

YASB Renesas Synergy Secure Bootloader

Introduction

The YASB (Yet Another Synergy Bootloader) bootloader is a secure bootloader for Synergy devices which supports cryptographically signed update images. Signing an image can ensure it originated from a trusted source and has not been altered since being signed by that trusted source. A Python program is used to sign update images. The signature uses ECSDA (Elliptic Curve Digital Signature Algorithm) with the ECC curve secp256k1 and SHA256 hashing.

What this bootloader can do:

- Only allow official signed images to update the device and run on the device.
- Prevent images which have not been signed or signed with an invalid key to update and run on the device.
- Provide integrity that an image has not changed since being signed.

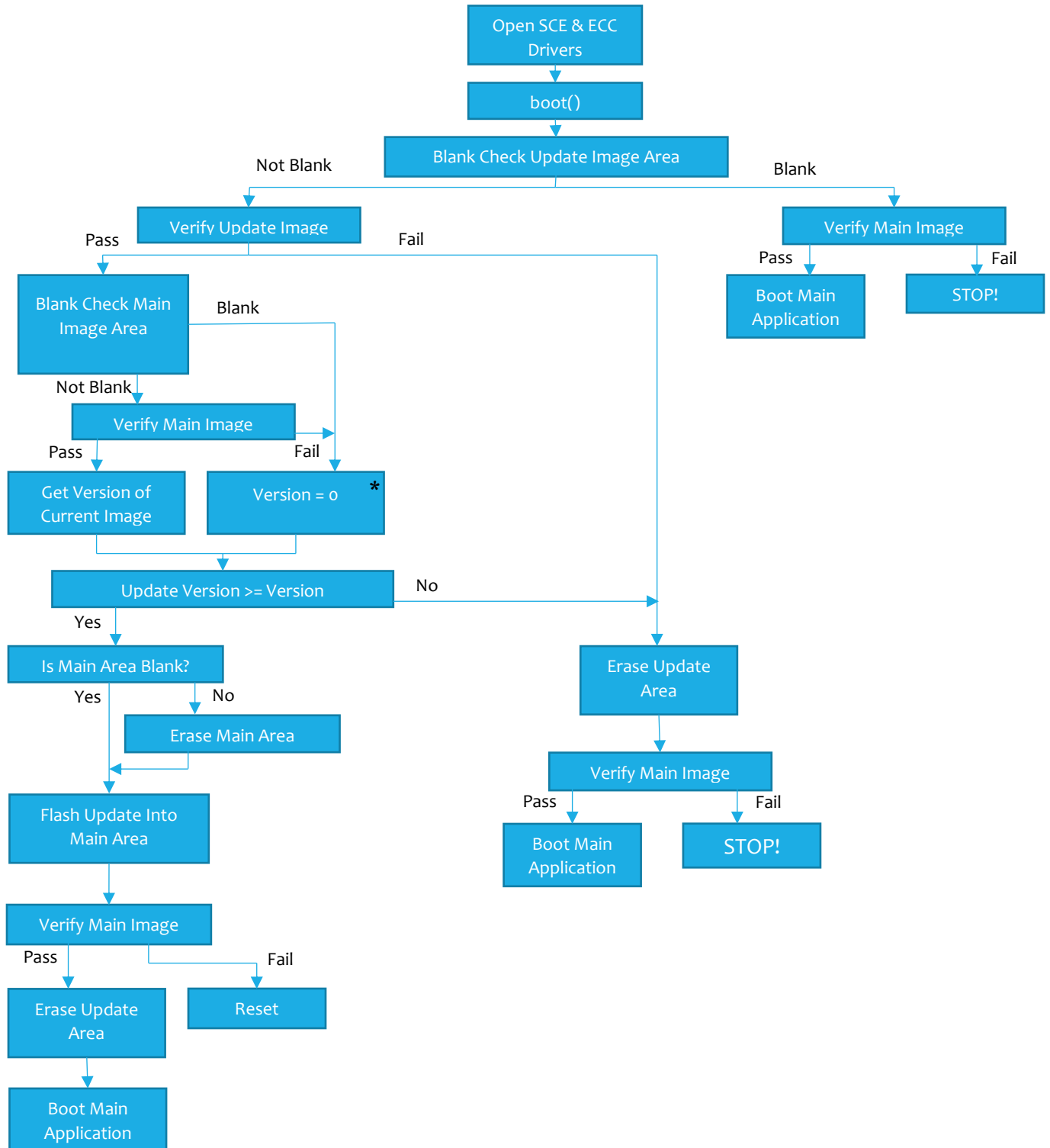
What this bootloader cannot do:

- Protect the image contents as images are not encrypted. So, the image contents could be loaded on a different device with a different boot program and run.

It is expected that updates will be delivered over a trusted medium and once delivered to the device are considered secure.

There is no download functionality built into the bootloader. It is expected that the application will download a new signed image into the update area and reset the device. On starting, the bootloader will look in the update area for a new valid image and update as necessary.

Flowchart



*** NOTE:**

This is an area of weakness.

This could allow an unwanted down grade to a previous version.

Scenario:

Valid update image in the upgrade area with a version equal or higher than the current version.

Bootloader starts to erase main application area so this area will fail future validation.

Device is stopped before applying the new image.

The update image is swapped with one with an earlier version number.

Restarting the bootloader will result in the application image being invalid and erased and the older update being applied.

This is an issue if it is possible to replace the update image without the need of the application.

When the update is in internal memory then this scenario can be mitigated against.

If the update is in external memory then consider preventing an update if there is not a valid image in the application area. This does come with a risk of being able to brick the device.

The trade-off is between recovering from a corrupted application image (from an interrupted update) against a down grade attack.

To prevent this possible attack at the risk of bricking stop at this point rather than setting the version to zero and continuing.

Image format

Magic Number	Signature Ecc-256	Length	Version	Padding	Image
--------------	-------------------	--------	---------	---------	-------

```
# Magic number - 4 bytes
# Signature - 64 bytes for ECC-256
# Length - 4 bytes
# Version - 4 bytes
# Padding - Space added to make header up to the specified header_size
# Binary image - Image dependent
#
# Header is Magic Number, Signature, Length, Version, Padding
#
# The Signature is from (and including) the Length field
# The Length field is from (and including) the Version field
# So, total size of the image is:
# Length + 4 (Length field) + 64 (Signature) + 4 (Magic Number)
```

Space for the header is 0x100 (256) bytes.

Signing tool

A Python 3 based program is supplied which:

- Generates ECC public and private key pair.
- Prints the public key and copies to the clipboard for inclusion in the bootloader source.
- Signs an update binary adding the fields shown in Image Format above.

```
usage: python3 yasb.py [-h] [-i INPUTFILE] [-k KEYFILE] [-v VERSION] [-o OUTPUTFILE] {sign,keygen,print}
```

Sign an image or create and show (print) keys for signing.

positional arguments:

{sign,keygen,print} Operation to perform - sign/keygen/print

optional arguments:

-h, --help show this help message and exit

-i INPUTFILE, --inputfile INPUTFILE

 Input image file to be signed

-k KEYFILE, --keyfile KEYFILE

 Key file used for signing the image (input only)

-v VERSION, --version VERSION

 Version number for the signed image

-o OUTPUTFILE, --outputfile OUTPUTFILE

 Output file, either the signed image or generated key file

e.g. Signing:

```
python yasb.py sign -i app.bin -k signingkey.bin -v 2 -o app_signed.bin
```

Generating an ECC secp256r1 keypair:

```
python yasb.py keygen -o signingkey.bin
```

Showing the signing key public part and copying to the clipboard:

```
python yasb.py print -k signingkey.bin
```

When generating a key pair the public key component should replace the contents of `g_public_key[]` in `/src/keys.c`

Porting

The first release of the bootloader includes a port for the Synergy S5D9 based PK-S5D9 board. This port supports locating the update image in either internal flash or external QSPI flash memory. Control of whether the update area is in QSPI flash is controlled by including the macro `UPDATE_USES_QSPI_FLASH` in `/src/port.h`

The files `/src/port.h` and `/src/port.c` include the device and board specific configuration and functionality. If porting the bootloader to a different Synergy device and/or board change the macros and functions accordingly. In most cases only minor changes will be required.

The bootloader uses the SCE7 to provide hardware acceleration of the ECC signature verification. If the bootloader is ported to a Synergy device with the SCE5 then ECC functionality will not be available. As such a software implementation of ECC signature verification will be necessary. In this case `/src/image_verify.c` will need to be modified to use the software routines. Take care choosing software implementations to minimize the risk of side channel attacks on the crypto operations.

Modify an Application to Use With the Bootloader

For an application to be compatible with the bootloader the linker script file requires a couple of modifications:

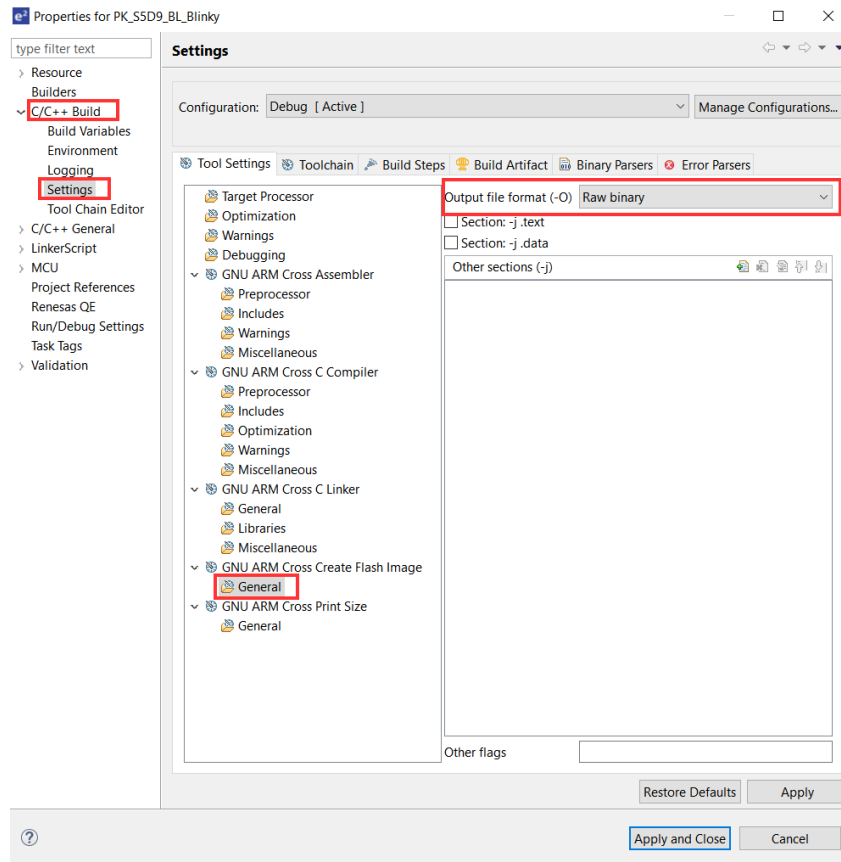
1. Change the FLASH section start address to the same value as `MAIN_IMAGE_START_ADDRESS` + the size of the header (default) 0x100 (256) bytes. So, for the supplied S5D9 version the FLASH section should start at 0x10100

```
/* Linker script to configure memory regions. */  
MEMORY  
{  
    FLASH (rx) : ORIGIN = 0x00010100, LENGTH = 0x00F8000 /* 968K */  
    RAM (rwx) : ORIGIN = 0x1FFE0000, LENGTH = 0x00A0000 /* 640K */  
    DATA_FLASH (rx) : ORIGIN = 0x40100000, LENGTH = 0x0010000 /* 64K */  
    QSPI_FLASH (rx) : ORIGIN = 0x60000000, LENGTH = 0x4000000 /* 64M, Change in QSPI section below also */  
    SDRAM (rwx) : ORIGIN = 0x90000000, LENGTH = 0x2000000 /* 32M */  
    ID_CODES (rx) : ORIGIN = 0x0100A150, LENGTH = 0x10 /* 16 bytes */  
}
```

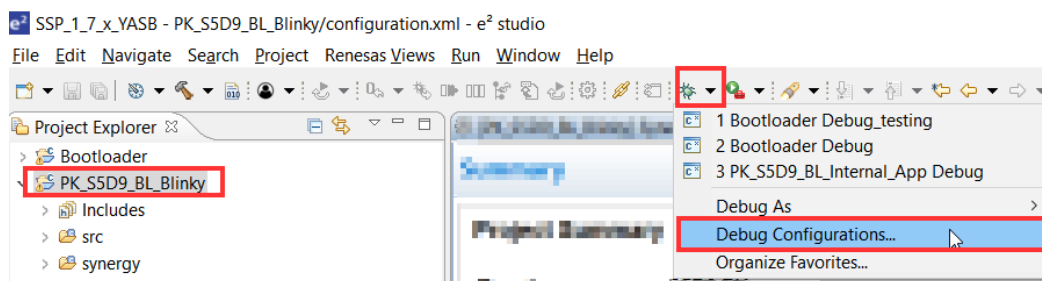
2. Modify the ID_CODE section so it is marked as NOLOAD. The ID codes are located in flash based registers which cannot be modified by application level flash programming and so will be configured by the settings in the bootloader.

```
.id_code_1 (NOLOAD):  
{  
    __ID_Codes_Start = .;  
    KEEP(*(.id_code_1*))  
    ID_Codes_End = .;  
} > ID_CODES
```

3. The signing tool expects the image to be in binary format. By default, the output from the build tools is ELF and SREC. Change the SREC output to Raw binary. Project -> Properties then expand C/C++ Build and select Settings then General in GNU ARM Cross Create Flash Image and change the Output File Format to Raw binary.



4. To debug the application located at the non-zero starting flash address add the following to the debug configuration for the project.
 - a. Select the project in the Project Explorer pane and click on the arrow next to the bug button and select “Debug configurations”.

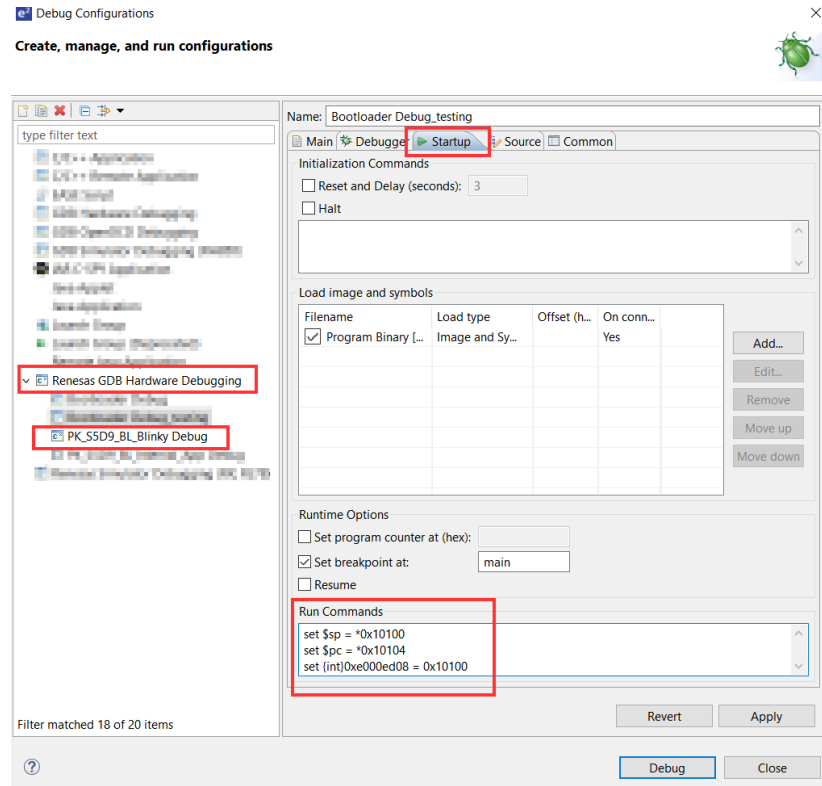


- b. Making sure the correct project is selected under “Renesas GDB Hardware Debugging” in the left pane select the Startup tab. In the Run Commands box enter:


```

set $sp = *0x10100
set $pc = *0x10104
set {int}0xe000ed08 = 0x10100

```



This sets the PC (Program Counter) and SP (Stack Pointer) to valid values for the project.

Import the Bootloader and Example Application

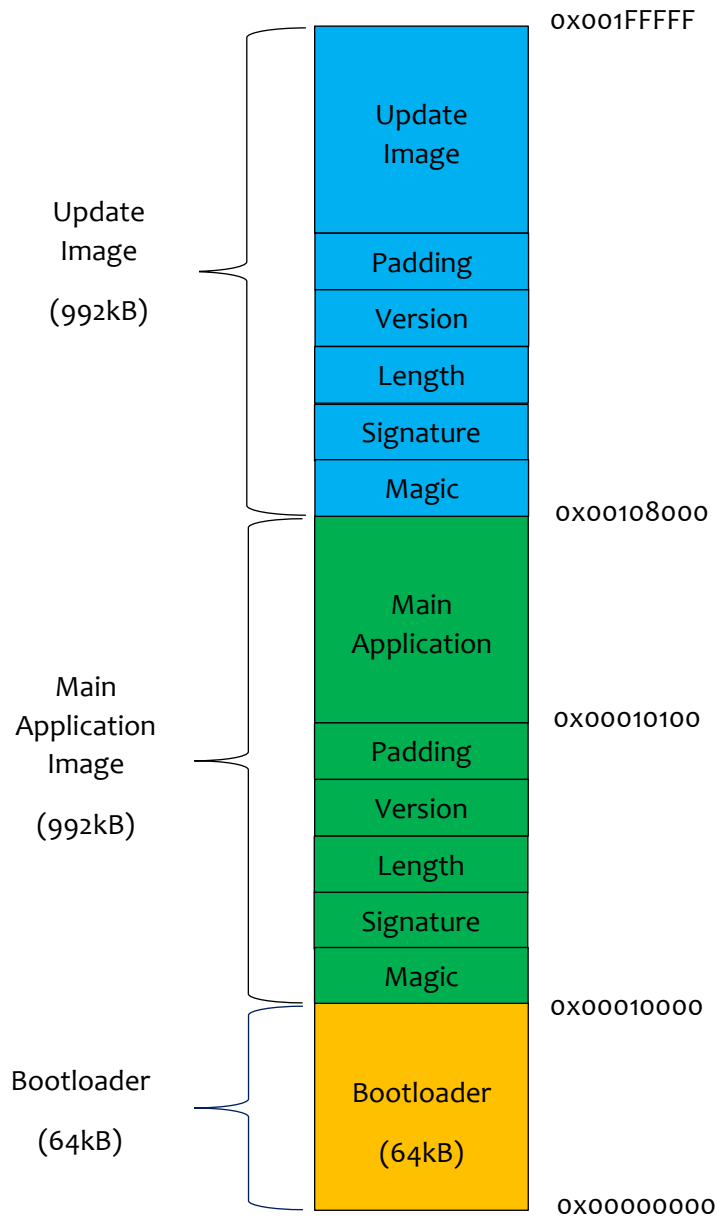
Download or clone the GitHub repo which contains the bootloader project and a RTOS based Blinky project configured for use with the bootloader. The projects require SSP 2.0.0 and e2studio 2021-04. The signing program has been tested with Python 3.8.3, Cryptography library 2.9.2 and Clipboard library 0.0.4.

Install the required Python libraries:

```
pip install cryptography==2.9.2
```

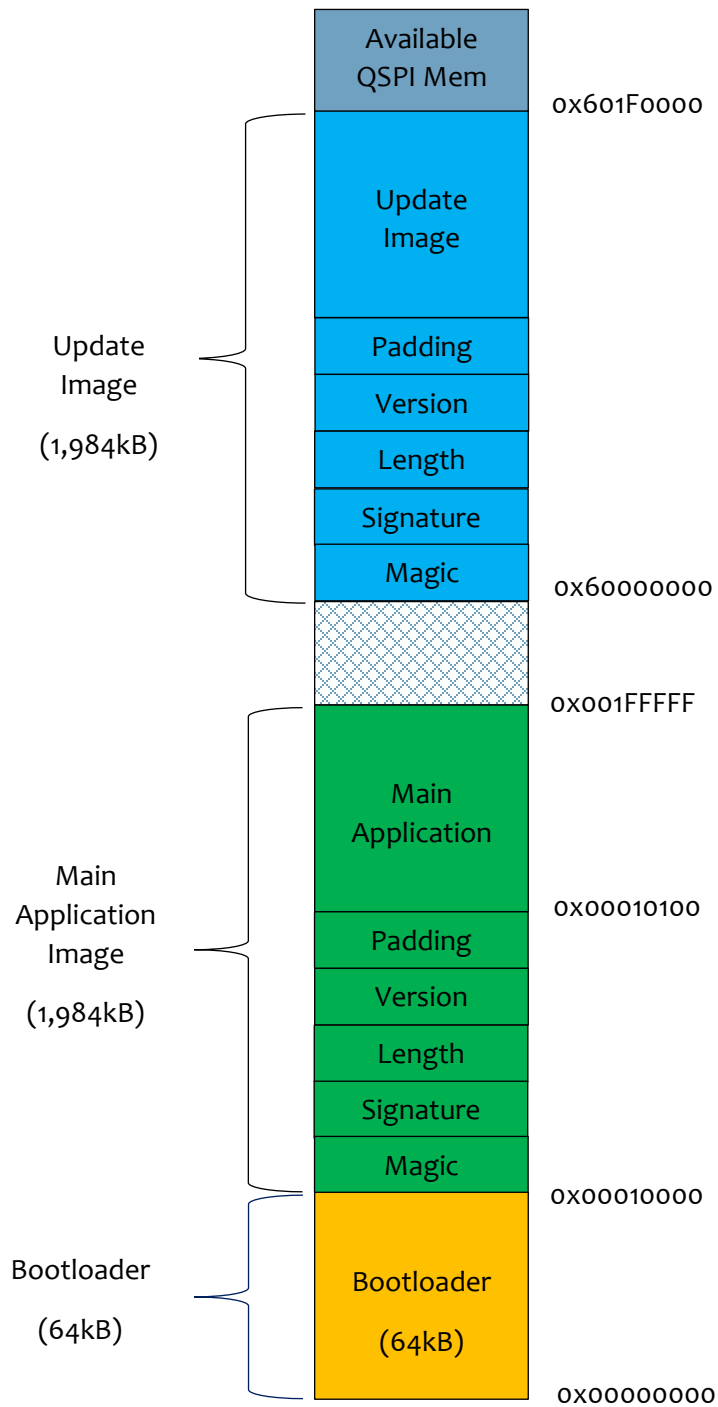
```
pip install clipboard==0.0.4
```

Memory Map when using the internal flash for the update image storage.



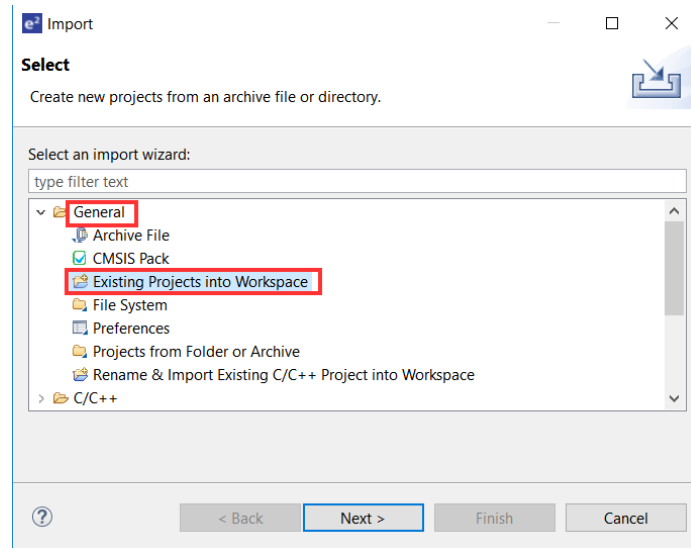
PK-S5D9 Bootloader Using Internal Flash Only

Memory Map when using the external QSPI flash for the update image storage.

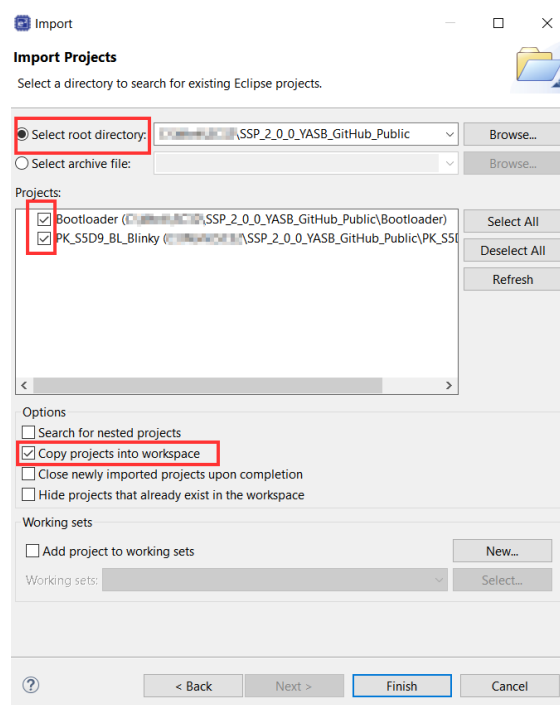


PK-S5D9 Bootloader Using QSPI Flash

1. Start e2studio.
2. File -> Import
3. Expand General and select “Existing Projects into Workspace” and click Next.



4. Select the “Select root directory” radio button and click “Browse” and navigate to the downloaded (unzipped) repo folder.
5. Make sure both projects – “Bootloader” and “PK_S5D9_BL_Blinky” – are selected
6. Make sure “Copy projects into workspace” is selected and click Finish.



7. Expand the project “PK_S5D9_BL_Blinky” in the Project Explorer pane on the left.
8. Double-click (open) “configuration.xml” and click “Generate Project Content”.



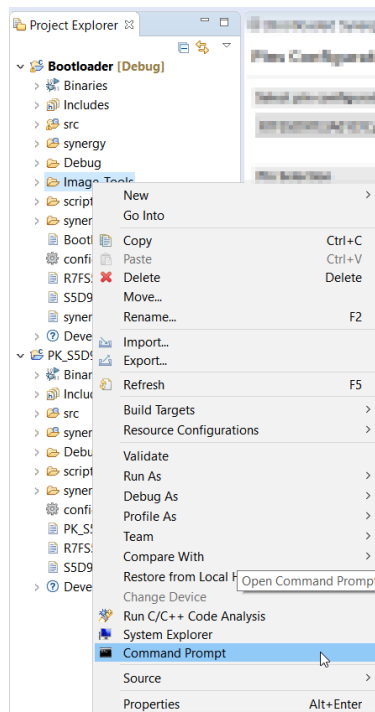
9. Build this project – CTRL-B or click the hammer button.



10. Expand the project “Bootloader” in the Project Explorer pane on the left.
11. Double-click (open) “configuration.xml” and click “Generate Project Content”.
12. Build this project – CTRL-B or click the hammer button.

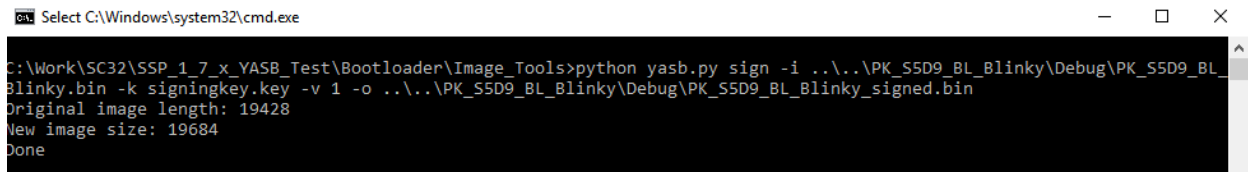


13. Right-click on the “Image_Tools” folder and select “Command Prompt”.



14. In the command prompt window run the signing tool (the below command line has been split across multiple lines for clarify but should be one line on the command line):

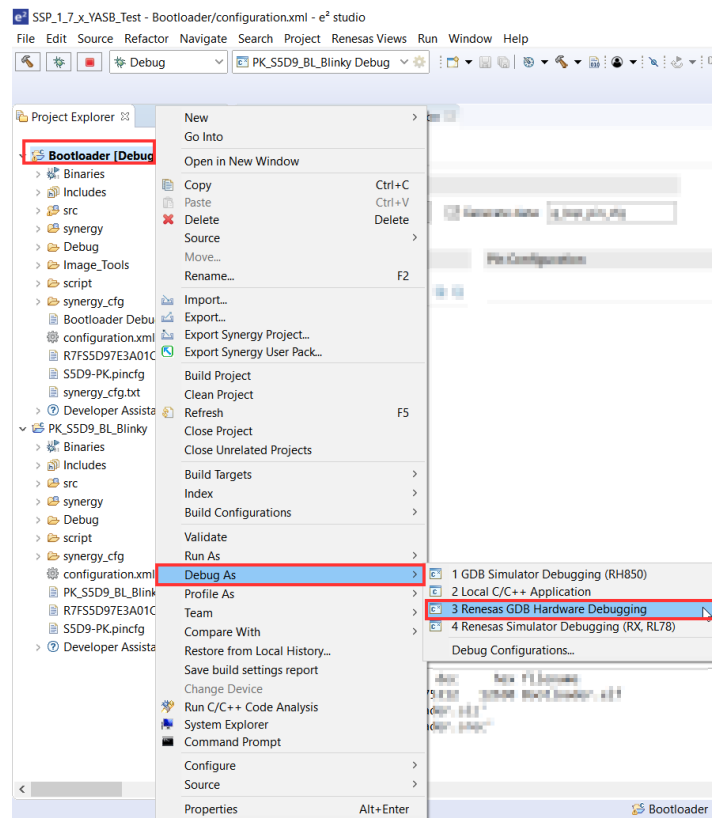
```
python yasb.py sign
-i ../../PK_S5D9_BL_Blinky\Debug\PK_S5D9_BL_Blinky.bin
-k signingkey.key
-v 1
-o ../../PK_S5D9_BL_Blink\Debug\PK_S5D9_BL_Blinky_signed.bin
```



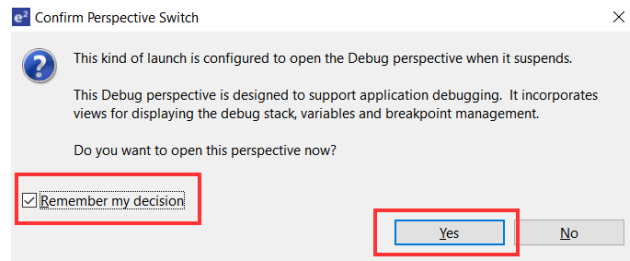
```
Select C:\Windows\system32\cmd.exe
C:\Work\SC32\SSP_1_7_x_YASB_Test\Bootloader\Image_Tools>python yasb.py sign -i ../../PK_S5D9_BL_Blinky\Debug\PK_S5D9_BL_Blinky.bin -k signingkey.key -v 1 -o ../../PK_S5D9_BL_Blinky\Debug\PK_S5D9_BL_Blinky_signed.bin
Original image length: 19428
New image size: 19684
Done
```

This should create the signed image binary “PK_S5D9_BL_Blinky_signed.bin” in the Debug folder of the PK_S5D9_BL_Blinky project.

15. Back in e2studio right click on the “Bootloader” project and select “Debug As -> Renesas GDB Hardware Debugging”.

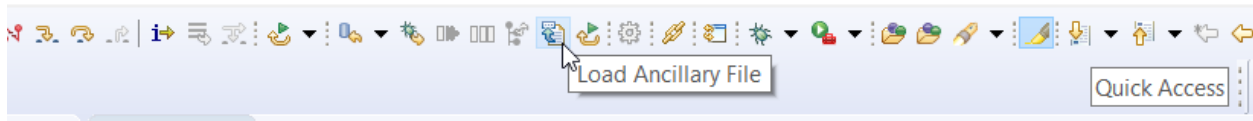


16. Confirm Perspective Switch if shown.



At this point the bootloader has been downloaded and is ready to run. However, there is no application image to boot. This can be fixed by downloading our signed image into the update area.

17. Click on the “Load Ancillary File” button.



18. Navigate to the signed image binary “PK_S5D9_BL_Blinky_signed.bin” in the Debug folder of the PK_S5D9_BL_Blinky project.
19. Tick “Load as raw binary image” and set the Address to 0x108000 which is the start of the update area in internal flash.



20. Click OK to download the signed image into the update area.
21. Now, run the bootloader by clicking Resume or F8 a couple of times.



The bootloader will run, find the update image in flash, verify it, copy it to the main application area, verify it, erase the update area and jump to the application flashing the user LEDs.

To test the use of QSPI for the update storage.

22. Disconnect the debugging session.



23. In the project “PK_S5D9_BL_Blinky” open the file src/blinky_thread_entry.c

24. Edit line 50 to flash only 2 of the 3 user LEDs so we can tell that a new program has been loaded and run by the bootloader.

```

37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
--
while (1)
{
    /* Determine the next state of the LEDs */
    if(IOPORT_LEVEL_LOW == level)
    {
        level = IOPORT_LEVEL_HIGH;
    }
    else
    {
        level = IOPORT_LEVEL_LOW;
    }

    /* Update all board LEDs */
    for(uint32_t i = 0; i < leds.led_count - 1; i++)
    {
        g_ioport.p_api->pinWrite(leds.p_leds[i], level);
    }

    /* Delay */
    tx_thread_sleep (delay);
}

```

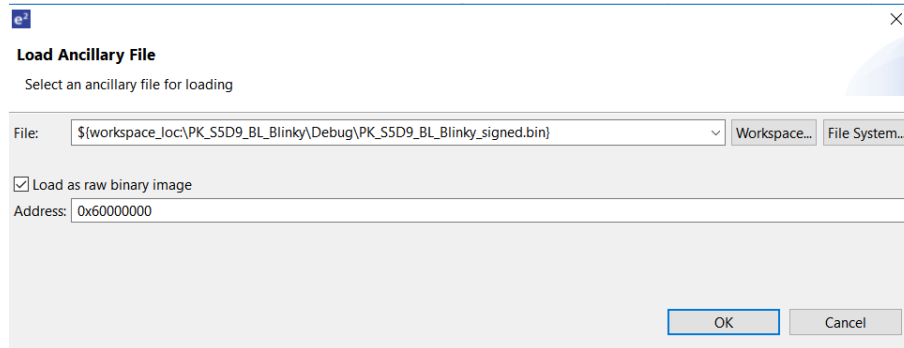
25. Build the project.

26. Repeat step 13 to sign the new image.

27. In the “Bootloader” project open the file /src/port.h and uncomment the macro “UPDATE_USES_QSPI_FLASH” on line 18.

28. Build the “Bootloader” project.

29. Right click on the “Bootloader” project and select “Debug As -> Renesas GDB Hardware Debugging”.
30. Click on the “Load Ancillary File” button.
31. Navigate to the signed image binary “PK_S5D9_BL_Blinky_signed.bin” in the Debug folder of the PK_S5D9_BL_Blinky project.
32. Tick “Load as raw binary image” and set the Address to 0x60000000 which is the start of the update area in QSPI flash.



33. Resume the “Bootloader” application. This time it will take several seconds for the application to be updated. This is because the erasing of the update area in QSPI flash takes longer than internal flash.

The bootloader should now be operational supporting updates from either internal flash or external QSPI flash.

Protecting the Bootloader

The bootloader should be made immutable so it cannot be altered to boot incorrectly signed images, or the public key changed. This can be done using the Flash Access Window (FAW) register settings. This allows the bootloader flash blocks to be configured so they cannot be erased or reprogrammed. See the device hardware user manual for more details.

Additionally, the Security MPU should be configured to secure the bootloader memory and prevent access from non-secure application memory. Further details can be obtained from the device user manual.