AN12352

LPC54S0xx Execute In Place with Secure Boot

Rev. 2 — 18 September 2020 Application Note

Contents

1 Introduction....1
1.1 Terminology....3
2 Implementation....4
2.1 Overview....4
2.2 Divide the image binary....4
2.3 Create the image (MCUXpresso IDE)....6
2.4 Program the secure bootable and non-secure part images....10
2.5 Convert key file generated by elftosb....13
2.6 Program 128 bits AES key and related OTP bit fields to enable secure boot....14
3 Demonstration....15
3.1 Environment....15
3.2 Steps and result....15
4 Revision history....16

1 Introduction

Secure boot is an important feature for LPC54S0xx parts.

Secure boot can ensure that unauthorized images (code) are not executed on a given product. The secure bootloader in ROM is immutable code forming the Root of Trust. When secure boot is enabled, the boot ROM examines the user executable image loaded in on-chip RAM to determine the authenticity of the code. If the code is authentic, the control is transferred. This process establishes a chain of trusted code from ROM to the user boot code.

The secure bootloader in ROM loads the user code into on-chip RAM and executes it in RAM after authentication or decryption. When the secure boot is enabled, the image size, code size + RO size + RW size, should be smaller than one of the RAM blocks, SRAMX or SRAM0. The maximum bootable size, code size + RO size + RW size, is 192 KB.

Figure 1 shows the boot process.

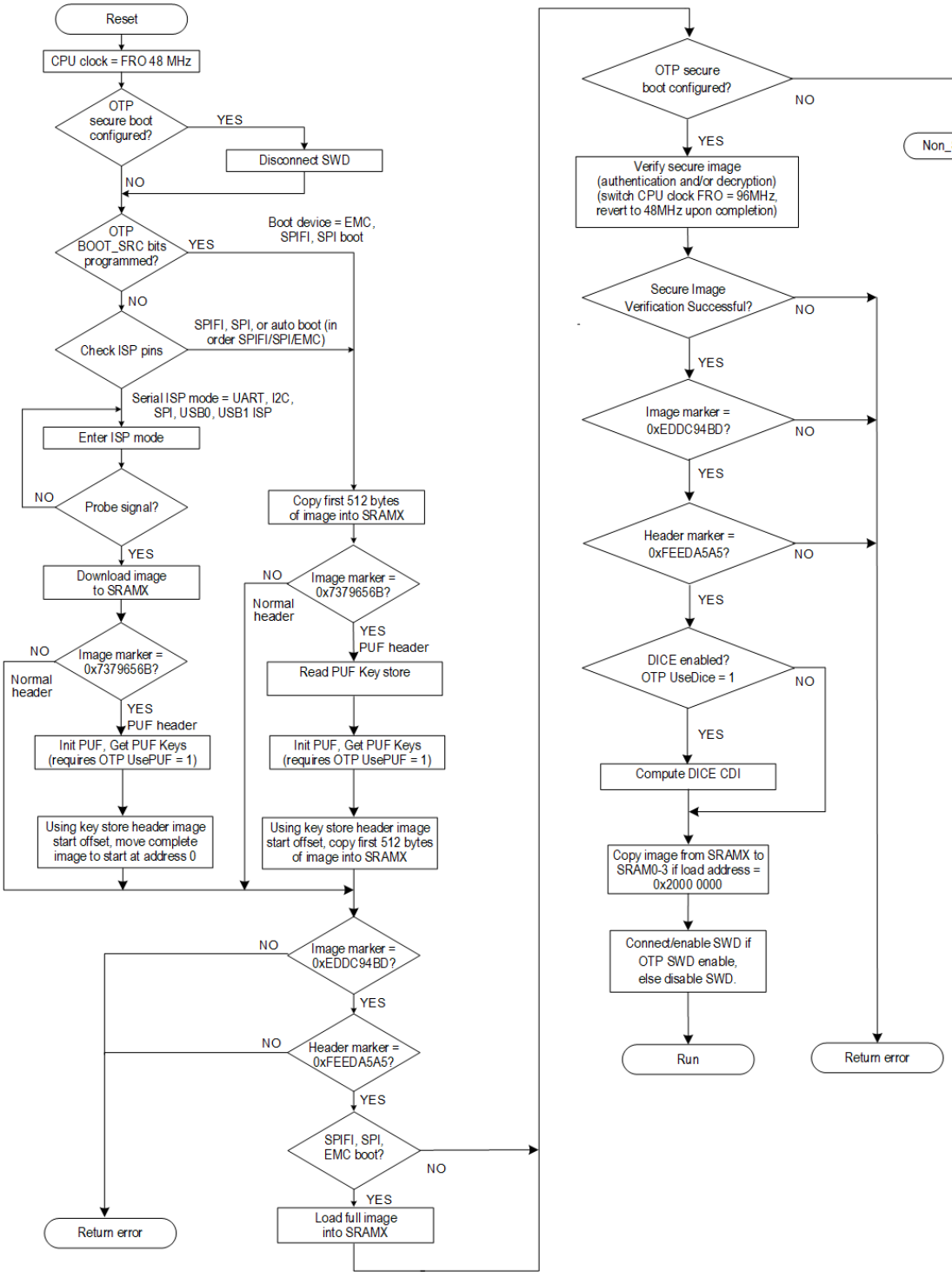

Figure 1. Secure boot process

When the secure boot is enabled, there is a size limitation and additional code limitation. The code is not executed in place (XIP) from QSPI Flash. To solve the aforementioned limitations, this application note describes a simple demo. The demo shows how to split the image into bootable part and XIP part. The bootable part contains secure bootable code, whereas the XIP part contains plain-text code. The secure bootable part is useful to secure the core code via image encryption and/or authentication.

NOTE

In this document, enabling secure boot is required to make secure boot, which is done by configuring secure boot type field in OTP. As an example, this document configures it to Enforce Encryption, with the `OTP_SECURE_BOOT_TYPE` field set to **b'10**.

NOTE

Modifying the OTP is a one-time operation and is not reversed. Thus, care must be taken before writing to OTP secure boot type field and other related fields.

1.1 Terminology

Table 1 lists the terminology used in the following application note sections.

Table 1. Terminology

Items	Description
Secure Bootable Image	A bootable image that is encrypted or signed, and so on. Additionally, it meets the requirements of the secure boot type.
Non-Secure (NS) Image	Plain text image.
Flashloader	The Flashloader is the secondary bootloader program loaded into the on-chip RAM of LPC54S0xx to support *blhost*. The project is located in SDK as a bootloader demo.
DFU Utility	The DFU utility is the host application used to load the Flashloader binary into the internal RAM memory of LPC540xx device connected to the host in USB DFU mode. *dfu-util.exe* is an open source command-line application. To download the tool, see dfu-util.
blhost	PC Command-Line Interface (CLI) tools to implement MCUBOOT protocol, it is part of MCUBOOT software package. The *blhost.exe* utility is an example host program used to interface with LPC54S0xx running the Flashloader program. This tool can be downloaded from MCUBOOT.
HxD	HxD is a binary file editor. It is easy to use and HxD is free of charge for private and commercial use.
elftosb	The *elftosb* tool creates a binary output file that contains the user application image along with a series of bootloader commands. The output file is known as a *Secure Binary* or SB file for short. These files have the **.sb* extension. The tool uses an input command file to control the sequence of bootloader commands present in the output file. This command file is called a *boot descriptor file* or BD file for short. This tool can be downloaded from MCUBOOT.
elftosb-gui	The *elftosb-gui* is a GUI tool with a main focus to help the user prepare a secure application image, as well as other useful security operation specific to target MCU platform. The *Elftosb-gui* tool provides intuitive graphical interface on top of `elftosb` and `blhost` command-line applications and it guides user in preparation of secure boot images required by ROM bootloader. This tool can be downloaded from MCUBOOT.

2 Implementation

This section introduces how to split the code into two parts.

- **Secure Bootable part (up to 192 kB)**
  - Contains confidential code that may or may not be performance sensitive (vector table, time constrained critical algorithm, and so on.)
  - Encrypted or signed as per secure image formats based on secure boot type.

The secure bootloader in ROM will load this Secure Bootable Part into RAM, and executes it after successful verification. Since the secure bootloader disables the XIP, it is required that SPIFI is initialized to enable XIP.

- **Non-Secure part (XIP)**
  - Contains code that is not confidential.
  - XIP code and the code loaded into RAM are placed in flash in plain text format.

2.1 Overview

The following steps are required to create a separate image.

1. Divide the image into two parts by modifying linker script.
   - Divide the image into secure bootable part and non-secure part through linker script. The division helps place the code identified as protected in the secure bootable part and the non-protected code in the non-secure part.
2. Create the image.
   - After coding, compile the code. The binary is generated based on the linker script.
   - Use tools to split the image into two parts: Secure Bootable Part and Non-Secure Part and then process them.
3. Program the two parts of the image into the flash.
   - Use MCUXpresso and CMSIS-DAP to program the images.
4. Program the 128 bits AES key to OTP.
5. Program the related OTP bit fields to enable secure boot.
   - Secure boot type.
   - Secure boot enable.

2.2 Divide the image binary

Figure 2 shows an example of a special image layout.

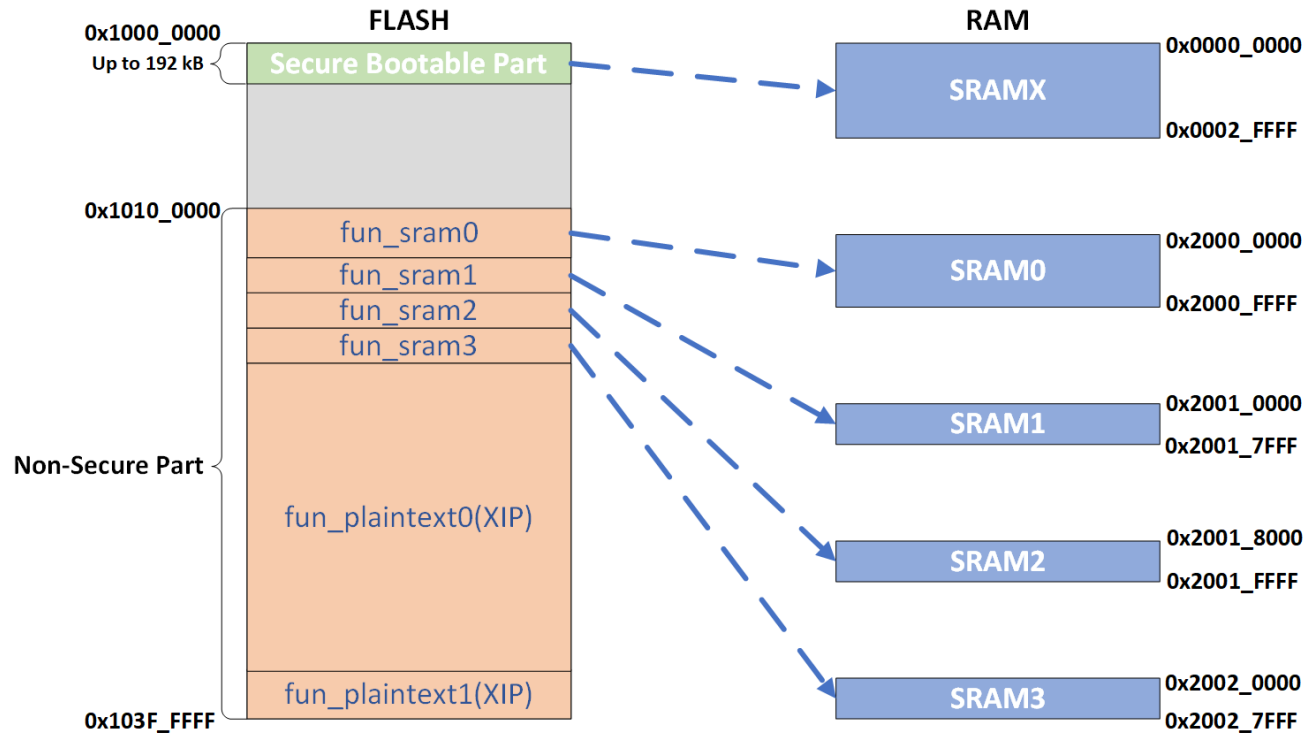

Figure 2. Flash and RAM map

NOTE

Before the code is executed, the **fun_sram0**, **fun_sram1**, **fun_sram2**, and **fun_sram3** sections are loaded into the RAM.

In the demo project, MCUXpresso IDE, these sections are loaded into execution address by **ResetISR** provided by the SDK.

In the MCUXpresso IDE environment, the **ld** files of the project are modified to achieve this image layout.

As shown in Figure 3, section **fun_plaintext1** is defined and placed in the non-secure part. **fun_plaintext1** starts at `0x103F_FF00`.

```
.text_plaintext1 0x103FFF00 : AT(0x103FFF00)
{
    *(.fun_plaintext1)
} > BOARD_FLASH_NS
```

Figure 3. Linker scripts

The non-confidential code is placed in one of the above Non-Secure (NS) sections of the image through **attribute** directive when declaring functions. The following code snippet places **ns_print_with_banner** function code in section **func_plaintext1** placed in the non-secure part:

```
__attribute__ ((section(".fun_plaintext1"))) void ns_print_with_banner(void)
{
    PRINTF("<NS-FLASH:>I'm from non-secure part of QSPI Flash.\r\n");
    PRINTF("<NS-FLASH:>My address: 0x%08X.\r\n", ns_print_with_banner);
}
```

Figure 4. Example code of attribute function

2.3 Create the image (MCUXpresso IDE)

The steps to create the image in MCUXpresso IDE environment are:

1. Initialize SPIFI to enable XIP
2. Build and generate the image
3. Split the image as secure-plain text and non-secure
4. Create the secure bootable part image based on secure-plain text image

2.3.1 Initialize SPIFI to enable XIP

This step is a must to enable XIP if the application code is larger than 192 KB and secure boot is enabled.

Complete the SPIFI initialization in the secure bootable part.

The code to initialize SPIFI for XIP is shown in Figure 5.

```
void app_spifi_init(void)
{
    spifi_config_t config = {0};
    uint32_t sourceClockFreq;
    spifi_command_t command[COMMAND_NUM] = {
        {PAGE_SIZE, false, kSPIFI_DataInput, 1, kSPIFI_CommandDataQuad, kSPIFI_CommandOpcodeAddrThreeBytes, 0x6B},
        {PAGE_SIZE, false, kSPIFI_DataOutput, 0, kSPIFI_CommandDataQuad, kSPIFI_CommandOpcodeAddrThreeBytes, 0x32},
        {1, false, kSPIFI_DataInput, 0, kSPIFI_CommandAllSerial, kSPIFI_CommandOpcodeOnly, 0x05},
        {0, false, kSPIFI_DataOutput, 0, kSPIFI_CommandAllSerial, kSPIFI_CommandOpcodeAddrThreeBytes, 0x20},
        {0, false, kSPIFI_DataOutput, 0, kSPIFI_CommandAllSerial, kSPIFI_CommandOpcodeOnly, 0x06},
        {1, false, kSPIFI_DataOutput, 0, kSPIFI_CommandAllSerial, kSPIFI_CommandOpcodeOnly, 0x31}};

    RESET_PeripheralReset(kSPIFI_RST_SHIFT_RSTn);

    /* Set SPIFI clock source */
    CLOCK_AttachClk(kFRO_HF_to_SPIFI_CLK);
    sourceClockFreq = CLOCK_GetFroHfFreq();

    /* Set the clock divider */
    CLOCK_SetClkDiv(kCLOCK_DivSpifiClk, sourceClockFreq / EXAMPLE_SPI_BAUDRATE, false);

    /* Initialize SPIFI */
    SPIFI_GetDefaultConfig(&config);
    SPIFI_Init(EXAMPLE_SPIFI, &config);

#if defined QUAD_MODE_VAL
    /* Enable Quad mode */
    enable_quad_mode();
#endif

    /* Setup memory command to enable XIP */
    SPIFI_SetMemoryCommand(EXAMPLE_SPIFI, &command[READ]);
}
```

Figure 5. Example code of SPIFI initialization

2.3.2 Build and generate the image

When the software of the project is completed, the project is compiled and then the **.axf* file is generated.

The simplest way to create a one-off binary or a hex file is to open up the **Debug**, or **Release**, folder in the Project Explorer. Right-click on the **.axf* file, and select the **Binary Utilities > Create binary** option as shown in Figure 6.

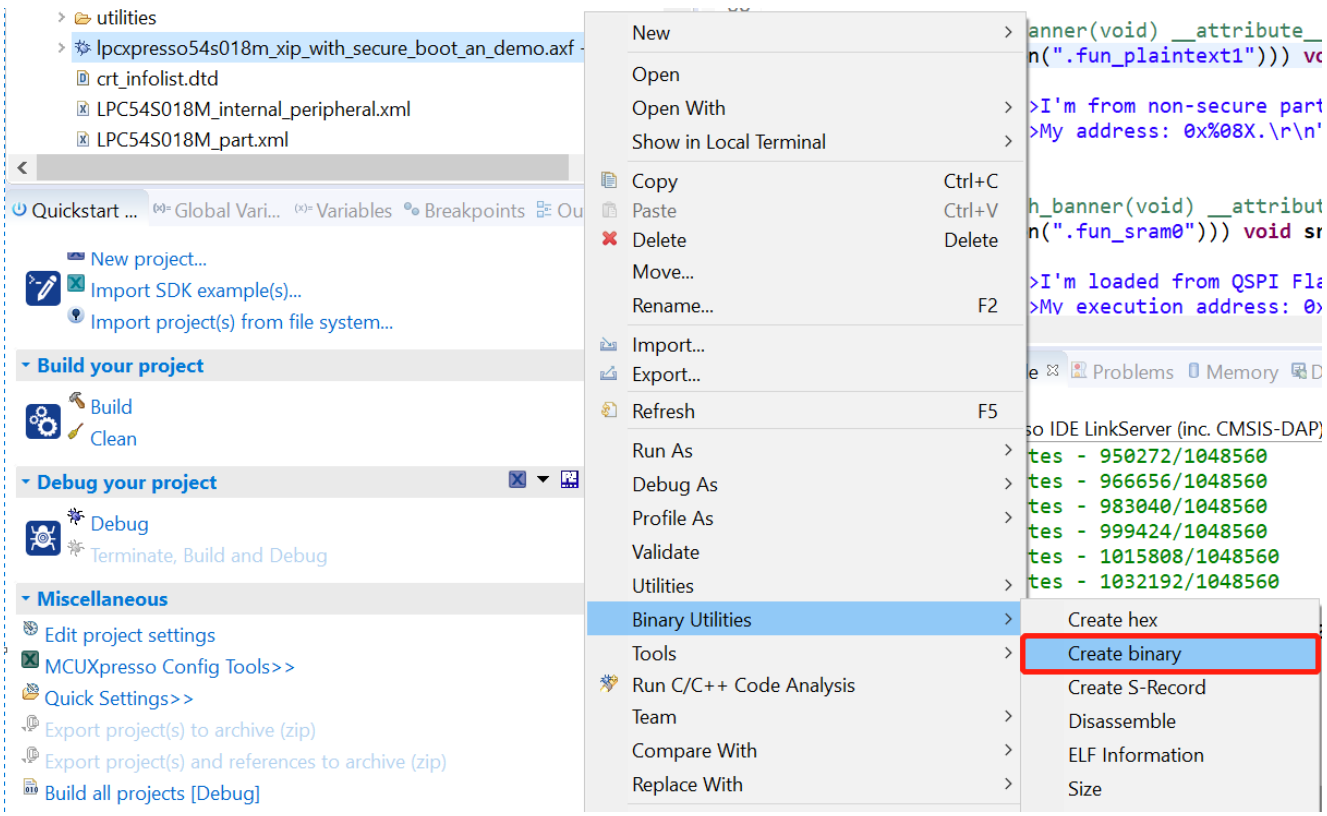

Figure 6. Generate the binary

2.3.3 Split the image as secure-plain text and non-secure

It is recommended to use HxD, to split the image.

Table 2 describes the plain image layout.

Table 2. Plain image layout

Offset	Block	Value	Description
0x00	Arm Vector table	__initial_sp	Stack pointer
0x04	Arm Vector table	__initial_pc	Image execution start address
……	……	……	……
0x28	HEADER_OFFSET	HEADER_OFFSET	A typical offset value is **0x160**.
……	……	……	……
HEADER_OFFSET+0x0C	Image_length	xxxxx	**Total length of the image -4.** The length

Table continues on the next page...

Table 2. Plain image layout (continued)

Offset	Block	Value	Description
			does not include the four bytes that make up the CRC value field.
……	……	……	……

The image length is obtained from the image header.

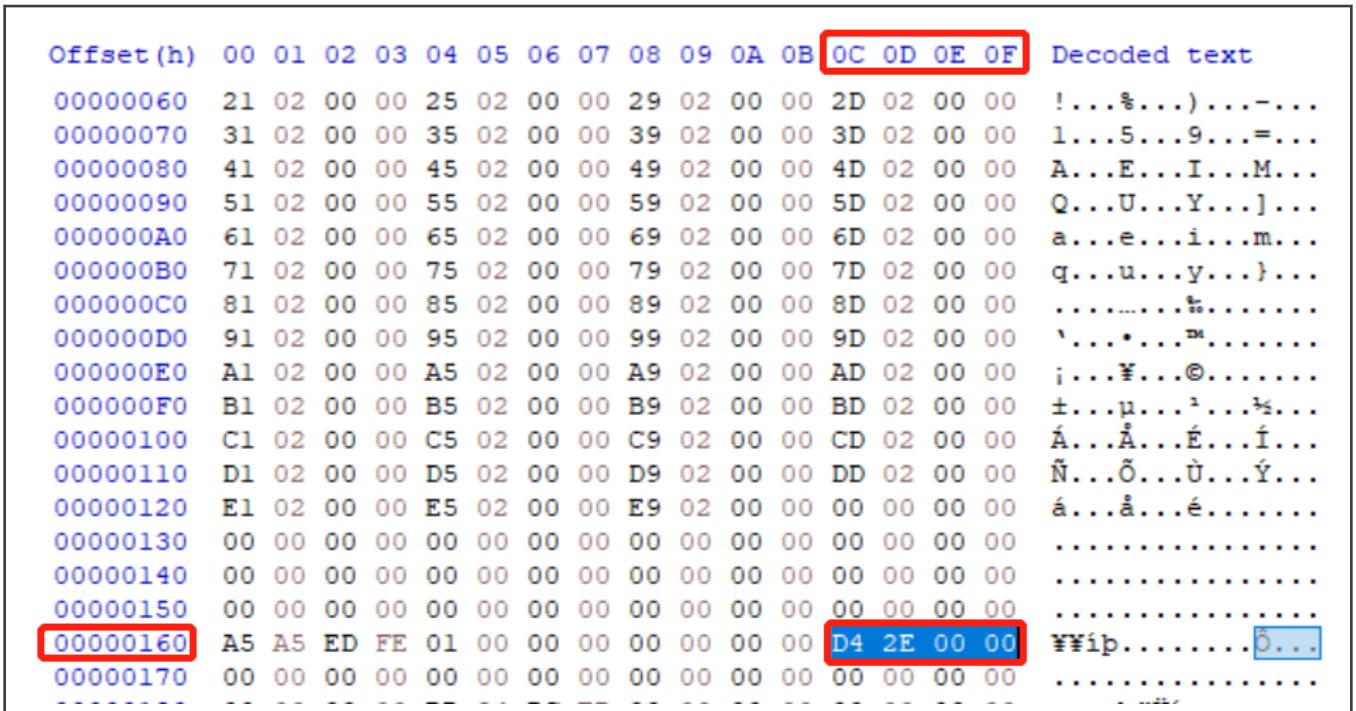

Figure 7. Image_length (image_length in this header equals 0x0000_2ED4)

The total length of the image in bytes= image_length + 4.

In order to generate secure-bootable part and non-secure part images, the original image binary is split into secure-plain text image and non-secure image.

The secure-plain text image is from address 0 to address (total length of the image – 1) of the original image binary. This image is used to create the secure-bootable part image.

The non-secure image is from address 0x0010_0000 (0x1010_0000 - 0x1000_0000) to the end of the original image. This image is as non-secure part image.

2.3.4 Create the secure bootable part image based on secure-plain text image

Use the *elftosb* and *elftosb-gui* to create the secure-bootable part image.

- Generate 128 bits AES key.

  Use the following command to generate 128 bits AES key.

  ```
  elftosb.exe --keygen 128 aes128_key.key
  ```

  Where "*aes128_key.key*" is the name of AES key file which stores AES128 key.

- Create the secure-bootable image

Open *elftosb-gui*, to create the secure-bootable image by following the steps shown in Figure 8

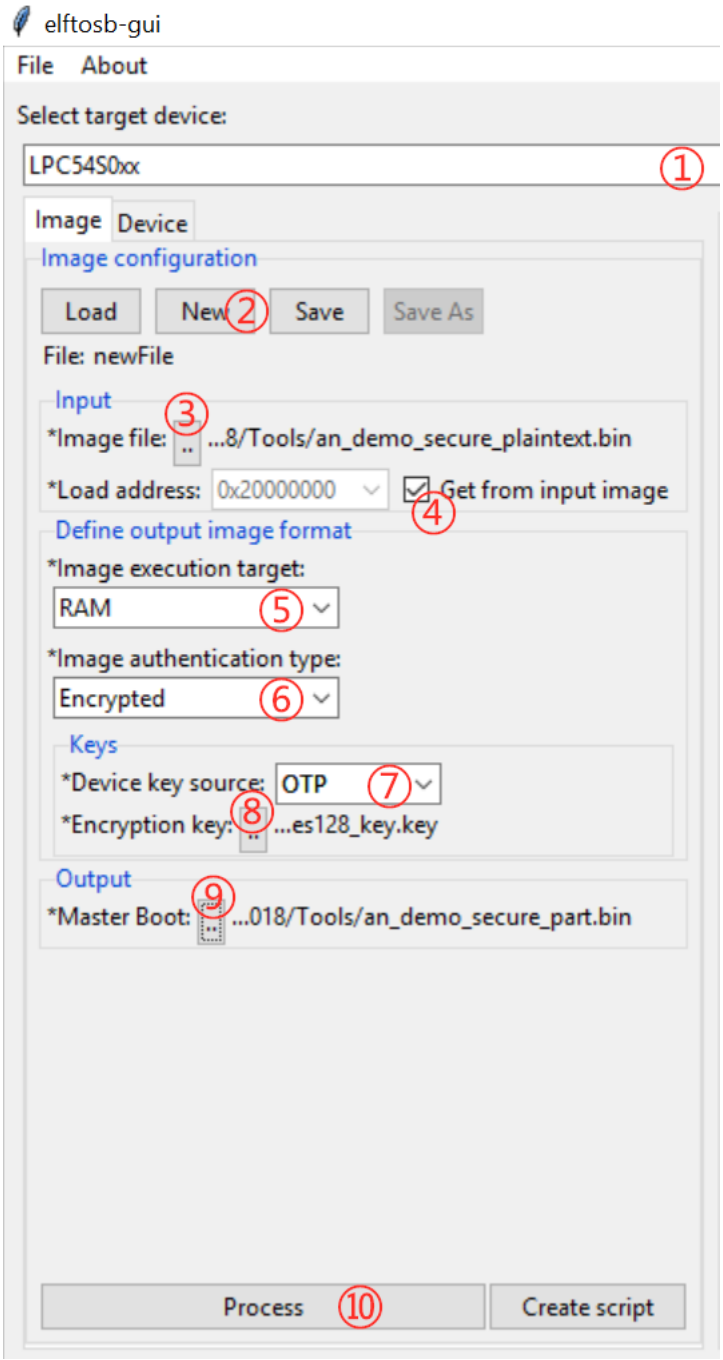

Figure 8. Create the Secure Bootable image by *elftosb-gui*

1. Select the LPC54S0xx device.
2. Create a new configuration.
3. Select the secure-plain text binary image.
4. Get the load address from the input image.
5. Select the image execution target as **RAM**.
6. Select the image authentication type as **Encrypted**.

7. Select the device key source as **OTP**.
8. Select the encryption key (the 128 bits AES key generated before).
9. Select the path and name of the output encrypted image.
10. Click the **Process** button to create the secure-bootable part image.

2.4 Program the secure bootable and non-secure part images

To program the Flash, it is recommended to use MCUXpresso and CMSIS-DAP.

NOTE

Jflash is not recommended. Jflash fills the checksum into the image during the programing process, because the image cannot pass verification during the secure boot.

2.4.1 Program secure bootable part image into Flash

1. Open MCUXpresso IDE with any SDK project of LPC54S018M or LPC54S018.

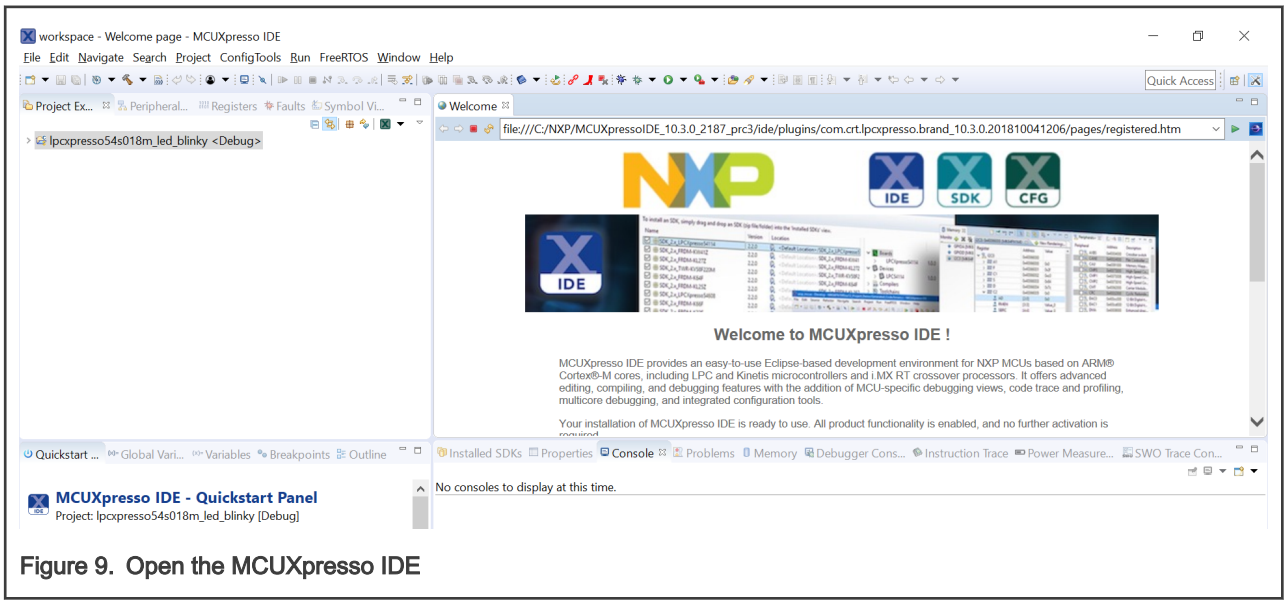

Figure 9. Open the MCUXpresso IDE

2. Open **MCUXpresso IDE LinkServer (inc.CMSIS-DAP) probes** by clicking the button in the order as shown in Figure 10.

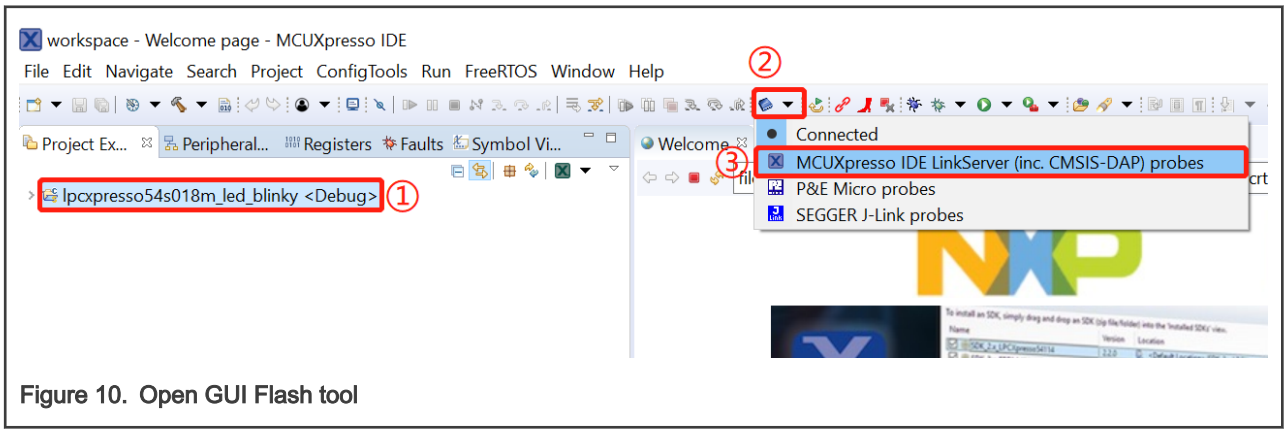

Figure 10. Open GUI Flash tool

The result is as shown in Figure 11.

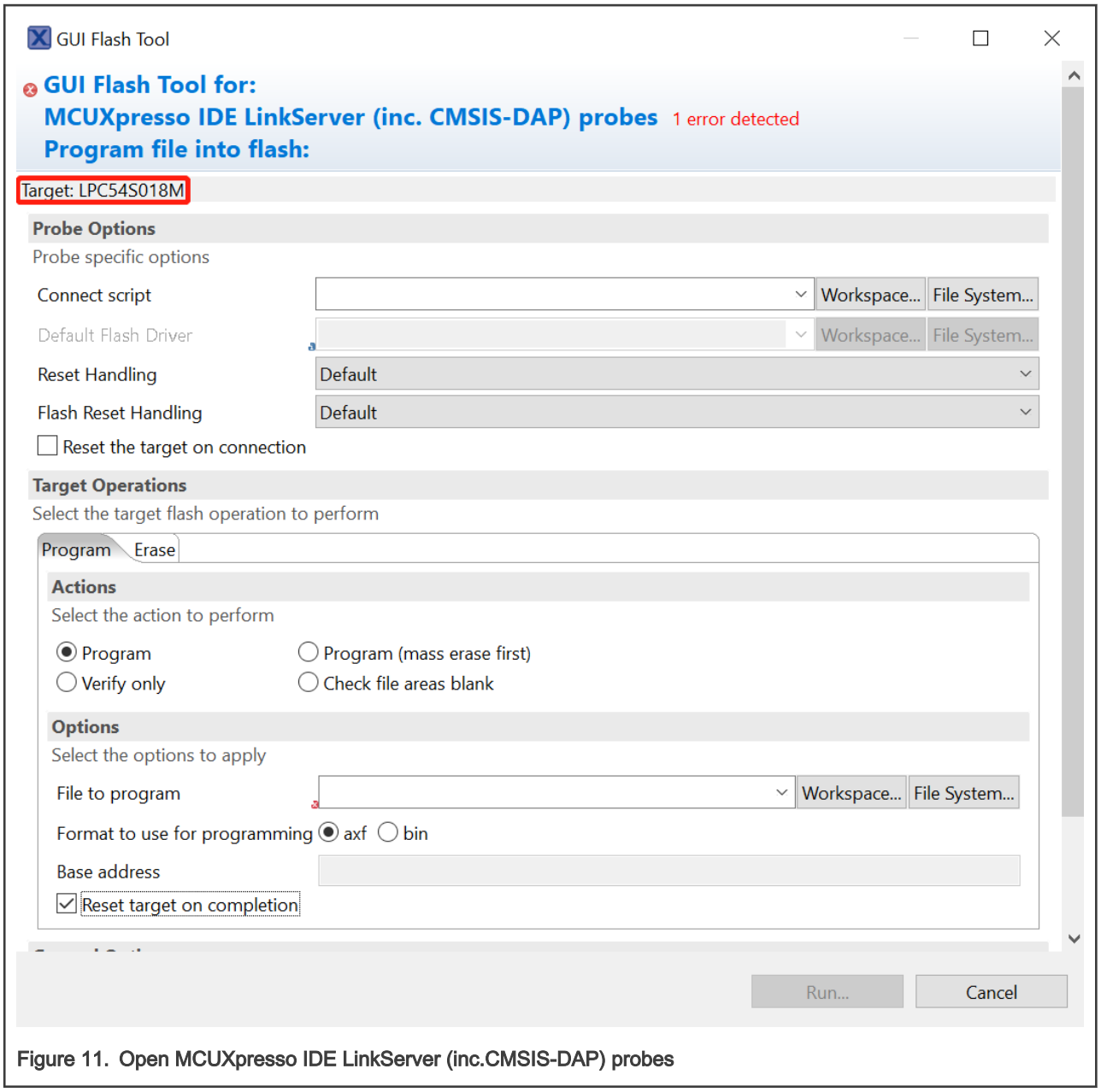

Figure 11. Open MCUXpresso IDE LinkServer (inc.CMSIS-DAP) probes

Follow the screenshot shown in Figure 12 to configure it, especially the red parts of the screenshot.

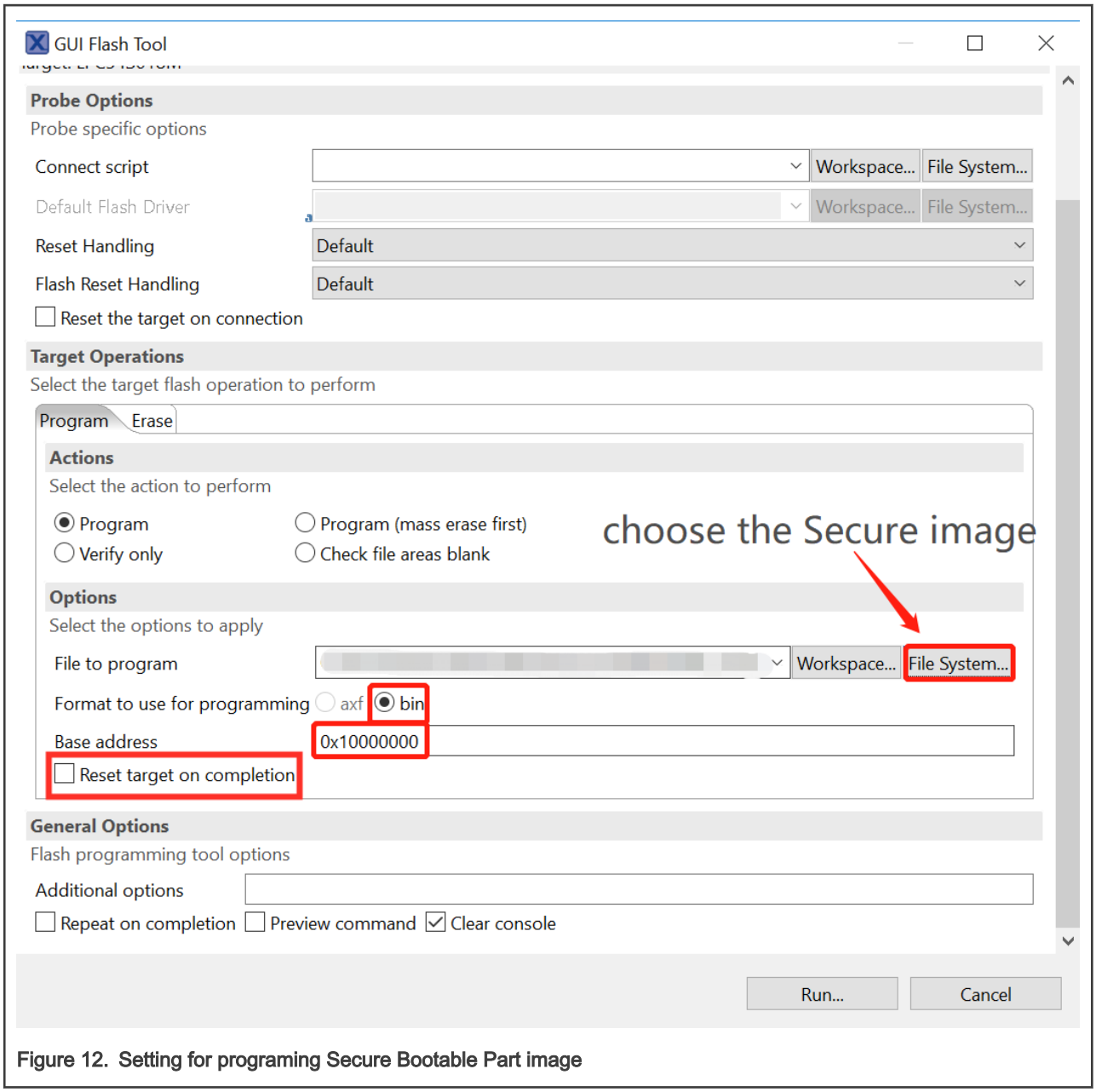

Figure 12. Setting for programing Secure Bootable Part image

3. Click the **Run** button to program the Secure Bootable Part image into the Flash.

2.4.2 Program the non-secure part image into Flash

1. Follow Step 1 and Step 2.
2. Change the configuration as shown in Figure 13, especially the red parts in the screenshot.

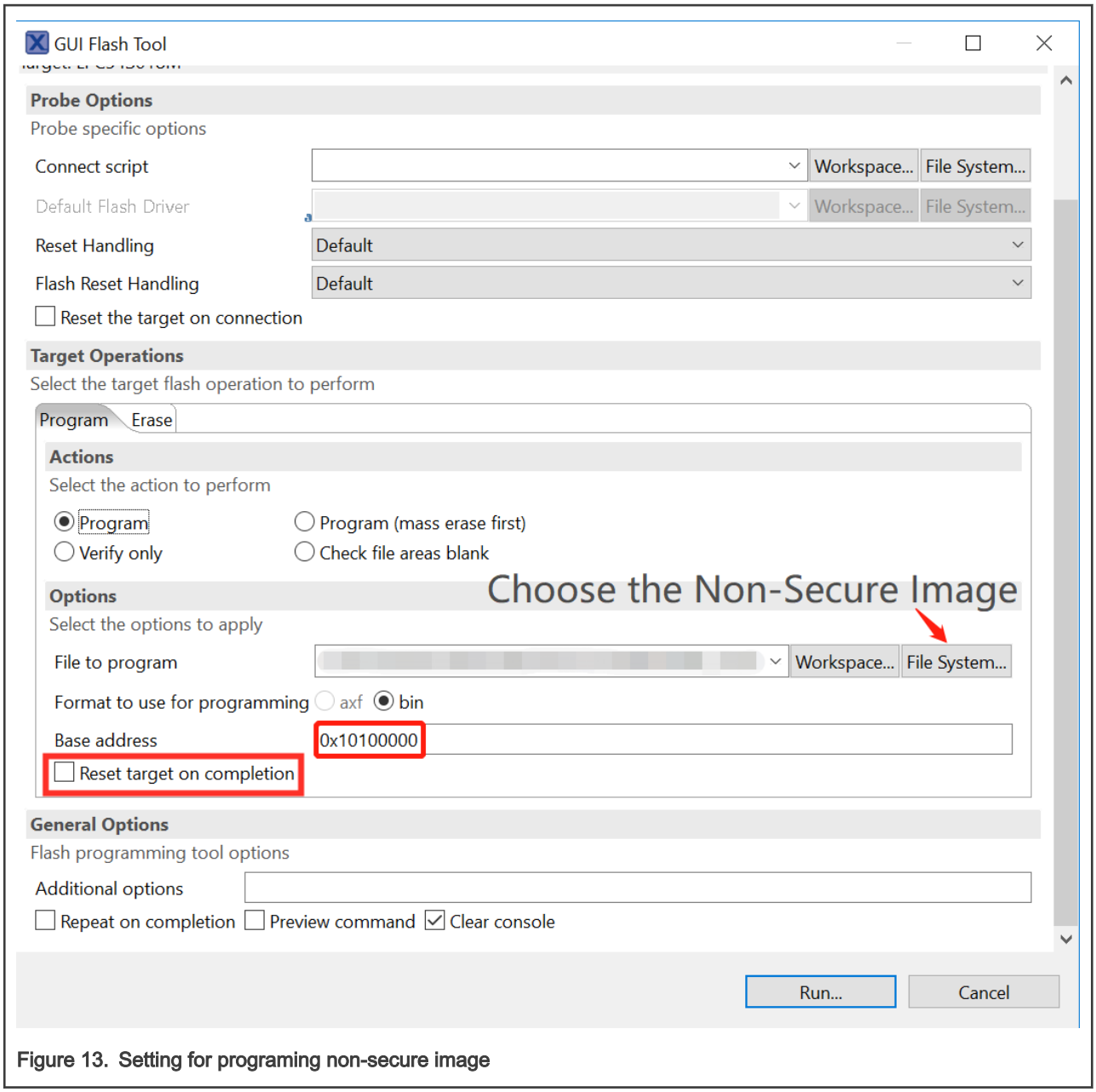

Figure 13. Setting for programing non-secure image

Click the **Run** button to program the non-secure image into the Flash.

2.5 Convert key file generated by elftosb

The key file generated by *elftosb* is in the ASCII format. It should be converted to hexadecimal format for blhost,

Figure 14 and Figure 15 show how to convert the key file.

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000   30 45 30 31 37 37 39 35 37 33 44 32 37 45 43 38  0E01779573D27EC8
00000010   31 38 39 34 35 31 30 30 31 36 42 34 39 39 42 32  1894510016B499B2
```

Figure 14. aes128_key.key

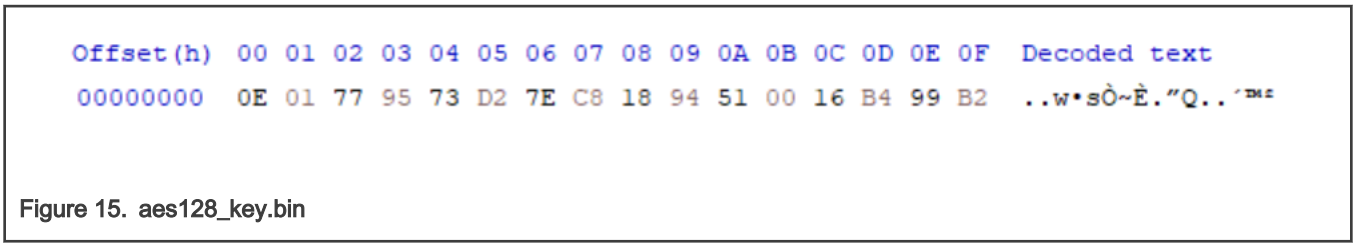

Figure 15. aes128_key.bin

2.6 Program 128 bits AES key and related OTP bit fields to enable secure boot

It is recommended to use ***blhost*** to program the OTP bits. For LPC54S0xx, the flashloader should be load into on-chip RAM and then the ***blhost*** will be available.

2.6.1 Use DFU to load the flashloader into the RAM

Configure the ISP pins to make the chip enter the USB0 DFU boot mode.

Table 3. Boot source based on ISP pins

Boot mode	ISP2 PIO0_6 pin	ISP1 PIO0_5 pin	ISP0 PIO0_4 pin	Description
USB0 DFU boot	LOW	HIGH	LOW	USB DFU class is used to download image over the USB0 full-speed port into SRAM.

Connect the LPC54S0xx device USB0 and PC with USB.

Use the following command to load the flashloader into the RAM. ***flashloader.bin*** is located in ***an_lpc54s0_xip_with_secureboot***. It can also be generated by compiling the sdk project which is located in ***sdk\boards\lpcxpresso54s018\bootloader_examples\flashloader***.

```
dfu-util.exe -D flashloader.bin
```

2.6.2 Use blhost to program 128 bits AES key and related OTP bit fields

Once the flashloader binary is downloaded on the device connected in USB DFU mode and starts its execution on the LPC54S0xx platform, there remains a physical USB connection between the LPC54S0xx platform **USB1 (High-Speed)** and host. The flashloader will be ready to receive the commands.

2.6.2.1 128 bits AES key

Use the following command to program 128 bits AES key.

```
blhost.exe -u 0x1fc9,0x01a2 -- program-aeskey aes128_key.bin
```

2.6.2.2 Secure boot type bit field

Use the following command to program the Secure boot type as **Enforce Encryption**.

```
blhost.exe -u 0x1fc9,0x01a2 -- efuse-program-once 12 00000010
```

2.6.2.3 Secure boot enable bit field

Use the following command to enable the secure boot.

```
blhost.exe -u 0x1fc9,0x01a2 -- efuse-program-once 12 00000004
```

3 Demonstration

This section describes the environment and the demo steps and results.

3.1 Environment

This section describes the hardware and software environment.

3.1.1 Hardware environment

- Board
  - LPCXpresso54S018 (LPC54S018-EVK) or LPCXpresso54S018M (LPC54S018M-EVK)
- Debugger
  - Integrated CMSIS-DAP debugger on the board
- Miscellaneous
  - Two Micro USB cables
  - PC

3.1.2 Software environment

- Tool chain
  - MCUXpresso IDE v10.3.0
- Software package
  - an_lpc54s0_xip_with_secureboot.zip

3.2 Steps and result

The basic steps are as follows:

1. **Build & Compile**

   Build and compile the demo project located in ***an_lpc54s0_xip_with_secureboot/an_demo***.

2. **Process image**

   Process the image according to Split the image as secure-plain text and non-secure and Create the secure bootable part image based on secure-plain text image .

3. **Download**

   Follow Program the secure bootable and non-secure part images to download images.

4. **Program the AES key.**

   Follow Convert key file generated by elftosb to program the AES key.

5. **Program the related OTP bit fields**

   Follow Program 128 bits AES key and related OTP bit fields to enable secure boot to program the related OTP bit fields.

6. **Run**

Reset the board to run by pressing the **Reset** button on the board.

7. **Result**

Figure 16 shows the messages printed on the terminal, 115200+8+N+1, by the demo code.

```
<S:>The image has been loaded into SRAMX from flash by ROM code.
<S:>The encrypted image has been descrypted by ROM code.
<S:>Executed the image located in SRMAX.
<S:>Enable the SPIFI for executing the plain-text code located in flash.
<S:>Call a function located in flash(XIP).

<NS-FLASH:>I'm from non-secure part of QSPI Flash.
<NS-FLASH:>My address: 0x103FFF0D.

<NS-SRAM0:>I'm loaded from QSPI Flash, and excuted from SRAM0.
<NS-SRAM0:>My execution address: 0x2000100D.

<NS-SRAM1:>I'm loaded from QSPI Flash, and excuted from SRAM1.
<NS-SRAM1:>My execution address: 0x20010001.

<NS-SRAM2:>I'm loaded from QSPI Flash, and excuted from SRAM2.
<NS-SRAM2:>My execution address: 0x20018001.

<NS-SRAM3:>I'm loaded from QSPI Flash, and excuted from SRAM3.
<NS-SRAM3:>My execution address: 0x20020001.

<S:>The LED will blink per second.
<S:>Enter any character, which will be echoed to terminal.
```

Figure 16. Messages printed on the terminal

The information with banner, ***<S:>***, means it is printed in Secure Bootable Part image. The information with banner, ***<NS:>***, means it is printed in Non-Secure part image.

As described in the print information displayed on the terminal, the program will echo each entered character.

The onboard LED3 will also blink per second.

4 Revision history

Table 4 summarizes the changes since the initial release.

Table 4. Revision history

Revision number	Date	Substantive changes
0	18 February 2019	Initial release
1	25 February 2019	Updated Figure 13 and tools path in Terminology.
2	18 September 2020	• Updated Table 1 • Updated Convert key file generated by elftosb

Table continues on the next page...

Table 4. Revision history (continued)

Revision number	Date	Substantive changes
		• Updated Program 128 bits AES key and related OTP bit fields to enable secure boot • Updated Use blhost to program 128 bits AES key and related OTP bit fields • Added 128 bits AES key

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 18 September 2020

Document identifier: AN12352

arm