitstream format is mode-independent. Therefore, regular bitstreams do not have the configuration mode note field on the bitstream header, whereas the encrypted bitstreams do.

The ispVM System uses the note field of the bitstream header to ensure the bitstream is encrypted and the configuration mode is selected, and prompts you accordingly. If the configuration mode selected is SPI or SPIm, then you are prompted to convert the configuration mode and save it as a temporary file. If the configuration mode selected is Slave SCM, no conversion is necessary.

Note: When prompted to convert an encrypted bitstream for SPI or SPIm configuration modes, do not overwrite the original file. Instead, save the changed file to a different file. If the encrypted bitstream has the configuration mode changed to Slave SCM then programmed into the SPI Flash, the Lattice devices fail configuration. Only the encrypted bitstream for SPI or SPIm configuration mode can be programmed into the SPI Flash devices.

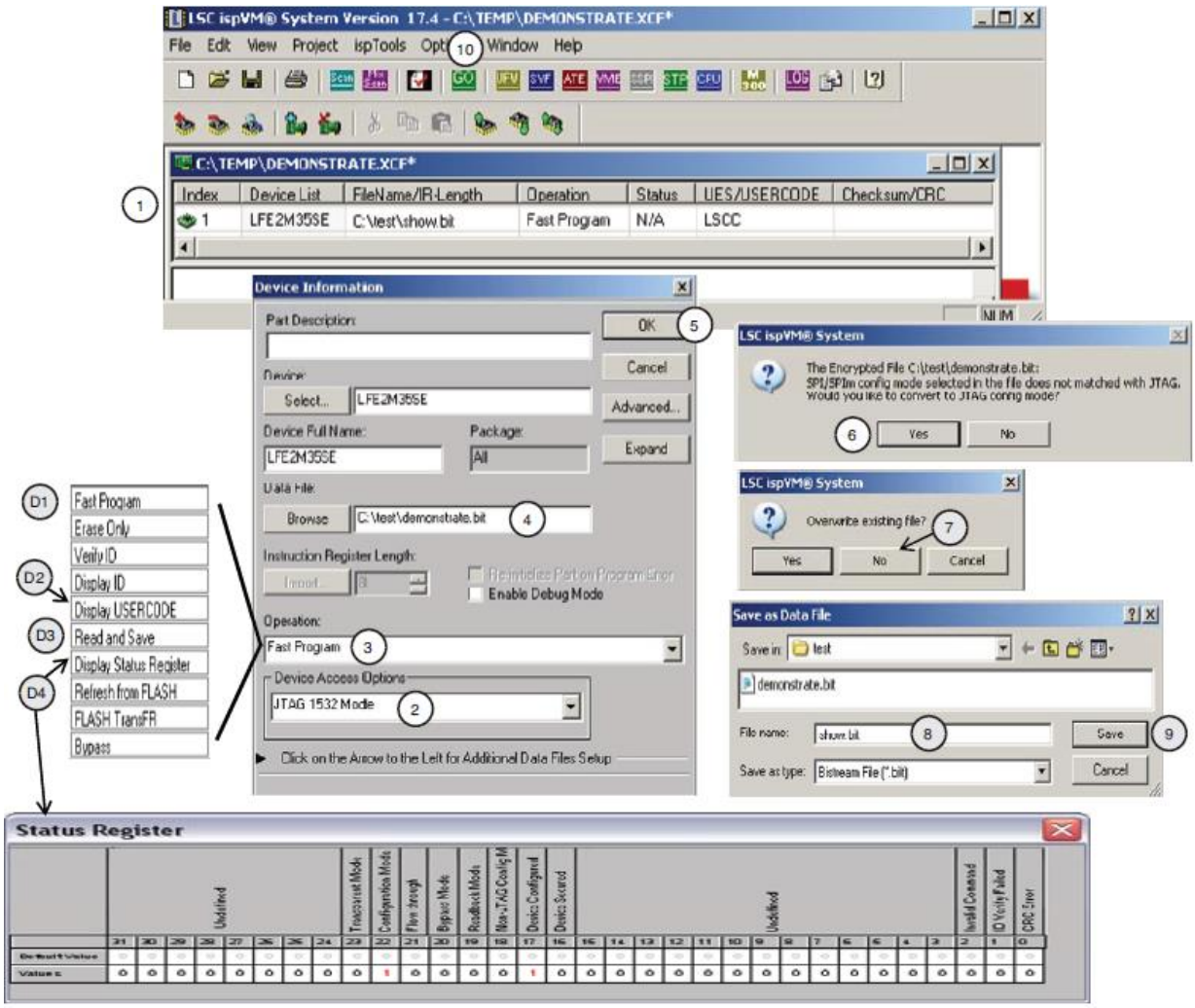

Figure A.4. Configuring a Device with an Encrypted Bitstream

Table A.3. Configuring a Device with an Encrypted Bitstream

Step	Description			
1	Scan or select the LatticeECP3, LatticeECP2MS and LatticeECP2S device.			
2	Select **JTAG 1532** under **Device Access Options**.			
3	Browse for the encrypted bitstream. **Note:** The software checks the bitstream configuration mode and asks to convert to the JTAG burst mode if the mode is different.			
4	Select the operation from the **Operation** list.			
	Operation	**Comments**		
	Fast Program	Uses the JTAG burst instruction to program the device. See Note D1.		
	Erase Only	Erase the device.		
	Verify ID	Read and compare the device's ID.		
	Display ID	Read and display the device's ID.		
	Display USERCODE	Read and display the device's USERCODE. See Note D2.		
	Read and Save	Read and save the device's data into a file. See Note D3.		
	Display Status Register	Read and display the device's status register. See Note D4.		
	Refresh from FLASH	Boot the data from an external SPI Flash. All I/O are tri-state while booting.		
	FLASH TransFR	Boot the data from an external SPI Flash. All I/O are held by the BSCAN cells while booting.		
5	Click **OK** to close the device setup dialog. If the configuration mode of the encrypted bitstream selected in step 3 is not Slave SCM (not for JTAG), the prompt to convert appears and the following optional steps becomes mandatory.			
6	Mandatory when the prompt appears. Click **Yes** to convert.			
7	Mandatory when the prompt appears. Click **No** so that the converted file is written to a different file name.			
8	Mandatory when the prompt appears. Enter a different file name to keep the original file.			
9	Mandatory when the prompt appears. Click **Save** to convert then write the converted file into the file name entered above. The cursor is returned to the **Project** menu.			
10	Click **Go** to program the device with the selected bitstream.			
Note	**Comments**			
D1	Fast Program is the operation that uses the JTAG port to clock the bitstream into the FPGA to configure it. Since clocking the bitstream is a continuous operation, terms such as blast, pump and burst are used to describe it.			
D2	The 32-bit USERCODE is a standard feature available on all PLDs. It is accessible through the JTAG port. One popular use of the USERCODE is to store the CRC of the bitstream. By reading the USERCODE, the bitstream version programmed into the device can then easily be found. The default USERCODE in the bitstream is all 0s.			
D3	Read and Save a bitstream from the Lattice device is valid only when the device has already been configured successfully and the security bit is not programmed. Thus, this operation is not valid for encrypted bitstreams since the security bit is always programmed. The Read and Save operation is for debugging purposes only. It is not intended for the configuration pass/fail decision.			
D4	This operation is for debugging purposes only. The expected bit states relevant to regular users include:			
	Status Bit	**Bit Function**	**When Status Value = 0**	**When Status Value = 1**
	17	Device Configured?	No, done fuse is not programmed.	Yes, done fuse is programmed.
	16	Device Secured?	No, security fuse is not programmed.	Yes, security fuse is programmed.
	2	Invalid Command?	No, no bitstream or Okay.	Fail, bitstream has invalid command.
	1	ID Verify Fail?	No, no ID Verify or Okay.	Fail, bitstream has ID Verify and Fail.
	0	CRC Compare Fail?	No, no bitstream or Okay.	Fail, CRC fail to compare.
	ispVM checks bits 17, 2, 1 and 0 to confirm if whether the Lattice device has passed configuration in the Fast Pro- gram operation.			

A.4. Programming Manufacturing Encryption Key

When performing manufacturing (production) encryption key programming, verifying the key code using the encrypted bitstream and programming the key lock fuse both become mandatory.

The encrypted bitstream can verify the key code because if the key code programmed into the key fuses exactly matches the one that was used to encrypt the bitstream, then the encrypted bitstream configures the device and both INITN and DONE go high. Otherwise, both remain low. The ispVM System only needs to check the status register for the DONE bit state = 1 to flag to determine a pass.

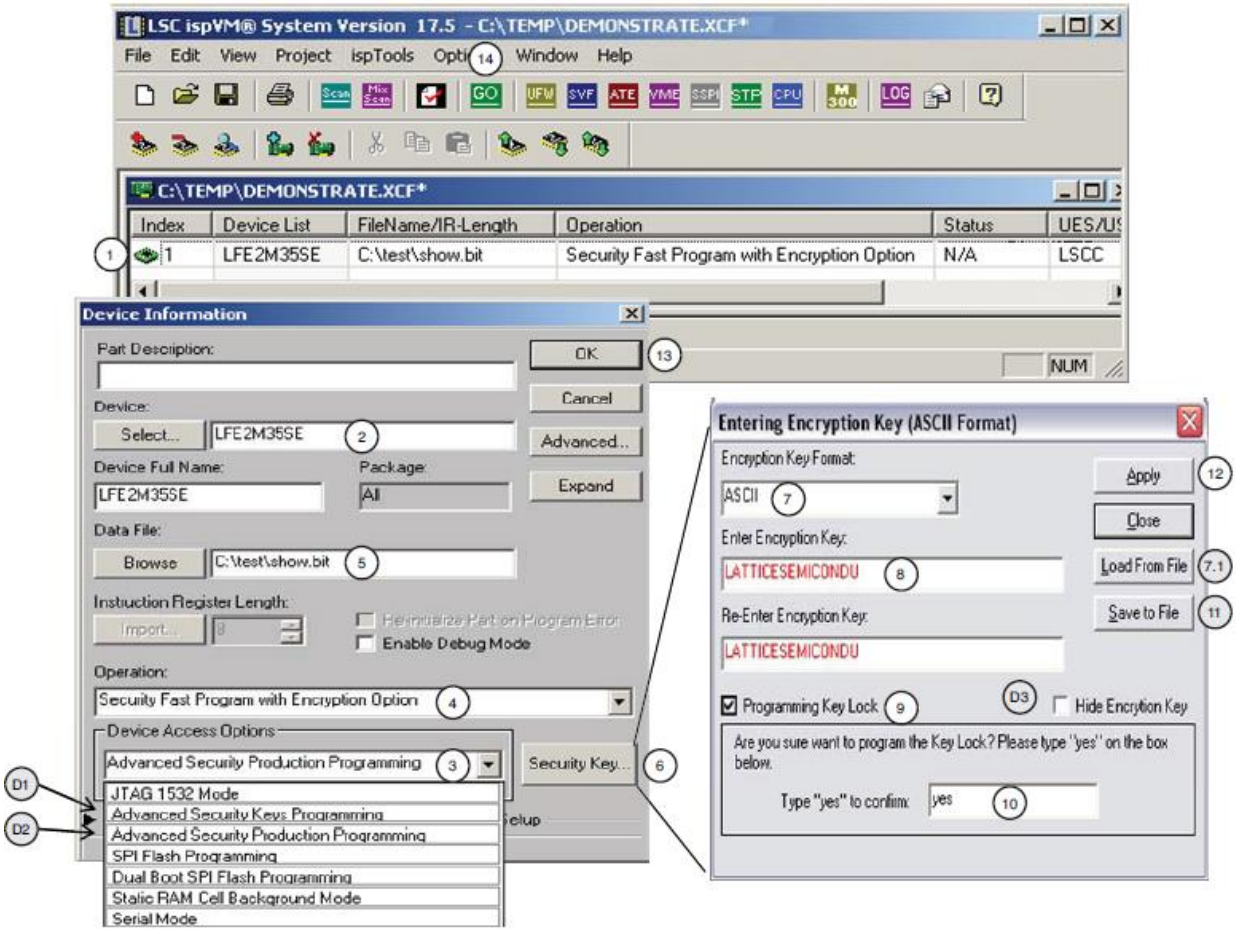

Figure A.5. Encryption Key Production Programming Flow

Table A.4. Configuring a Device with an Encrypted Bitstream

Step	Description
1	Launch the device selection menu.
2	Select the FPGA device.
3	Select **Advanced Security Production Programming** under **Device Access Options** in ispVM 17.5 or later.
4	Select **Security Fast Program With Encryption Operation** to select the flow as follows: Program the encryption key. Use the encrypted bitstream to verify the encryption key is OK. Program the key lock fuse if selected (strongly recommended). The key lock fuse is purposely the last to provide some debugging capability by reading out the fuse state in the event that failure happens at step a or b.
5	Select the encrypted bitstream so that the encryption key can be verified correctly. If the encrypted bitstream is not available, use ispVM to encrypt the regular sample bitstreams first. **Note:** Select JTAG configuration mode when encrypting the bitstream. If the mode is not JTAG, a prompt appears to guide users to convert.
6	Launch the **Security Key** entry menu.
7	Select the encryption key format under **Encryption Key Format**. The default is ASCII, with an option for Hexa-decimal.
7.1	This step is optional. Load the Encryption key from an existing .bek file by clicking the **Load From File** button.
8	Enter the encryption key.
	Re-enter the encryption key.
9	This step is mandatory. Check the box. For production, the key lock fuse is strongly recommended to be programmed at the same time. If it is not, then follow Figure A.3 to program the key lock fuse afterward.
10	Enter **Yes** to confirm programming of the key lock fuse at the same time.
11	This step is optional. Only the encryption-enabled version of ispVM System supports saving the encryption key into a .bek file. If the save to file is not selected, then the encryption key that you entered is saved into the .xcf file.
12	Select the **Apply** button to accept the entry.
13	Select **OK** to save the entry onto the .xcf file.
14	Select **Go** to start the programming action.
Notes	**Comments**
D1	This is for the group of operations for board development described in Figure A.3.
D2	This is the operation added into ispVM 17.5 or later to program the encryption key and verify with an encrypted bitstream the same time.
D3	This is a standard feature provided by all user interfaces that accept password entry. It keeps the key code private while it is being entered.

Appendix B. Using Lattice Model 300 Programmer to Program Encryption Key

The Lattice Model 300 desktop programmer supports off board (on-socket) programming. It is ideal for small volume programming. Large volume programming should be done using Lattice's approved third-party programmers. The procedure for using the Model 300 programmer is shown in Figure B.1.

The Model 300 desktop programmer has the following three components:

- USB or parallel port ispDOWNLOAD cable for PC connection
- Model 300 programmer base unit for socket adapters and for providing power
- Model 300 socket adapter for mounting packaged devices

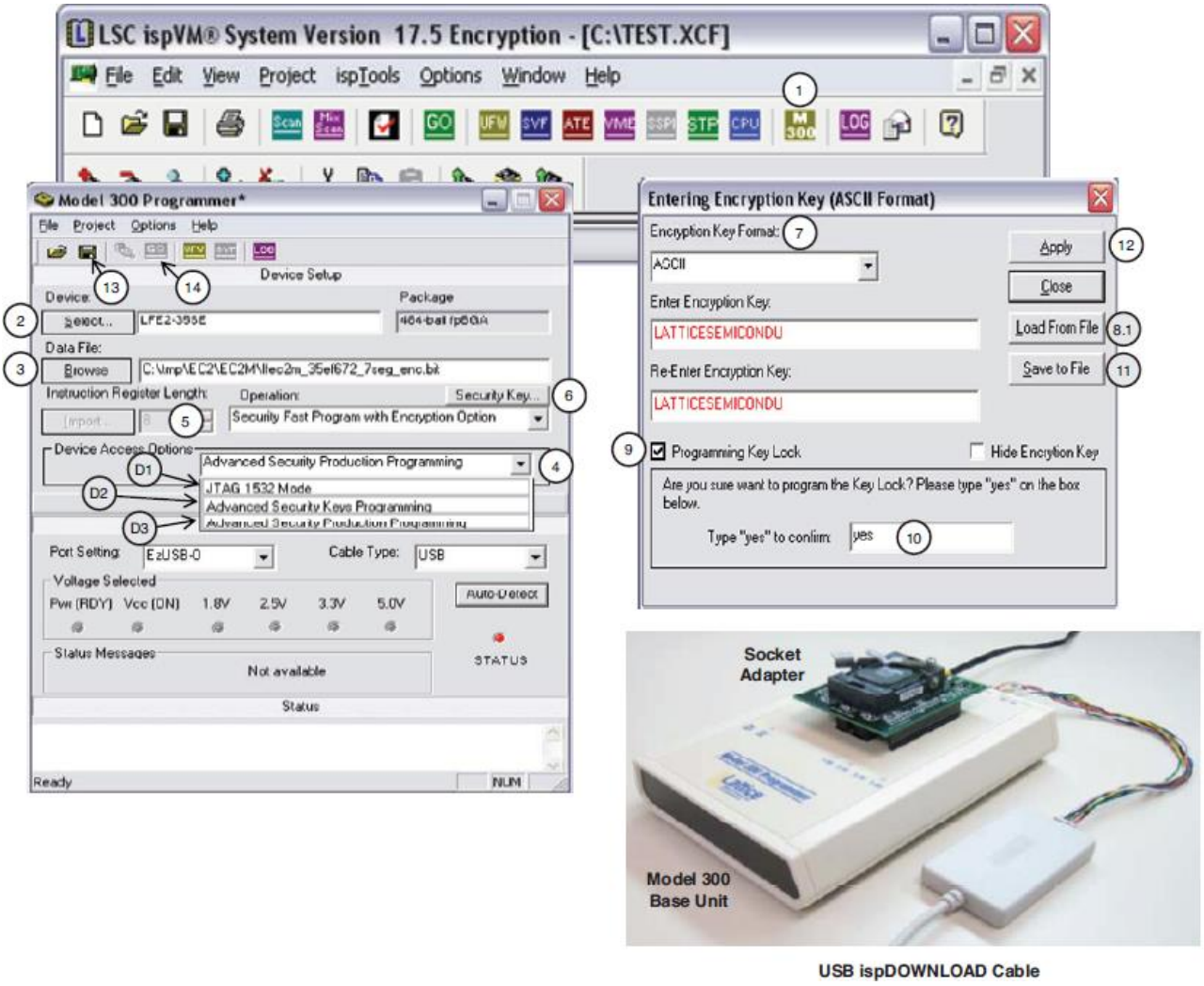

Figure B.1. Off board Encryption Key Code Programming on the Model 300 Programmer

Table B.1. Off board Encryption Key Code Programming

Step	Description
1	Launch the Model 300 using the ispVM System.
2	Select the FPGA device.
3	Browse to the encrypted bitstream. If an encrypted bitstream is not available, use ispUFW to encrypt the sample regular bitstream provided with the ispVM System software first. **Note**: The software checks the bitstream configuration mode and asks to convert to the JTAG burst mode if the mode is different.
4	Select **Advanced Security Production Programming** under **Device Access Options**.
5	Select **Security Fast Program With Encryption Operation** to select the flow as follows: a. Program the encryption key. b. Use the encrypted bitstream to verify that the encryption key is OK c. Program the key lock fuse if selected (strongly recommended). d. The key lock fuse is the last program to provide some debugging capability by reading out the fuse state in the event that a failure occurs during steps a or b.
6	Launch the Encryption Key dialog by clicking on the **Security Key** button.
7	Select the encryption key format under **Encryption Key Format**. To use the hexadecimal format, select **Hexadecimal**.
8	Enter the encryption key.
	Re-enter the encryption key.
8.1	This step is optional. Load the encryption key from an existing .bek file by clicking the **Load From File** button.
9	This step is mandatory. Select the **Programming Key Lock** option to program the key lock fuse at the same time.
10	Type **Yes** to confirm programming of the key lock fuse at the same time.
11	This step is optional. Save the encryption key to a .bek file by clicking the **Save To File** button. This feature is available only on the encryption-enabled versions of ispVM System software.
12	Click **Apply** to return to the Device Information dialog.
13	The Model 300 programmer generates a chain description .xcf file that contains the encryption key if it was not saved as a .bek file in step 11.
14	Select the **Go** button to program the encryption key and verify use of the encrypted bitstream.

Appendix C. ispVM Advanced Encryption Key Programming Flows

The ispVM System software version 17.5 or later supports a command line option for encryption fuse programming, and also supports passing the key code through the command line. The ispUFW included in ispVM System 17.5 or later can also accept the key code through the command line to encrypt an unencrypted bitstream file.

The ispVM System and ispUFW are flexible tools designed to implement customized programming flows. Customized programming flows are usually defined by using a script file. The script file can be launched by your user interface program or ATE.

The .xcf file is still needed to launch the ispVM.exe command line. Use the ispVM user interface to generate the .xcf file first. The .xcf file contains the following information required by ispVM to run:

- Device name
- Operation
- Encrypted bitstream file name
- Encryption key code

The following examples illustrate the key serialization:

1. Launch the ispUFW command line to encrypt the bitstream with the first key code. Select Slave SCM Mode for JTAG.
2. Launch the ispVM command line with the .xcf file already generated to program the key code into the FPGA device and verify the correct key code using the encrypted bitstream obtained in step 1.

   The -key switch lets the encryption key that follows replace the one in the .xcf file.
3. Repeat steps 1 and 2 after incrementing the key code.

If you wish to program the encrypted bitstream into the SPI Flash device as well, follow this procedure:

1. Launch the ispUFW command line to encrypt the bitstream to Key Code 1. Select Slave SCM mode for JTAG.
2. Launch the ispUFW command line to encrypt the bitstream to Key Code 1. Select SPI mode for SPI Flash.
3. Launch the ispVM command line with the encryption key programming .xcf file to program the key code into the FPGA device and verify the correct key code using the encrypted bitstream obtained in step 1.
4. Launch the ispVM command line with the SPI Flash programming .xcf file to program the SPI Flash device using the encrypted bitstream obtained in step 2.
5. Repeat steps 1 and 2 after the key code has been incremented.

Table C.1. Encryption Feature Command Line Syntax of ispVM and ispUFW

Program	Command Line Syntax	
ispVM	**Switch**	**Description**
	-encryption	Specifies the encryption options.
	-key	Replaces the key in the .xcf with the encryption key that follows.
	-hex:	Specifies the encryption key in hexadecimal (32 characters max.)
	-ascii:	Specifies the encryption key in ASCII (16 characters max.)
	Examples: ispVM.exe -infile c:\test.xcf -encryption -key -hex "000000004C6174746963652053656D69” ispVM.exe -infile c:\test.xcf -encryption -protect -ascii "_LATTICE”	
ispUFW	**Switch**	**Description**
	-encryption:	Specifies the encryption options.
	-key	Replaces the key in the .xcf with the encryption key that follows.
	-hex	Specifies the encryption key in hexadecimal (32 characters max.)
	-ascii	Specifies the encryption key in ASCII (16 characters max.)
	-config_mode:	Specifies the configuration mode of the encrypted bitstream. Possible values are “jtag burst”, “spi”, “spim”, “slave_scm”, and “slave_pcm”]
	Examples: ispUFW.exe -device LFE2M50SE -infile "ecp2m50e.bit" -encryption -key -hex "000000004C6174746963652053656D69" -config_mode slave_scm -oft -pcm -outfile “ecp2m50e_encryption.bit" **Notes:** • This option only available on the encryption-enabled version of ispVM. • This option only supports the LatticeECP3, LatticeECP2MS or LatticeECP2S device families.	

References

For more information, refer to the following resources:

- LatticeECP2/M sysCONFIG Usage Guide (TN1108)
- LatticeECP2/M S-Series Configuration Encryption Usage Guide (TN1109)
- LatticeECP3 sysCONFIG Usage Guide (FPGA-TN-02192)
- LatticeXP2 Advanced Security Programming Usage Guide (TN1212)
- ECP5 sysCONFIG Usage Guide (FPGA-TN-02032)

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.8, June 2024

Section	Change Summary
Disclaimer	Updated the disclaimer.
Abbreviations in This Document	Moved the terms from the Glossary section to this section.
Encrypted Bitstream JTAG Programming	Updated part of the description for the SPIm mode in step 10 of the Setting Security and Encryption Using the Deployment Tool section as follows: *The encrypted bitstream must be located only in one pattern, either primary or golden.*
Appendix A. ispVM Applications	Updated part of the description for the SPIm mode in Table A.1. Encrypting a Regular Bitstream as follows: *The encrypted bitstream must be located only in one pattern, either primary or golden.*
Technical Support Assistance	Added a reference to the Lattice Answer Database.

Revision 1.7, December 2020

Section	Change Summary
Overview of the Encryption Key Programming Algorithm	Added SSPI to Slave Modes in Table 4.2. Encrypted Bitstream Format and Configuration Mode Dependency.

Revision 1.6, November 2020

Section	Change Summary
All	• Updated document title to Advanced Security Encryption Key Programming Guide for ECP Device Family. • Changed document number from TN1215 to FPGA-TN-02202. • Updated document template. • Applied minor editorial changes.
Disclaimers	Added this section.
Acronyms in This Document	Added this section.
Overview of the Encryption Key Programming Algorithm	Updated encryption key readback for ECP5 and incorporated ECP5-5G in procedures.
Encrypted Bitstream JTAG Programming	
Appendix A. ispVM Applications	

Revision 1.5, January 2016

Section	Change Summary
Overview of the Encryption Key Programming Algorithm	Updated Setting Security and Encryption for FPGA Devices section. Corrected procedure numbering.
Appendix B. Using Lattice Model 300 Programmer to Program Encryption Key	Updated Appendix B. Using Lattice Model 300 Programmer to Program Encryption Key section. Corrected final step number in the procedure table.

Revision 1.4, October 2015

Section	Change Summary
All	• Changed document title to Advanced Security Encryption Key Programming Guide for ECP5, ECP5-5G, LatticeECP3, and LatticeECP2/MS Devices. • Added support for ECP5-5G device family. • Added support for ECP5-5G device family.
Technical Support Assistance	Updated Technical Support Assistance section.

Revision 1.3, March 2014

Section	Change Summary
All	• Changed document title to Advanced Security Encryption Key Programming Guide for ECP5, LatticeECP3, and LatticeECP2/MS Devices. • Added support for ECP5 device family.

Revision 1.2, March 2014

Section	Change Summary
All	• Changed document title to Advanced Security Encryption Key Programming Guide for ECP5, LatticeECP3, and LatticeECP2/MS Devices. • Added support for ECP5.
Introduction	Added features for ECP5 in Introduction section.
Glossary	Arranged Glossary section in alphabetical order.
Overview	Added description of Encryption Only feature for ECP5 in Overview section.
Overview of the Encryption Key Programming Algorithm	Updated Table 5.1 footnote 1.
Encrypted Bitstream JTAG Programming	• Added information on file conversion options in Setting Security and Encryption Using the Deployment Tool section. • Updated Diamond Programmer startup procedure in Encryption Key Programming section.
References	Added FPGA-TN-02032 to References section.
Technical Support Assistance	Updated Technical Support Assistance information.

Revision 1.1, September 2012

Section	Change Summary
All	• Updated document with new corporate logo. • Updated for Diamond Programmer/Diamond Deployment Tool usage. • Procedure to Encrypt a Regular Bitstream diagram – updated corresponding table with notes to D1 and D5.
Introduction	Expanded the Introduction section.

Revision 1.0, October 2010

Section	Change Summary
All	Initial release.

www.latticesemi.com