# AM62X_OTP_Keywriter_User_Guide

Sitara

Exported on 08/03/2023

# Table of Contents

1 Introduction ........ 5
1.1 Supported Key Types and Fields ........ 5
1.2 Components ........ 6
1.3 VPP requirements ........ 7
2 Setting up the build environment ........ 8
2.1 Linux Host ........ 8
2.1.1 Install the MCU+ SDK for AM62X ........ 8
2.1.2 Install keywriter addon package ........ 8
2.1.3 Build the Keywriter Certificates ........ 8
2.1.4 Build the example: ........ 9
2.2 Windows Host ........ 9
2.2.1 Install MCU+ SDK for AM62X ........ 9
2.2.2 Install keywriter addon package ........ 10
2.2.3 Prerequisites for windows ........ 10
2.2.4 Build Keywriter Certificates ........ 10
2.2.5 Build the example: ........ 10
3 Keywriter Certificate Generation ........ 12
3.1 Generating an x509 Certificate from a Customer's HSM ........ 12
3.1.1 Flow Diagram for Creating the Keywriter Binary ........ 13
3.2 Reference Commands ........ 14
3.2.1 Incremental Programming ........ 14
3.2.2 Program Everything in One Shot ........ 16
4 Booting and Running the Keywriter ........ 17
4.1 UART Boot ........ 17
4.1.1 For Linux Users: ........ 17
4.1.2 For Windows Users: ........ 17
4.2 OSPI Boot ........ 18
4.2.1 UART Uniflash ........ 18
4.3 USB-DFU Boot ........ 19
4.3.1 For Linux Users: ........ 19
4.3.2 For Windows Users: ........ 19

4.4 Keywriter Logs and Debugging ....... 21

5 Checking Device State (HS-FS/HS-SE) ....... 22

6 Appendix ....... 24

6.1 Keywriter app build flow and directory structure ....... 24

6.2 Creating x509 Certificates for Incremental Programming ....... 24

6.3 OTP Keywriter Logs ....... 25

6.4 X509 Configuration Template ....... 28

6.5 OTP keywriter Validations ....... 31

6.6 Programming USB0 PID VID, MAC Device ID into Extended OTP ....... 32

6.6.1 Examples: ....... 33

6.6.1.1 Program USB VID: ....... 33

6.6.1.2 Program MAC Device ID: ....... 34

- Introduction(see page 5)
  - Supported Key Types and Fields(see page 5)
  - Components(see page 6)
  - VPP requirements(see page 7)
- Setting up the build environment(see page 8)
  - Linux Host(see page 8)
    - Install the MCU+ SDK for AM62X(see page 8)
    - Install keywriter addon package(see page 8)
    - Build the Keywriter Certificates(see page 8)
    - Build the example:(see page 9)
  - Windows Host(see page 9)
    - Install MCU+ SDK for AM62X(see page 9)
    - Install keywriter addon package(see page 10)
    - Prerequisites for windows(see page 10)
    - Build Keywriter Certificates(see page 10)
    - Build the example:(see page 10)
- Keywriter Certificate Generation(see page 12)
  - Generating an x509 Certificate from a Customer's HSM(see page 12)
    - Flow Diagram for Creating the Keywriter Binary(see page 13)
  - Reference Commands(see page 14)
    - Incremental Programming(see page 14)
    - Program Everything in One Shot(see page 16)
- Booting and Running the Keywriter(see page 17)
  - UART Boot(see page 17)
    - For Linux Users:(see page 17)
    - For Windows Users:(see page 17)
  - OSPI Boot(see page 18)
    - UART Uniflash(see page 18)
  - USB-DFU Boot(see page 19)
    - For Linux Users:(see page 19)
    - For Windows Users:(see page 19)
  - Keywriter Logs and Debugging(see page 21)
- Checking Device State (HS-FS/HS-SE)(see page 22)
- Appendix(see page 24)
  - Keywriter app build flow and directory structure(see page 24)
  - Creating x509 Certificates for Incremental Programming(see page 24)
  - OTP Keywriter Logs(see page 25)
  - X509 Configuration Template(see page 28)
  - OTP keywriter Validations(see page 31)
  - Programming USB0 PID VID, MAC Device ID into Extended OTP(see page 32)
    - Examples:(see page 33)
      - Program USB VID:(see page 33)
        - Extended OTP Generation Commands:(see page 34)
          - To Program OTP Word 31(see page 34)
          - To Program OTP Word 30(see page 34)
      - Program MAC Device ID:(see page 34)
        - Extended OTP Generation Commands:(see page 34)
          - To Program OTP Word 31(see page 34)
          - To Program OTP Word 30(see page 34)
          - To Program OTP Word 29(see page 34)

# 1 Introduction

**Note**

- This document is applicable to HS-FS device variants.
- For an in-depth understanding of keywriter, read this document along side the TISCI User Guide - Key Writer[1] documentation.

OTP (One Time Programmable) Keywriter is a software tool for provisioning customer keys into a device's eFuses, primarily for enforcing secure boot and establishing root of trust. Secure boot requires an image to be encrypted (optional) and signed using customer keys[2]. This image will then be verified by the SoC using the active MPK Hash (to verify the signature) and the MEK (for decryption). And in case the customer's MPK and MEK are compromised, there is a backup set of MPK and MEK fields that can be used after updating key revision. However, this set of backup keys are completely optional.

Additionally, in order to burn the active and/or backup customer keys into the SoC, OTP Keywriter requires an x509 format certificate that was signed by the customer keys.

**Important**

**Once the keys are burned into SoC eFuses, there is no going back. This action of burning the keys is irreversible.**

## 1.1 Supported Key Types and Fields

Keywriter supports programming of following key types and fields described in the below table.

| Key | Description | Notes |
|---|---|---|
| BMEK | Backup Manufacturer Encryption Key | 256-bit Customer encryption key for encrypted boot |
| BMPKH | Backup Manufacturer Public Key Hash | BMPK is a 4096-bit customer RSA signing key |
| EXTENDED OTP | (Reserved for future use) Extended OTP array | 1024 bit extended otp array |
| KEYCNT | Key Count | • 2 if both BMPK & SMPK are used<br>• 1 if only SMPK is used<br>• 0 if none is used (HS-FS) |

1 https://software-dl.ti.com/tisci/esd/latest/6_topic_user_guides/key_writer.html
2 http://downloads.ti.com/tisci/esd/latest/6_topic_user_guides/secure_boot_signing.html

| Key | Description | Notes |
|---|---|---|
| KEYREV | Key Revision | This is the revision for the key that is active<br>• 1 for SMPK<br>• 2 for BMPK<br>Can have a maximum value = key count |
| MSV | Model Specific Value | 20 bit value with 12 bit BCH code |
| SMEK | Secondary Manufacturer Encryption Key | 256-bit Customer encryption key for encrypted boot |
| SMPKH | Secondary Manufacturer Public Key Hash | SMPK is a 4096-bit customer RSA signing key |
| SWREV-BOARDCONFIG | Software Revision of the Secure Board Configuration | 124 bit value (62 w/ double redundancy)<br>***DO NOT PROGRAM* - initial values set by TI** |
| SWREV-SBL | Software Revision of the Secondary Bootloader | 96 bit value (48 w/ double redundancy)<br>***DO NOT PROGRAM* - initial values set by TI** |
| SWREV-SYSFW | Software Revision of the System Firmware | 96 bit value (48 w/ double redundancy)<br>***DO NOT PROGRAM* - initial values set by TI** |

## 1.2 Components

OTP Keywriter contains the following:

1. MCU+ SDK keywriter app source code
    a. It boots on the primary boot core of the device without any other bootloader.
    b. It's responsible for sending the OTP configuration to the secure keywriter component, which runs on the security subsystem, to blow the keys into the device's eFuses.
    c. It contains an **x509 certificate** configuration template, as well as a **generation script** which can be used as-is with the customer’s keys, or integrated within the customer’s HSM environment.
2. Encrypted TIFS keywriter binary and certificate
    a. It's the secure OTP keywriter firmware which runs on the secure subsystem of the device.
    b. It's responsible for verifying the OTP configuration from the x509 certificate and blowing the data into the efuses.

> **Note**
>
> This binary differs from the standard TIFS binary used in production.

3. TI FEK (public) key - TI Field Encryption Key
    a. This is required by the certificate generation script as a means to ensure confidentiality of the customer's key material.
    b. The public key will be used to encrypt content
    c. The private key (not shared, but part of encrypted TIFS keywriter binary) will be used to decrypt the binary
        i. This is done at runtime - inside of the secure subsystem

## 1.3 VPP requirements

1. The VPP pin of the SoC must be powered (1.8v) when the efuses are being programmed
    a. Refer to the device's datasheet for details on VPP requirements.
    b. While designing boards, make sure to provision hardware, using headers/switches, or software, via GPIO/PMIC, to enable/disable VPP
2. An AM62X SK controls VPP via GPIO pin of the IO expander.
    a. The IO expander pin is controlled via I2C.
    b. Refer to the example keywriter applications to understand how VPP is enabled by the IO expander and I2C.
3. Due to the unique design of all boards, it is the responsibility of the user to enable VPP.
    a. Keywriter application contains a file called `board.c` which can be referenced for how to implement this VPP control.
    b. This control must be implemented to meet all requirements.

# 2 Setting up the build environment

## 2.1 Linux Host

### 2.1.1 Install the MCU+ SDK for AM62X

**Note:**

1. This step is optional for users who already have version 8.6 of the MCU+ SDK (or greater).
2. Users of the linux SDK must install the MCU+ SDK in order to use keywriter.

The keywriter example app works as an add-on for the MCU+ SDK, as it requires some of its libraries.

1. Download and install latest MCU+ SDK release for AM62X from link.
    - MCU-PLUS-SDK-AM62X[3]
    - Select "MCU PLUS SDK Linux Installer" for Linux hosts.
2. Download and install following dependent packages from the same link.
    - CCS[4] 12.2.0
    - SYSCONFIG[5] 1.15
3. Ideally, the tools listed above can be installed anywhere in the system, but we recommend installing them at the default location: ~/ti/
    - We will be referring to this location as <MCU_PLUS_SDK_INSTALL_DIR>

### 2.1.2 Install keywriter addon package

1. Download the keywriter add-on package from mySecureSW.
    a. It will be a .run file
2. Install the add-on package at `<MCU_PLUS_SDK_INSTALL_DIR>/source/security`.
    a. This will create a folder named "sbl_keywriter" upon successful installation.

### 2.1.3 Build the Keywriter Certificates

The following is an example of generating a certificate with MSV data. For information on how to generate certificates for other fields, see section 3.

Before starting on the steps below, make sure openssl and python3 are installed on your system.

Make sure openssl and python3 are installed on your system before following below steps.

1. Go to the following directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/scripts/cert_gen/am62x`

3 https://www.ti.com/tool/download/MCU-PLUS-SDK-AM62X/
4 https://www.ti.com/tool/CCSTUDIO
5 https://www.ti.com/tool/SYSCONFIG

2. Execute the following: `./gen_keywr_cert.sh --msv 0xC0FFE -t tifek/ti_fek_public.pem`
    a. This will generate a certificate with MSV data at `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/scripts/x509cert/final_certificate.bin`
3. Convert the certificate binary to .h format.
    a. Go to the x509 script directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/scripts/x509cert`
    b. Run: `python3 ../../../../../tools/bin2c/bin2c.py final_certificate.bin keycert.h KEYCERT`
    c. This will generate a C header file called keycert.h
        i. Make sure keycert.h is placed inside script/x509cert folder, as it will need to be included for the build

### 2.1.4 Build the example:

1. Go to the following directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/am62x-sk/r5fss0-0_nortos/ti-arm-clang`
2. Clean the example: `make -sj clean PROFILE=debug`
3. Then run: `make -sj PROFILE=debug`
    a. This will build the example and generate tiboot3.bin in the same location.
4. Use the tiboot3.bin to boot the application using the supported boot modes. (Refer to section 4)

## 2.2 Windows Host

### 2.2.1 Install MCU+ SDK for AM62X

**Note:**

This step is optional for users who already have version 8.6 of the MCU+ SDK (or greater).

The keywriter example app works as an add-on for the MCU+ SDK, as it requires some of its libraries.

1. Download and install the latest MCU+ SDK release for AM62X.
    - MCU-PLUS-SDK-AM62X[6]
    - Select "MCU PLUS SDK Windows Installer" for Windows hosts.
2. Download and install the following dependent packages from the same link.
    - CCS[7] 12.2.0
    - SYSCONFIG[8] 1.15
3. Ideally, the tools listed above can be installed anywhere in the system, but we recommend installing them at the default location: c/ti/
    a. We will be referring to this location as <MCU_PLUS_SDK_INSTALL_DIR>

6 https://www.ti.com/tool/download/MCU-PLUS-SDK-AM62X/
7 https://www.ti.com/tool/CCSTUDIO
8 https://www.ti.com/tool/SYSCONFIG

### 2.2.2 Install keywriter addon package

1. Download the keywriter add-on package from mySecureSW.
   a. It will be an .exe file
2. Install the add-on package at `<MCU_PLUS_SDK_INSTALL_DIR>/source/security`.
   a. This will create a folder named "sbl_keywriter" upon successful installation.

### 2.2.3 Prerequisites for windows

- Download and install **git bash** terminal for windows from the link: https://git-scm.com/download/win
- Set up **gmake** by referring to the MCU+ SDK documentation about make[9]: Developer Guides → Using SDK with Makefiles → Enabling "make" in Windows.

### 2.2.4 Build Keywriter Certificates

The following is an example of generating a certificate with MSV data. For information on how to generate certificates for other fields, see section 3.

⚠ Before starting on the steps below, make sure openssl and python3 are installed on your system.

Run the following commands in a **git bash** terminal.

1. Go to the following directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/scripts/cert_gen/am62x`
2. Execute the following: `./gen_keywr_cert.sh --msv 0xC0FFE -t tifek/ti_fek_public.pem`
   a. This will generate a certificate with MSV data at `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/scripts/x509cert/final_certificate.bin`
3. Convert certificate binary to .h format.
   a. Go to the x509 script directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/scripts/x509cert`
   b. Run: `python3 ../../../../../tools/bin2c/bin2c.py final_certificate.bin keycert.h KEYCERT`
   c. This will generate a C header file called keycert.h
      i. Make sure keycert.h is placed inside script/x509cert folder, as it will need to be included for the build

### 2.2.5 Build the example:

1. Go to the following directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/am62x-sk/r5fss0-0_nortos/ti-arm-clang`

9 https://software-dl.ti.com/mcu-plus-sdk/esd/AM64X/08_02_00_08/exports/docs/api_guide_am64x/MAKEFILE_BUILD_PAGE.html#autotoc_md175

2. Clean the example: `gmake -sj clean PROFILE=debug`
3. The run: `gmake -sj PROFILE=debug`
   a. This will build the example and generate tiboot3.bin in the same location.
4. Use the tiboot3.bin to boot the application using the supported boot modes. (Refer to section 4)

# 3 Keywriter Certificate Generation

Invoke gen_keywr_cert.sh with -h for additional help.
./gen_keywr_cert.sh -h

## 3.1 Generating an x509 Certificate from a Customer's HSM

The customer keys are supposed to be private, rather than be distributed in the open. For the purpose of this document, **Customer HSM** stands for *the secure server/system* where the customer generates the x509 certificate. Once the certificate is generated, it means the customer key information is encrypted, and thus it is safe to share the certificate in the open without revealing the actual keys.

The customer keys consist of both the public key hashes (xMPKH) as well as the secret/secure symmetric keys (xMEK). In order to maintain security, the symmetric keys are encrypted on the secure server (referred to here as Customer HSM), so that the resulting x509 certificate can be exported and embedded in the keywriter binary without exposing the secret keys.

**Note**

OpenSSL (1.1.1 11 Sep 2018) or later(1.1.1 series) is required for building the OTP Keywriter.

- You can check if OpenSSL is installed by typing “openssl version” in your command prompt.
    - If it's not installed, download and install OpenSSL for your OS.
- For Windows:
    - The easiest way is to download and install it is by using Strawberry Perl[10].
        - The Strawberry Perl installer automatically installs and sets up OpenSSL.
        - Alternatively, users can also use any of these Third Party OpenSSL Distributions[11] for Windows. Refer to individual links for instructions on how to setup OpenSSL using a particular distribution.
- For Linux:
    - Execute the command `sudo apt-get install openssl` within a linux command prompt.

**Note**

OTP keywriter firmware supports a maximum certificate length of 5400 bytes.

- This is a limit for individual certificates -- not the `x509cert/final_certificate.bin`.
- Make sure to check the size of `primary_cert.bin` and `seconday_cert.bin` (optional depending on BMPK).
    - If certificate length exceeds the limit, the keys can be programmed using an incremental programming approach instead.

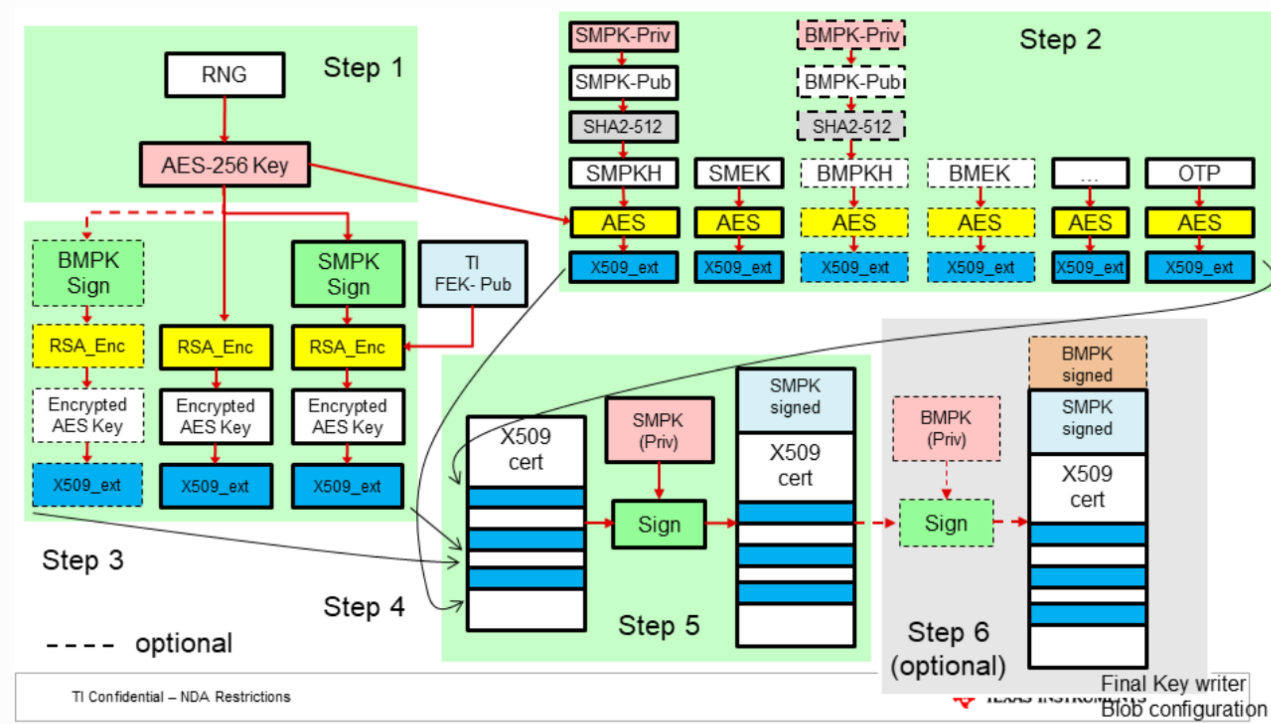
The **scripts** folder contains the necessary tools for generating an x509 certificate for eFuse programming. It can be copied to a secure server (Customer HSM) where the customer keys are stored, and used to generate the x509 certificate.

- `./gen_keywr_cert.sh -g`

10 http://strawberryperl.com/download/5.28.0.1/strawberry-perl-5.28.0.1-64bit.msi
11 https://wiki.openssl.org/index.php/Binaries

- This will generate random keys in the `keys/` directory that can be used for testing.
- `gen_keywr_cert.sh` will also generate a SHA-512 hash of the SMPK, SMEK, BMPK, and the BMEK, and store them in `verify_hash.csv`.
- M4 UART logs will also print out the SHA-512 hash of the keys.
    - `verify_hash.csv` can be used for quick comparison with M4 logs.

### 3.1.1 Flow Diagram for Creating the Keywriter Binary




1. The OEM generates a random 256-bit number to be used as an AES encryption key for protecting the OTP data.
2. The AES-256 key from step 1 is used to encrypt all x509 extension fields that require encryption protection.
3. Next, x509 extensions are created from the following:
    - Encrypting the AES-256 key with the TI FEK.
    - Signing the AES-256 key with the SMPK, and encrypting that with the TI FEK.
    - (Optional) Signing the AES-256 key with the BMPK, and encrypting that with the TI FEK (refer to step 6)
4. All of the x509 extensions from steps 1-3 are combined into an x509 configuration.
5. This x509 configuration is signed using SMPK (priv).
6. (Optional) If the OEM chooses to write the BMPK/BMEK fields, the x509 configuration from step 5 needs to be signed using BMPK (priv).

## 3.2 Reference Commands

### 3.2.1 Incremental Programming

**Warning**

Write protecting the software revision of SYSFW and SBL field will also write protect the software revision of BCFG field and the key revision field since these three fields share a common row on AM62X.

- If the software revision fields and key revision field needs to be write protected (***NOT RECOMMENDED FOR FUTURE UPDATES***), they must be written together in one of the following ways:
    a. During the last step of incremental programming using "key revision write protect".
    b. Or by using "key revision write protect", during the one-shot programming step.
- If the software revision fields or key revision field is write protected, you will not be able to update the revisions in future.

The current software revision of SYSFW, SBL and BCFG is already 1. These fields do not need to be programmed now. ***DO NOT PROGRAM* - initial values set by TI**

**Note**

- Each command in the table below results in a fresh boot of the device.
    - In other words, with each command we program certain fields, build a new keywriter app with a new certificate, and reboot the device.
- Until the KEYREV value is set to either 1 or 2, the device is considered an HS-FS device, and key values can continue being programmed incrementally.
    - This allows key programming to be done in multiple passes.
- However, once the KEYREV value is set to 1 or 2, the device becomes an HS-SE device, and the OTP keywriter application will no longer boot since the user root key has now taken over as the root of trust.
    - So, programming the KEYREV should be left to the final step.

| Command | Description |
|---|---|
| ./gen_keywr_cert.sh --msv 0xC0FFE -t tifek/ti_fek_public.pem --msv-wp | • Generates certificate with MSV value = 0xC0FFE for programming<br>• --msv-wp is optional, and should only be passed if the row needs to be write protected |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --sr-sbl 1 --sr-sysfw 1 --sr-sysfw-ovrd --sr-sbl-ovrd | • Generates a certificate for setting the SBL software revision to 1, and the system-firmware software revision to 1 (can be skipped if software revision update is not required) |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --sr-bcfg 1 --sr-bcfg-ovrd | • Generates a certificate for setting the board configuration software revision to 1 (can be skipped if software revision update is not required) |

| Command | Description |
| --- | --- |
| ./construct_ext_otp_data.sh -extotp 0x80000001 -indx 0 -size 32<br>./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-indx 0 --ext-otp-size 32 --extotp-wprp 00000000000000010000000000000000 | • Generates a certificate for configuring extended OTP with index 0 and size 32<br>• --ext-otp-wprp is optional, and should only be passed if the row needs to be write protected |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem -b keys_devel/bmpk.pem --bmek keys_devel/bmek.key -b-wp --bmek-wp | • Generates a certificate to program the BMPK and BMEK<br>• The keys are taken from the keys_devel folder.<br>    • To program your own keys, replace this with the path of your keys<br>• -b-wp, -bmek-wp is optional, and should only be passed if the row needs to be write protected |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem -b-def --bmek-def -b-wp --bmek-wp | • Generates a certificate to program the default BMPK and BMEK<br>• The keys are taken from the keys_devel folder.<br>• -b-wp, -bmek-wp is optional, and should only be passed if the row needs to be write protected<br>• This step is optional and should only be done if previous step is not executed |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem -s keys_devel /smpk.pem --smek keys_devel/smek.key -s-wp --smek-wp | • Generates a certificate to program the SMPK and SMEK<br>• The keys are taken from the keys_devel folder.<br>    • To program your own keys, replace this with the path of your keys<br>• -s-wp, -smek-wp is optional, and should only be passed if the row needs to be write protected |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem -s-def --smek-def -s-wp --smek-wp | • Generates a certificate to program the default SMPK and SMEK<br>• The keys are taken from the keys_devel folder.<br>• -s-wp, -smek-wp is optional, and should only be passed if the row needs to be write protected<br>• This step is optional and should only be done if previous step is not executed |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --keycnt 2 --keycnt-wp | • Generates a certificate for setting the key count to 2<br>• -keycnt-wp is optional, and should only be passed if the row needs to be write protected |

| Command | Description |
| --- | --- |
| ./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --keyrev 1 | • Generates a certificate for setting the program key revision to 1<br>  • This device now becomes HS-SE and enforces secure booting.<br>  • **Note**: To add write protection to the software revisions and the key revision (not recommended), --keyrev-wp can be added in this step |

### 3.2.2 Program Everything in One Shot

```
./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --msv 0xC0FFE -b keys_devel/bmpk.pem
--bmek keys_devel/bmek.key -s keys_devel/smpk.pem --smek keys_devel/smek.key --keycnt
2 --keyrev 1
```

- Above is the example command given for programming multiple fields at once (in one boot).
  - The resulting certificate given will program the MSV, SMPK, SMEK, BMPK, BMEK, key count, and key revision all at once.
  - After a reboot or a power on reset, the device will become an HS-SE device and enforce secure booting.
- One thing to keep in mind for this method is that the certificate should be less than 5400 bytes.
  - If the certificate does not fit within 5400 bytes, use the incremental method instead.
- Additionally, if the extended OTP needs to be programmed via keywriter (for USB/PCIE VID/PID), make sure to program the extended OTP before converting the device to an HS-SE device.

```
./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --msv 0xC0FFE -b-def -bmek-def -s-def
-smek-def --keycnt 2 --keyrev 1
```

The above command programs everything at once with all dummy keys.

# 4 Booting and Running the Keywriter

This section provides the steps for UART, OSPI and USB-DFU boot modes for an AM62x SK. However, it is recommended to go through the SDK documentation for more details on the SoC boot procedures. In total though, the keywriter application has been validated with the following boot modes:

- AM62x SK
    - UART
    - OSPI
    - USB-DFU

## 4.1 UART Boot

SW2 SW1

UART BOOT MODE

### 4.1.1 For Linux Users:

Use the **uart_bootloader.py** python script that's provided in the sbl_keywriter/tools directory. But before doing so, make sure you have the necessary python dependencies already installed.

1. Setup up the SK for UART boot mode by configuring the boot mode switches.
2. Go to the following directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/tools`
3. Run: `python3 uart_bootloader.py -p "/dev/ttyUSB0" -b  <path_to_tiboot3.bin>`
    a. Update -p <port_number> and -b <binary_path> as necessary

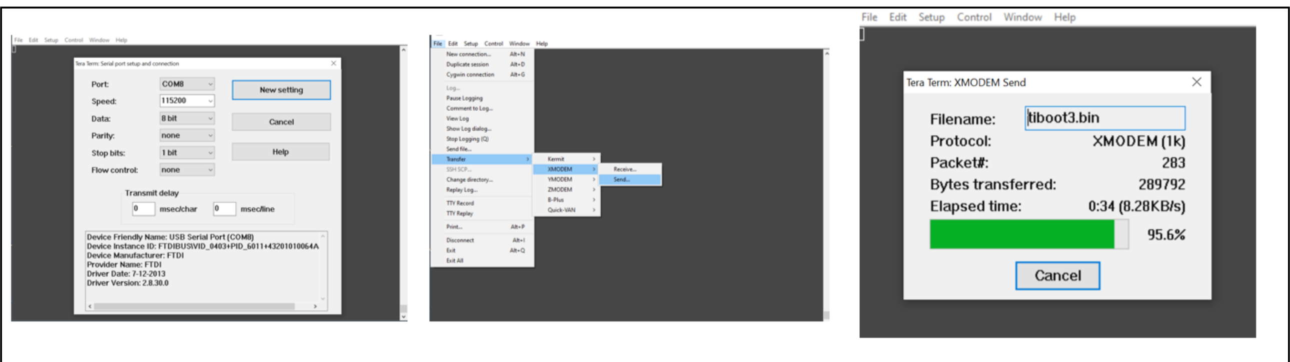
### 4.1.2 For Windows Users:

The uart_bootloader.py python script mentioned above can also be used on windows if you have all of it's dependencies installed.

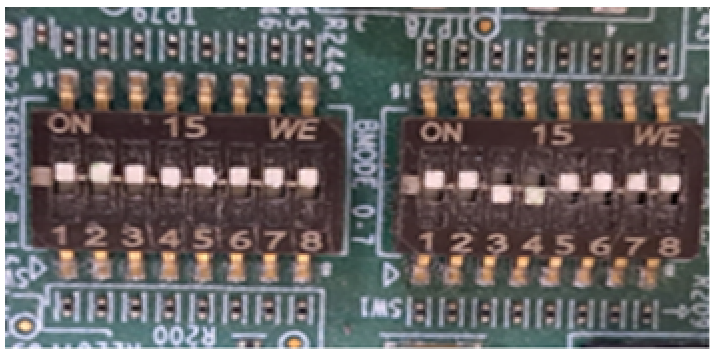
An alternative and quick way to boot on windows is to use Tera Term as shown below:

1. Download and install Tera Term from the following link:
    a. https://ttssh2.osdn.jp/index.html.en
2. Set up the serial port by selecting Setup → Serial Port.

   a. Select an appropriate COM port.
   b. Copy the settings shown in the picture below.
3. As shown in the picture below, send the tiboot3.bin binary over XMODEM by selecting File → Transfer → XMODEM → Send




## 4.2 OSPI Boot




SW2 SW1

OSPI BOOT MODE

The MCU+ SDK currently supports Flash Writer. Flashing the binary can be done using the UART uniflash as described below.

Once flashing is complete:

1. Set up the SK in OSPI boot mode by reconfiguring the boot mode switches.
2. Then, reset the SoC
3. And boot from OSPI memory

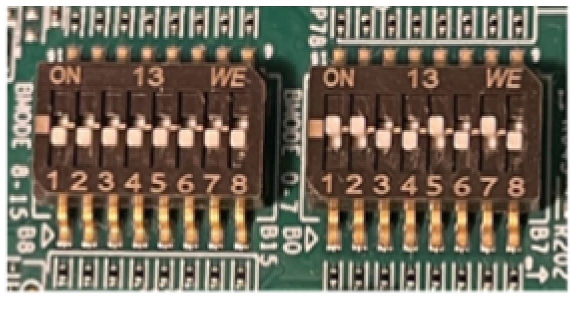
### 4.2.1 UART Uniflash

Flashing is achieved using the SDK's sbl_uart_uniflash, as shown below. Additionally, example source code can be found at <MCU_PLUS_SDK_INSTALL_DIR>/examples/drivers/boot/sbl_uart_uniflash

1. Set up the SK for UART boot mode by configuring the boot mode switches.
2. Go to the following directory: <MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/tools/hsfs_flashwriter
3. Run the uart_uniflash.py script to flash the keywriter binary onto flash memory.
   - AM62X-SK:

    - python3 ../../../../../tools/boot/uart_uniflash.py -p /dev/ttyUSB0 --cfg=am62x/keywr_sbl_am62.cfg
- AM62X-SK-LP:
    - AM62-SK-LP: python3 ../../../../../tools/boot/uart_uniflash.py -p /dev/ttyUSB0 --cfg=am62x/keywr_sbl_am62q.cfg

## 4.3 USB-DFU Boot




### 4.3.1 For Linux Users:

Follow the steps below to send files using the USB-DFU boot mode:

1. Install USB-DFU:
    a. Run **sudo apt-get install dfu-util** in your ubuntu PC.
2. Setup up the SK for USB-DFU boot mode by configuring the boot mode switches.
3. Connect the SK (running DFU) to the PC with a USB cable
4. Verify the installation and device detection:
    a. Run **sudo dfu-util -l**

```
Found DFU: [0451:6165] ver=0200, devnum=114, cfg=1, intf=0, path="1-10.2", alt=1, name="SocId", serial="01.00.00.00"
Found DFU: [0451:6165] ver=0200, devnum=114, cfg=1, intf=0, path="1-10.2", alt=0, name="bootloader", serial="01.00.00.00"
```

5. Send the boot binary:
    a. Run **sudo dfu-util -c 1 -i 0 -a bootloader -D <path_to_tiboot3.bin>**
        i. Update -a <alt_number> -D <binary_path> as necessary.

### 4.3.2 For Windows Users:

An alternative method for using USB-DFU boot mode on windows is shown below:

1. Download the dfu-util tool (dfu-util-0.6-win32.zip.bz2) from the following:
    a. http://dfu-util.sourceforge.net/releases/
    b. Then unzip to a designated folder (say c:\dfu)
2. Download the zadig application from the following:
    a. https://sourceforge.net/projects/libwdi/files/zadig/
    b. Then unzip and execute the zadig application.
    c. Also make sure to install the winUSB driver.
3. Connect the SK (running DFU) to the windows PC with a USB cable.
    a. Windows host will detect the SK as "DFU Gadget".

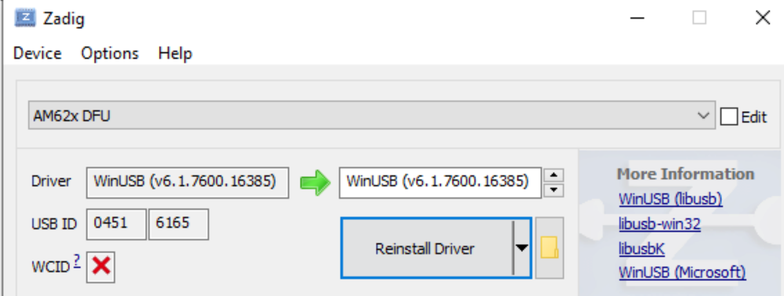
i. In devices, "am62x DFU" can be seen.

4. Open the command prompt and go to the directory containing the dfu-util.exe (say the c:\dfu folder).
5. Run **dfu-util -l**
   a. One of the two devices should be listed now

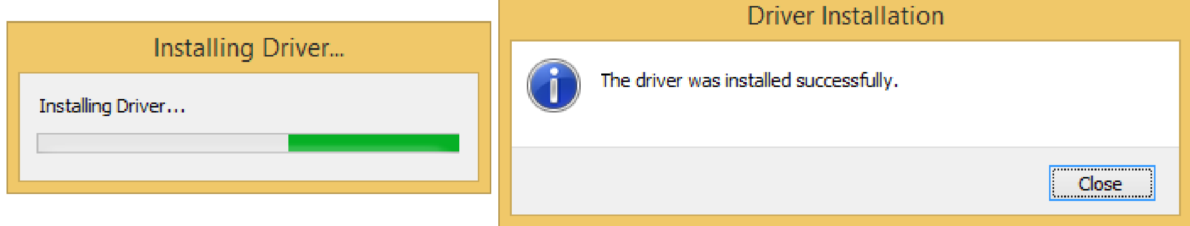
```
Found DFU: [0451:6165] devnum=0, cfg=1, intf=0, alt=0, name="bootloader"
Found DFU: [0451:6165] devnum=0, cfg=1, intf=0, alt=1, name="SocId"
```

```
Found DFU: [0451:6165] devnum=0, cfg=1, intf=0, alt=0, name="UNDEFINED"
Found DFU: [0451:6165] devnum=0, cfg=1, intf=0, alt=1, name="UNDEFINED"
```

6. Go to the zadig application and select “List All Devices” under the “Options” dropdown menu.
   a. "AM62x DFU" will be listed




7. Click on "Install Driver".




8. Now run one of the following commands:
   a. **dfu-util -c 1 -i 0 -a 0 -D <path to tiboot3.bin>**
   b. **dfu-util -c 1 -i 0 -a bootloader -D <path to tiboot3.bin>**
      i. Update -a <alt_number> and -D <binary_path> as necessary.

1. Use -a bootloader if name gets identified as shown in 2nd image of 5th step.

```
Opening DFU capable USB device... ID 0451:6165
Run-time device DFU version 0110
Found DFU: [0451:6165] devnum=0, cfg=1, intf=0, alt=0, name="bootloader"
Claiming USB DFU Interface...
Setting Alternate Setting #0 ...
Determining device status: state = dfuIDLE, status = 0
dfuIDLE, continuing
DFU mode device DFU version 0110
Device returned transfer size 512
No valid DFU suffix signature
Warning: File has no DFU suffix
bytes_per_hash=5530
Copying data from PC to DFU device
Starting download: [##################################################] finished!
state(6) = dfuMANIFEST-SYNC, status(0) = No error condition is present
state(2) = dfuIDLE, status(0) = No error condition is present
Done!
```

## 4.4 Keywriter Logs and Debugging

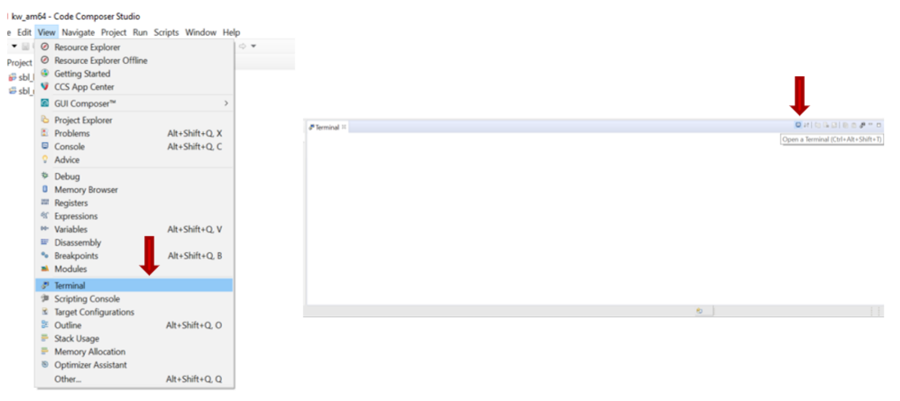
- Once the keywriter application is booted, the R5 SBL and M4 logs can be collected on UART0 and UART1 respectively.
  - M4 logs can also be read from trace memory buffer location[12].
  - See the appendix for example R5 and M4 logs.
- An LED will be used for indicating whether or not VPP is set to high during key programming.
  - A yellow LED(LD 11) will glow on the SK
- For additional help with blowing eFuses, the `debug_response` from TISCI_MSG_KEY_WRITER[13] has additional information.
  - The enumeration documented here[14] lists out the bit positions which are used by the `debug_response` for debugging the specific key programming issue.
  - A `debug_response` of 0x00 means programming was successful.

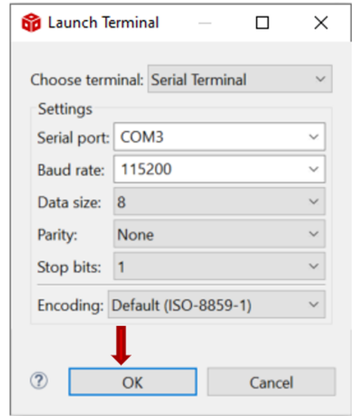
12 https://software-dl.ti.com/tisci/esd/latest/4_trace/trace.html#trace-memory-buffer-location
13 http://downloads.ti.com/tisci/esd/latest/2_tisci_msgs/security/keywriter.html#sec-api-keywr-program-keys
14 http://downloads.ti.com/tisci/esd/latest/2_tisci_msgs/security/keywriter.html#sec-api-keywr-program-keys

# 5 Checking Device State (HS-FS/HS-SE)

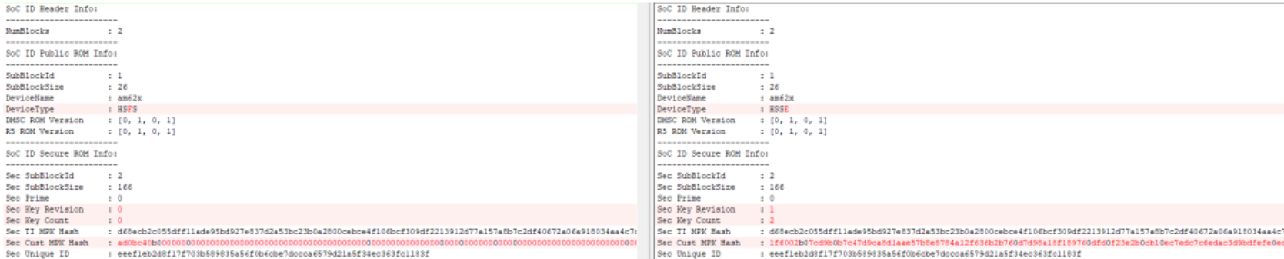
When UART boot mode is selected, boot ROM reports the SoC ID via UART, for both GP and HS devices. Additionally, this can determine the device configuration, which can be used to check if device was converted from HS-FS to HS-SE after key programming. And to make the output easier to read, the **parse_uart_boot_socid.py** parser can be used to convert the hexadecimal numbers into human readable text. Below are the steps for using this parser:

- Set the boot mode switches into UART boot mode.
- Open CCS and select view → terminal.
- Open the launch terminal window and select "Serial Terminal" for the R5 UART port.
    - COM port on windows
    - /dev/ttyUSBxx on linux.
- Use the rest of the default settings, and press OK.






- Reset the SoC by pressing the reset button.
    - The terminal will print the SoC ID followed by "CCC" (ping characters)
- Copy the entire string up until "CCC" (ping characters)
- Paste the string in `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/tools/socid.txt`
- Run the command below in the following directory: `<MCU_PLUS_SDK_INSTALL_DIR>/source/security/sbl_keywriter/tools`
    - $ `python parse_uart_boot_socid.py socid.txt`
        - This will parse the SoC ID and show the information pictured below.



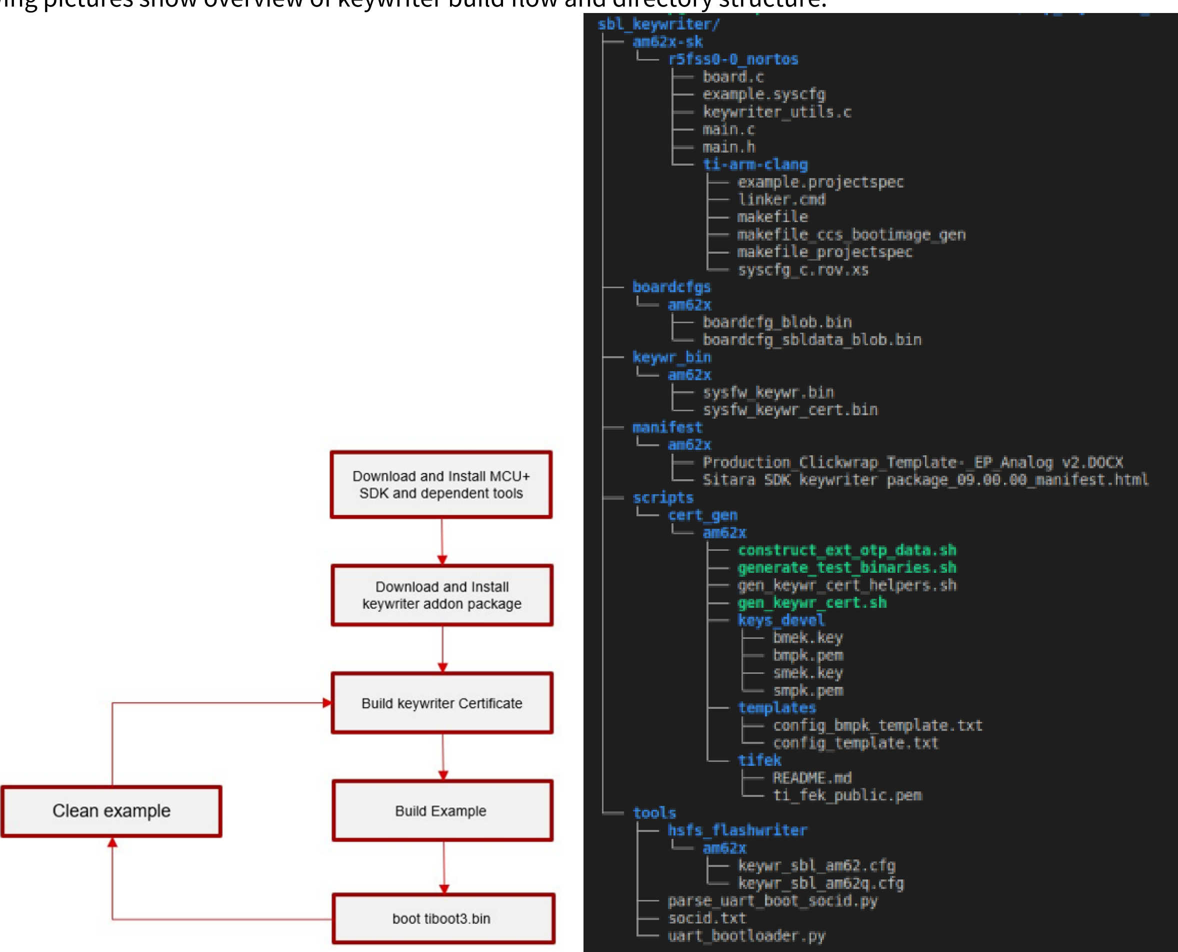

Pictured on the left is the parsed SoC ID of an HS-FS device, while pictured on the right is the SoC ID of an HS-FS device that was converted to an HS-SE device after programming the keys with the keywriter application. This conversion is verified as successful because of the new values of the Cust MPK Hash, Key Count and Key Revision seen on right.

Additionally, to confirm secure boot functionality on HS-SE devices, check out the HS boot support sections within the MCU+ SDK documentation (Developer Guides → Enabling Secure Boot), as well as the boot example/applications.

# 6 Appendix

## 6.1 Keywriter app build flow and directory structure

Following pictures show overview of keywriter build flow and directory structure.




## 6.2 Creating x509 Certificates for Incremental Programming

**Pass 1**

When supplied with an index and bit size, OTP keywriter can program an extended OTP array. However, both index and size have to be a multiple of 8.

Here's how to construct an extended OTP array

```
# Step 1: Generate an array called ext_otp_data.bin
./construct_ext_otp_data.sh -extotp 0xFF -indx 0 -size 8

# Step 2: Edit ext_otp_data.txt with any additional changes
vim ext_otp_data.txt

# Step 3: Convert the .txt file into .bin file
```

```
xxd -r -p ext_otp_data.txt > ext_otp_data.bin

# Setp 4: Generate the x509 certificate
./gen_keywr_cert.sh -t ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-
indx 0 --ext-otp-size 8
```

**Note**

Steps 2 and 3 can be skipped if no changes are required for the extended OTP array.
Additionally, we could program another offset of "ext_otp" after the certificate is created, the example keywriter app is built, and ext_otp[0:7] is programmed.

**Pass 2**

```
./construct_ext_otp_data.sh -extotp 0x1234 -indx 8 -size 16
./gen_keywr_cert.sh -t ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-
indx 8 --ext-otp-size 16 --ext-otp-wprp 00000000000000010000000000000000
```

**Note**

This will create another certificate. The keywriter example app needs to be built after every change in certificate. This step can be repeated, so long as there is no overlap in the "ext_otp" index/size parameters that are used.
Additionally, the index and size arguments should be the same in both commands.

**Pass 3**

Since the KEYREV value has not been modified until now, we can still use OTP keywriter to program the other keys.

The following code snippet sets the MSV, BMPKH, SMPKH, KEYCNT and KEYREV.

**Note**

It is recommended to write protect the SMPKH, SMEK, BMPKH, BMEK eFuse rows by running the `gen_keywr_cert.sh` script with the necessary arguments. It is also recommended to write protect key count eFuse rows after 2 sets of keys have been programmed.

```
./gen_keywr_cert.sh -s keys/smpk.pem -s-wp
--smek keys/smek.key --smek-wp -t ti_fek_public.pem
-a keys/aes256.key --msv 0xC0FFE --msv-wp --keycnt 1 --keyrev 1
```

- KEYCNT should match the number of keys programmed into the eFuses.
- KEYREV should always be less than or equal to the KEYCNT

## 6.3 OTP Keywriter Logs

Below logs were captured for incremental mode backup keys programming.

**R5 Log**

```
Starting Keywriting
Enabled VPP
Using keys Certificate found: 0x43c14900
Keywriter Debug Response: 0x0
Success Programming Keys
```

**M4 Log**

```
0x409031
0x800023
#
# Decrypting extensions..
#
MPK Options:  0x21
MEK Options:  0x1
MPK Opt P1:  0x1
MPK Opt P2:  0x1
MEK Opt   :  0x1
SMPKH extension disabled
SMEK extension disabled
* BMPKH Part 1 BCH code: 8180dd66

* BMPKH Part 2 BCH code: 81873fe5

* BMPK Hash (part-1,2):

384be278a7a50eb25afdffac2e8bd306f82a3b51a770f8056c9ddeb9f31b0d3d01

3ea0063e6de3127a47c8a1443fc7e10dadffb51601aeaeb499d607e02874cd8001

* BMEK BCH code: c1bbe2be

* BMEK Hash:
9352f2c8069698ad4a1e6dfb381723ba4a15948a5e00c5ac004f574f194efe66bc701d378b01ec0bf1a36
bef6e7a931d466dbdb38bd2be0aad8afc756aa5ea7a

EXT OTP extension disabled
MSV extension disabled

KEY CNT extension disabled

KEY REV extension disabled

SWREV extension disabled

FW CFG REV extension disabled

* KEYWR VERSION:  0x20000

#
# Programming Keys..
```

```
#

* MSV:
[u32] bch + msv:  0x8BAC0FFE
MSV extension disabled
[u32] bch + msv:  0x8BAC0FFE

* SWREV:
[u32] SWREV-SBL:  0x1
[u32] SWREV-SYSFW  :  0x1
SWREV extension disabled
[u32] SWREV-SBL:  0x1
[u32] SWREV-SYSFW  :  0x1

* FW CFG REV:
[u32] SWREV-FW-CFG-REV:  0x1
SWREV SEC BCFG extension disabled
[u32] SWREV-FW-CFG-REV:  0x1

* EXT OTP:
EXT OTP extension disabled

* BMPKH, BMEK:
Programmed 11/11 rows successfully
Programmed 2/2 rows successfully
Programmed 11/11 rows successfully
Programmed 2/2 rows successfully
WP: bl: 1, r: 44, prot: 4
WP: bl: 1, r: 45, prot: 4
WP: bl: 1, r: 46, prot: 4
WP: bl: 1, r: 47, prot: 4
WP: bl: 1, r: 48, prot: 4
WP: bl: 1, r: 49, prot: 4
WP: bl: 1, r: 50, prot: 4
WP: bl: 1, r: 51, prot: 4
WP: bl: 1, r: 52, prot: 4
WP: bl: 1, r: 53, prot: 4
WP: bl: 1, r: 54, prot: 4
WP: bl: 1, r: 55, prot: 4
WP: bl: 1, r: 56, prot: 4
WP: bl: 1, r: 57, prot: 4
WP: bl: 1, r: 58, prot: 4
WP: bl: 1, r: 59, prot: 4
WP: bl: 1, r: 60, prot: 4
WP: bl: 1, r: 61, prot: 4
WP: bl: 1, r: 62, prot: 4
WP: bl: 1, r: 63, prot: 4
WP: bl: 1, r: 64, prot: 4
WP: bl: 1, r: 65, prot: 4
WP: bl: 1, r: 66, prot: 4
WP: bl: 1, r: 67, prot: 4
WP: bl: 1, r: 68, prot: 4
Programmed 11/11 rows successfully
```

```
Programmed 2/2 rows successfully
WP: bl: 1, r: 85, prot: 4
WP: bl: 1, r: 86, prot: 4
WP: bl: 1, r: 87, prot: 4
WP: bl: 1, r: 88, prot: 4
WP: bl: 1, r: 89, prot: 4
WP: bl: 1, r: 90, prot: 4
WP: bl: 1, r: 91, prot: 4
WP: bl: 1, r: 92, prot: 4
WP: bl: 1, r: 93, prot: 4
WP: bl: 1, r: 94, prot: 4
WP: bl: 1, r: 95, prot: 4
WP: bl: 1, r: 96, prot: 4
WP: bl: 1, r: 97, prot: 4

* SMPKH, SMEK:
SMPKH extension disabled
SMEK extension disabled

* KEYCNT:
[u32] keycnt:  0x0
KEY CNT extension disabled
[u32] keycnt:  0x0

* KEYREV:
[u32] keyrev:  0x0
KEY REV extension disabled
[u32] keyrev:  0x0
```

## 6.4 X509 Configuration Template

The following x509 configuration template is used by `gen_keywr_cert.sh` to create a new x509 certificate. Each key has an extension field (OID 1.3.6.1.4.1.294.1.64 - 1.3.6.1.4.1.294.1.81), which contains information such as the SHA-512 value, size, IV, or RS used in the AES encryption.

Details about individual extensions can be obtained from the Security x509 Certificate Documentation[15].

```
[ req ]
distinguished_name = req_distinguished_name
x509_extensions = v3_ca
prompt = no
dirstring_type = nobmp

# This information will be filled by the end user.
# The current data is only a place holder.
# System firmware does not make decisions based
# on the contents of this distinguished name block.
[ req_distinguished_name ]
C = oR
```

15 http://downloads.ti.com/tisci/esd/latest/2_tisci_msgs/security/sec_cert_format.html#extensions

```
ST = rx
L = gQE843yQV0sag
O = dqhGYAQ2Y4gFfCq0t1yABCYxex9eAxt71f
OU = a87RB35W
CN = x0FSqGTPWbGpuiV
emailAddress = kFp5uGcgWXxcfxi@vsHs9C9qQWGrBs.com

[ v3_ca ]
basicConstraints = CA:true
1.3.6.1.4.1.294.1.64 = ASN1:SEQUENCE:enc_aes_key
1.3.6.1.4.1.294.1.65 = ASN1:SEQUENCE:enc_smpk_signed_aes_key
1.3.6.1.4.1.294.1.66 = ASN1:SEQUENCE:enc_bmpk_signed_aes_key
1.3.6.1.4.1.294.1.67 = ASN1:SEQUENCE:aesenc_smpkh
1.3.6.1.4.1.294.1.68 = ASN1:SEQUENCE:aesenc_smek
1.3.6.1.4.1.294.1.69 = ASN1:SEQUENCE:plain_mpk_options
1.3.6.1.4.1.294.1.70 = ASN1:SEQUENCE:aesenc_bmpkh
1.3.6.1.4.1.294.1.71 = ASN1:SEQUENCE:aesenc_bmek
1.3.6.1.4.1.294.1.72 = ASN1:SEQUENCE:plain_mek_options
1.3.6.1.4.1.294.1.73 = ASN1:SEQUENCE:aesenc_user_otp
1.3.6.1.4.1.294.1.74 = ASN1:SEQUENCE:plain_key_rev
1.3.6.1.4.1.294.1.76 = ASN1:SEQUENCE:plain_msv
1.3.6.1.4.1.294.1.77 = ASN1:SEQUENCE:plain_key_cnt
1.3.6.1.4.1.294.1.78 = ASN1:SEQUENCE:plain_swrev_sysfw
1.3.6.1.4.1.294.1.79 = ASN1:SEQUENCE:plain_swrev_sbl
1.3.6.1.4.1.294.1.80 = ASN1:SEQUENCE:plain_swrev_sec_brdcfg
1.3.6.1.4.1.294.1.81 = ASN1:SEQUENCE:plain_keywr_min_version

[ enc_aes_key ]
# Replace PUT-ENC-AES-KEY with acutal Encrypted AES Key
val = FORMAT:HEX,OCT:PUT_ENC_AES_KEY
size = INTEGER:PUT_SIZE_ENC_AES

[ enc_bmpk_signed_aes_key ]
# Replace PUT-ENC-BMPK-SIGNED-AES-KEY with acutal Encrypted BMPK signed AES Key
val = FORMAT:HEX,OCT:PUT_ENC_BMPK_SIGNED_AES_KEY
size = INTEGER:PUT_SIZE_ENC_BMPK_SIGNED_AES

[ enc_smpk_signed_aes_key ]
# Replace PUT-ENC-SMPK-SIGNED-AES-KEY with acutal Encrypted SMPK signed AES Key
val = FORMAT:HEX,OCT:PUT_ENC_SMPK_SIGNED_AES_KEY
size = INTEGER:PUT_SIZE_ENC_SMPK_SIGNED_AES

[ aesenc_smpkh ]
# Replace PUT-AESENC-SPMKH with acutal Encrypted AES Key
val = FORMAT:HEX,OCT:PUT_AESENC_SMPKH
iv = FORMAT:HEX,OCT:PUT_IV_AESENC_SMPKH
rs = FORMAT:HEX,OCT:PUT_RS_AESENC_SMPKH
size = INTEGER:PUT_SIZE_AESENC_SMPKH
action_flags = INTEGER:PUT_ACTFLAG_AESENC_SMPKH

[ aesenc_smek ]
# Replace PUT-AESENC-SMEK with acutal Encrypted AES Key
val = FORMAT:HEX,OCT:PUT_AESENC_SMEK
```

```
iv = FORMAT:HEX,OCT:PUT_IV_AESENC_SMEK
rs = FORMAT:HEX,OCT:PUT_RS_AESENC_SMEK
size = INTEGER:PUT_SIZE_AESENC_SMEK
action_flags = INTEGER:PUT_ACTFLAG_AESENC_SMEK

[ aesenc_bmpkh ]
# Replace PUT-AESENC-BMPKH with acutal Encrypted BMPKH
val = FORMAT:HEX,OCT:PUT_AESENC_BMPKH
iv = FORMAT:HEX,OCT:PUT_IV_AESENC_BMPKH
rs = FORMAT:HEX,OCT:PUT_RS_AESENC_BMPKH
size = INTEGER:PUT_SIZE_AESENC_BMPKH
action_flags = INTEGER:PUT_ACTFLAG_AESENC_BMPKH

[ aesenc_bmek ]
# Replace PUT-AESENC-BMEK with acutal Encrypted BMEK
val = FORMAT:HEX,OCT:PUT_AESENC_BMEK
iv = FORMAT:HEX,OCT:PUT_IV_AESENC_BMEK
rs = FORMAT:HEX,OCT:PUT_RS_AESENC_BMEK
size = INTEGER:PUT_SIZE_AESENC_BMEK
action_flags = INTEGER:PUT_ACTFLAG_AESENC_BMEK

[ plain_msv ]
# Replace PUT-PLAIN-MSV with actual MSV value
val = FORMAT:HEX,OCT:PUT_PLAIN_MSV
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_MSV

[ plain_mpk_options ]
# Replace PUT-PLAIN-MPK-OPT with actual MPK OPT value
val = FORMAT:HEX,OCT:PUT_PLAIN_MPK_OPT
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_MPK_OPT

[ plain_mek_options ]
# Replace PUT-PLAIN-MEK-OPT with actual MEK OPT value
val = FORMAT:HEX,OCT:PUT_PLAIN_MEK_OPT
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_MEK_OPT

[ plain_key_rev ]
# Replace PUT-PLAIN-KEY-REV with actual Key Rev value
val = FORMAT:HEX,OCT:PUT_PLAIN_KEY_REV
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_KEY_REV

[ plain_key_cnt ]
# Replace PUT-PLAIN-KEY-CNT with actual Key Count value
val = FORMAT:HEX,OCT:PUT_PLAIN_KEY_CNT
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_KEY_CNT

[ plain_swrev_sysfw ]
# Replace PUT-PLAIN-SWREV-SYSFW with actual SWREV SYSFW value
val = FORMAT:HEX,OCT:PUT_PLAIN_SWREV_SYSFW
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_SWREV_SYSFW

[ plain_swrev_sbl ]
# Replace PUT-PLAIN-SWREV-SBL with actual SWREV SBL value
```

```
val = FORMAT:HEX,OCT:PUT_PLAIN_SWREV_SBL
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_SWREV_SBL

[ plain_swrev_sec_brdcfg ]
# Replace PUT-PLAIN-SWREV-SEC-BRDCFG with actual SWREV SEC BRDCFG value
val = FORMAT:HEX,OCT:PUT_PLAIN_SWREV_SEC_BRDCFG
action_flags = INTEGER:PUT_ACTFLAG_PLAIN_SWREV_SEC_BRDCFG

[plain_keywr_min_version ]
# Replace PUT-PLAIN-KEYWR-MIN-VER with actual KEYWR-MIN-VER value
val = FORMAT:HEX,OCT:PUT_PLAIN_KEYWR_MIN_VER

[ aesenc_user_otp ]
# Replace PUT-AESENC-USER-OTP with actual Encrypted OTP
val = FORMAT:HEX,OCT:PUT_AESENC_USER_OTP
iv = FORMAT:HEX,OCT:PUT_IV_AESENC_USER_OTP
rs = FORMAT:HEX,OCT:PUT_RS_AESENC_USER_OTP
wprp = FORMAT:HEX,OCT:PUT_WPRP_AESENC_USER_OTP
index = INTEGER:PUT_INDX_AESENC_USER_OTP
size = INTEGER:PUT_SIZE_AESENC_USER_OTP
action_flags = INTEGER:PUT_ACTFLAG_AESENC_USER_OTP
```

## 6.5 OTP keywriter Validations

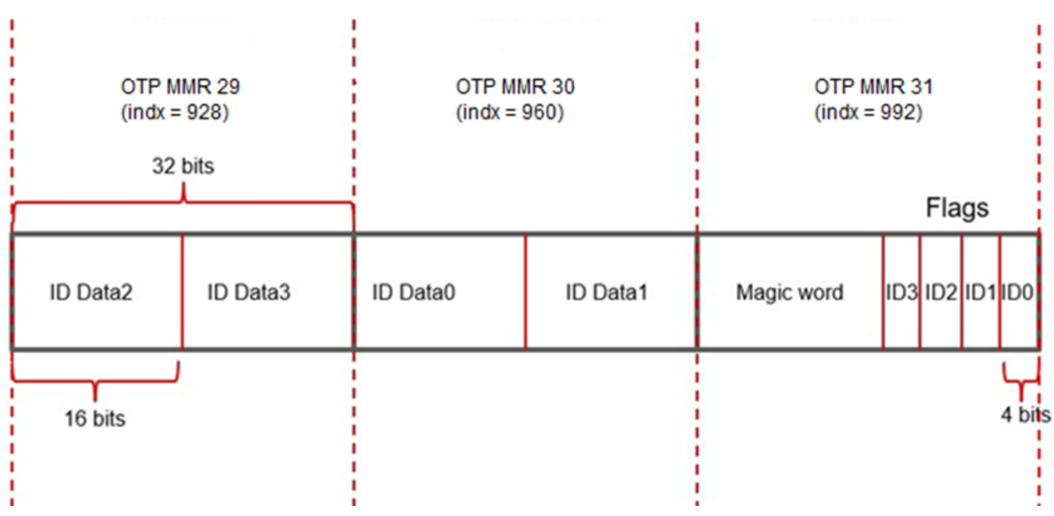
The following is an example of the OTP keywriter validation procedure, which comes from the following script: `<keywriter_dir>/scripts/cer_gen/am62x/generate_test_binaries.sh`

| Pass | Details | Value | Purpose |
|---|---|---|---|
| 1 | Program the MSV | 0xC0FFE | Demonstrates that the MSV can be programmed successfully |
| 2 | (~WP) SWREV (SBL+SYSFW) | SBL: 1, SYSFW: 1 | |
| 3 | (~WP) SWREV (SEC BCFG) | sec-bcfg: 1 | |
| 4 | EXT OTP (31:0) bits (32 bits) | 0x80000001 | Ensures that you can correctly program the ext_otp area with a size larger than 25 bits |
| 5 | EXT OTP (39:32) bits (8 bits) | 0x81 | Ensures that incremental programming is possible for ext otp |
| 6 | (WP) Program BMPKH, BMEK (set as the BMPK dummy key value) | | Dummy keys are programmed |

| Pass | Details | Value | Purpose |
|---|---|---|---|
| 7 | (WP) Program SMPKH, SMEK (set as the SMPK dummy key value) | | Dummy keys are programmed |
| 8 | Program KEYCNT 2 | 2 (0x0303) | Key count programmed |
| 9 | (~WP) Program KEYREV 1 | 1 (0x0101) | After reboot, the device will be an HS-SE device. Additionally, make sure to use SMPK for signing the SBL and board configs |

## 6.6 Programming USB0 PID VID, MAC Device ID into Extended OTP

The diagram below shows ROM's interpretation of the extended OTP region for DEVICE IDs. ROM reads the extended OTP array to determine if the various device IDs (USB VID/PID, MAC Device ID etc.) were programmed into the customer eFuses. If the customer OTP data matches the OTP magic word, then ROM will process the values from the customer OTP eFuses and flags. When processing the data from the customer OTP, ROM will process the flags independent of the DEVSTAT values. Then, ROM will read and update these values in the control MMR. (See the TRM for details on the control MMRs for USB and MAC device IDs)




| | #bits | fields | Description |
|---|---|---|---|

| Mandatory | 16 | Magic word (13 bits) [12:0]<br>Flag (3 bits) [15:13] | • Magic word of **BC8** will indicate whether or not the eFuse is programmed<br>• Flag is used to indicate how many more eFuse bits are required for boot<br>• 000-110 Reserved<br>• 111 - ID Flags only (32 - 96 bits) |
|---|---|---|---|
| Optional | 16 | Flags | Split into 4 independent 4 bit values (ID0 to ID3):<br>• 0000 - unused<br>• 0001 - USB VID<br>• 0010 - USB PID<br>• 0011 - PCIe VID<br>• 0100 - PCIe PID<br>• 0101 - MAC Chunk0<br>• 0110 - MAC Chunk1<br>• 0111 - MAC Chunk2 |
| Optional | 16 | ID Data0 | Content interpreted based on Flags[3:0] |
| Optional | 16 | ID Data1 | Content interpreted based on Flags[7:4] |
| Optional | 16 | ID Data2 | Content interpreted based on Flags[11:8] |
| Optional | 16 | ID Data3 | Content interpreted based on Flags[15:12] |

## 6.6.1 **Examples:**

### 6.6.1.1 Program USB VID:

Magic word with flags = 0xEBC8

Flags = 0x1 (Use ID Data0 for USB VID)

OTP word 31 = 0xEBC8_0001

Since we are using ID0 flags, put VID into ID Data0 (as an example, say we are using 0xA0A0 as VID)

OTP word 30 = 0xA0A0_0000

Extended OTP Generation Commands:

To Program OTP Word 31

./construct_ext_otp_data.sh -extotp 0x0100C8EB -indx 992 -size 32

./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-indx 992 --ext-otp-size 32

To Program OTP Word 30

./construct_ext_otp_data.sh -extotp 0x0000A0A0 -indx 960 -size 32

./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-indx 960 --ext-otp-size 32

#### 6.6.1.2 Program MAC Device ID:

Magic word with flags = 0xEBC8

Flags = 0x5 (Use ID Data0 to put MAC Chunk 0)

Flags = 0x6 (Use ID Data1 to put MAC Chunk 1)

Flags = 0x7 (Use ID Data2 to put MAC Chunk 2)

OTP word 31 = 0xEBC8_0765

As we are using ID0, ID1, ID2 flags, put MAC in these (say we are using 0x8D41A7B9B02C as MAC Device ID)

OTP word 30 = 0xB02C_A7B9

OTP word 29 = 0x8D41_0000

Extended OTP Generation Commands:

To Program OTP Word 31

./construct_ext_otp_data.sh -extotp 0x6507C8EB -indx 992 -size 32

./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-indx 992 --ext-otp-size 32

To Program OTP Word 30

./construct_ext_otp_data.sh -extotp 0xB9A72CB0 -indx 960 -size 32

./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-indx 960 --ext-otp-size 32

To Program OTP Word 29

./construct_ext_otp_data.sh -extotp 0x0000418D -indx 928 -size 32

./gen_keywr_cert.sh -t tifek/ti_fek_public.pem --ext-otp ext_otp_data.bin --ext-otp-indx 928 --ext-otp-size 32

**Note**

For storing user specific data in MMR31, it is recommended to not write the magic word (0xebc8) into the first 16 bits of MMR31. Otherwise, the word placed in MMR30 will be considered part of the USB PID/VID of the device.

**Important**

1. Only USB PID VID or MAC Device ID can be programmed into a device at a time. As per requirement, both cannot be programmed at the same time.
2. Both USB VID and PID should be programmed to a non-zero value in the USB MMR.