Payment Solutions: Secure Boot LAB Guide

Rev. 0.2

SECURE CONNECTIONS FOR A SMARTER WORLD

1 Purpose

This document is provided as a hands-on lab guide for the NXP Technology day in Irvine California. The intent of the lab is to demonstrate how the SLN-POS-RDR performs a secure boot. For the case of this lab, we will use the FRDM-K82F as the target hardware.

The lab will direct the user through using KBOOT and its tools (BLHOST and ELFTOSB) to sign application firmware. The signed application firmware will be programmed to the target device (FRDM-K82F). Then we will show a couple of test cases that prove that the firmware is protected with authentication before executing.

By the end of this lab the user will learn how to use:

- KBOOT bootloader
- Host tools (ELFTOSB and BLHOST)
- Mbed TLS cryptography

2 Resources

The following resources are for reference. If you could review the below before completing the lab it is highly recommended.

Secure boot White Paper: http://www.nxp.com/docs/en/white-paper/SECURITY-WP.pdf

Secure boot Webinar part 1: http://www.nxp.com/video/how-to-protect-your-firmware-against-malicious-attacks-using-the-latest-kinetis-development-board:SECURE-YOUR-FIRMWARE-WITH-KINETIS?elq_mid=5166&elq_cid=2193576

Secure boot Webinar part 2: http://www.nxp.com/video/designing-secure-iot-devices-starts-with-a-secure-boot:DESIGNING-SECURE-IOT-DEVICES?elq_mid=5166&elq_cid=2193576

LINK: Download KBOOT 2.0 Package

Link to SDK Builder: https://mcuxpresso.nxp.com/en/welcome

3 Overview and Essential Background

Bench Setup

This LAB uses the FRDM-K82F and a PC running Windows. The FRDM-K82F has been updated to use the Segger JLink profile for the on-board debugger CMSIS-DAP.

This LAB assumes that the PC is preloaded with the IAR Embedded work bench version 8 and the related drivers. The path to the Lab Software is:

C:\NXP\ SDK_2.2_FRDM-K82F_Irvine_Training

Two USB micro cables are needed to connect from the PC to the FRDM-K82F board on the connector highlighted below.

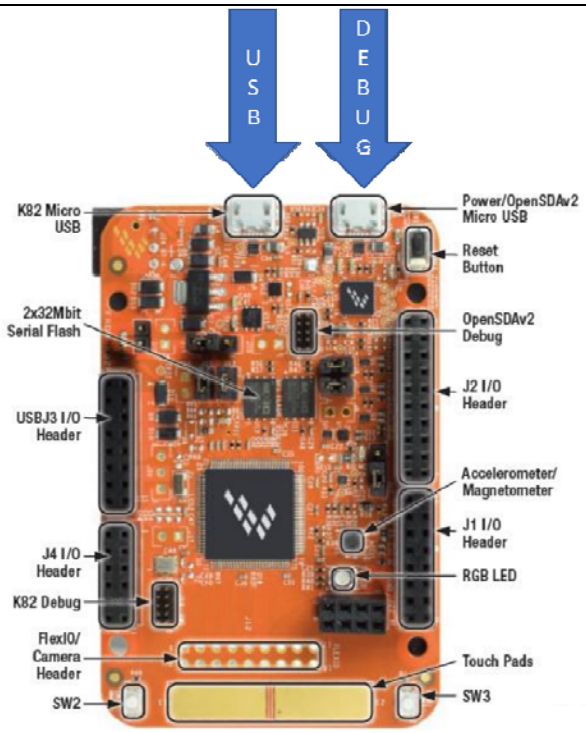


Figure 2. FRDM-K82F primary component placement

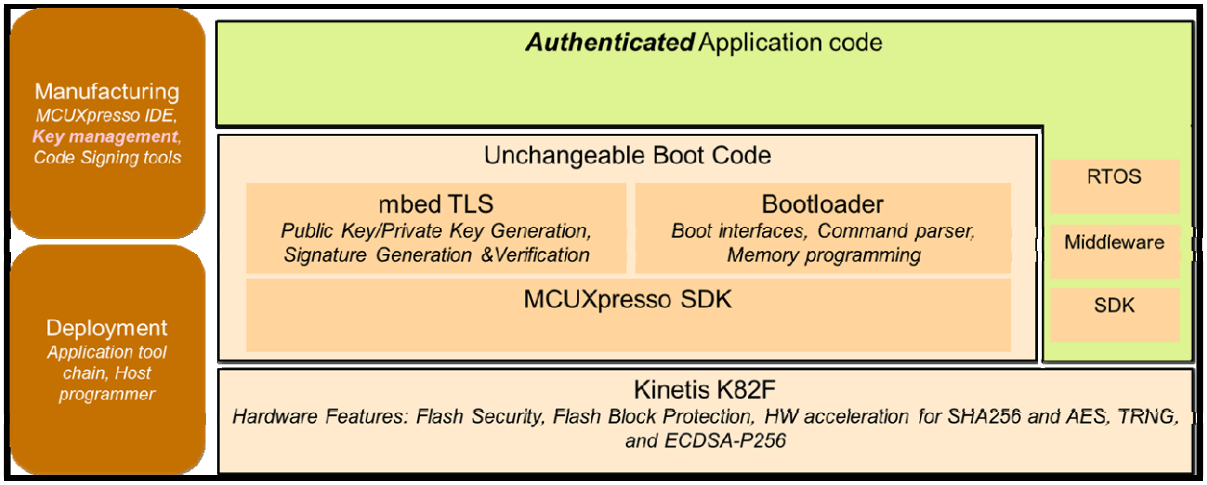
Secure Boot Architecture

The following diagram details the components of a secure boot. At the base, there is the hardware providing physical and logical security, the Kinetis K82 MCU. This is where microcontroller capabilities are necessary to protect data, perform cryptography and monitor access to memories and peripherals.

Sitting above the hardware must be unchangeable boot code. This code must always run when the device is powered. This boot code contains low level drivers to set up relevant security peripherals, a cryptography stack for authentication and or confidentiality of data and a way to load application code (a bootloader).

With the unchangeable boot code present on the right hardware, application code that is loaded on the device is authenticated upon every boot. Application code can be changed but the cryptographic authentication applied to the code by the secure boot ensures that the changes are only and always provided by a trusted entity. Application code can make use of all or a portion of the microcontroller resources as determined by the secure boot code. This is because upon boot, the secure boot code is always executed first, ensuring proper resource management.

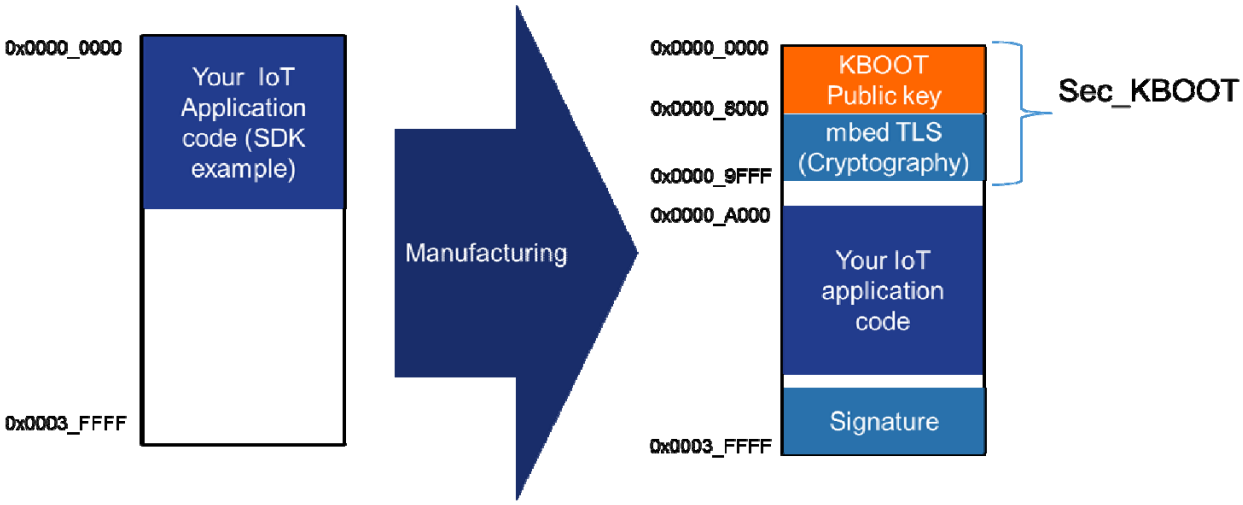
Represented on the left are tools used in the manufacturing and deployment of the device. The microcontroller must be programmed, so tools for key management, creating firmware files and connecting and downloading firmware into the device are needed to implement the secure boot design.



Memory Map view

The following diagram shows the memory map view for the secure boot implementation. Any application running on the target hardware can be modified to work with a Security Kboot that includes the cryptography. This lab will detail the linker file changes to typical application code (shown on the left). This lab will also detail how Security KBoot must be configured to work with the application code memory map.

- **Typical Application Development**
- **Final production image**



Tools and manufacturing flow

The Secure boot implementation requires two separate configurations of the target hardware. These are labeled Factory configuration and Production Configuration.

Factory Configuration: This configuration is for use in a secure manufacturing environment. The main security functions in addition to bootloader functions are to generate a PUB/PRIV key pair and to generate the signature for application code using the **private key**.

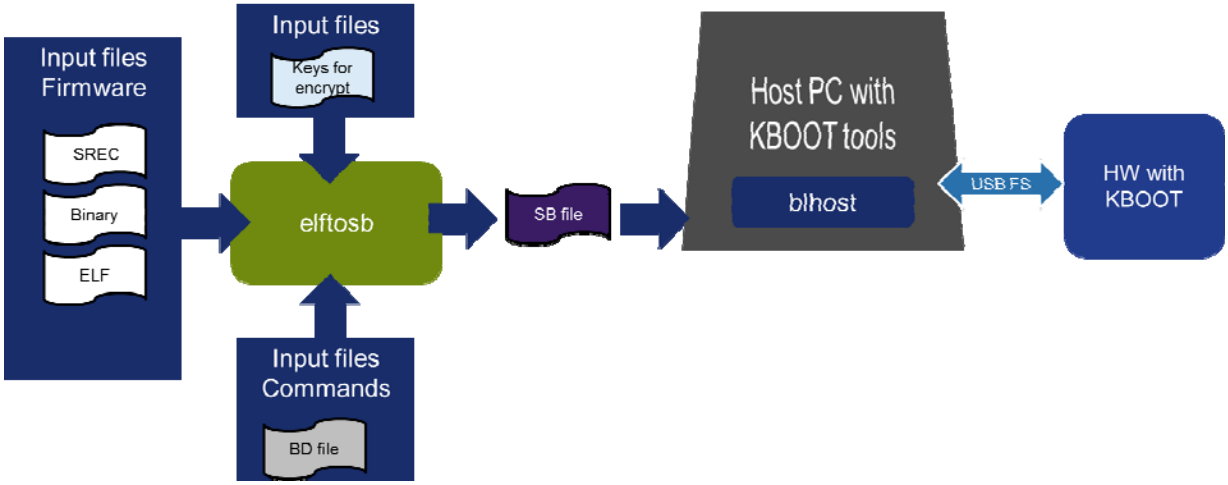
Production Configuration: This configuration is for use in a deployed device. The main security functions in addition to bootloader functions are to check the signature of application code using the **public key**, and only allow execution of the application code if the signature is authentic

For both configurations, the KBoot Tools (ElftoSB and BLhost) and BD and SB files are used.

BD file: Short for boot descriptor file. This is an input command file to be used by elftosb for created SB files

SB file: Short for secure binary file. This is the output of elftosb which is used to pass commands and data (for example application firmware) to a Kinetis MCU running KBOOT

The following diagram shows how these files are used with KBOOT tools.
Input files are processed by the elftosb tool using a BD file to generate SB files.
SB files are used by blhost tool to provision target hardware running KBOOT.



Lab sections

The major sections of this lab are:

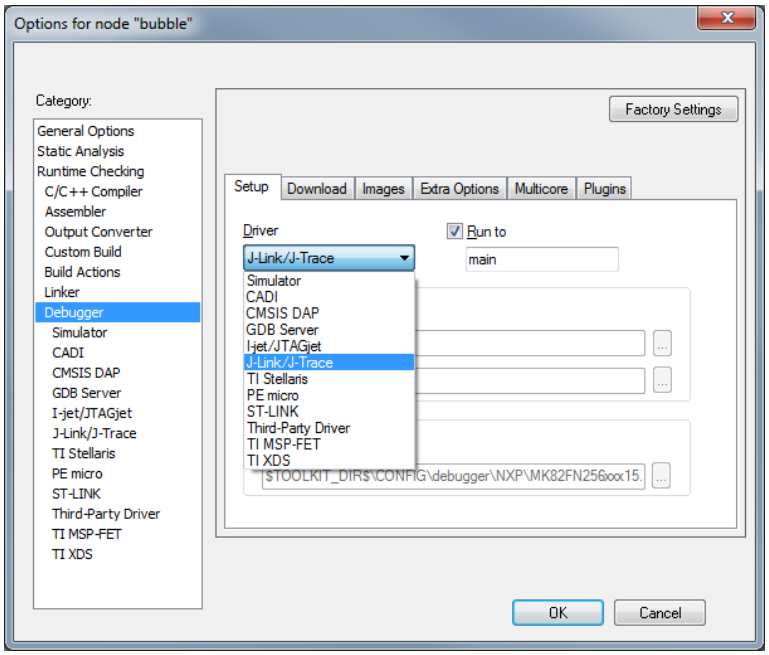
- *Preparing Application Code and Security KBoot*
- *Using Security KBoot in Factory Configuration*
- *Using Security KBoot for provisioning production image (Production Configuration)*
- *Firmware Authentication Test Cases*

4 Preparing Application Code and Security KBOOT

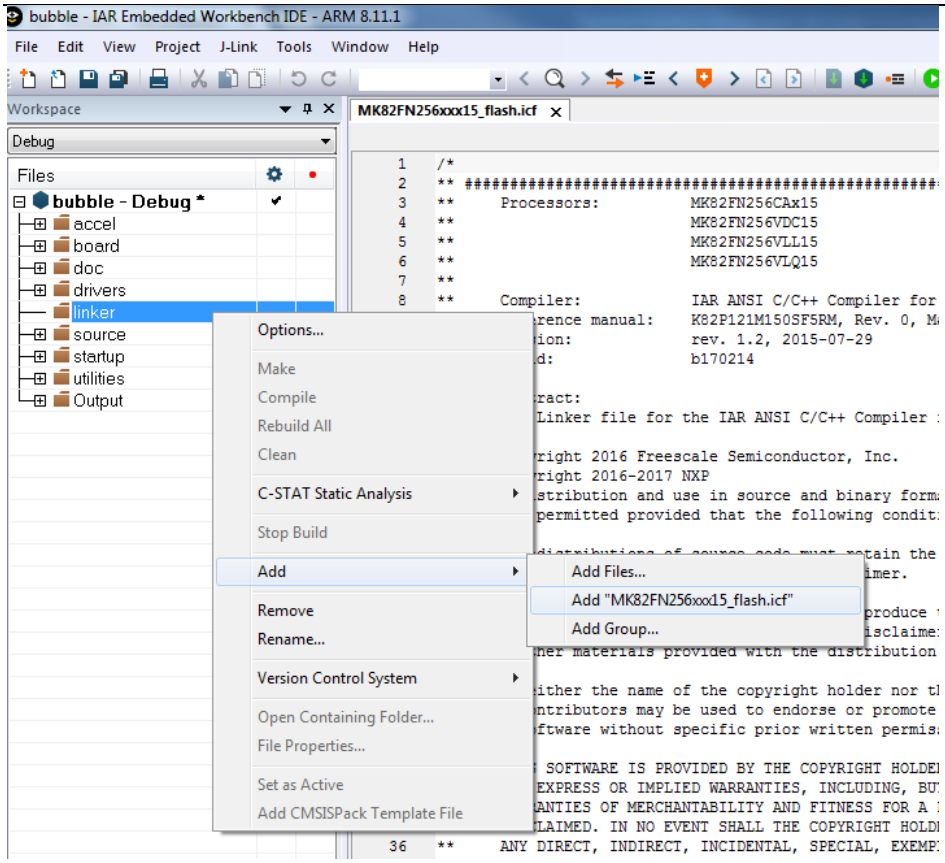
The end result of this section is that two separate SREC files will be created and stored in the tools folder.

The following steps **were done previously** to the default SDK software example (bubble). No Action is required for the following 2 points.

1) Change debugger to Segger JLink

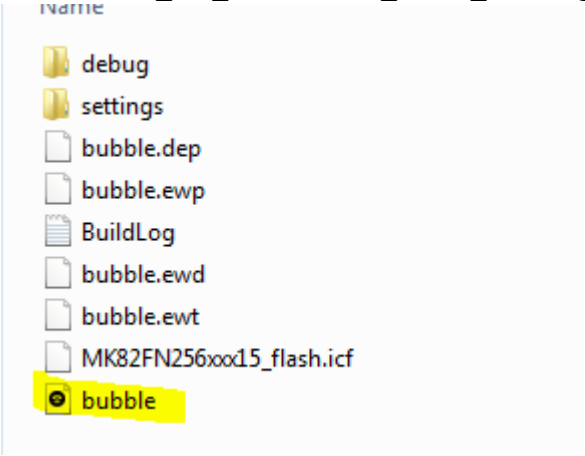


2) Add a folder with the project linker file



Open the application workspace

Navigate and open the SDK example by double left clicking the project at:
C:\NXP\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\bubble\iar



Update linker file to relocate application code to begin at address 0xA000

Update the following lines of the Linker File to adjust the target memory to make room for the security bootloader. The updates offset the memory definitions by 0xA000.

Lines 52, 53, 55, 56, 57, 58, 59 and 61 must be updated.

```
52  define symbol m_interrupts_start        = 0x0000A000;
53  define symbol m_interrupts_end          = 0x0000A3BF;
54
55  define symbol m_bootloader_config_start = 0x0000A3C0;
56  define symbol m_bootloader_config_end   = 0x0000A3FF;
57
58  define symbol m_flash_config_start      = 0x0000A400;
59  define symbol m_flash_config_end        = 0x00000A40F;
60
61  define symbol m_text_start              = 0x0000A410;
62  define symbol m_text_end                = 0x0003FFFF;
63
```

Add bootloader configuration to startup_MK82F25615.s

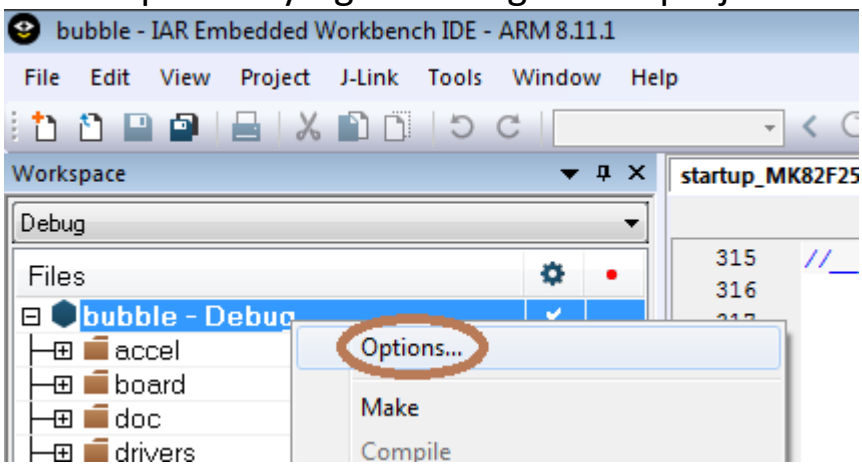
Expand the startup folder and open the startup_MK82F25615.s
Copy and paste the below at line 315 of the startup file.
This is applying the bootloader configuration data which is used by Kboot standard bootloader functions.

NOTE: This step is already done – just confirm that the definition is present in the startup file

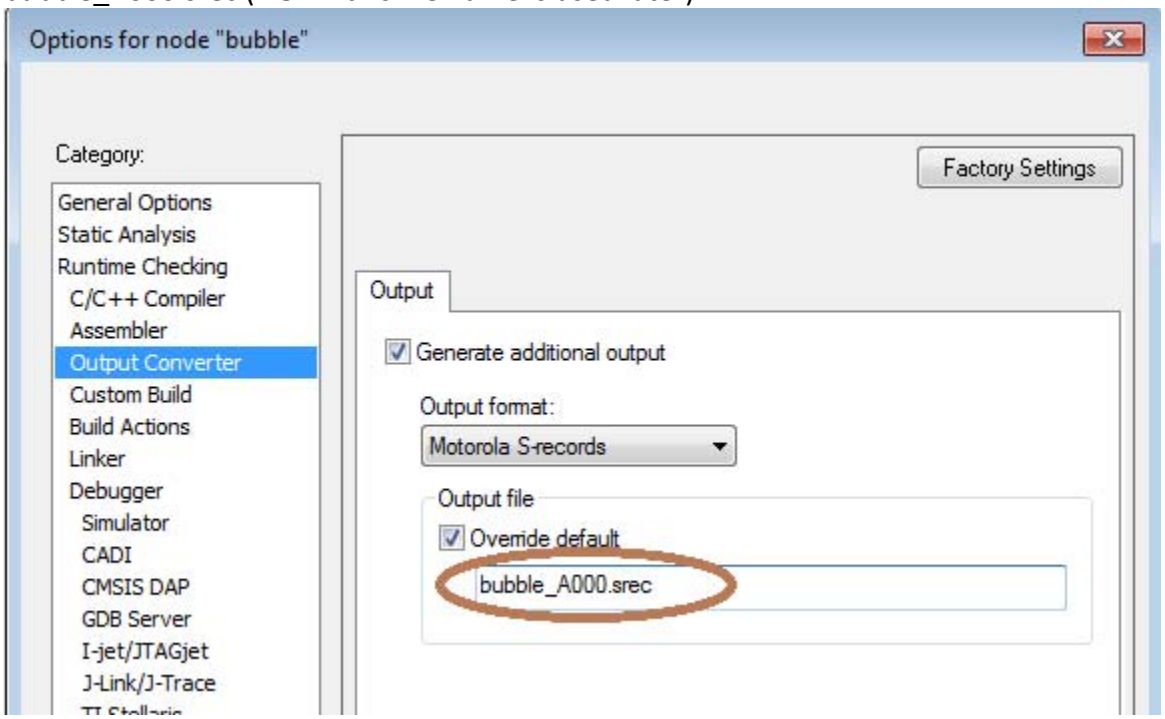
```
//__bootloaderConfigurationArea ; 0x3c0
    DCD    'kcfg'      ; [00:03] tag - Tag value used to validate the bootloader configuration data. Must be set to 'kcfg'.
    DCD    0xFFFFFFFF  ; [04:07] crcStartAddress
    DCD    0xFFFFFFFF  ; [08:0b] crcByteCount
    DCD    0xFFFFFFFF  ; [0c:0f] crcExpectedValue
    DCB    0xFF        ; [10:10] enabledPeripherals
    DCB    0xFF        ; [11:11] i2cSlaveAddress
    DCW    3000        ; [12:13] peripheralDetectionTimeoutMs - Timeout in milliseconds for peripheral detection before jumping to application code
    DCW    0xFFFF      ; [14:15] usbVid
    DCW    0xFFFF      ; [16:17] usbPid
    DCD    0xFFFFFFFF  ; [18:1b] usbStringsPointer
    DCB    0xFF        ; [1c:1c] clockFlags - High Speed and other clock options
    DCB    0xFF        ; [1d:1d] clockDivider - One's complement of clock divider, zero divider is divide by 1
    DCW    0xFFFF      ; [1e:1f] reserved
    ; Fill to align with flash configuration field.
    REPT   (0x400-0x3e0)/4  ; 0x3E0 - 0x3FF
    DCD    0xFFFFFFFF      ; Reserved for user TRIM value
    ENDR
```

Configure project to generate SREC

Select options by right clicking on the project

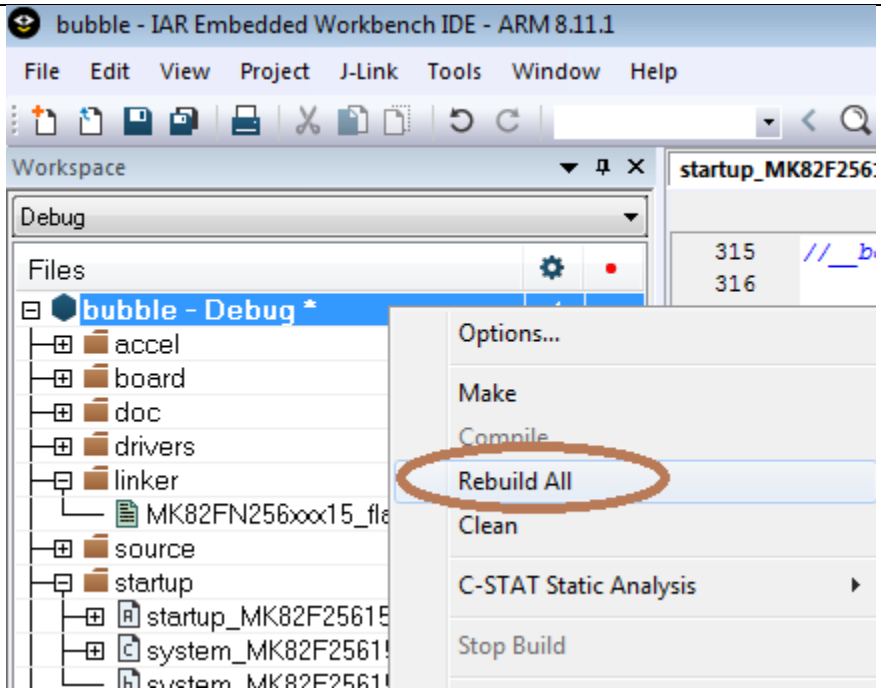


Select output converter. Click on Generate additional output. Overide the default name and type bubble_A000.srec (**NOTE**: this file name is used later)



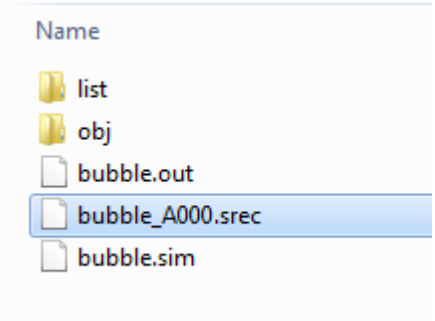
Rebuild all to generate bubble_A000.srec

Right click on project name and select rebuild all



Move the SREC that has been generated

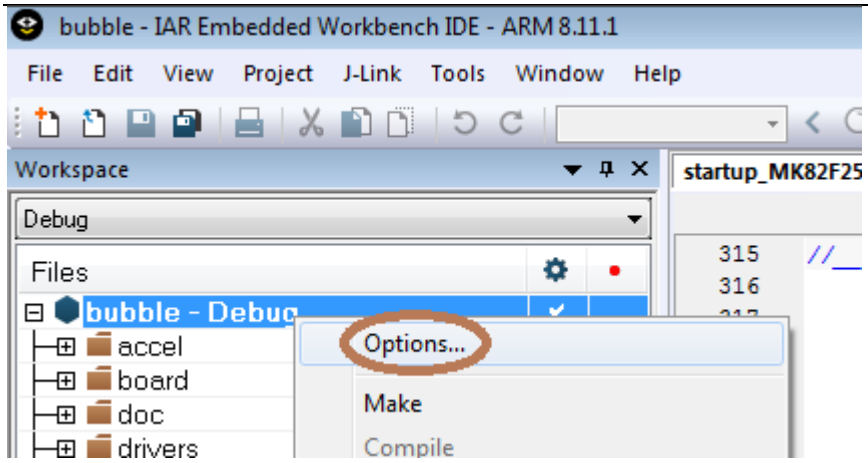
Check the folder for the srec file using windows explorer.
boards\frdmk82f\demo_apps\bubble \iar\debug



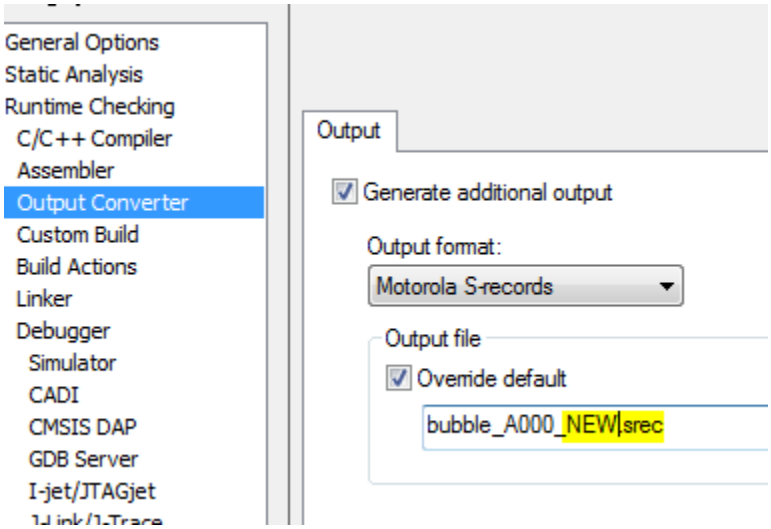
Copy the SREC and paste in c:\nxp\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\tools\blhost & elftosb

Configure project to generate a new srecrod

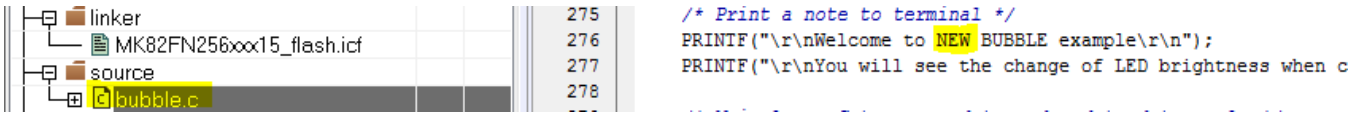
Select options by right clicking on the project



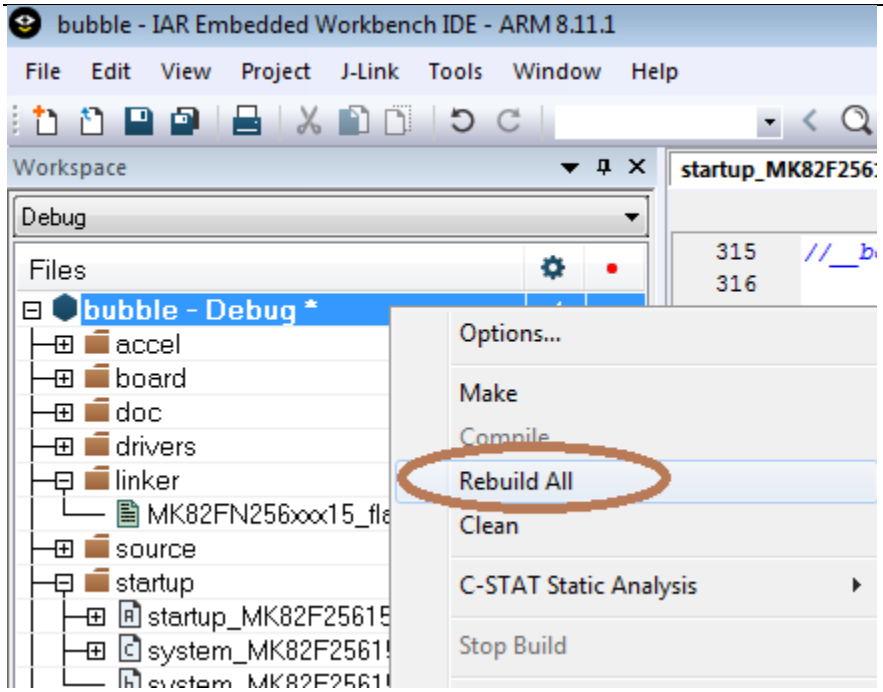
Select output converter. Click on Generate additional output. Overide the default name and type **bubble_A000_NEW**.srec



Open the bubble.c source file and change the text at line 276 to say NEW Bubble example.

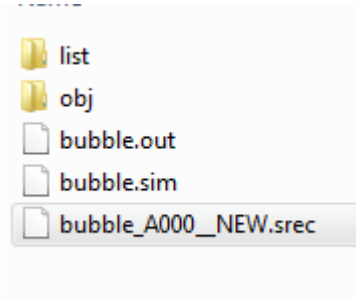


Right click on project name and select rebuild all



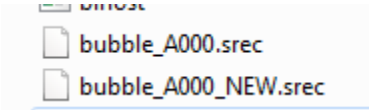
Move the NEW SREC that has been generated

Check the folder for the srec file using windows explorer.
boards\frdmk82f\demo_apps\bubble \iar\debug



Copy the SREC and paste in c:\nxp\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\tools\blhost & elftosb

Check that there are 2 SREC files in this folder.

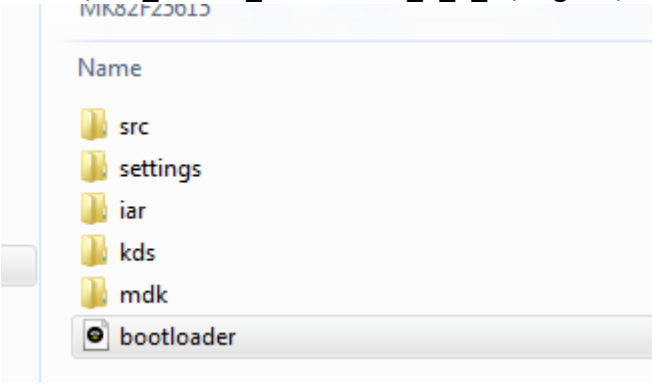


Security KBoot

Security KBoot is a customized version of KBOOT2.0 release. It has been modified to add cryptography for performing secure boot. The following section highlights the important attributes of this component.

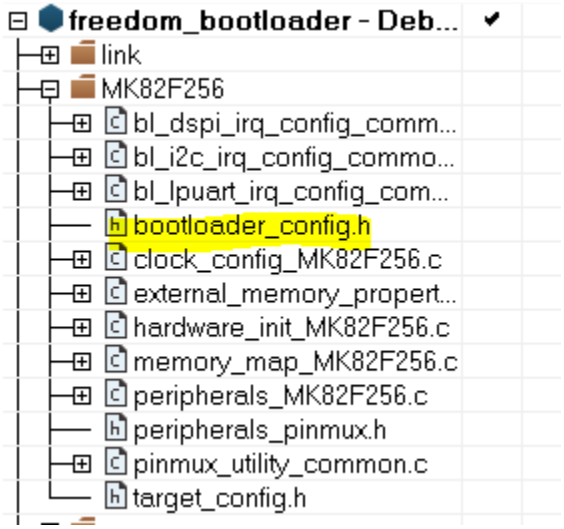
Navigate to the below folder and open the security KBoot Project

C:\nxp\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\NXP_Kinetis_Bootloader_2_0_0\targets\MK82F25615



Examine bootloader_config.h file

Open bootloader_config.h from the target specific folder of the freedom_bootloader project highlighted below.



Examine the following definitions

At line 228, this definition sets the start of the application code space. If your application code is relocated, then this definition must change.

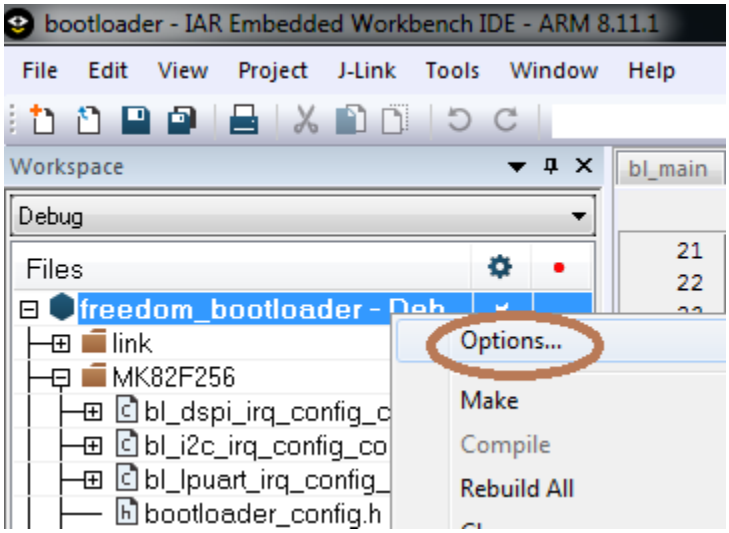
```
142 // The bootloader will check this address for the application vector table upon startup.
143 #if !defined(BL_APP_VECTOR_TABLE_ADDRESS)
144 #define BL_APP_VECTOR_TABLE_ADDRESS 0xa000
145 #endif
```

At line 205, this definition sets factory configuration. If BOOTLOADER_FACTORY is defined, then the bootloader is in factory configuration.

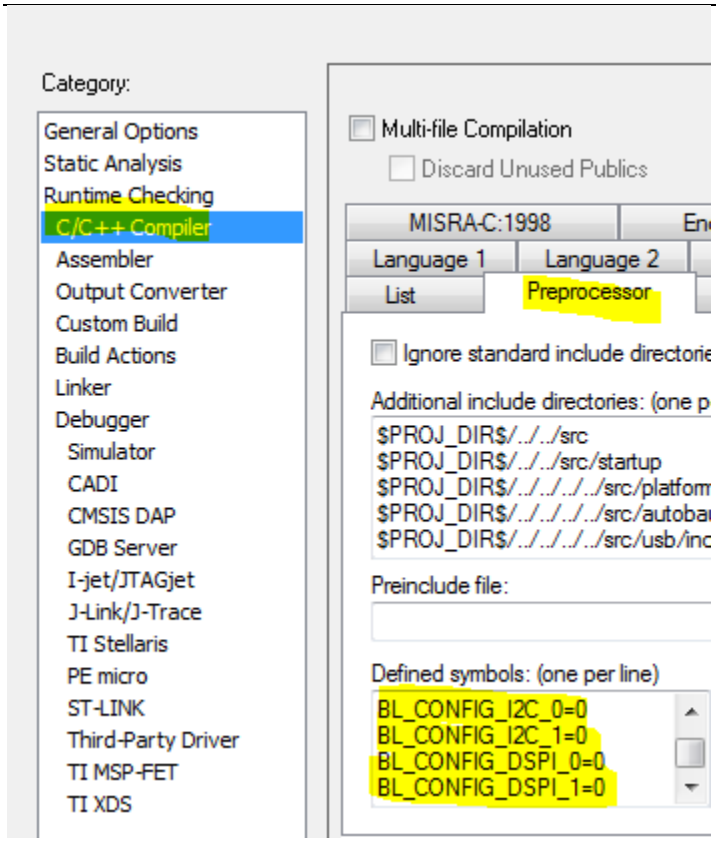
```
204  // if this isn't defined, then the bootloader is for product
205  #define BOOTLOADER_FACTORY
```

Examine freedom_bootloader project configurations

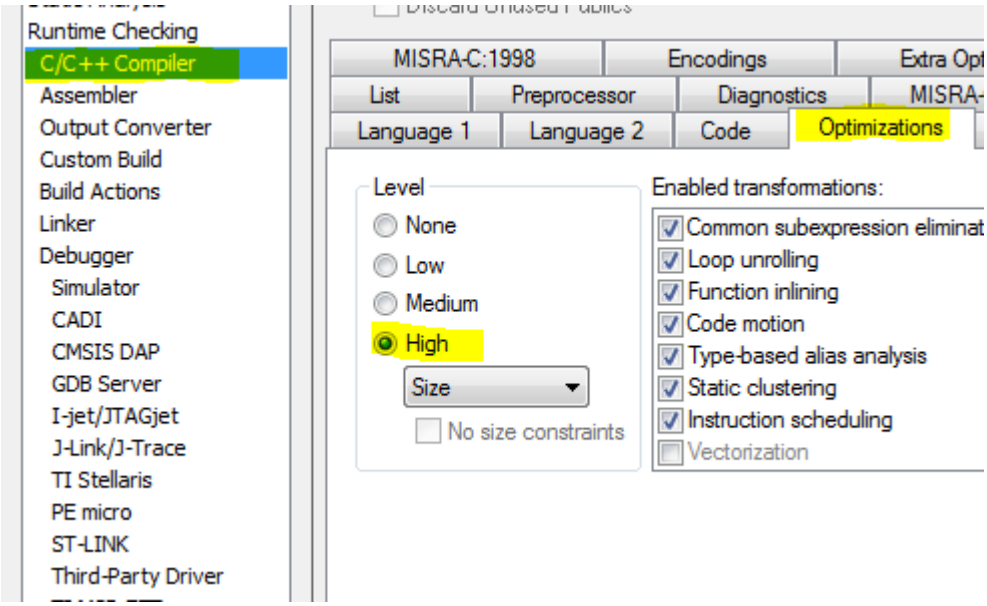
Open the options of the feedom_bootloader project. Right click and select options.



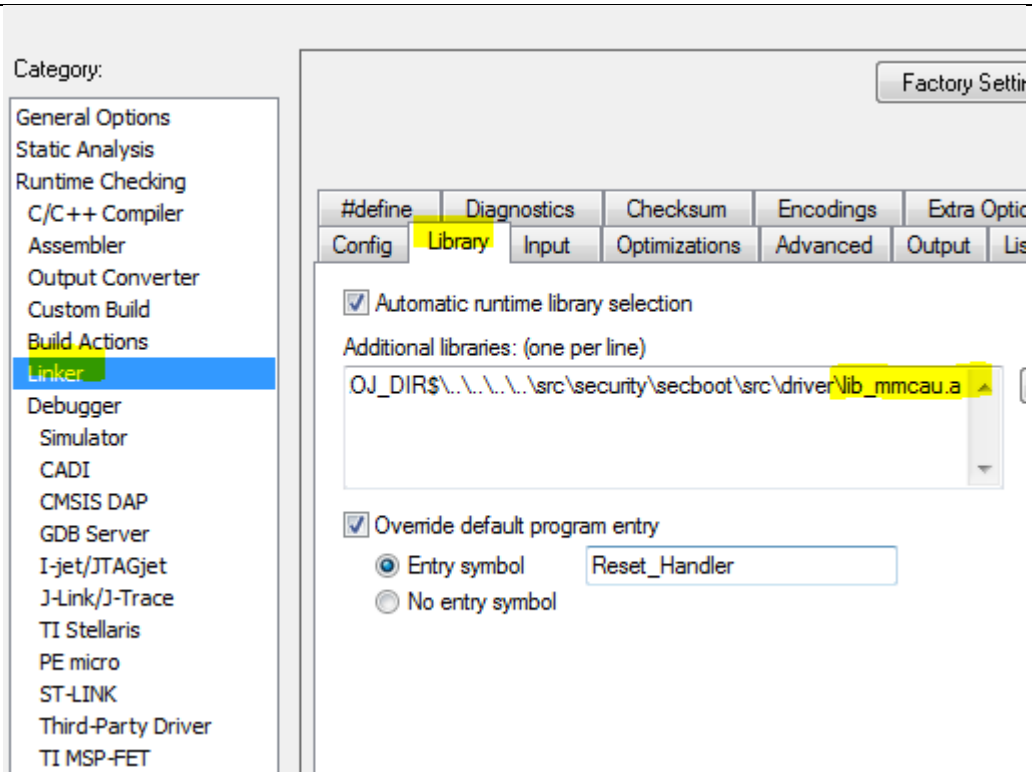
Select C/C++ Compiler, and the preprocessor tab. Scroll down in the defined symbols. Note that all unused interfaces are disabled. This is done to save code size and remove un needed interfaces.



Select C/C++ Compiler and the optimizations tab. Not that Optimization are set to high. As the internal flash is a precious resource and takes away space for application code, the security KBoot was optimized to save flash space.

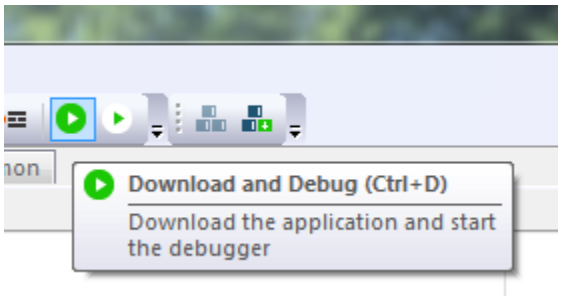


Select Linker and the library tab. Note that for hashing functions, the security Kboot uses the mmcau library.

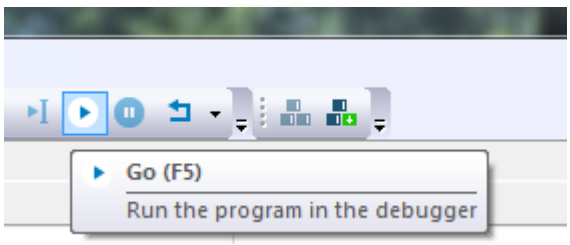


Program and test factory configuration

Download and debug the factory configuration of the security Kboot



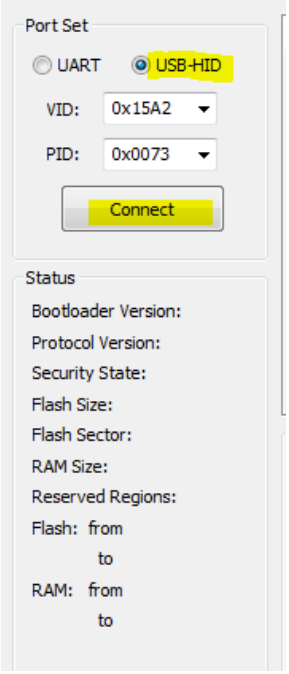
Press go from the debug perspective



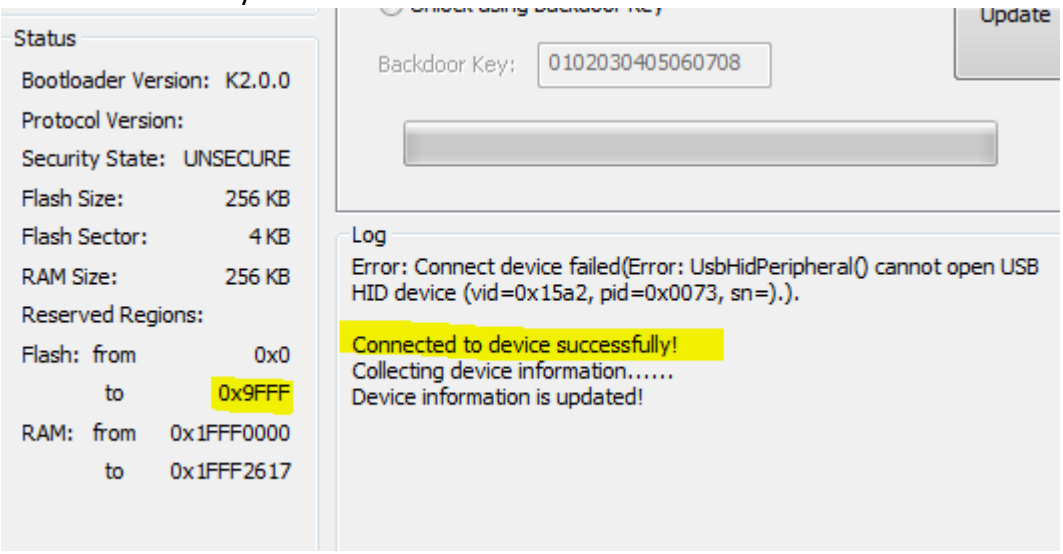
Open the Kinetis flash tool by selecting the icon on the Windows Task bar.
If the icon is not on your task bar, then the Kinetis Flash tool is in the following directory:

C:\NXP\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\NXP_Kinetis_Bootloader_2_0_0\bin\Tools\KinetisFlashTool\win

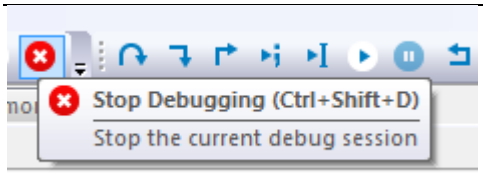
Select USB and press connect



Once connected you will see that the status is updated and an updated message (Connected to the device successfully.



Close the Kinetis Flash tool
Stop debugging



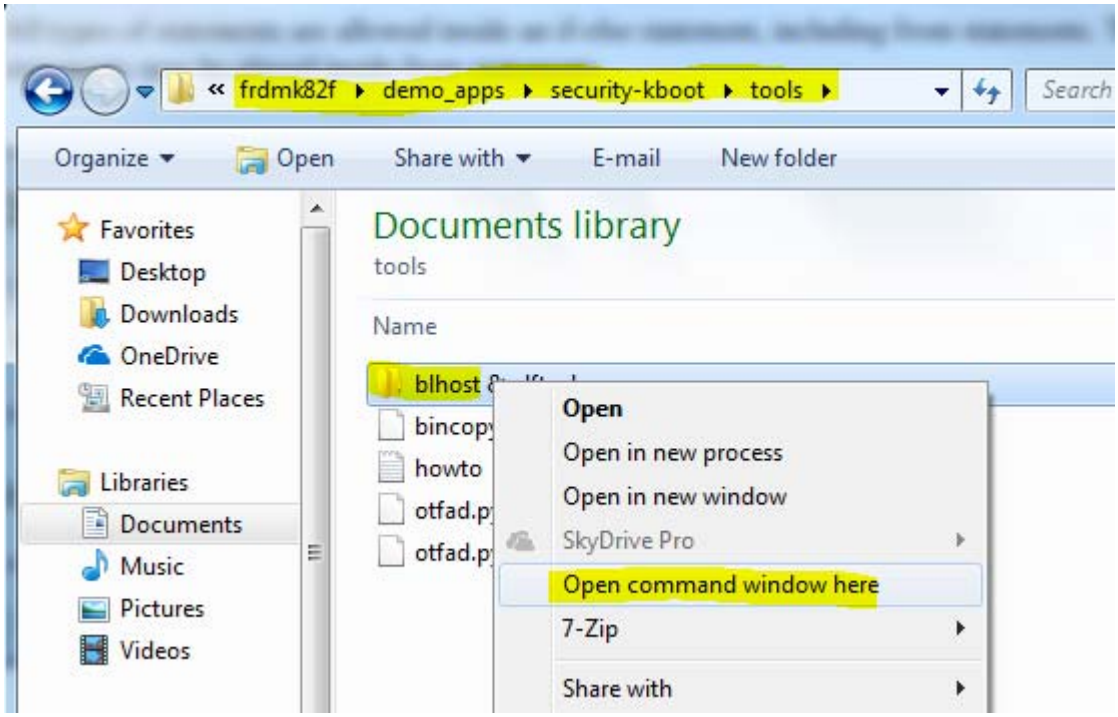
You are now ready to move on to the next section.

5 Using Security KBoot in Factory Configuration

In this section we will use the factory configuration of kboot to sign the two srecords which were previously generated.

Navigate to *SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\tools\blhost&elftosb*

Hold SHIFT + right click in the blhost & elftosb folder and select Open Command Window



Use elftosb to create factory.sb

In the command window type: elftosb -V -c factory.bd -o factory.sb

Note: the syntax is:
-V enables verbose output
-c is a command file prefix
-o is an output file prefix

The output should be

```
Boot Section 0x00000000:
  LOAD ¦ adr=0x20004000 ¦ len=0x00000200 ¦ crc=0x0719eed2 ¦ flg=0x0000
  ENA  ¦ adr=0x20004000 ¦ cnt=0x00000004 ¦ flg=0x0100
  ERAS ¦ adr=0x68000000 ¦ cnt=0x00010000 ¦ flg=0x0000
  ERAS ¦ adr=0x68010000 ¦ cnt=0x00010000 ¦ flg=0x0000
  ERAS ¦ adr=0x68020000 ¦ cnt=0x00010000 ¦ flg=0x0000
  ERAS ¦ adr=0x68030000 ¦ cnt=0x00010000 ¦ flg=0x0000
  ERAS ¦ adr=0x68040000 ¦ cnt=0x00010000 ¦ flg=0x0000
  ERAS ¦ adr=0x68050000 ¦ cnt=0x00010000 ¦ flg=0x0000
  ERAS ¦ adr=0x0003f000 ¦ cnt=0x00001000 ¦ flg=0x0000
  CALL ¦ adr=0x00004db5 ¦ arg=0x11223344 ¦ flg=0x0000
  LOAD ¦ adr=0x0000a000 ¦ len=0x000040a8 ¦ crc=0x8aaf3c16 ¦ flg=0x0000
  CALL ¦ adr=0x00004de3 ¦ arg=0x6d1a853c ¦ flg=0x0000
```

Use blhost to program factory.sb

In the command window type: blhost -u -- receive-sb-file factory.sb

NOTE: reset the board wait 1 second – then press enter to send the command

Note: the syntax is:
-u defines USB as the interface
-- is a command prefix (this is a double dash)
receive-sb-file is the command

the output should be

```
Inject command 'receive-sb-file'
Preparing to send 17424 (0x4410) bytes to the target.
Successful generic response to command 'receive-sb-file'
(1/1)100% Completed!
Successful generic response to command 'receive-sb-file'
Response status = 0 (0x0) Success.
Wrote 17424 of 17424 bytes.
```

The Green LED will blink to show a successful key and signature generation.

Extract the public key and signature

In the command window type: blhost -u -- read-memory 0x20010000 128 sign.bin

The output should be:

```
Inject command 'read-memory'
Successful response to command 'read-memory'
(1/1)100% Completed!
Successful generic response to command 'read-memory'
Response status = 0 (0x0) Success.
Response word 1 = 128 (0x80)
Read 128 of 128 bytes.
```

In the command window type: blhost -u -- read-memory 0x20010080 64 pubkey.bin

```
Inject command 'read-memory'
Successful response to command 'read-memory'
(1/1)100% Completed!
Successful generic response to command 'read-memory'
Response status = 0 (0x0) Success.
Response word 1 = 64 (0x40)
Read 64 of 64 bytes.
```

Use elftosb to create factory_NEW.sb

In the command window type: elftosb -V -c factory_NEW.bd -o factory_NEW.sb
Note: the syntax is:
-V enables verbose output
-c is a command file prefix
-o is an output file prefix

```
oot Section 0x00000000:
 LOAD | adr=0x20004000 | len=0x00000200 | crc=0x0719eed2 | flg=0x0000
 ENA  | adr=0x20004000 | cnt=0x00000004 | flg=0x0100
 ERAS | adr=0x0000a000 | cnt=0x00034000 | flg=0x0000
 CALL | adr=0x00004db5 | arg=0x11223344 | flg=0x0000
 LOAD | adr=0x0000a000 | len=0x00003fa4 | crc=0x8733c1e1 | flg=0x0000
 CALL | adr=0x00004de3 | arg=0x6d1a853c | flg=0x0000
```

Use blhost to program factory_NEW.sb

In the command window type: blhost -u -- receive-sb-file factory_NEW.sb
Note: the syntax is:
-u defines USB as the interface
-- is a command prefix (this is a double dash)
receive-sb-file is the command

The Green LED will blink to show a successful key and signature generation.

Extract the NEW signature

In the command window type: blhost -u -- read-memory 0x20010000 128 sign_NEW.bin
The output should be:

```
Inject command 'read-memory'
Successful response to command 'read-memory'
(1/1)100% Completed!
Successful generic response to command 'read-memory'
Response status = 0 (0x0) Success.
Response word 1 = 128 (0x80)
Read 128 of 128 bytes.
```

Note: public key does not need to be extracted.

You are now ready to move on to the next section.

6 Using production configuration

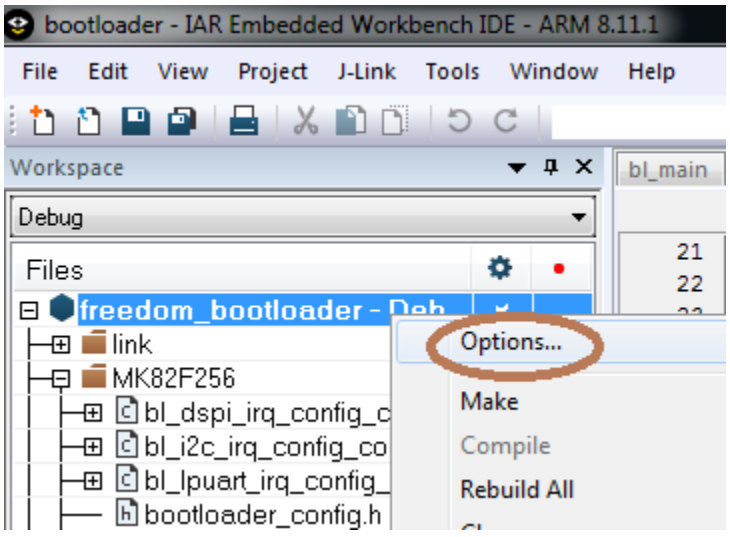
Production configuration is what will be programmed into every device that gets deployed. This configuration only performs signature verification based on the public key.

Update the freedom bootloader project for production configuration

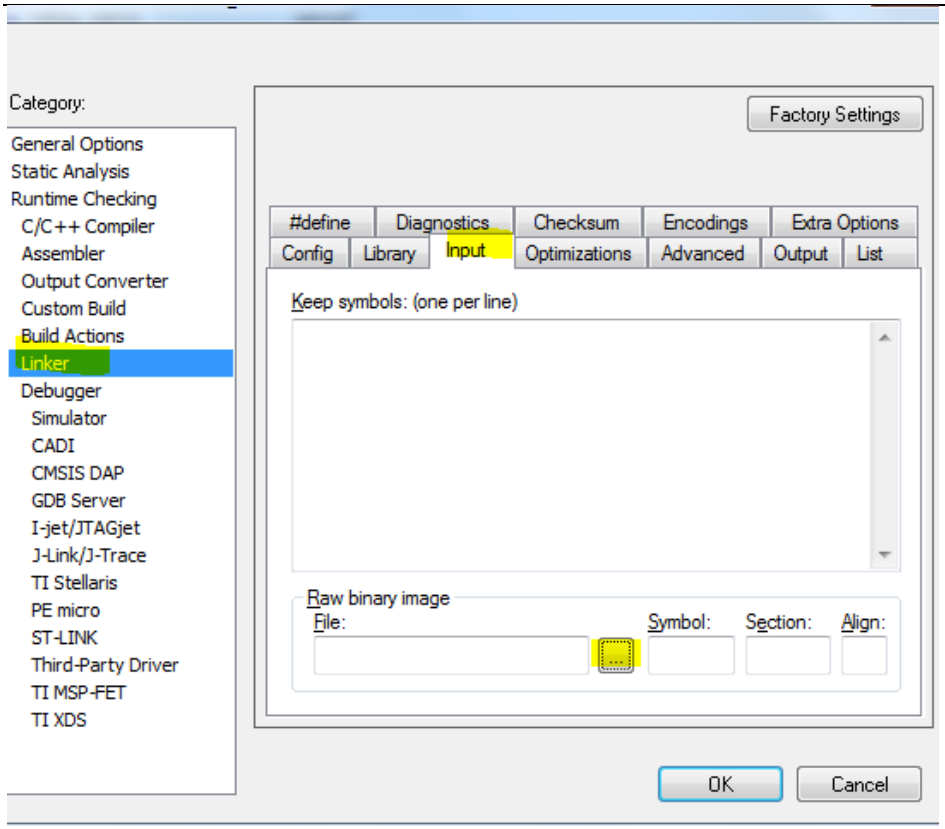
Comment line 205 in bootloader_config.h file

```
204  // if this isn't defined, then the bootloader is for product
205  //#define BOOTLOADER_FACTORY
```

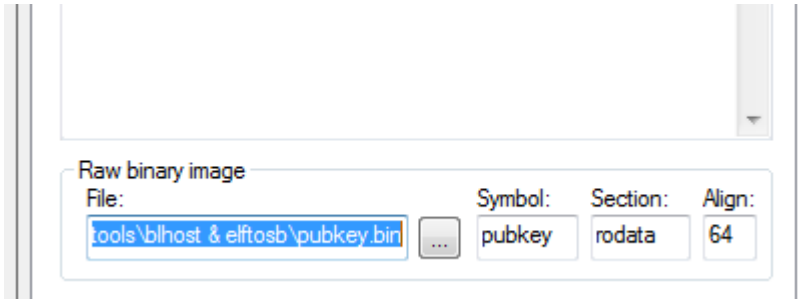
Open the project options to add the public key binary to the build process.



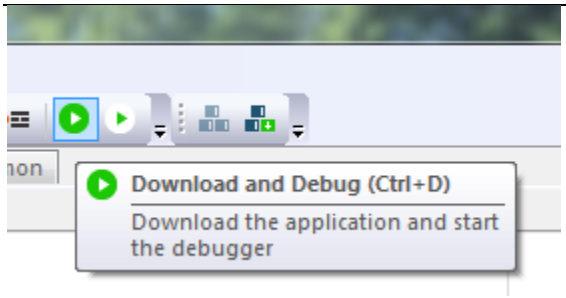
Select Linker-> Input and enter a file



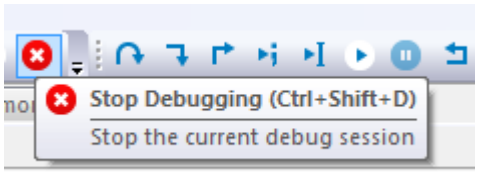
Navigate to C:\nxp\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\tools\blhost & elftosb and select the pubkey.bin file, fill in
Symbol: pubkey
Section: rodata
Align: 64



Download and debug the new configuration



Stop debugging



Green LED should turn on to show a signature verification failure.

Use elftosb to create the product.sb file

In the command window (same as was used to in factory configuration)

In the command window type: elftosb -V -c product.bd -o product.sb

The production.sb file contains the application firmware along with the signature binary.
The public key is part of the production configuration of the security Kboot.

The output should be:

```
Boot Section 0x00000000:
  LOAD ¦ adr=0x20004000 ¦ len=0x00000200 ¦ crc=0x0719eed2 ¦ flg=0x0000
  ENA  ¦ adr=0x20004000 ¦ cnt=0x00000004 ¦ flg=0x0100
  CALL ¦ adr=0x00004e55 ¦ arg=0x457a9d3f ¦ flg=0x0000
  LOAD ¦ adr=0x20001000 ¦ len=0x00000080 ¦ crc=0x2cade15a ¦ flg=0x0000
  LOAD ¦ adr=0x0000a000 ¦ len=0x00004098 ¦ crc=0xac49d77b ¦ flg=0x0000
  CALL ¦ adr=0x00004e87 ¦ arg=0x6d1a853c ¦ flg=0x0000
```

Use blhost to program production.sb

In the command window type: blhost -u -- receive-sb-file product.sb

Reset the board (Button adjacent to SDA USB micro connector) – Then press Enter in the command window.

The output should be:

```
Inject command 'receive-sb-file'
Preparing to send 17440 (0x4420) bytes to the target.
Successful generic response to command 'receive-sb-file'
(1/1)100% Completed!
Successful generic response to command 'receive-sb-file'
Response status = 0 (0x0) Success.
Wrote 17440 of 17440 bytes.
```

The green LED will flash when complete.
Reset the board.
After a short delay – the Signature Verification will run- If the signature verification passes, the green LED will flash, then the application code will run.

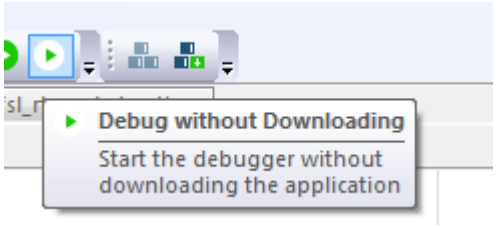
You are now ready to move on to the next section.

7 Firmware Authentication Test Cases

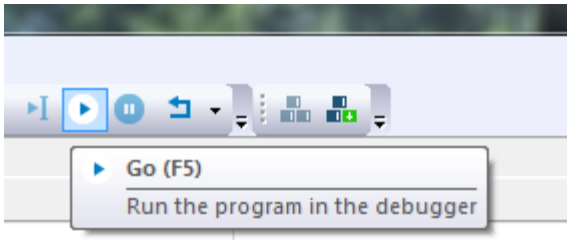
The following two test cases demonstrate that the signature verification process is occurring.

Debugger test case

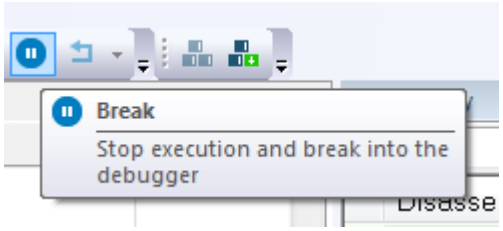
From the bootloader workspace, press the debug without downloading button



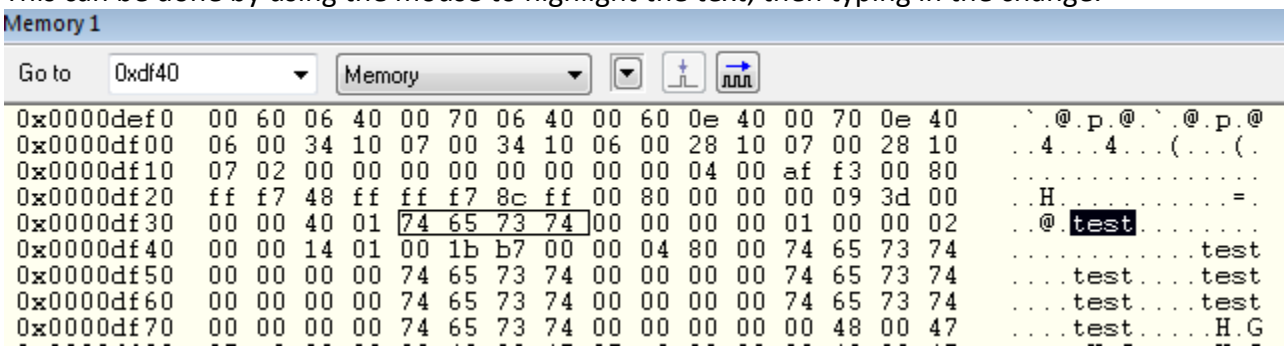
Press the Go button to see normal operation.



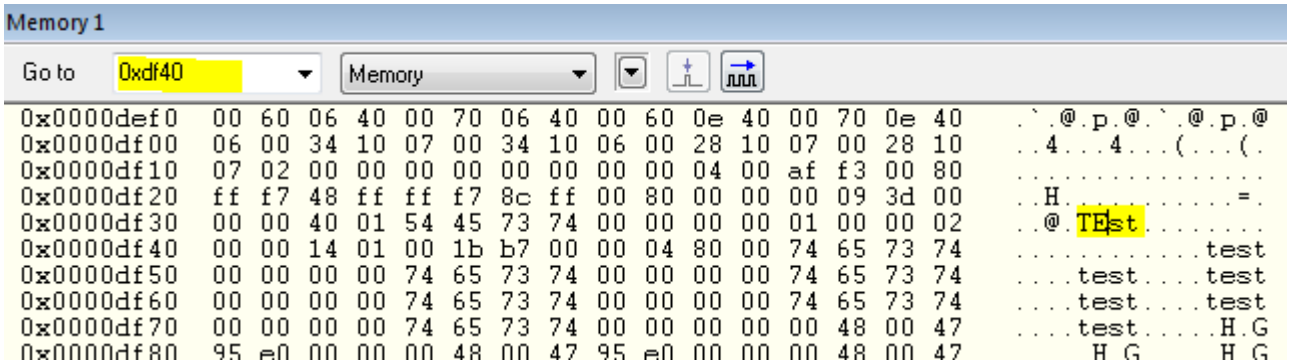
Press the break button



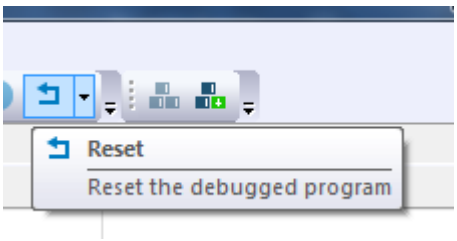
From the memory window (should be set to show range at 0XDF00) change the text.
This can be done by using the mouse to highlight the text, then typing in the change.

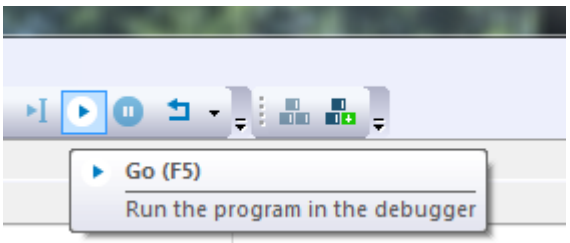


Change lower case test to upercase TEST like the below.



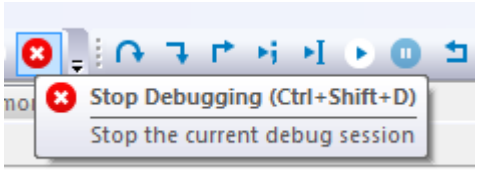
RESET and press GO to see signature verification fail.





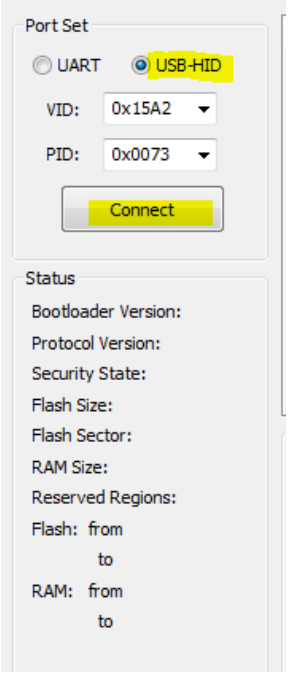
The Green LED should stay on.

Stop debugging

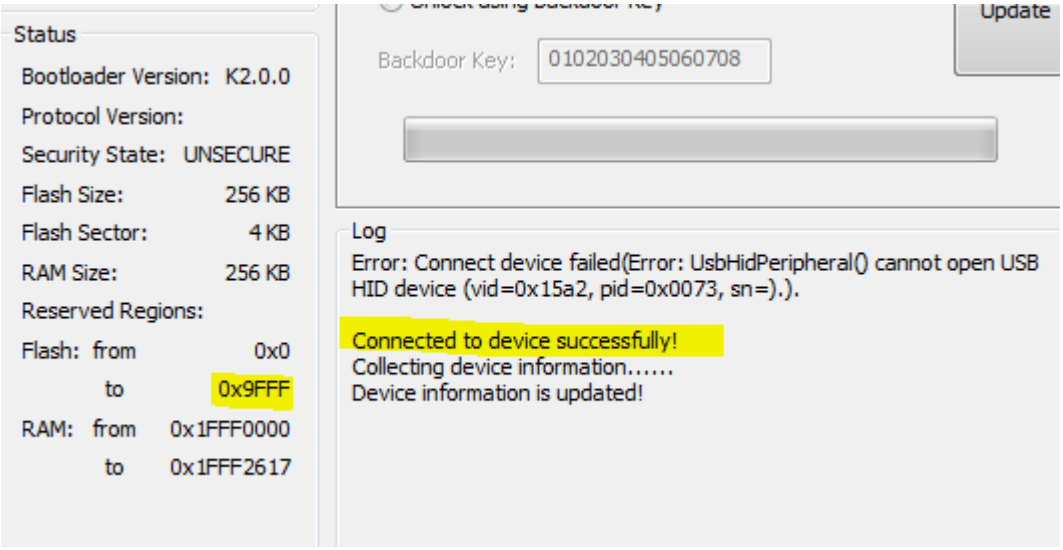


Open the Kinetis flash tool by selecting the icon on the Windows Task bar.
If the icon is not on your task bar, then the Kinetis Flash tool is in the following directory:
C:\NXP\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\NXP_Kinetis_Bootloader_2_0_0\bin\Tools\KinetisFlashTool\win

Reset the board – then select USB and press connect



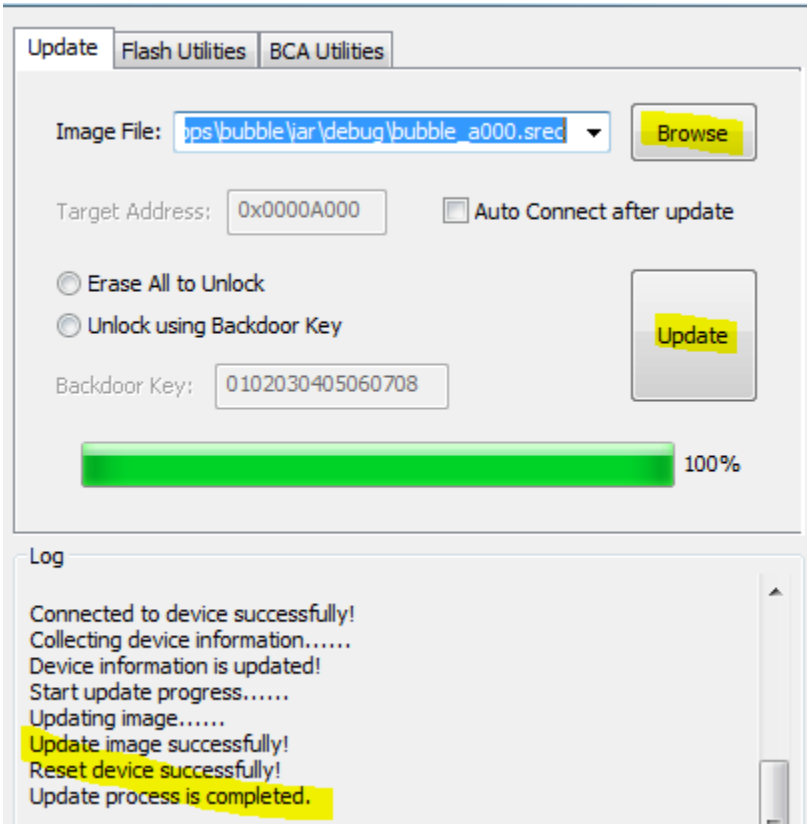
Once connected you will see that the status is updated and an updated message (Connected to the device successfully.



Select browse and navigate to:
C:\nxp\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\tools\blhost & elftos

Select the bubble_A000.srec

Then press update to revert the firmware to the original state.

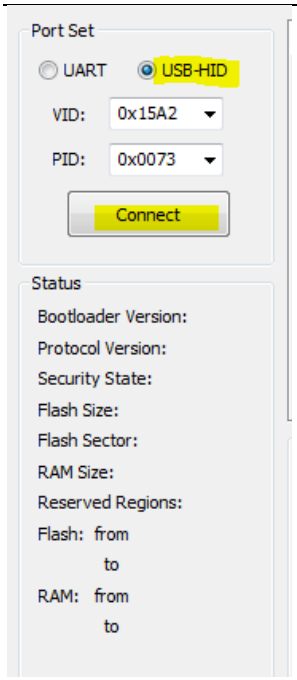


The board should reset and the signature verification pass (application will run).

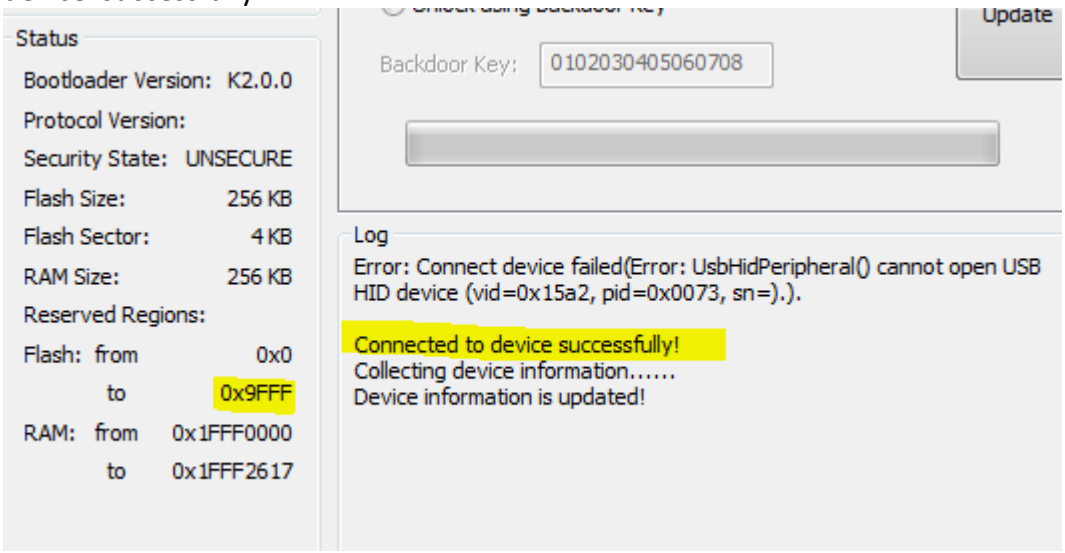
Kinetis Flash Tool test case

Open the Kinetis flash tool by selecting the icon on the Windows Task bar.
If the icon is not on your task bar, then the Kinetis Flash tool is in the following directory:
C:\NXP\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\NXP_Kinetis_Bootloader_2_0_0\bin\Tools\KinetisFlashTool\win

Reset the board – then select USB and press connect



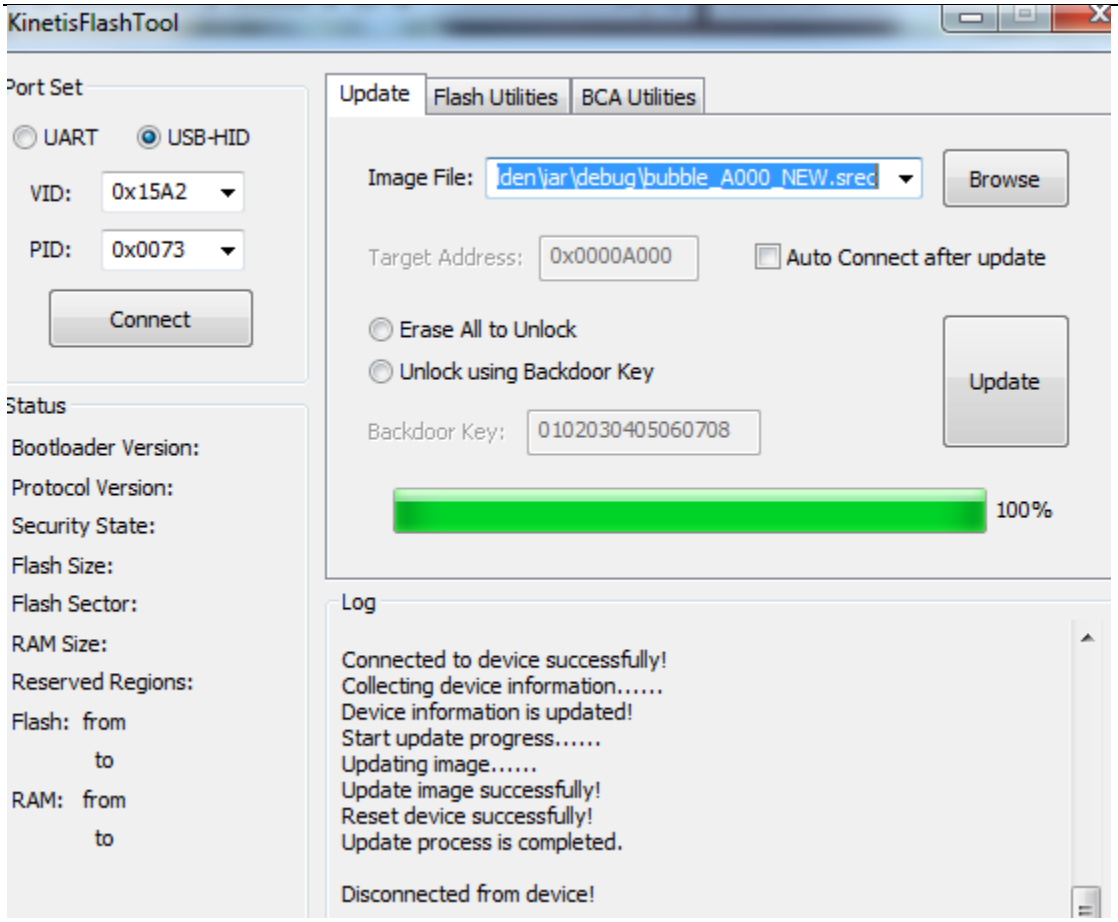
Once connected you will see that the status is updated and an updated message (Connected to the device successfully.



Select browse and navigate to:
C:\nxp\SDK_2.2_FRDM-K82F_Irvine_Training\boards\frdmk82f\demo_apps\security-kboot\tools\blhost & elftosb

Select the bubble_A000_NEW.srec

Then press update to revert the firmware to the New version.



The board should reset and the signature verification **will fail**. GREEN LED will be on.

8 Updating signature to support new application code

In this section, we will use blhost tool to update the signature to use the new application srecord.

From the command window we will erase the available application space

Type: blhost -u -- flash-erase-all

Reset then enter the command

This will erase the signature region.

```
Inject command 'flash-erase-all'
Successful generic response to command 'flash-erase-all'
Response status = 0 (0x0) Success.
```

Next program the new signature binary

Type: blhost -u -- write-memory 0x3FF80 sign_NEW.bin

```
Inject command 'write-memory'
Preparing to send 128 (0x80) bytes to the target.
Successful generic response to command 'write-memory'
(1/1)100% Completed!
Successful generic response to command 'write-memory'
Response status = 0 (0x0) Success.
Wrote 128 of 128 bytes.
```

Next we can program the new SREC

Type: blhost -u -- flash-image bubble_A000_NEW.srec erase

```
Inject command 'flash-image'
Successful generic response to command 'flash-erase-region'
Wrote 16540 bytes to address 0xa000
Successful generic response to command 'write-memory'
(1/1)100% Completed!
Successful generic response to command 'write-memory'
Response status = 0 (0x0) Success.
```

Reset the board and see that the verification process succeeds.

9 Conclusion

Thanks for completing the secure boot lab. Please explore the resources section.