*Application Note*

# *Shared Key Boot Loader for TM4C129 Devices With AES Hardware Encryption*

*Bob Crosby and Ralph Jacobi*

**ABSTRACT**

This application report provides an infrastructure where the product owner can create an image for field upgrade of their product in their factory using a pair of 256-bit shared keys. One key is used to generate a signature. The other key is used to encrypt the image. The product in the field has a boot loader that shares the same two keys. It decrypts the signature and the image and programs them into the flash with one key. Then, it verifies the signature against the image with the second key. Only if the image verifies, then the boot loader transfers execution to the new image.

This example provides a mechanism to protect against accidental or unsophisticated attempts to upload a code image other than one created by the tools described in this report. It is not intended to imply that this method is sufficient to protect against more sophisticated attacks. It is the your responsibility to determine if this method is appropriate for your application.

Project collateral discussed in this document can be downloaded from the following URL: https://www.ti.com/lit/zip/spma083.

## Table of Contents

**1 Implementation**.....2
1.1 Flash Boot Loader Project.....2
1.2 Image Creation Project.....4
1.3 Key Image Project.....4
1.4 EK-TM4C129EXL Example Application Project.....4
1.5 DK-TM4C129X Example Application Project.....4
1.6 RAM-Based EEPROM Erase Project.....4
**2 Example Walk Through**.....4
2.1 Build Environment.....4
2.2 Importing the Examples into Code Composer Studio.....5
2.3 Setting Keys and Variables.....8
2.4 Running the shared_key_image_encrypt Tool.....9
2.5 Running the Shared Key Serial Boot Loader.....15
2.6 Returning to the Boot Loader.....20
**3 Summary**.....21

## List of Figures

Figure 2-1. Import CCS Projects Step 1.....5
Figure 2-2. Import CCS Projects Step 2.....6
Figure 2-3. Import CCS Projects Step 3.....7
Figure 2-4. Import CCS Projects Step 4.....8
Figure 2-5. Executing the *shared_key_image_encrypt* Program.....10
Figure 2-6. Identifying the COM Port.....10
Figure 2-7. Selecting the COM Port in Tera Term.....11
Figure 2-8. Configuring the COM Port in Tera Term.....11
Figure 2-9. Executing the *shared_key_image_encrypt* Program.....12
Figure 2-10. *shared_key_image_encrypt* Command Menu.....12
Figure 2-11. Loading the Key Image.....13
Figure 2-12. Keys Successfully Loaded.....13



Copyright © 2022 Texas Instruments Incorporated

Figure 2-13. Loading the Program Image....14
Figure 2-14. Verifying the Program Image....15
Figure 2-15. Erasing Flash with Code Composer Studio....16
Figure 2-16. Loading eeprom_erase.out....16
Figure 2-17. Configuring LM Flash Programmer to Program the Boot Loader....17
Figure 2-18. Programming the Boot Loader With LM Flash Programmer....18
Figure 2-19. Configuring LM Flash Programmer to Program the Application Code....19
Figure 2-20. Programming the Application Code with LM Flash Programmer....20

## Trademarks

Code Composer Studio™ is a trademark of Texas Instruments.
All trademarks are the property of their respective owners.

## 1 Implementation

The code encryption and verification method will be implemented with five example projects:

- Flash boot loader project
- Image creation project
- Key creation project
- Example application project for the EK-TM4C129EXL LaunchPad Development Kit
- Example application project for the DK-TM4C129X Development Kit

There is also a sixth project included that is used to erase the keys in EEPROM. This is simply an supplemental resource for use during development.

### 1.1 Flash Boot Loader Project

The flash boot loader project *shared_key_boot_serial* is an example based on the current flash serial boot loader implementation with these additional features:

- The boot loader exists in the first 16KB of flash space
- The boot loader checks for 256-bit keys stored in two blocks of EEPROM.
- If no valid keys are found (keys that are all FFs are invalid):
  - The boot loader enters a mode waiting to upload an image without decrypting that image. If the image starting at 0x4000 contains only 1 to 4 valid keys, those keys are copied to the EEPROM and the sector at 0x4000 is erased.
  - The boot loader checks that JTAG is disabled. If not, it disables JTAG, and write protects the first sector of flash (release configuration only).
- If a valid key image is found:
  - The boot loader decrypts the incoming data stream using AES decryption with a 256-bit key before programming the data.
  - The incoming image should consist of a full flash image starting at address 0x4000, with the 16 bytes at the end of flash containing the authentication signature.
    - The flash end address, `APP_END`, is defined in the header file `linker_defines.h`. This file is used by the key creation project and the boot loader project. The flash application end address can be defined smaller than the actual flash end to reduce the time to update a device, but once defined in the boot loader, that becomes the maximum size of the image that can be uploaded into that device.
  - The boot loader computes an AES-CBCMAC signature on the data from address 0x4000 to the end of flash minus 16 bytes.
  - If that signature matches the one stored at the end of flash, the boot loader hides the EEPROM blocks that contain the keys and jumps to the application code.

> A method to change the encryption keys would be to add a new function that can revoke the current key using a valid hash with the current key and a specific command.



Copyright © 2022 Texas Instruments Incorporated

### 1.1.1 Changes to the Example Project boot_serial

#### *1.1.1.1 Changes to bl_config.h*

- Line 29: Modify blank line by adding “**#include** "linker_defines.h"”. This defines the flash application start address
- Line 80: Modify to “**#define** APP_START_ADDRESS APP_BASE”.
- Line 218: Uncomment FLASH_CODE_PROTECTION.
- Line 1402: Uncomment “void MyReinitFunc(void);”.
- Line 1405: Uncomment “#define BL_INIT_FN_HOOK MyReinitFunc”.
- Line 1442: Uncomment “void MyEndFunc(void);”.
- Line 1445: Uncomment “#define BL_END_FN_HOOK MyEndFunc”.
- Line 1453: Uncomment “void MyDecryptionFunc(unsigned char *pucBuffer, unsigned long ulSize);” and change “long” to “int”.
- Line 1460: Uncomment “#define BL_DECRYPT_FN_HOOK MyDecryptionFunc”.
- Line 1471: Uncomment “unsigned long MyCheckUpdateFunc(void);”.
- Line 1479: Uncomment “#define BL_CHECK_UPDATE_FN_HOOK MyCheckUpdateFunc”.

#### *1.1.1.2 New Functions Added*

The following new functions are added in the file `shared_key_functions.c`.

##### 1.1.1.2.1 MyCheckUpdateFunc

This routine is called at the beginning of the boot loader and determines if the boot loader should run the application code, or wait for a new image. It checks the following five conditions:

- If the code is running on a device that does not support the AES hardware encryption, the boot loader will not execute an application program.
- If the device does not have valid keys stored in the EEPROM, the application program will not be executed. Instead, the boot loader enters a state where it waits for an unencrypted image of the keys to be uploaded.
- *MyCheckUpdateFunc* will call the user-defined function *CheckGPIOPin*. If this function returns true, the boot loader will not execute the application program, rather it will wait to upload a new encrypted application image. In the example boot loader program, it checks if the EK-TM4C129EXL LaunchPad *SW2* button is being held low.
- *MyCheckUpdateFunc* checks if a software reset occurred and if a RAM-based predetermined value is set signifying the boot loader was called by the application. If these are true, the boot loader waits to upload a new encrypted application image.
- *MyCheckUpdateFunc* computes an AES-CBCMAC hash on the application code area and compares the result to the last 16 bytes of the application image. If the images match, the boot loader executes the application code. If the images do not match, it waits to upload a new encrypted application image.

##### 1.1.1.2.2 MyReinitFunc

Since the keys can only be read from the EEPROM after a reset but before the application is invoked, the boot loader will not work when directly called from the application. Therefore, this function sets a predefined value in a static RAM location and then causes a software reset. The *MyCheckUpdateFunc* function checks for a software reset with the predefined RAM value and uses that condition to stay in the boot loader.

##### 1.1.1.2.3 MyEndFunc

This function is used to load a new key image into EEPROM. It is called after all of the data has been programmed but before execution is passed to the new program. If the device contains valid keys or the device does not have an AES module, it will return without executing. If the device does not contain valid keys and has an AES module, then this function will check the loaded image to see if it is a valid key image. If so, it will program the keys into EEPROM, hide the EEPROM blocks, erase the flash at `APP_BASE`, and finally reset the part.

##### 1.1.1.2.4 MyDecryptionFunc

This function decrypts the input buffer using AES-CBC with a 256-bit key. The input buffer must be a size of 16, 32, 48, or 64 bytes.


Copyright © 2022 Texas Instruments Incorporated

### 1.2 Image Creation Project

The image creation project, *shared_key_image_encrypt*, only runs on a TM4C129 part with AES module. It uses UART0 to run a command line program. That program executes the following commands:

- K – Load two 256-bit key pairs and save them in EEPROM
- D - Disable JTAG (only if keys are already in EEPROM)
- L – Load a program that starts at address 0x4000, using XMODEM transfer.
- X – Execute the program loaded at address 0x4000
- O – Output a binary image that is signed and encrypted.
- H – Print help menu

### 1.3 Key Image Project

*key_image* is a simple assembly language project that is used to create a binary image of the encryption and verification keys.

### 1.4 EK-TM4C129EXL Example Application Project

A simple application project, *shared_key_boot_demo*, can be used to demonstrate the process of creating an application, signing and encoding it, then programming it with the secure boot loader. When run, the application blinks an LED until the *SW1* button is pressed and held ,which returns the application to the boot loader.

### 1.5 DK-TM4C129X Example Application Project

A simple application project, *shared_key_boot_demo_DK*, that can be used to demonstrate the process of creating an application, signing and encoding it, then programming it with the secure boot loader. When run, the application prints a message on the LCD screen. When the *Down* switch is pressed, the application returns to the boot loader.

### 1.6 RAM-Based EEPROM Erase Project

Keys stored in the EEPROM will remain unless the device goes through an “unlock” procedure, or they are explicitly erased by an application that runs from reset. The project *eeprom_erase* erases the keys when run by Code Composer Studio™ after a system reset. This project is simply an aid for use during the boot loader development to clear any loaded keys from EEPROM without needing to go through the more tedious device unlock process.

## 2 Example Walk Through

### 2.1 Build Environment

These examples were built and tested using:

- EK-TM4C129EXL LaunchPad Development Kit
- Code Composer Studio v10.2
- TI Arm C Compiler v20.2.5.LTS
- TivaWare v2.2.0.295



Copyright © 2022 Texas Instruments Incorporated

## 2.2 Importing the Examples into Code Composer Studio

There are six CCS project examples attached as collateral to this document. Project collateral discussed in this document can be downloaded from the following URL: https://www.ti.com/lit/zip/spma083. The projects can be unzipped into a folder or kept in the zip file. Both formats can be imported to the CCS.

1. To import the project into CCS, first select “File” -> ”Import”.

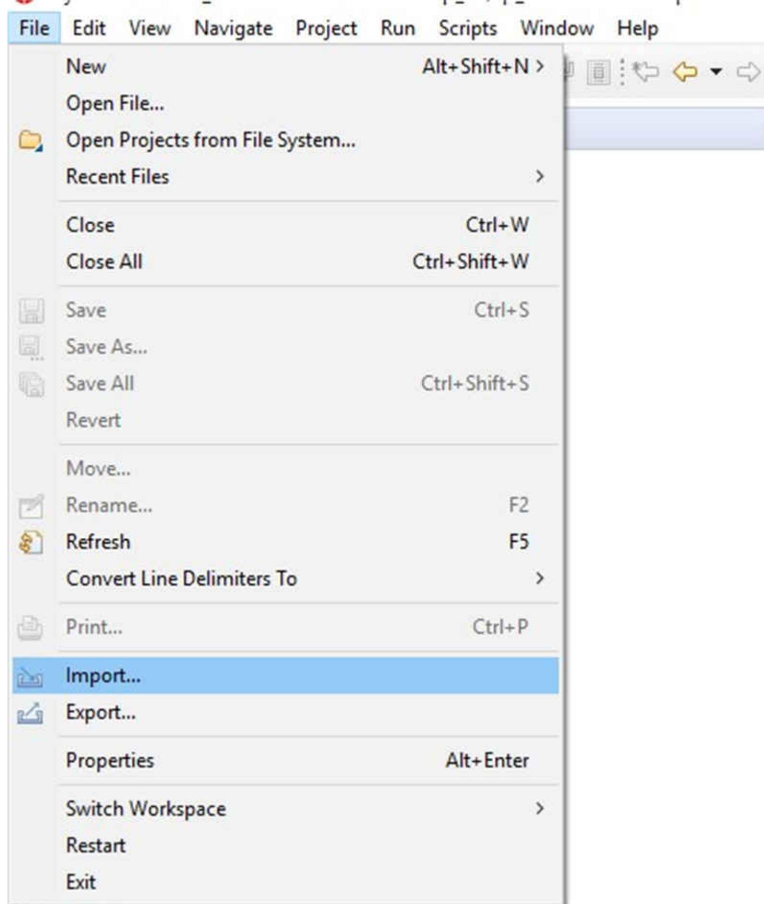


**Figure 2-1. Import CCS Projects Step 1.**


Copyright © 2022 Texas Instruments Incorporated

2. Select “CCS Projects” to import the examples and then click “Next”.

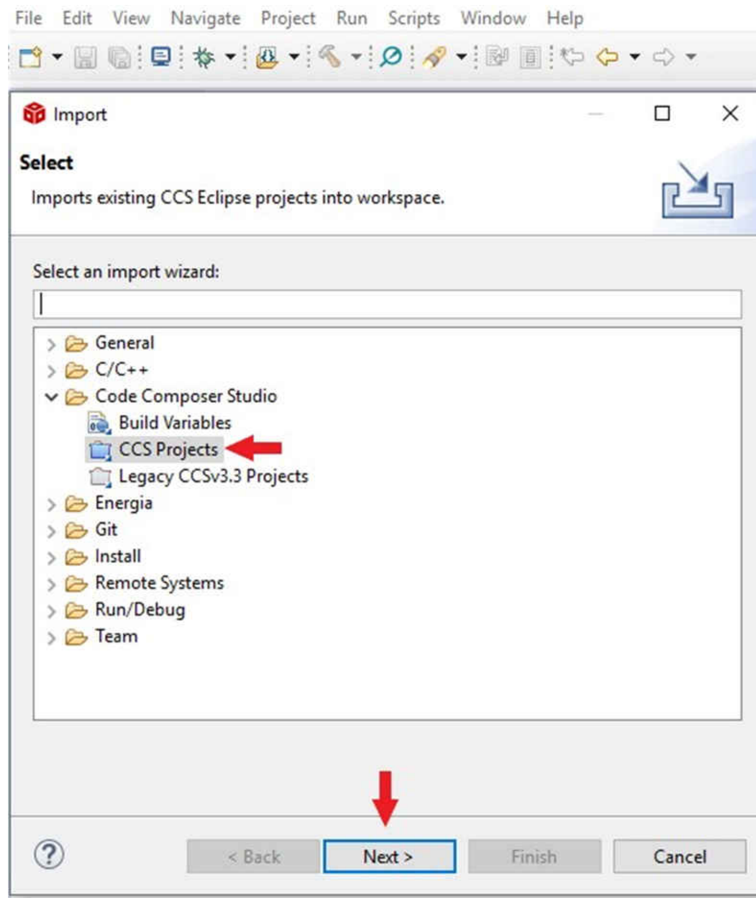


**Figure 2-2. Import CCS Projects Step 2.**



Copyright © 2022 Texas Instruments Incorporated

3. Next, provide the path to either the unzipped project by selecting the first radio button or import the zip file directly by selecting the second radio button. Select the “Copy projects into workspace” box.

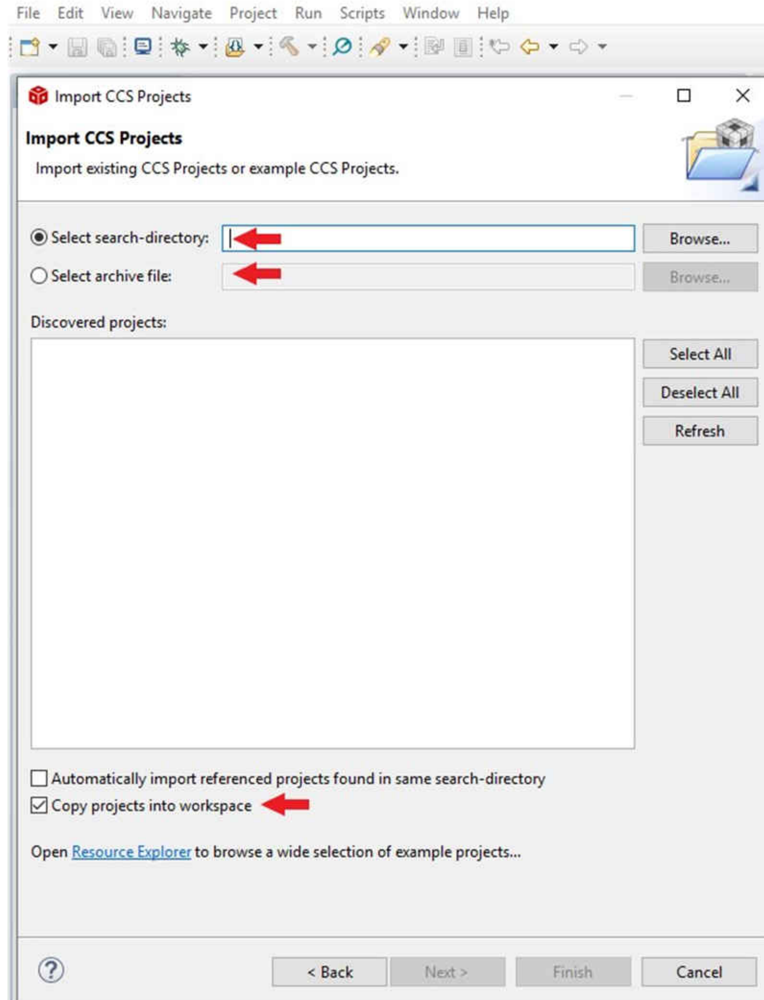


**Figure 2-3. Import CCS Projects Step 3.**



Copyright © 2022 Texas Instruments Incorporated

4. After the project path is provided, a total of six discovered projects will show up. First click the “Select All” button and then click the “Finish” button to complete the import.

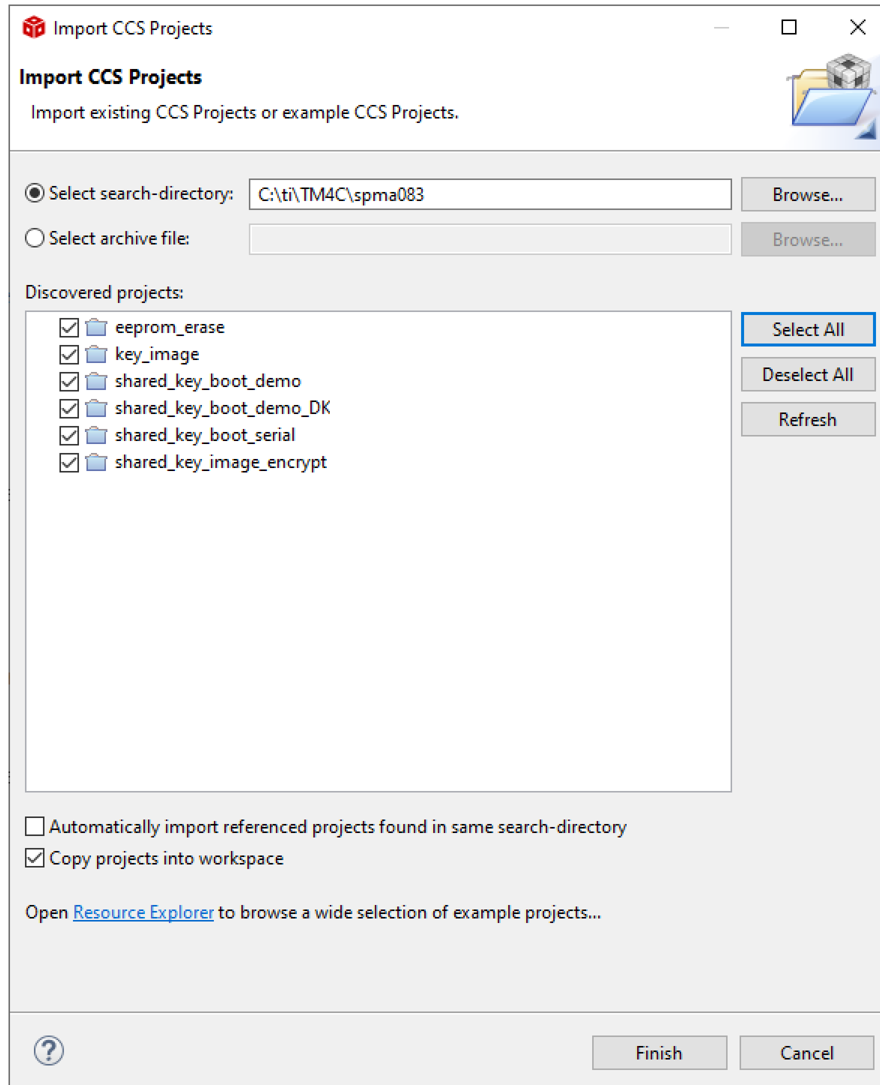


**Figure 2-4. Import CCS Projects Step 4.**

## 2.3 Setting Keys and Variables

### 2.3.1 Keys

The keys are defined in the file `key_image.asm` that is in the `key_image` directory. This file is used in the *key_image* project and also, by link, in the *shared_key_boot_serial* example project.

```
 .sect ".keyimage"
 .global key
key
; Encryption/Decryption Key 0
     .word 0x3ed44417, 0x8d849ffc, 0x4719e4dc, 0x71583965
     .word 0x84b6ba7d, 0xa96ff68a, 0xb79d9da8, 0x5f747e02
; Authentication Key 0
     .word 0xf74e59bf, 0xd19cd4e1, 0x630303f8, 0xe5bb8089
     .word 0x0e3bc945, 0xffc85239, 0x53289e9c, 0x5f906df8
; Encryption/Decryption Key 1
     .word 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF
     .word 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF
; Authentication Key 1
     .word 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF
     .word 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF
```



Copyright © 2022 Texas Instruments Incorporated

As seen in the code, there are actually four 256-bit keys defined. The first key is for encryption/decryption. The second key is for authentication. The last two keys are currently not used.

For initial evaluation and development the keys given in this example can be used, but for a real project these need to be changed and kept confidential in a secure location.

### 2.3.2 Initialization Vector

The initialization vector is defined in the file `initialization_vector.c`. This file is in the project *shared_key_image_encrypt* and is referenced by link in the project *shared_key_boot_serial*. In this example the initialization vector is all zeros. It is suggested that a random 128-bit initialization vector be chosen for real products.

### 2.3.3 Application Start Address and Flash Size

The file `linker_defines.h`, located in the project *shared_key_image_encrypt*, defines the space in Flash memory for the application and the size of RAM. It is used in the source code and link command files for the *shared_key_image_encrypt* and *shared_key_boot_serial* projects.

```
/*
 * linker_defines.h
 */
#ifndef LINKER_DEFINES_H_
#define LINKER_DEFINES_H_

#define APP_BASE 0x00004000
#define APP_END 0x00100000
#define RAM_BASE 0x20000000

#endif /* LINKER_DEFINES_H_ */
```

#### *2.3.3.1 APP_BASE*

`APP_BASE` is the start of the flash area for the application. It must be on the start of a flash sector boundary. For the TM4C129 devices this means it must be a multiple of 0x4000. Since the boot loader exists in the first sector, `APP_BASE` cannot be zero. There is typically no need to change `APP_BASE` from the default value of 0x4000.

#### *2.3.3.2 APP_END*

`APP_END` is the address of the first byte beyond the flash memory available to the application. The AEC-CBCMAC authentication signature is stored in the 16 bytes before `APP_END`. The default value for devices with 1MB of flash is 0x100000. For devices with 512KB of flash the default value is 0x80000.

The user may choose to use a smaller size flash to reduce the time to download a new image. If so, the value for `APP_END` must be multiple of the sector size, 0x4000. The same value must be used for `APP_END` in the boot loader and in the tool that creates the encrypted image. The size of usable flash for the application cannot be changed once the boot loader is programmed into the device. Make sure to leave enough unused space in the flash to support future application program growth.

The reason a fixed size image is loaded each time is that one attack for AES-CBC encoded data is to extend the length with malicious code. Using a fixed length of data foils that attack method.

#### *2.3.3.3 RAM_BASE*

`RAM_BASE` defines the starting address of RAM on the part. There is usually no reason to change this value from the default of 0x2000.0000.

## 2.4 Running the shared_key_image_encrypt Tool

After making any modifications described in section 3.3, re-compile the `Debug` configurations of the projects *shared_key_image_encrypt* and *key_image*. Then recompile the `Debug_wKey` configuration of the project *shared_key_boot_serial*. The debug configurations do not lock the JTAG and are for use during development. The release configurations will lock JTAG and should be used for the production version of the shared key boot loader.


Copyright © 2022 Texas Instruments Incorporated

1. With the EK-TM4C129EXL LaunchPad Development Kit connected with a USB cable from the DEBUG port of the LaunchPad to a USB port on the PC, launch the *shared_key_image_encrypt* project. This can be done by highlighting the project in the Project Explorer window then clicking on the debug icon shown in yellow in Figure 2-5.

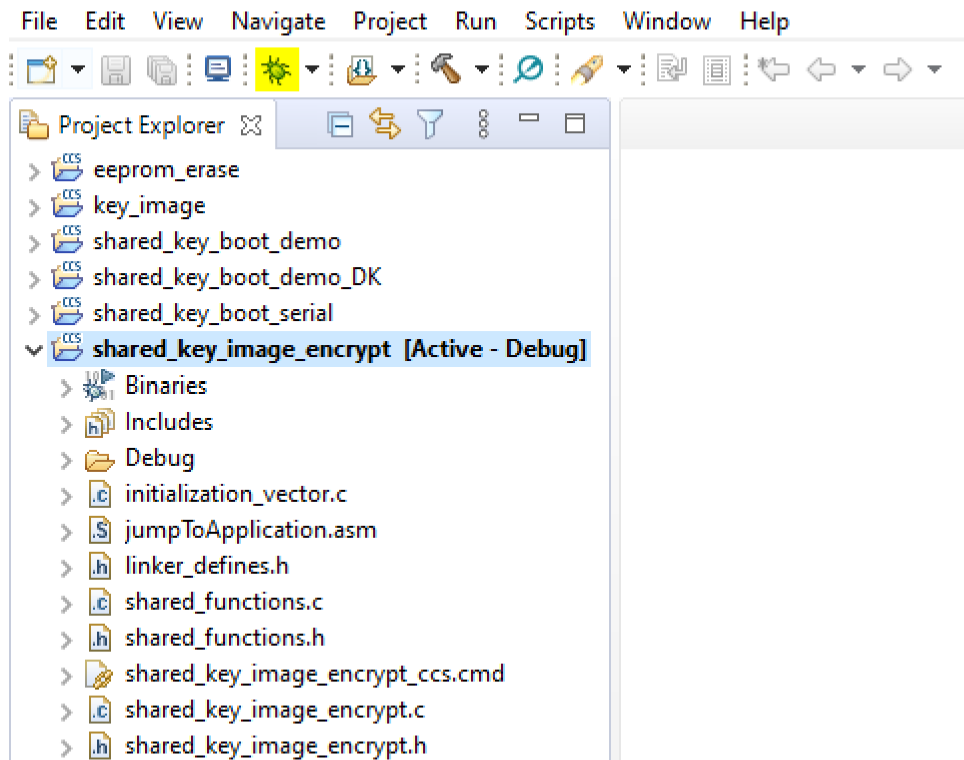


**Figure 2-5. Executing the *shared_key_image_encrypt* Program**

2. Identify that the serial port was assigned by the PC to the virtual serial port of the LaunchPad. This can be found by running Device Manager on the PC. In Figure 2-6, the LaunchPad was assigned to COM19.

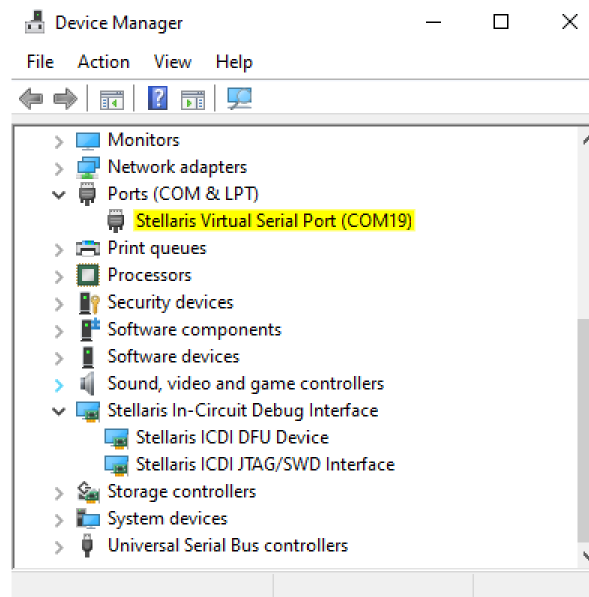


**Figure 2-6. Identifying the COM Port**



Copyright © 2022 Texas Instruments Incorporated

3. Open a terminal emulator that supports XMODEM communications. In this example "Tera Term" was used. Connect over serial to the identified COM port for the LaunchPad.

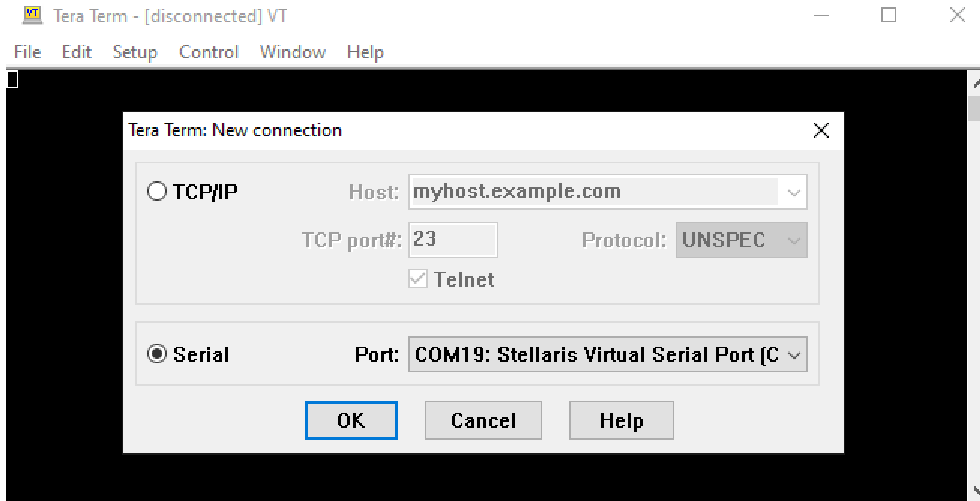


**Figure 2-7. Selecting the COM Port in Tera Term**

4. Configure the terminal emulator to use the serial port at 460800 baud, 8-bit data, 1 stop bit, and no parity.

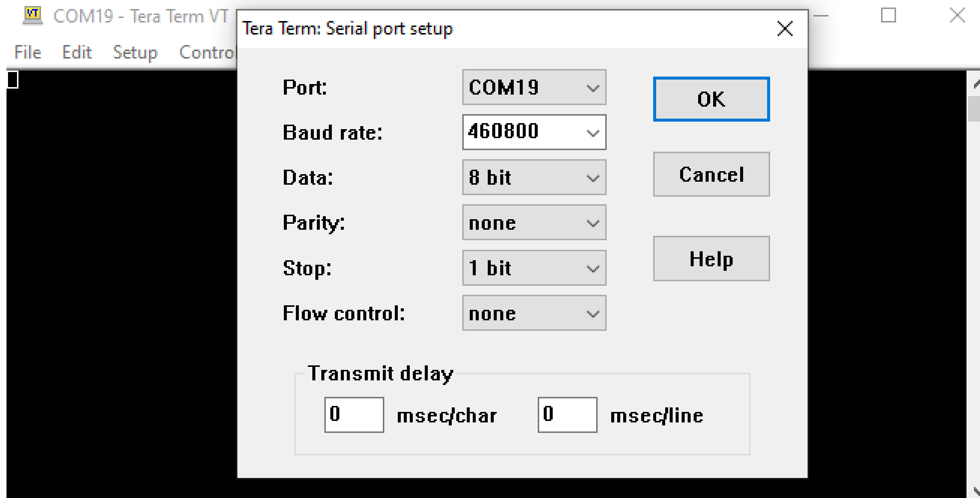


**Figure 2-8. Configuring the COM Port in Tera Term**



Copyright © 2022 Texas Instruments Incorporated

5. Execute the *shared_key_image_encrypt* program by clicking on the green arrow in Code Composer Studio or by pressing **F8**.

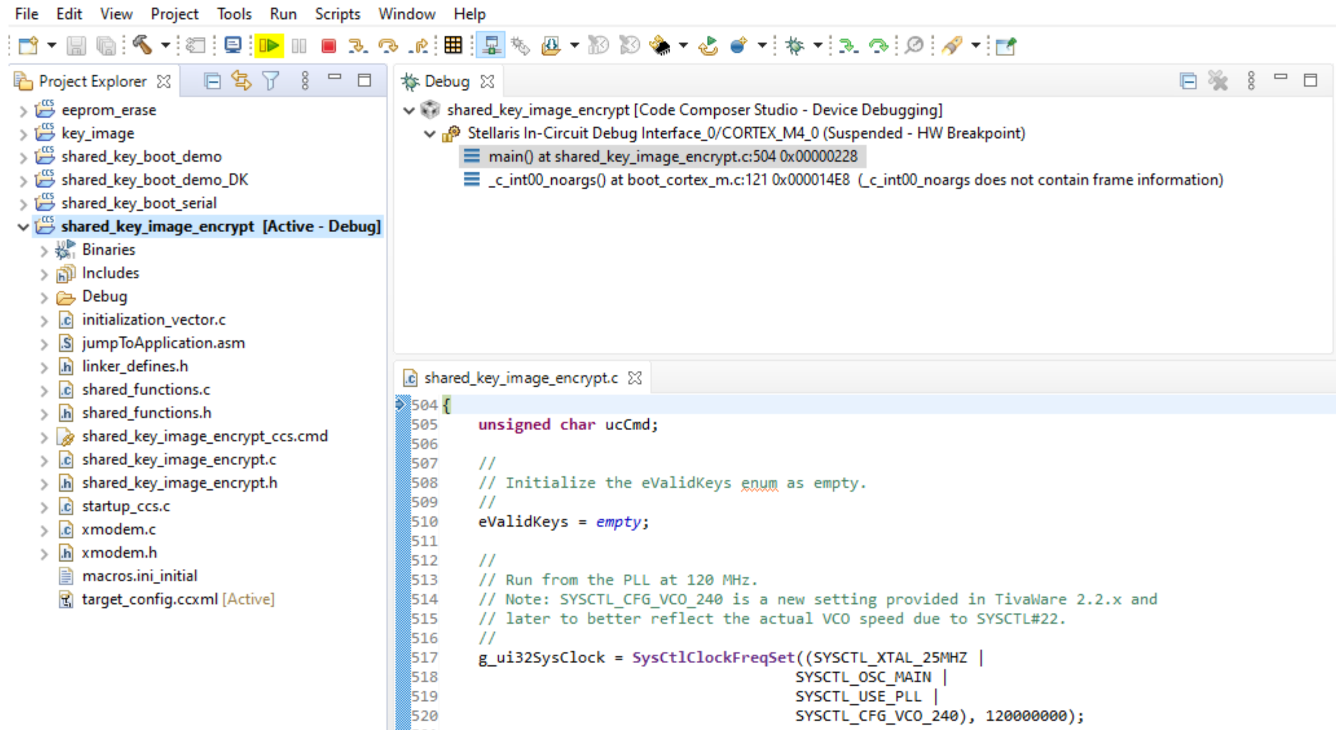

**Figure 2-9. Executing the *shared_key_image_encrypt* Program**

6. The following menu should now be displayed on the terminal.

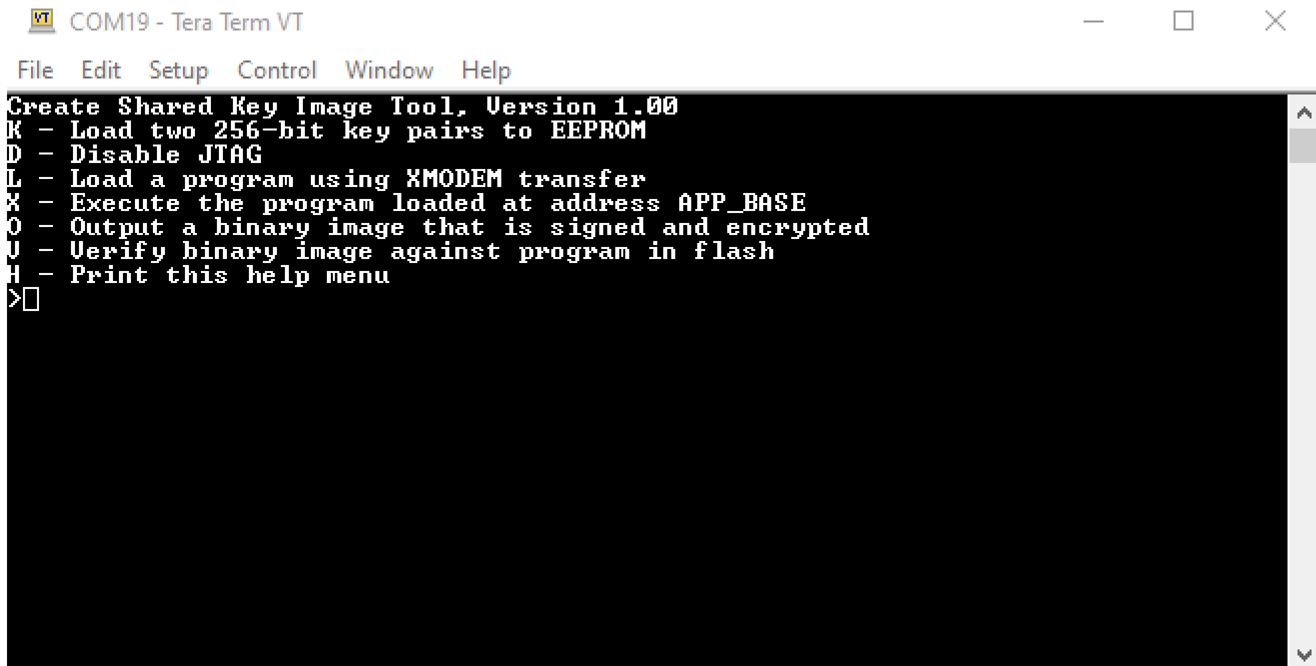


**Figure 2-10. *shared_key_image_encrypt* Command Menu**



Copyright © 2022 Texas Instruments Incorporated

7. To create a signed and encrypted image to be used by the boot loader follow these steps:
   a. Transfer the file `key_image.bin` from the debug subdirectory of the project *key_image* to the device using XMODEM protocol. Using Tera Term follow these steps:
      i. Type K in the terminal emulator (it takes lower-case k or upper-case K).
      ii. Select "File" -> "Transfer" -> "XMODEM" -> "Send".
      iii. Browse to `key_image\Debug\key_image.bin` and select "Open".

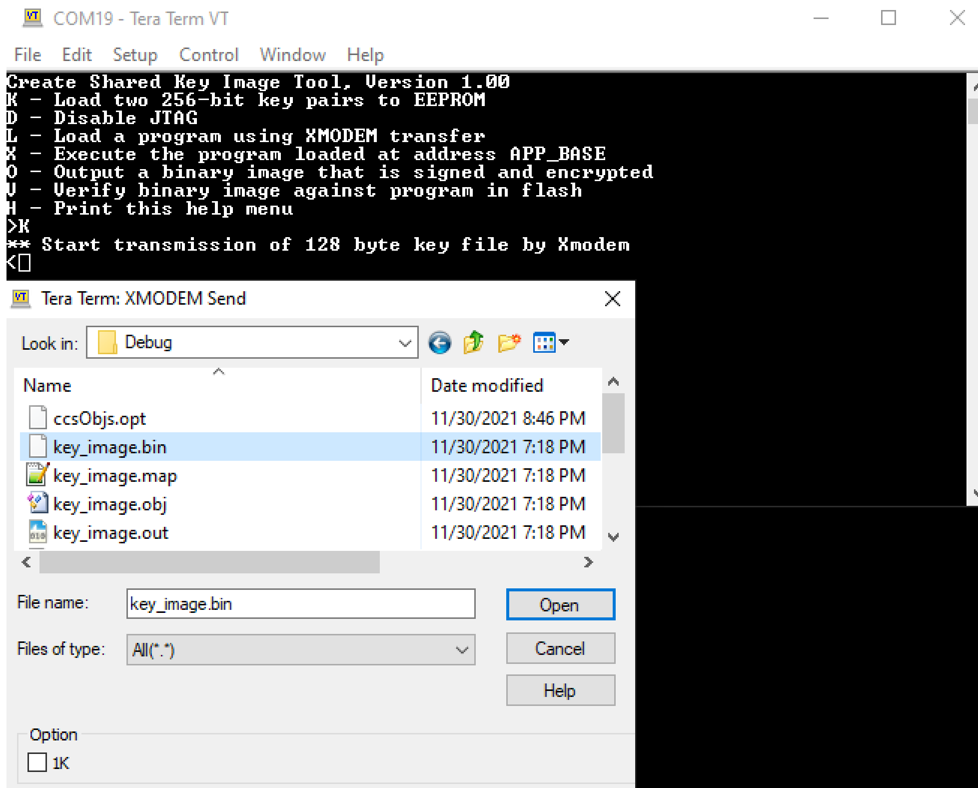


**Figure 2-11. Loading the Key Image**

      iv. The terminal should now display **Keys successfully programmed into EEPROM**.

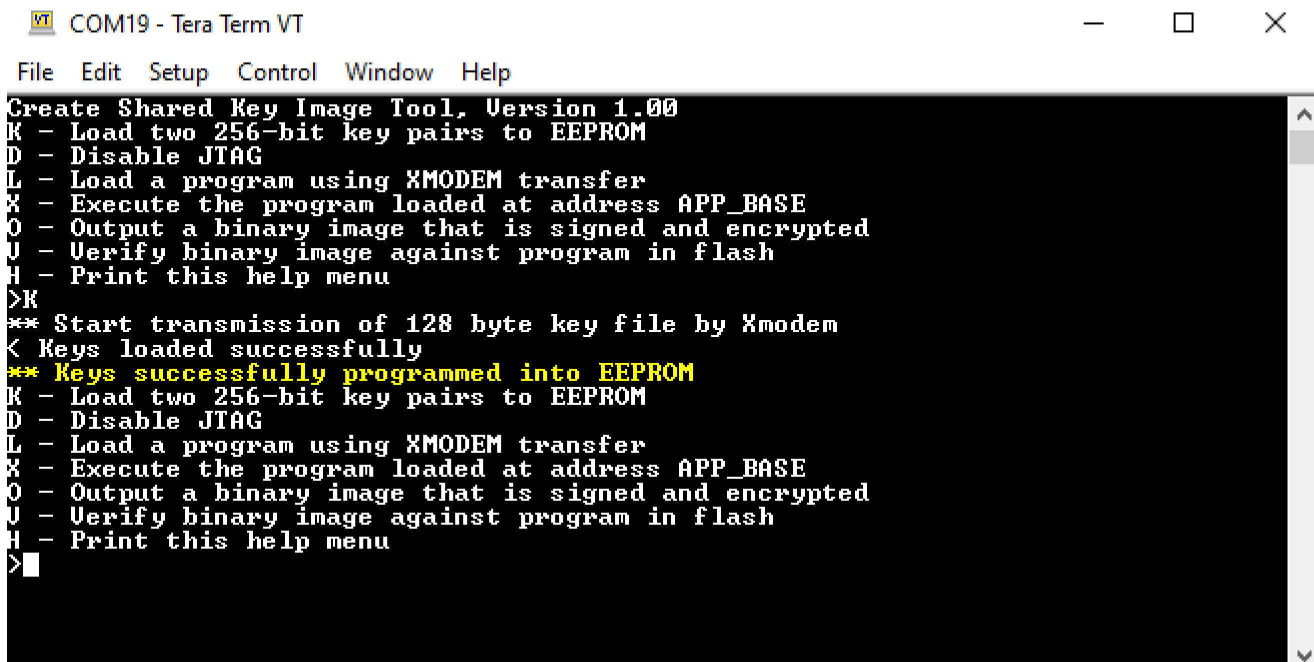


**Figure 2-12. Keys Successfully Loaded**



Copyright © 2022 Texas Instruments Incorporated

b. Transfer the application image into the device. Using Tera Term follow these steps:
   i. Type L in the terminal emulator.
   ii. Select “File” -> “Transfer” -> “XMODEM” -> “Send”.
   iii. Browse to `shared_key_boot_demo\Debug\shared_key_boot_demo.bin` and select “Open”.

   The transfer can take up to one minute for a full 1MB flash device. After completion, the message **File programming complete** will be displayed on the terminal.

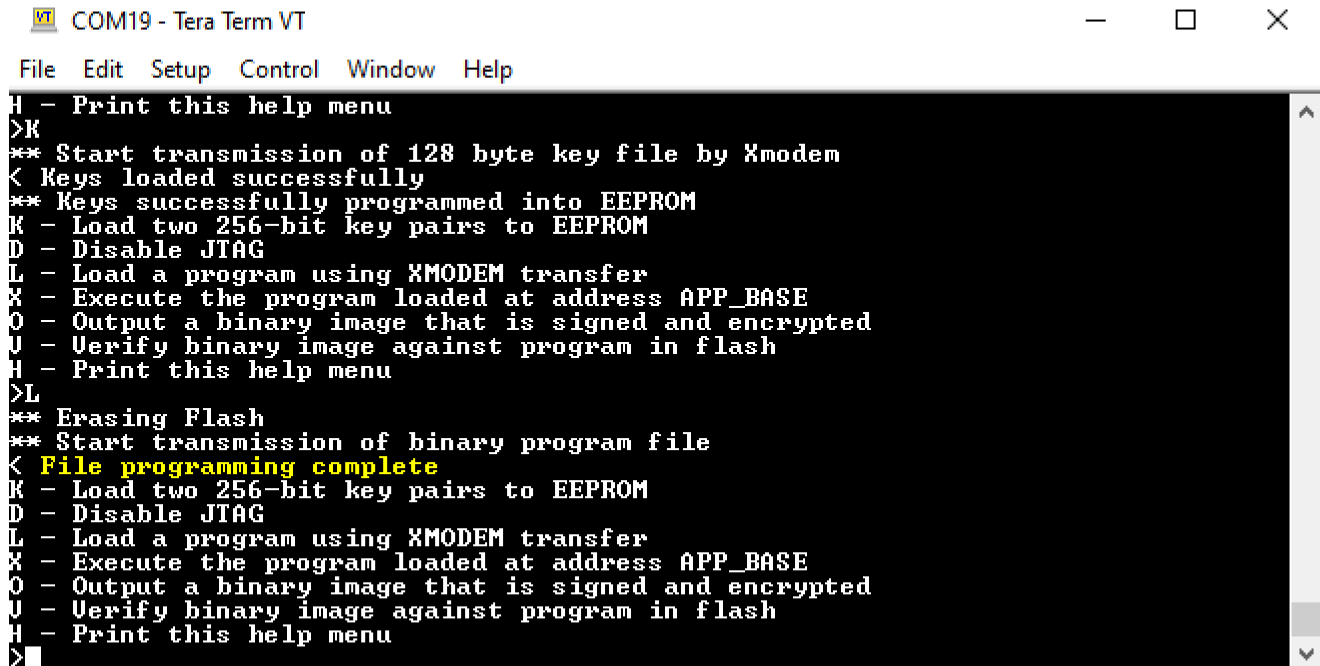


**Figure 2-13. Loading the Program Image**

c. Output a signed and encrypted version of the binary image. Using Tera Term follow these steps:
   i. Type O in the terminal emulator.
   ii. Select “File” -> “Transfer” -> “XMODEM” -> “Receive”.
   iii. Choose an output file name such as:

      `shared_key_boot_demo\Debug\shared_key_boot_demo_encrypted.bin` and select “Save”.

d. Optionally, it is possible to verify the encrypted file just created against the unencrypted image stored in flash by:
   i. Type V in the terminal emulator.
   ii. Select “File” -> “Transfer” -> “XMODEM” -> “Send”.
   iii. Browse to `shared_key_boot_demo\Debug\shared_key_boot_demo_encrypted.bin` and select “Open”.
   iv. The transfer will take about one minute for a full 1MB flash device. After completion, the message **File verification complete** will be displayed on the terminal if the encrypted file's hash value was autheticated successfully.


Copyright © 2022 Texas Instruments Incorporated

```
COM19 - Tera Term VT
File Edit Setup Control Window Help
O - Output a binary image that is signed and encrypted
V - Verify binary image against program in flash
H - Print this help menu
>O
** Start xmodem file receive
< Encrypted firmware image sent
K - Load two 256-bit key pairs to EEPROM
D - Disable JTAG
L - Load a program using XMODEM transfer
X - Execute the program loaded at address APP_BASE
O - Output a binary image that is signed and encrypted
V - Verify binary image against program in flash
H - Print this help menu
>V
** Start transmission of encrypted data file by Xmodem
< File verification complete
K - Load two 256-bit key pairs to EEPROM
D - Disable JTAG
L - Load a program using XMODEM transfer
X - Execute the program loaded at address APP_BASE
O - Output a binary image that is signed and encrypted
V - Verify binary image against program in flash
H - Print this help menu
>
```

**Figure 2-14. Verifying the Program Image**

e. Optionally, execute the application code that has been programmed into the device by:
   i. Type X in the terminal emulator.
   ii. The message **Hash value authenticated** will be displayed on the terminal and LED D1 on the LaunchPad will be flashing.
   iii. Reset the LaunchPad with either Code Composer Studio, or the reset button on the LaunchPad and the terminal will indicate that the program is back in the *shared_key_image_encrypt* tool.

## 2.5 Running the Shared Key Serial Boot Loader

While the boot loader could be programmed in a non-secure environment, that would open up a risk that the initialization vectors could be exposed. Therefore, the best method is to program the boot loader and the keys simultaneously in a secure environment. When the boot loader is programmed in the first sector, and an image of the keys is programmed in the sector at `APP_BASE`, the boot loader copies the keys into EEPROM and then erase the sector at `APP_BASE`. If the “Release” configuration is used, the boot loader then write protects itself and disables JTAG.

The “Debug” configuration of the boot loader should not be released in the field. Without much difficulty someone can connect to the JTAG port with an emulator and write code that would expose the keys. Then they could make their own programs that would load using these keys.

The user should consider whether or not to expose the JTAG pins on their circuit board. It is possible to program the initial boot loader and the keys using the ROM boot loader. Therefore, JTAG access is not required. Using the “Release” configuration of the boot loader disables the JTAG interface, but someone could still use the method of recovering a locked microcontroller to completely reset the device. This would erase the boot loader, the application code, and the keys which were in EEPROM. Afterwards, they could then program their own code into that device. It would be the same effect as if they desoldered the TM4C device and replaced it with a new one. For ball grid array (BGA) devices, JTAG can be hidden by not adding traces to the JTAG balls. For quad flat pack (QFP) devices the pins are exposed even if there are no traces to the pins.

The main disadvantage of not having access to the JTAG pins is that the device cannot be recovered externally. Therefore, the user must ensure there is a validated process to re-enter the boot loader. Also, it would not be possible to do analysis of failing parts without removing them from the printed circuit board.


Copyright © 2022 Texas Instruments Incorporated

### 2.5.1 Programming the Boot Loader

#### *2.5.1.1 Erasing Existing Code and Keys*

To emulate a true production environment, start with a fully erased part. There are two methods of achieving this:

**2.5.1.1.1 Erasing Flash and EEPROM With Code Composer Studio**

1. To erase the flash, select “Tools” -> “On-Chip Flash” from the pulldown menu. Enter the proper crystal frequency. It is 25 MHz for the EK-TM4C129EXL LaunchPad Development Kit. In the erase section make sure “Entire Flash” is selected, then click the “Erase Flash” button.

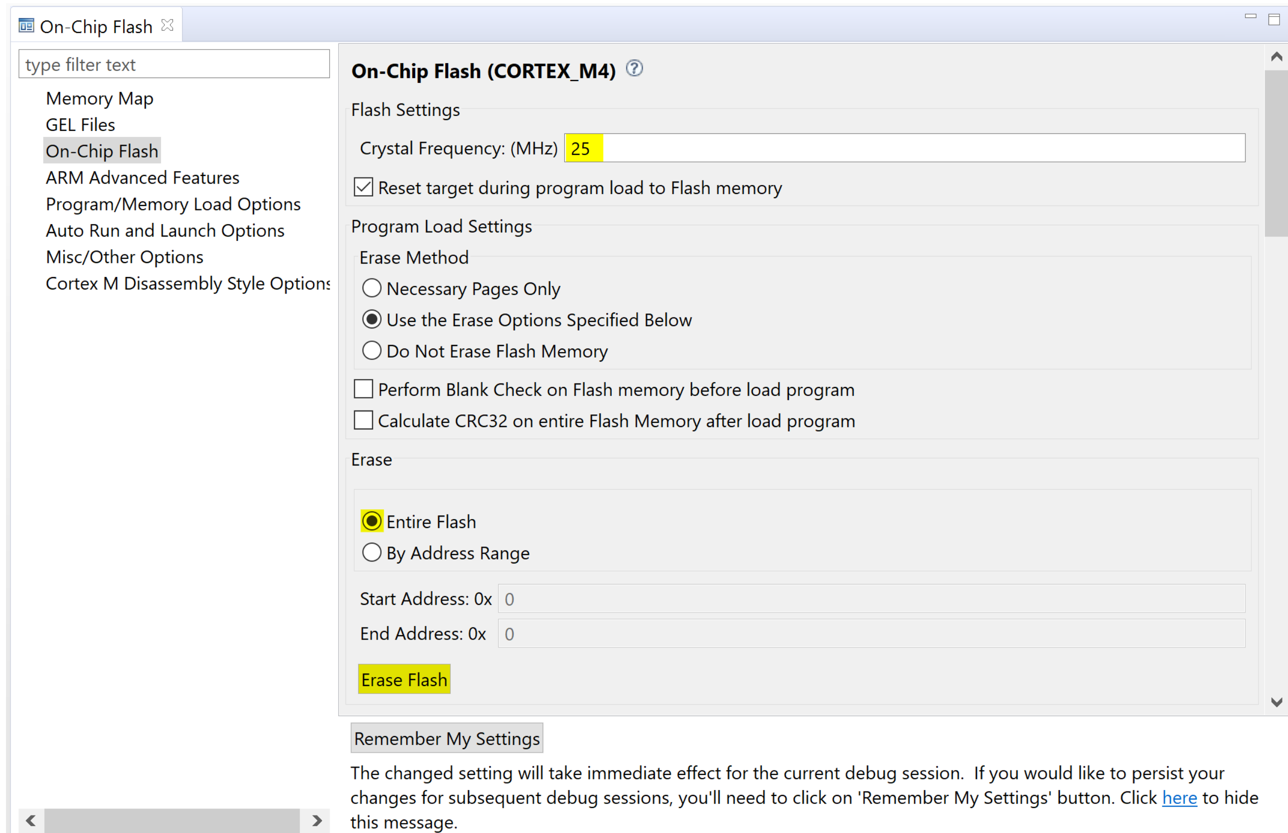


**Figure 2-15. Erasing Flash with Code Composer Studio**

2. To erase the EEPROM, load and execute the *eeprom_erase* program that was loaded into the workspace with this document’s collateral. From the pulldown menu select “Run” -> “Load” -> “Select Program to Load”. Then click “Browse project”. Select the file `eeprom_erase\Debug\eeprom_erase.out`, then click “OK”.
3. Click OK again to load and execute the *eeprom_erase* program.

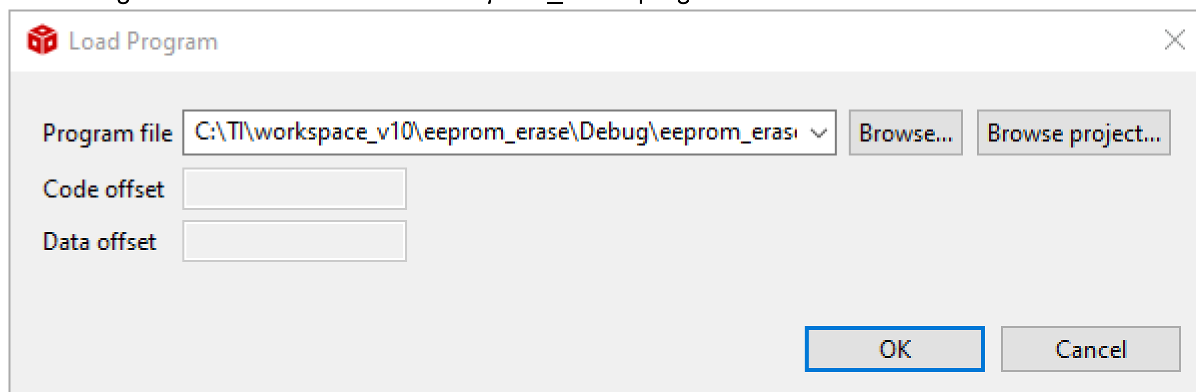


**Figure 2-16. Loading eeprom_erase.out**

4. Two LEDs on the LaunchPad will blink indicating that the program ran and erased the EEPROM.



Copyright © 2022 Texas Instruments Incorporated

#### 2.5.1.1.2 Erasing Flash and EEPROM by Using the Unlock Procedure

The unlock procedure causes all flash, EEPROM, and device settings to be restored to factory conditions. This causes the MAC address stored in the user registers of the LaunchPad to be erased. It is recommended to record the MAC address before unlocking the part so that it can be restored afterwords.

The unlock procedure using LM Flash Programmer is described in the *Executing Unlock Sequence* section in *Using TM4C12x Devices Over JTAG Interface*.

### *2.5.1.2 Using the ROM Boot Loader to Program the Shared Key Boot Loader*

Now that the flash is erased, the LaunchPad is executing the ROM boot loader. The prior example for the shared key boot loader utilized a serial interface. This ROM boot loader example uses the same serial interface to demonstrate how to load the shared key boot loader into a blank device.

---

**Note**

Make sure the TeraTerm terminal window is disconnected from the serial port and the LaunchPad is not connected to Code Composer Studio. The USB cable must still be connected.

---

1. Open LM Flash Programmer and set the items on the configuration tab as shown below, selecting the COM port associated with the LaunchPad.

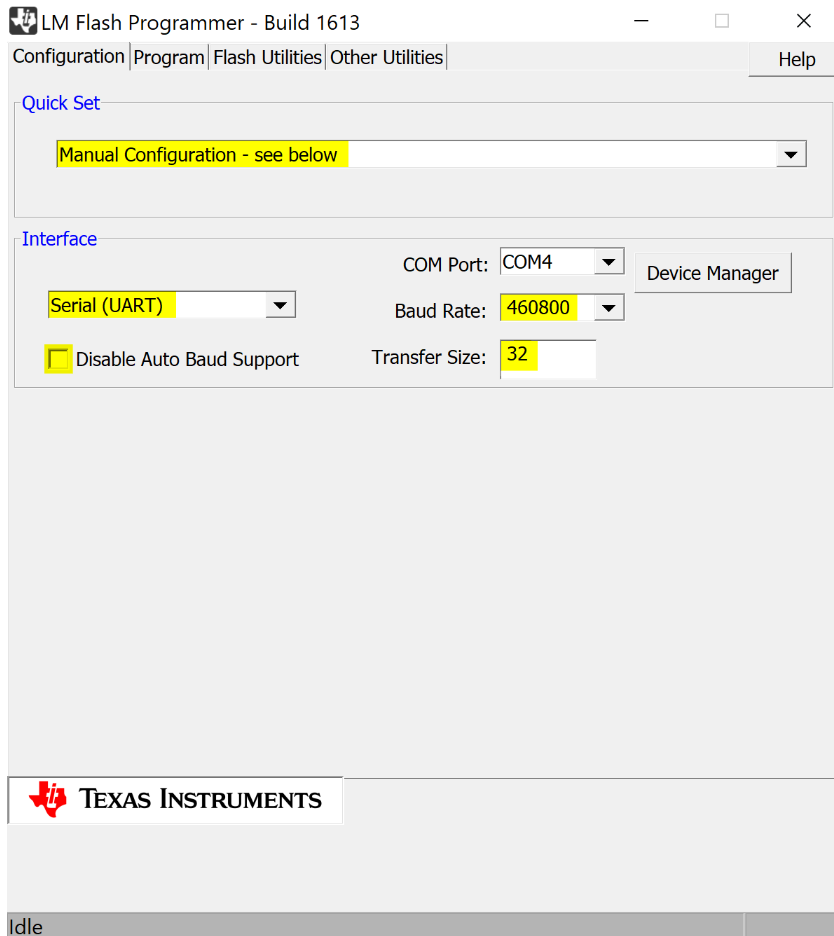


**Figure 2-17. Configuring LM Flash Programmer to Program the Boot Loader**


Copyright © 2022 Texas Instruments Incorporated

2. Then, on the "Program" tab, browse to the `Debug_wKey\shared_key_boot_serial.bin` file. Make sure "Reset MCU After Program" is checked and the "Program Address Offset" is set to 0.

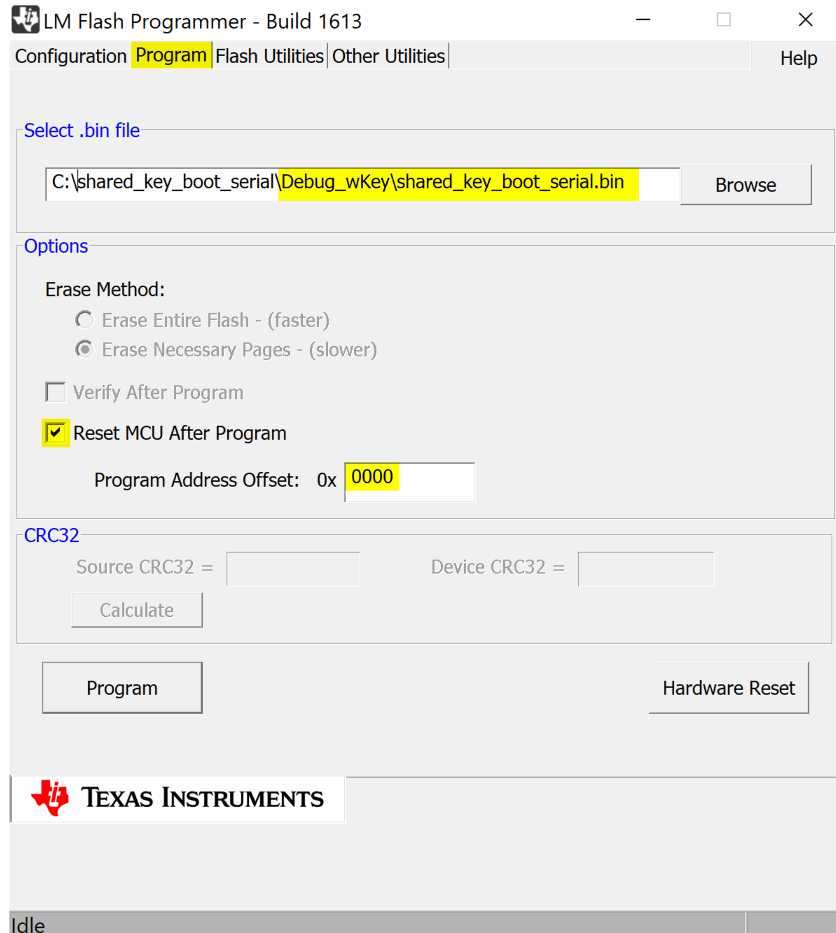


**Figure 2-18. Programming the Boot Loader With LM Flash Programmer**

3. Click the "Program" button.

After programming is completed, the boot loader executes and copies the keys into EEPROM before erasing them from flash. If the `Release_wKey\shared_key_boot_serial.bin` file is used instead of the debug version, the boot loader write protects the first sector of flash and lock the JTAG.



Copyright © 2022 Texas Instruments Incorporated

### 2.5.2 Using the Shared Key Boot Loader to Program the Application Code

The encrypted application code can now be programmed into the part using LM Flash Programmer. Since the keys are now hidden, this can be done in the field or in a less secure part of the factory.

1. On the configuration tab of LM Flash Programmer the “Disable Auto Baud Support” must be checked, and the baud rate must be set to the one chosen in the file `bl_config.h`. In this example, 460800 was used.

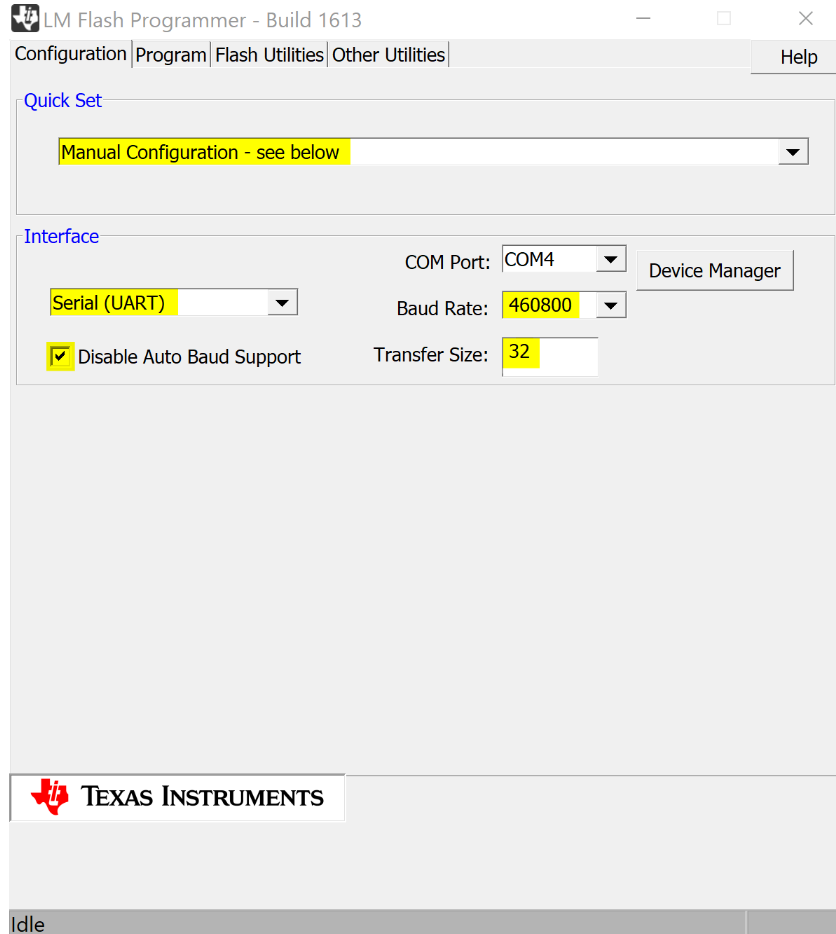


**Figure 2-19. Configuring LM Flash Programmer to Program the Application Code**


Copyright © 2022 Texas Instruments Incorporated

2. On the "Program" tab, select the encrypted application file. For this example, `shared_key_boot_demo\Debug\shared_key_boot_demo_encrypted.bin` is used.

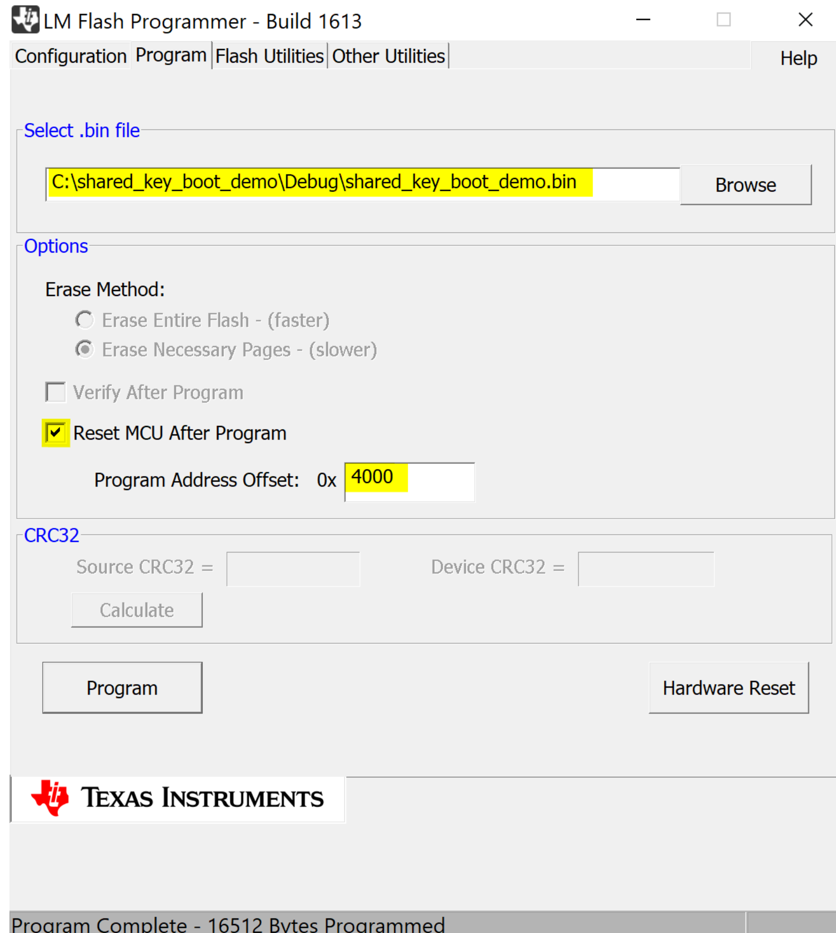


**Figure 2-20. Programming the Application Code with LM Flash Programmer**

3. Click the "Program" button.

Programming the application code space on a 1MB part will take about 100 seconds. When the *shared_key_boot_demo* application is running it will blink LED1.

## 2.6 Returning to the Boot Loader

In this example, there are two methods to return to the boot loader. The first is implemented in the boot loader. Coming out of reset, the boot loader checks if the *SW2* button on the LaunchPad is pressed. Press and hold *SW2* while pressing the reset button. LED1 will stop flashing and the boot loader will be ready to load a new encrypted application code. Simply press the reset button again without depressing *SW2* and the application code will run again.

The second method of returning to the boot loader is implemented in the application itself. This *shared_key_boot_demo* application code reads the LaunchPad *SW1* button. If it is depressed and held one second, the application code calls the boot loader.



Copyright © 2022 Texas Instruments Incorporated

## 3 Summary

The implementation of effective safeguards for systems continues to grow as a priority for system designers. This application report and the associated collateral are aimed at providing sensible options that can be considered for TM4C microcontrollers. Techniques such as leveraging EEPROM for key storage, validating firmware images with hash signatures, and locking JTAG access are all methods that can increase the protection of a system against unsophisticated or accidental attempts to override a code image. These concepts are applicable across all boot loader options as long as an emphasis is placed on ensuring images are able to be transferred in a manner that validates all packets were received correctly. While the focus for this application report was centered on devices with the hardware AES encryption peripheral, the same concepts can be applied for all TM4C devices provided software AES is leveraged instead with the tradeoffs being additional cost of code space and slower execution of the boot loading process. Ultimately, it is up to the system designers to define the criteria for what safeguards are required for proper system protection and assess the correct implementation to meet those requirements.


Copyright © 2022 Texas Instruments Incorporated

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2022, Texas Instruments Incorporated