# AN14070

## How to Run HSE Demo Application on Cortex-M7 Core of S32G2

**Rev. 0 — 5 October 2023** **Application note**

**AN14070**

## 1 Introduction

To help the customers get started with HSE on a S32G2 platform, NXP delivers the *HSE_DEMOAPP*, which is available for download from NXP's official website. This app-note explains in detail the steps to start the *HSE_DEMOAPP_S32G2XX_0_1_0_9* on *S32G-PROCEVB-S* Board. Besides, the app-note covers parts of *HSE_DEMOAPP_S32G2XX_0_1_0_9_ReadMe.pdf* that are relevant for successfully executing the *HSE_DEMOAPP* on the S32G2 hardware platform. The pdf is located at *<path>\HSE_DEMOAPP_S32G2XX_0_1_0_9*\, where <path> is the local directory in which the demo is stored. So, for a more detailed description with different setups, it is recommended to refer to the pdf. Moreover, it is important to note that there are different versions of *HSE_DEMOAPP* for different S32 devices.

This app-note includes a few terminologies, such as sys_img, pink image and blue image. In addition, each of these terms are described in the *NXP HSE HIGH FW FAQ.pdf*. The pdf is located at the following location - *<path>\HSE_DEMOAPP_S32G2XX_0_1_0_9\demo_app\docs*\.

Overview of steps involved.

1. Install the *HSE_DEMOAPP_S32G2XX_0_1_0_9* on Windows PC.
2. Import the HSE demo project from *HSE_DEMOAPP_S32G2XX_0_1_0_9* and generate an *.elf*.
3. Using *S32DS IDE v3.5*, create a new *Application bootloader* binary for M7 core of *S32G2*.
4. Generate a secure-boot blob image for M7 core by inserting a *HSE* pink image to the *IVT*.
5. Write the generated blob image to the target device's QSPI flash using *S32 Flash Tool*.
6. Load the generated HSE demo *.elf* to the SRAM using *Trace32 Debugger Tool*. .
7. Check the version and status of HSE FW.
8. Perform encryption and decryption. .
9. Verify the encrypted and decrypted results using an online *AES Conversion Tool*.

The following table shows the Acronyms used through out the document.

**Table 1. Acronyms and definition**

| Acronyms | Definition |
|---|---|
| PC | Personal Computer |
| POR | Power On Reset |
| ROM | Read Only Memory |
| HSE | Hardware Security Engine |
| IVT | Interrupt Vector Table |
| IDE | Integrated Development Environment |
| FW | Firmware |
| AES | Advanced Encryption Standard |
| SRAM | System Random Access Memory |

## 2 Installation

The following steps shows how to install the HSE_DEMOAPP_S32G2XX_0_1_0_9 on Windows PC

1. The HSE FW 0.1.0.9 SR Release for S32G2 can be downloaded from NXP.com. In case, the download is not permitted, please reach out to the NXP sales representative. After logging in, depending on the access right granted to the account, the user can find the packages listed under standard software offering, as shown in the following figure.

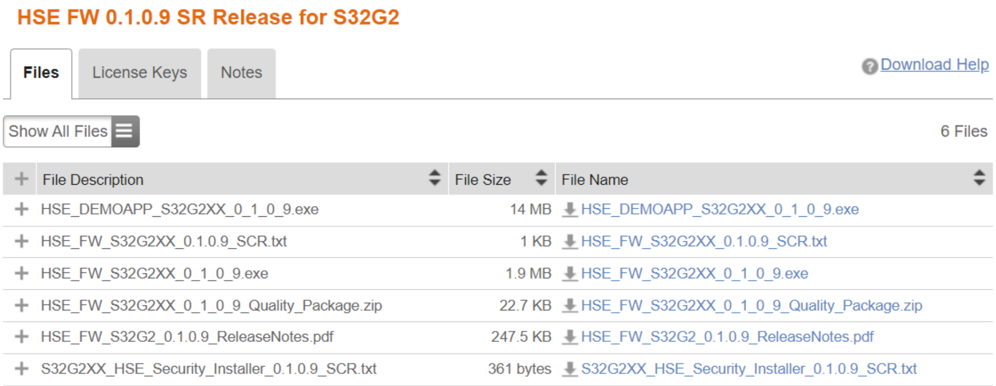


**Figure 1. HSE demo download**

2. From the HSE FW package, download the following files, the remaining ones are optional.
   - *HSE_DEMOAPP_S32G2XX_x.x.x.x.exe*
   - *HSE_FW_S32G2XX_x.x.x.x.exe*
3. After downloading them, consequently run *HSE_DEMOAPP_S32G2XX_x.x.x.x.exe* and *HSE_FW_S32G2XX_x.x.x.x.exe* and follow the instructions from the installation wizard.

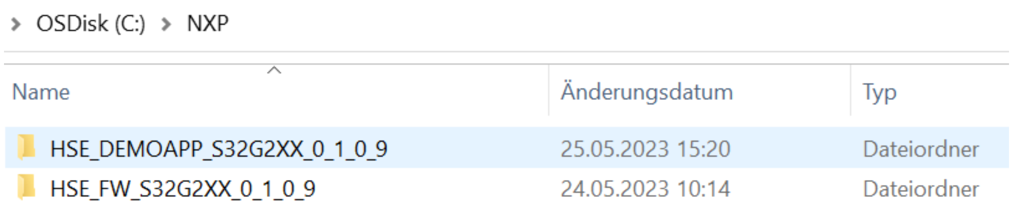


**Figure 2. HSE demo installation two**

4. After the installation is complete, navigate to the *<path>* to which the folder named *HSE_DEMOAPP_S32G2XX_x.x.x.x* is located – refer to above figure and open *HSE_DEMOAPP_S32G2XX_0_1_0_9_ReadMe.pdf*, as shown in the following figure.

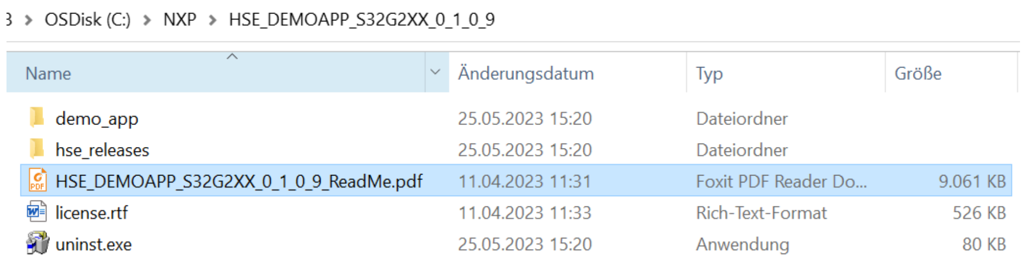


**Figure 3. Read me pdf location**

5. Although, the HSE demo app supports multiple NXP platforms, the focus of this app-note is on S32G2. However, with some fine-tuning, the same steps can be mirrored on other listed target devices.
6. Furthermore, the *HSE_DEMOAPP_S32G2XX_0_1_0_9_ReadMe.pdf* does a deep-dive into the *HSE_DEMOAPP_S32G2XX_0_1_0_9* package. The important highlights from the pdf are given below,
   - The section 5 and 6 detail the successful running of HSE FW using different resources.
   - The section 7 discusses ways to verify the status of the HSE FW, configure the relevant keys and services, perform different types of encryption and decryption on the target device. 322010932201093221.4.
7. Next, copy the subdirectories *hse* and *interface* from *<path>\HSE_FW_S32G2XX_0_1_0_9* to *<path>\HSE_DEMOAPP_S32G2XX_0_1_0_9\hse_releases\S32G2XX*, as shown in the following figure.

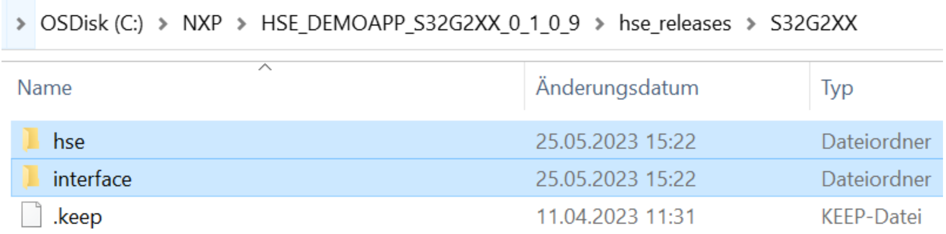


**Figure 4. Copying directories**

## 3 Load the HSE demo .elf to the SRAM

Import the HSE demo project from HSE_DEMOAPP_S32G2XX_0_1_0_9 and generate an .elf. The following steps shows how to import:

1. Once the installation of *HSE_DEMOAPP_S32G2XX_0_1_0_9* is complete, open the *S32DS v3.5* and navigate to *File -> Open projects from File Systems and Archives*. Next, from the dialogue box, click on *Directory* and navigate to *HSE_DEMO_S32G2XX*, as shown in the following figure.

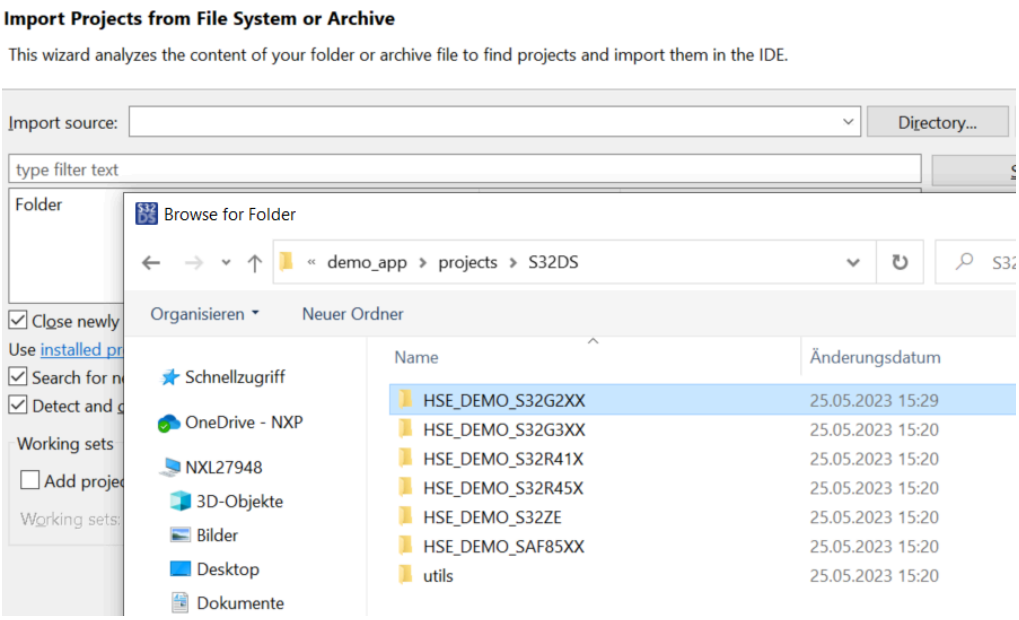


**Figure 5. HSE demo project**

2. The imported project will then appears in the *Project Explorer* window on the left-hand side.

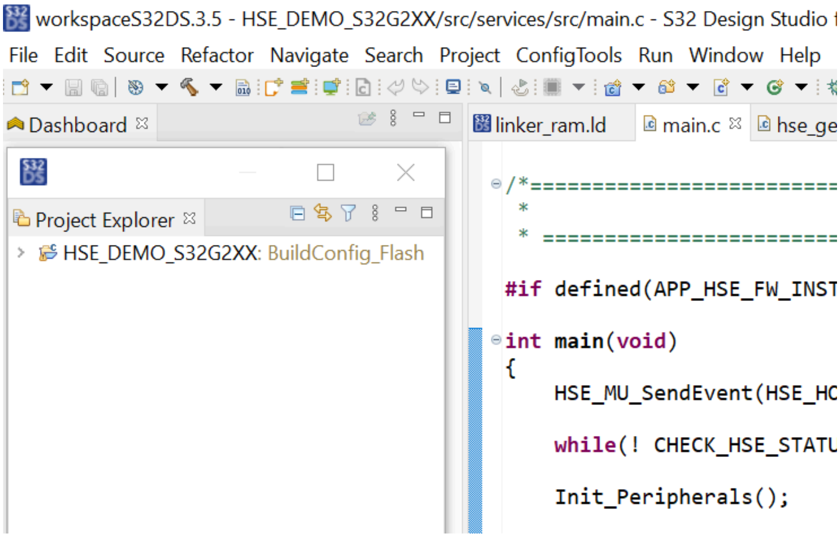


**Figure 6. HSE demo project two**

3. After the project is imported, go to the *Project Explorer* window and navigate to *HSE_DEMO_S32G2XX/src/services/src/main.c*. Though, the *main.c* file includes several functions that service HSE, for this specific use-case, the user may choose to simplify the code, as shown in the following figure.

```
int main(void)
{
    HSE_MU_SendEvent(HSE_HOST_PERIPH_CONFIG_DONE);

    while(! CHECK_HSE_STATUS(HSE_STATUS_INIT_OK | HSE_STATUS_RNG_INIT_OK) );

    Init_Peripherals();

    HSE_GetVersion_Example();

    HSE_Status();

    HSE_Config();

    HSE_Crypto();

    while(1);
}
```

**Figure 7. Code snippet from HSE demo project**

4. Additionally, as this app-note focusses only on *HSE_Aes_Example()*, the user may choose either to comment out or to remove the rest of the functions from *HSE_Crypto()*, as shown in the following figure.

```
void HSE_Crypto(void)
{
    ASSERT(CHECK_HSE_STATUS(HSE_STATUS_INSTALL_OK));

    HSE_Aes_Example();
}
```

**Figure 8. Code snippet from HSE demo project two**

5. It is important to note that if the above modifications are made to the original code, due to dependencies, the *srvResponse* will not return *HSE_SRV_RSP_OK* and therefore, the error handling will fail. Hence, it is necessary either to comment out or to remove the error handling, as shown in the following figures.

```
/*--------- AES ECB Encrypt Request ---------*/

memset(testOutput_encrypt, 0, BUFF_LEN);

/* Send the request */
srvResponse = HSE_AesEncrypt(HSE_CIPHER_BLOCK_MODE_ECB, aesEcbKeyHandle,
    NULL, aesEcbPlaintext, aesEcbPlaintextLength, testOutput_encrypt);

/* Check response and output */
/*if( (HSE_SRV_RSP_OK != srvResponse) ||
    (0 != memcmp(testOutput, aesEcbCiphertext, aesEcbCiphertextLength)) )
{
    goto exit;
}*/

/*--------- AES ECB Decrypt Request ---------*/

memset(testOutput_decrypt, 0, BUFF_LEN);

/* Send the request */
srvResponse = HSE_AesDecrypt(HSE_CIPHER_BLOCK_MODE_ECB, aesEcbKeyHandle,
    NULL, aesEcbCiphertext, aesEcbCiphertextLength, testOutput_decrypt);

/* Check response and output */
/*if( (HSE_SRV_RSP_OK != srvResponse) ||
    (0 != memcmp(testOutput, aesEcbPlaintext, aesEcbPlaintextLength)))
{
    goto exit;
}*/
```

**Figure 9. Code snippet from HSE demo project three**

The next step is to build the project. To do so, right-click on the project in the *Project Explorer* window and select *Build Project*. The *Build* action will *Compile* the project, *Link* the libraries and finally generate an *.elf*. The generated *HSE_DEMO_S32G2XX.elf*, as shown in the following figure and to be henceforth called HSE demo*.elf*, is located in the folder *<path>\NXP\HSE_DEMOAPP_S32G2XX_0_1_0_9\demo_app \projects\S32DS\HSE_DEMO_S32G2XX\BuildConfig_Flash*.

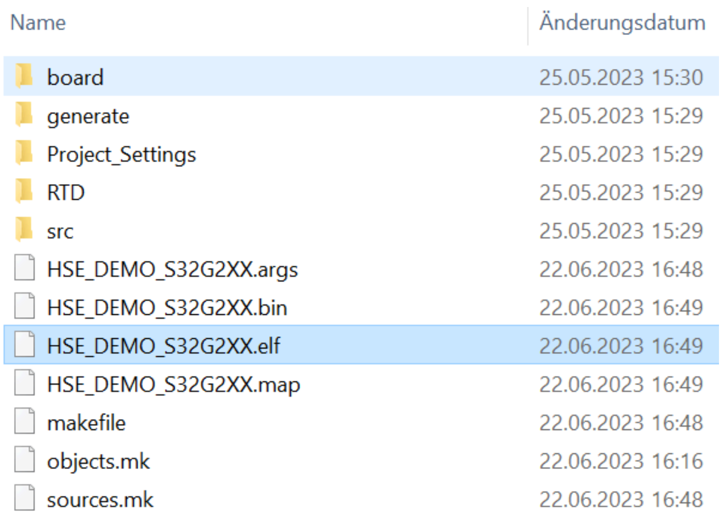


**Figure 10. Generated .elf**

# 4 New application bootloader creation

Use of *S32DS IDE v3.5*, to create a new *Application bootloader* binary for Cortex-M7 core of *S32G2*. The following steps shows how to create:

1. To create a new application bootloader *.bin*, first go to *File -> New -> S32DS Application Project*. Then, provided the relevant *S32GX* packages were installed, a list of processor will be displayed in the dialogue box. In case they were not, refer to the *S32DS IDE v3.5 Installation Guide*. Further, assuming that *S32G274A* is the target device, select the *S32G274A_Rev2 Cortex-M7* processor from the list, as shown in the following figure.

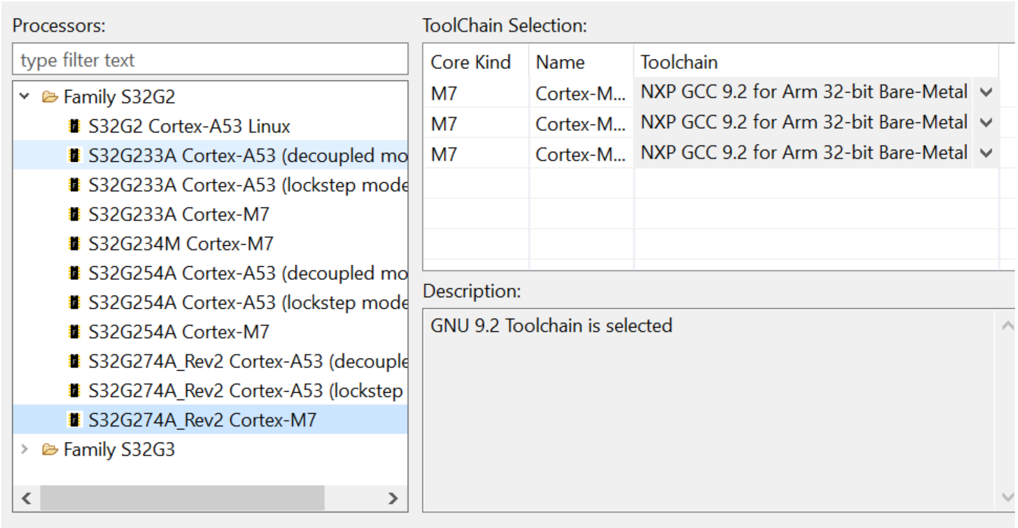


**Figure 11. Creating an application project**

2. Although *S32G274A* has three M7 cores in lockstep, only one lockstep core is sufficient to perform the secure boot operation. Hence, for the purpose of this app-note, only *Cortex-M7_0* core is advised to be enabled – refer to the following figure.

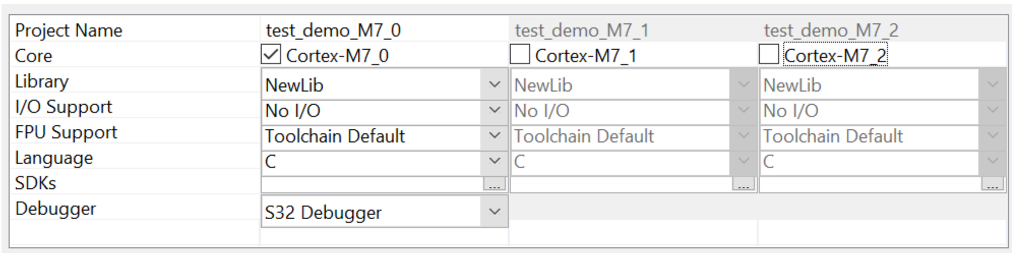


**Figure 12. Core selection**

3. Once these changes are incorporated, the user can customise the application bootloader code and subsequently, generate a *.bin* file. But, note that generating a *.bin* file is tricky. Unlike the *.elf* file, which is automatically generated post successful build, the generation of *.bin* file needs to be manually enabled. To enable it, first, right-click on the *<project>* -> *Properties*. Then, from the dialogue box click on *C/C++ Build -> Settings -> Tool Settings -> Create flash image*. Lastly, click on *Apply and Close* - refer to the following figure.

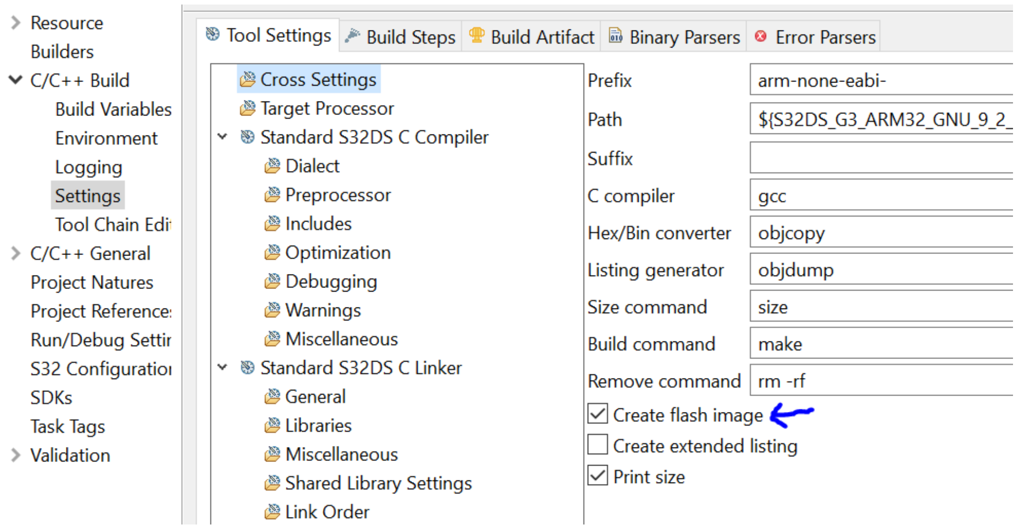


**Figure 13. Enabling flash image creation**

4. The flash binary is, however, not yet created, because the final step is still pending. To create the flash binary, reopen the dialogue box, as shown in the following figure. Follow the same sequence until *Tool Settings*. And in the *Tool Settings* tab, expand the option *Standard S32DS Create Flash Image* and click on *General*. From the drop-down menu of *Output file format*, select *Raw Binary* and click *Apply and Close*.

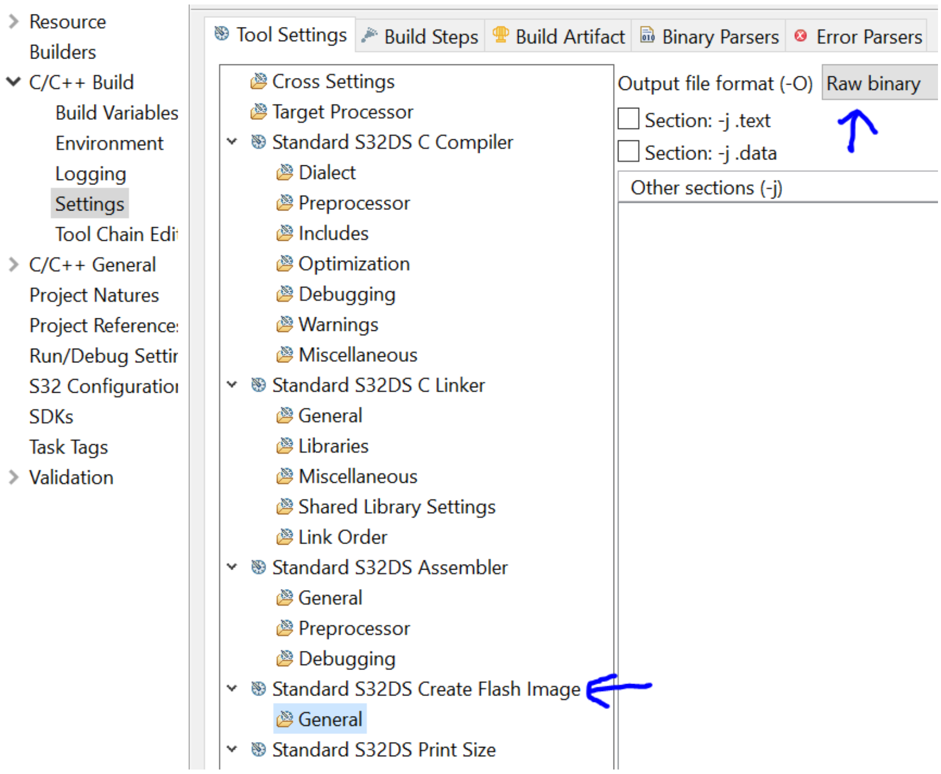


**Figure 14. Flash binary creation**

5. Then, rebuild the project as described in Section 3. The generated *<project_name>.bin* file will be located in the folder *<project_path>\Debug_RAM*, as shown in the following figure.

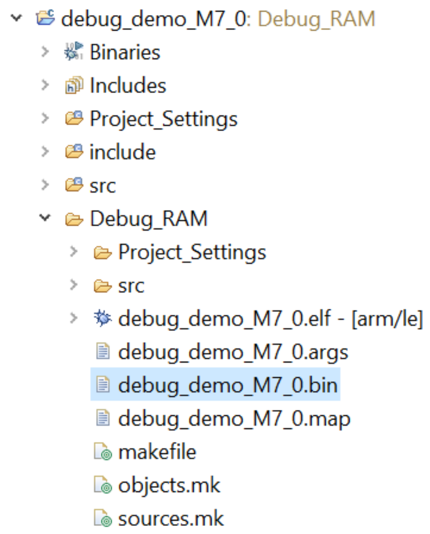


**Figure 15. Application code binary image**

## 5 Secure-boot blob image generation

Generate a secure-boot blob image for Cortex-M7 core by inserting a *HSE* pink image to the *IVT*. The following steps shows how to insert the image:

1. A blob image is a form of *.bin* file that, using a flash tool, is written to the storage device, such as a *Flash*, an *eMMC* or a *SD card*; in this case, the *S32 Flash Tool* is used to write the blob image to the *QSPI Flash*. Moreover, a blob image for *S32G2* contains following components in the same sequence: a self-test *DCD*, self-test *DCD* (backup), *DCD*, *DCD* (backup), *HSE*, *HSE* (backup), *application bootloader* and *application bootloader* (backup). However, depending on the use case, one or more of the listed component could be left-out. Further, there is another critical part of the blob image called *IVT* that holds the pointers to the binaries of each of the enabled components. In addition to this, the secure-boot blob image includes a *HSE* pink image.
2. For creating a secure boot blob image for M7 core, the users can use the project created in step 3; note that it is not related to the HSE demo package, which was imported in step 2. So, once the required binaries are available, run *S32DS IDE v3.5*, then open the project that was either created in step 3. Next, switch the view from *C/C++* to *IVT*, refer to the following figure.

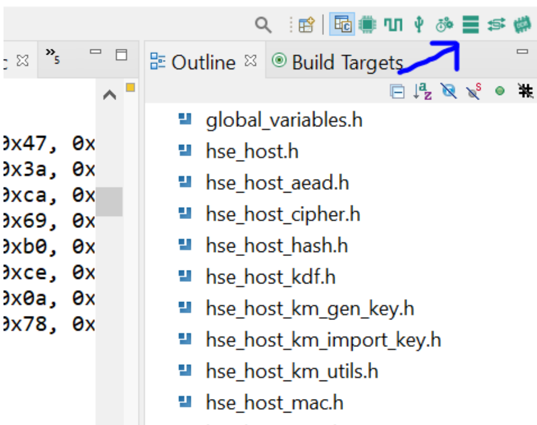


**Figure 16. Switching to IVT view**

3. The *IVTView*, shown in the following figure, displays the blocks *DCD*, *HSE*, *Application bootloader*, *Boot configuration* and *Automatic Align* that are relevant for the generation of the blob image.

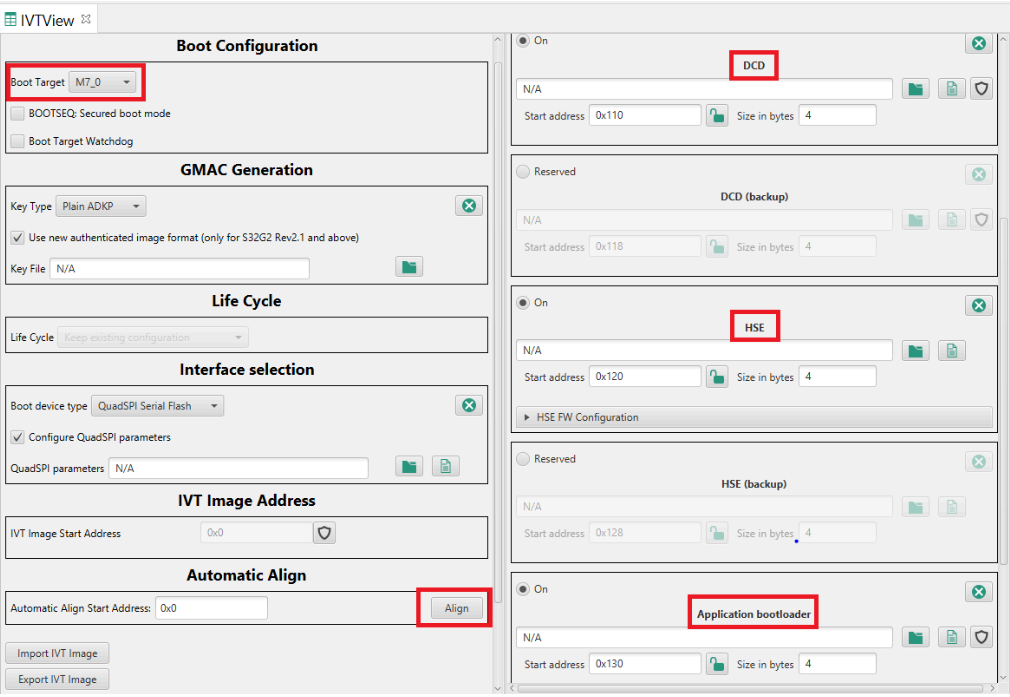


**Figure 17. IVT configuration**

4. The DCD *.bin* contains the data that will be used to configure *S32G2* after it comes out of reset. Moreover, the DCD is generally created using DCD Tool, but, in this case, the DCD *.bin* files are included in the demo project folder and can be directly loaded from *<path>NXP\HSE_DEMOAPP_S32G2XX_0_1_0_9\demo_app\images\S32G2XX*. So, from the four available binaries seen in the following figure, for general use-case load *dcd_init_sram.bin*.

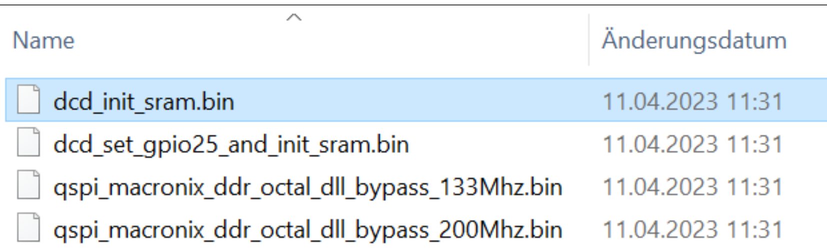


**Figure 18. DCD binary image**

5. Likewise, the HSE pink image is also delivered with the HSE demo project and is located in the folder *<path>\NXP\HSE_DEMOAPP_S32G2XX_0_1_0_9\hse_releases\S32G2XX\hse*\bin. However, before uploading the image, remember to modify its name from *<image_name>.bin.pink* to *<image_name>.bin*, as shown in the following figure.

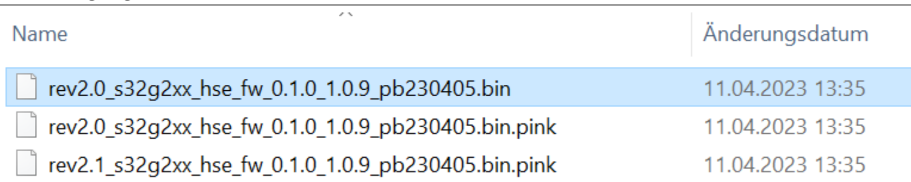


**Figure 19. HSE FW pink image**

6. Moreover, as the *sys-img* is not yet available, disable the *sys-img* pointers in the *HSE FW Configuration* window, as visible in the following figure.

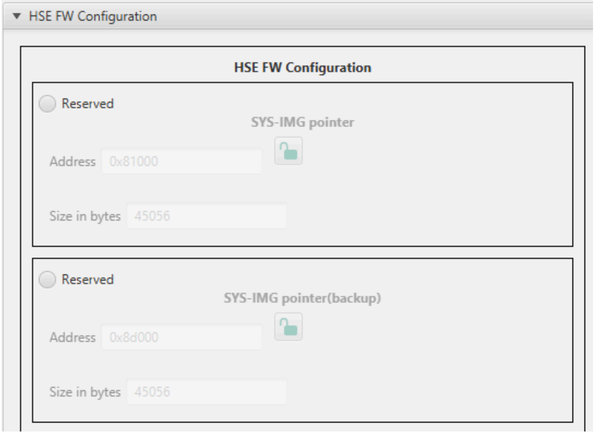


**Figure 20. HSE FW configuration in IVT**

7. Next, the *Application bootloader* image that was created in the step 3, is the third *.bin* that has to be linked to the IVT. Additionally, in the *Application Boot Image* block from the following figure, two addresses namely, *RAM start pointer* and *RAM entry pointer* must be populated in their respective fields.

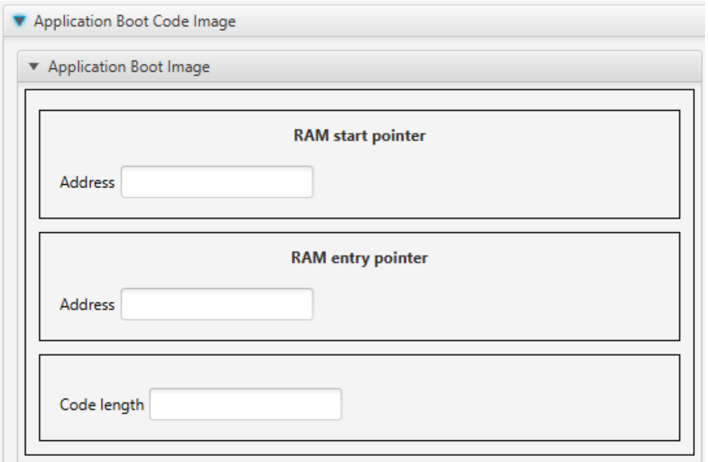


**Figure 21. Application boot code image configuration**

8. Both the parameters can be found in the project's linker file with *.ld* extension. For S32G2XX target device, usually, the file name is *S32G_M7_RAM.ld,* as shown in the following figure.

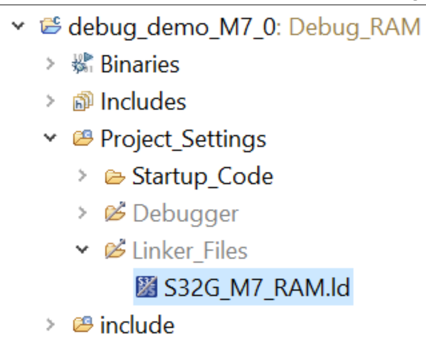


**Figure 22. Application code linker file**

9. To find the pointer addresses, scroll down *S32G_M7_RAM.ld* to the *Memory* section. Since, in this case, the $0^{th}$ M core is in use, the *RAM start pointer* is *0x34001000*, as shown in the following figure. Further, unless

an explicit mismatch is found in the *.bss* section of the linker file, the *RAM entry pointer* is the same as *RAM start pointer* i.e. *0x34001000*.

```
/* Linker script to configure memory regions. */
MEMORY
{
    CM7_0_RAM (rw) : ORIGIN = 0x34001000, LENGTH = 0x0FF000
    CM7_1_RAM (rw) : ORIGIN = 0x34100000, LENGTH = 0x100000
    CM7_2_RAM (rw) : ORIGIN = 0x34200000, LENGTH = 0x100000
}
```

**Figure 23. Linker code snippet**

10. Next, in the left-most panel of the *IVTView*, verify whether the *Boot Target* matches the core - here *M7_0*. In addition, before exporting the final secured boot blob image, make sure that none of the IVT fields are marked in red, as shown in the following figure. If they are, it indicates that an address overlap has occurred.

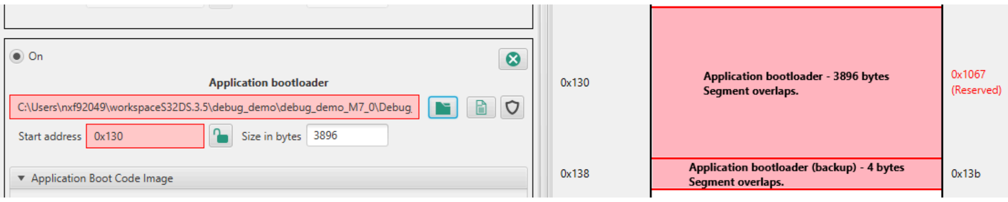


**Figure 24. IVT automatic size alignment**

11. The solution is to either recalculate the memory usage and manually add buffers between the conflicting memory segments or use the *Automatic Align* feature, as visible in the following figure, and simply click on *Align* button.

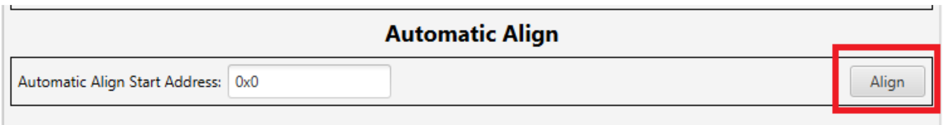


**Figure 25. IVT automatic size alignment two**

12. The last step is to export the blob image; please refer the following figure.

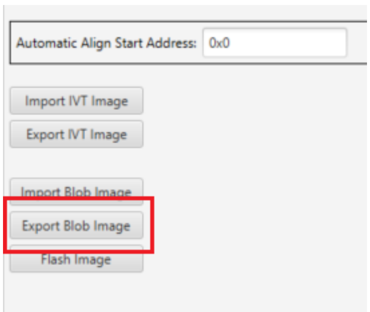


**Figure 26. Export final blob image**

## 6 Writing the blob image to the QSPI flash

Use the *S32 Flash Tool* to write in the generated blob image to the target device's QSPI flash. The following steps shows how to write in the blob:

1. The blob image generated in step 4 needs to be delivered to the target device's *QSPI Flash* memory. Although, there are a few means to perform this task, this app-note discusses doing it serially using *S32 Flash Tool* and *Trace32 Debugger*. Moreover, the target device is *S32G-PROCEVB-S* Board and, with

corresponding modifications, can be replicated on other devices. Further, the *S32 Flash Tool* can be downloaded from NXP.com, provided the necessary access rights already exist. If not, reach out to the designated sales representative from NXP; note that the *Trace32 Debugger* is, however, a proprietary of *Lauterbach* and must therefore be purchased accordingly.

2. Next, once the *S32G-PROCEVB-S* has entered serial boot mode and all the necessary prerequisites are fulfilled - refer sections 2 to 4 of AN12422 for detailed description, load the generated blob image, from step 4, serially to the *S32G-PROCEVB-S* Board. To do so, first find the COM port to which the target device is serially connected. To locate the COM port on *Windows OS*, open the device manager, go to *Connections (COM & LPT)* and look for *USB Serial Port*, as shown in the following figure.


**Figure 27. Configuring serial communication port**

3. Next, open the *S32 Flash* Tool, enter the *COM* port, perform *Test connection check*, set the device *Target* and choose the flash hardware *Algorithm*, as described in the following figure. Furthermore, the *Upload target and algorithm to hardware.* option uploads the algorithm linked to the target and QSPI flash device.

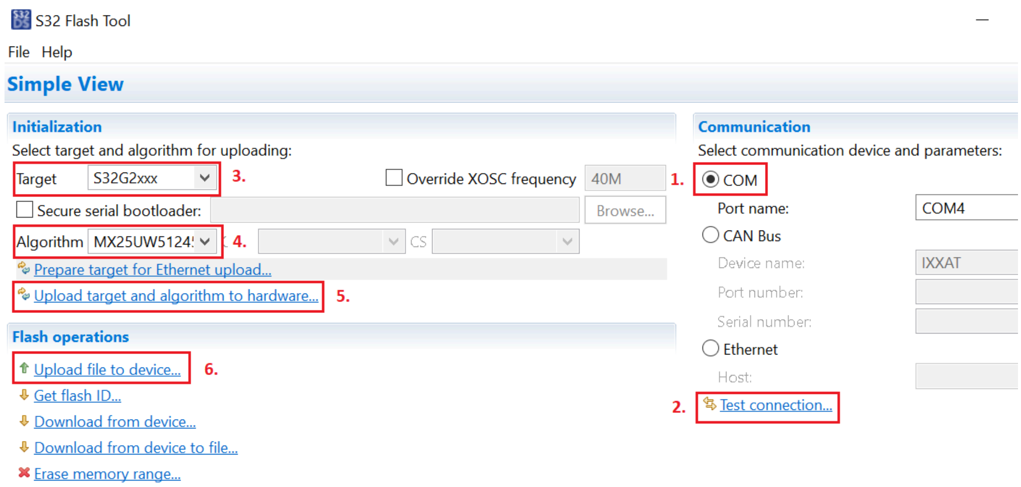


**Figure 28. Configuring S32 flash tool**

4. Once the algorithm upload is complete, the log window displays a message confirming the successful deployment of the algorithm to the hardware – refer to the following figure.

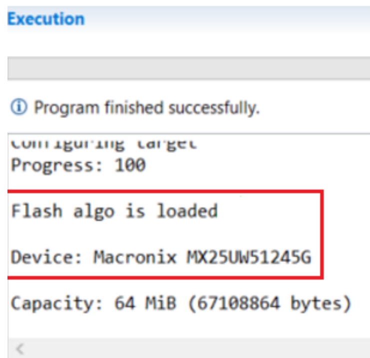


**Figure 29. Binary image successfully loaded**

5. Lastly, click on *Upload file to device* to upload the blob image. The log window will then display the progress of data-packets being transferred. Depending on the size of the HSE secure-boot blob image, the writing might take between a few seconds to a few minutes. After the successful loading of the blob image is complete, switch the mode from serial boot to QSPI boot – refer sections 4 to 6 of AN12422 for detailed description.

## 7 How to load the HSE demo .elf to the SRAM

Load the generated HSE demo *.elf* to the SRAM using *Trace32 Debugger Tool*. The following steps shows how to load in the blob:

1. In secure boot, immediately after POR, HSE M7 core executes the BootROM. The BootROM then reads IVT, loads its content into SRAM and after execution, passes the control over to HSE FW. So, once the target device is fully up and running, meaning the application code that was stored in the QSPI flash is loaded into the SRAM and, in this case, the M7 core has taken charge from HSE M7 core, the task then is to check the status of HSE FW and test a few of the HSE services. However, to begin with the HSE status check, the HSE demo *.elf*, that was generated in step 2, needs to be loaded to the SRAM; note that both the symbols and the code must be loaded. Also remember that this app-note uses *Lauterbach*'*s Trace32 PowerView for ARM* as the debugging tool.
2. Next, connect the debugger probe between PC and the target device, run the executable and configure the *Debugger Tool*. Moreover, the initial configuration can be automated with the help of a *.cmm* file. So, to create a *.cmm* file, go to *File -> New Script*. Then, copy the below debugger-specific configuration code to it and close the script.

```
sys.CPU S32G274A-M7
core.ASSIGN 1.
sys.CONFIG.DEBUGPORTTYPE JTAG
SYStem.Option dualport on
SYStem.Option TRST OFF
SYStem.JtagClock 10MHz
sys.MemAccess DAP
ETM.OFF
```

3. Then, open the *Trace32 Debugger* tool and go to *CPU* -> *System Settings*. In the dialogue box, as shown in the following figure, verify whether the system parameters from the .cmm file, namely *CPU*, *dualport, DEBUGPORTTYPE*, *TRST*, *JtagClock* and *MemAccess* match.

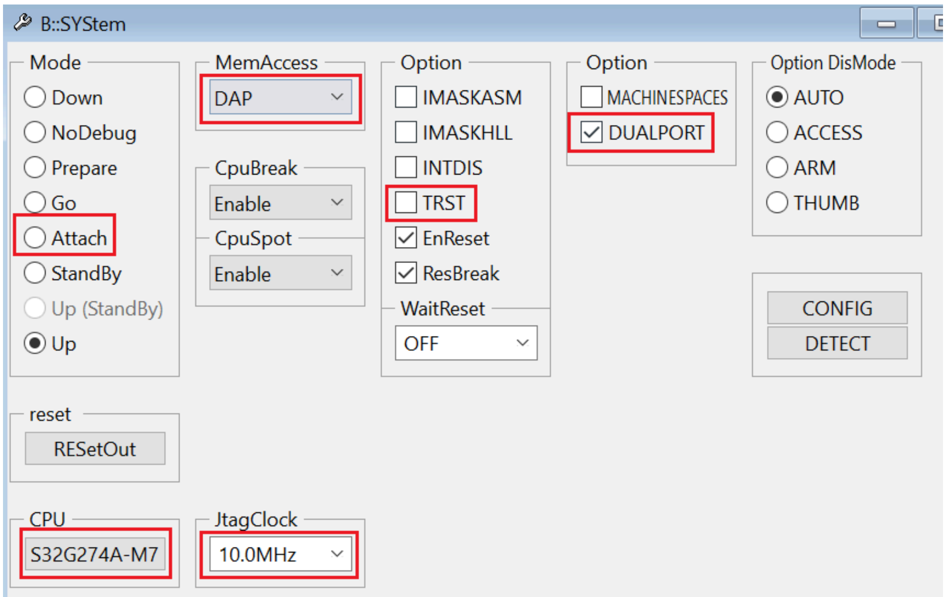


**Figure 30. Configuring Trace32 debugger**

4. Next, to attach to the running firmware, click on *Attach*; a green bar at the bottom-right corner of the GUI indicates that the attach was successful. After this, the HSE demo *.elf* file must be loaded. However, before loading the *.elf* file, pause the run by clicking on break, as shown in the following figure, or else the debugger will throw a *target running* error.

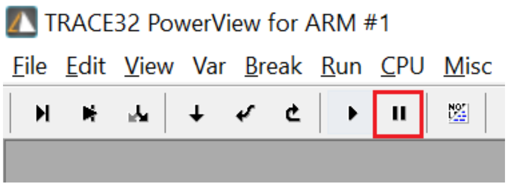


**Figure 31. Trace32 debugger setting**

5. Then, in the *command* bar, enter the command *data.list*. It displays the paused instance of the assembly code that was running on the M7 core - refer to the following figure.

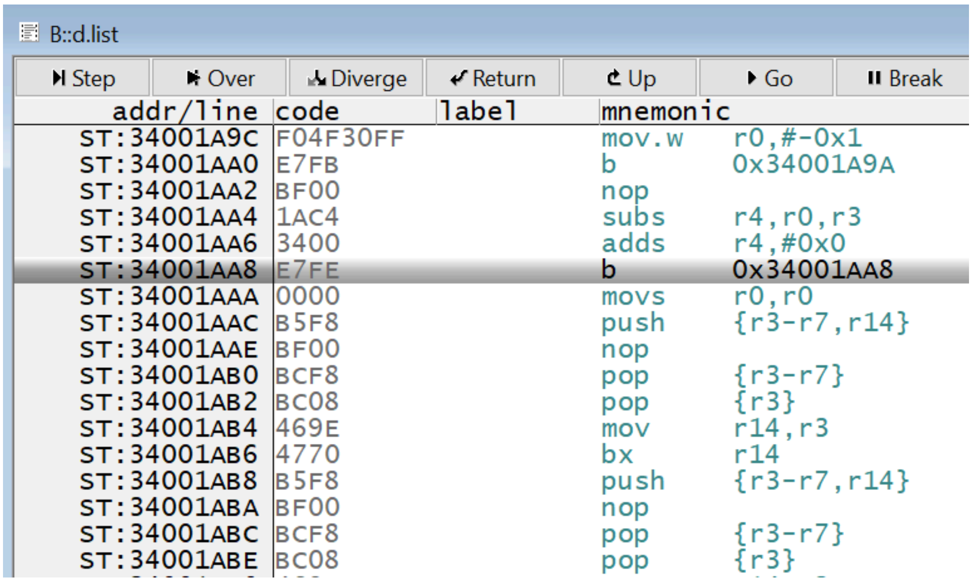


**Figure 32. Stepwise debugging using Trace32 debugger**

6. Next, to load the HSE demo .elf, enter the command *data.load <path to the HSE demo .elf>*

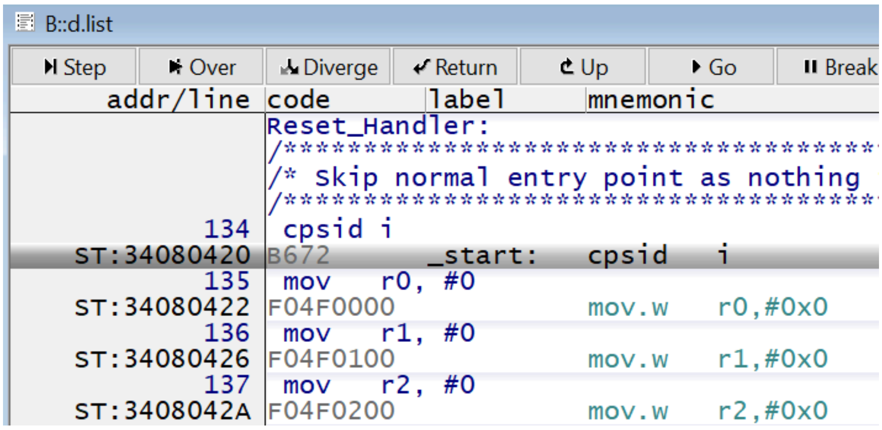


**Figure 33. Stepwise debugging using Trace32 debugger two**

7. Usually, the *.elf*, used for debugging and the blob image loaded to the flash originate from the same application code. However, here, they both originate from two different sources; *.elf* from HSE demo project and the *.bin* from an existing or imported M7 project. Therefore, this method is specifically devised for testing HSE with similar conditions; note that for a lean debugging experience in the *List* window, click the *Mode* button. The window then displays only the high level code, as shown in the following figure.

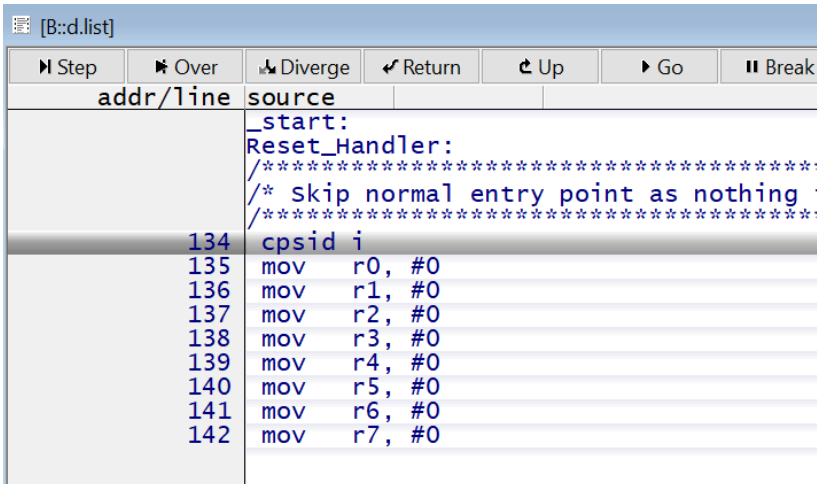


**Figure 34. Stepwise debugging using Trace32 debugger three**

8. As the function calls necessary for executing the HSE services are in the *main()*, it is optimal to directly jump to that function and skip the initialization block. To do so, enter *go main* in the command bar. Moreover, from this point onwards, stepwise debug the HSE demo code. To do so, use *Step*, *Over* and *Up* buttons to step-into, step-over and step-up, respectively, as shown in the following figure.

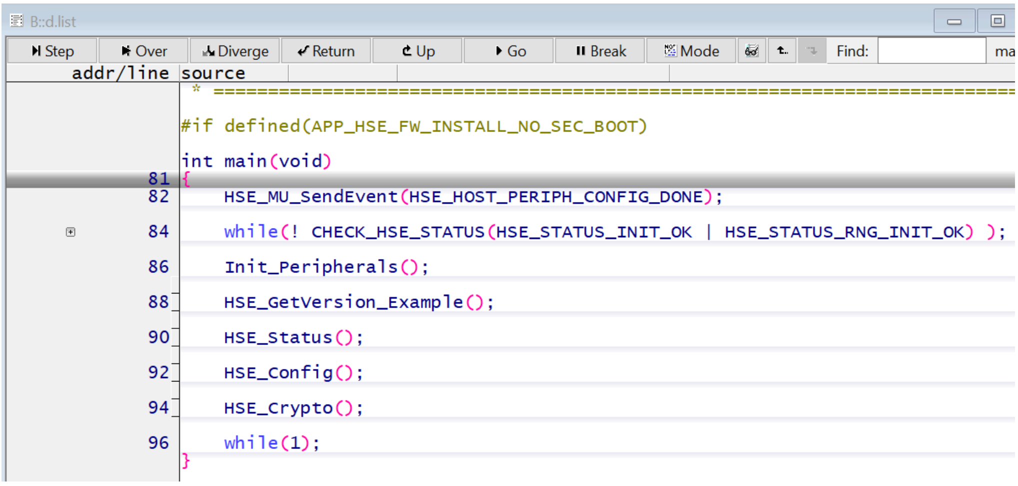


**Figure 35. Stepwise debugging using Trace32 debugger four**

## 8 Status and version check of HSE FW

The following steps shows how to check the version and status of HSE FW:

1. The *HSE_GetVersion_Example()* on line 88 of HSE services contains the code that display's the HSE's FW version. However, to view the FW version, first open the serial terminal such as *PuTTY* on the PC, then configure the *COM* port and the *baudrate* accordingly, as shown in the following figure.

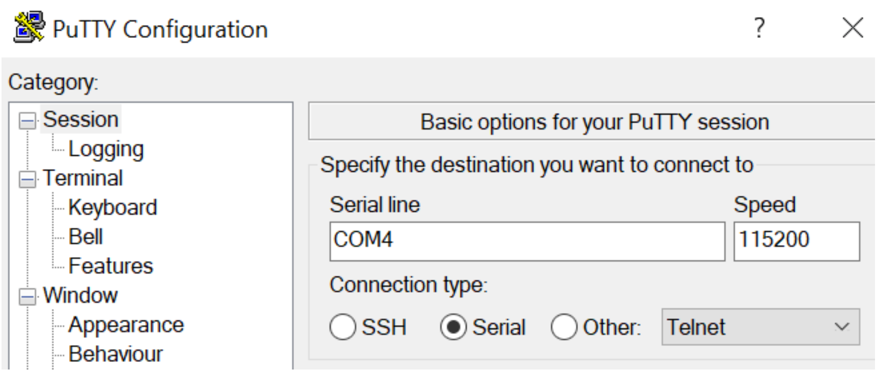


**Figure 36. Configuring serial console**

2. Moreover, before executing the code, either set a breakpoint on line 90 and click on *Go* or right-click on that line and select *Go Till*. Additionally, after execution, the parameters *HSE FW Version* and *HSE FW Image* is printed on the serial terminal. It must, therefore, be verified that the printed parameters on *PuTTY* matches the corresponding version number and image type – please refer the following figure.

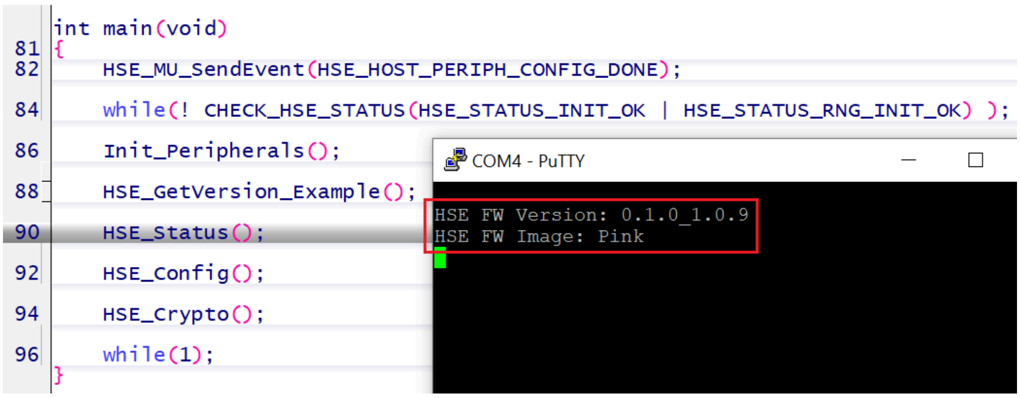


**Figure 37. Code snippet displaying HSE_status()**

3. Furthermore, the version can be verified against the last four digits of the installation folder name – as shown in the following figure.

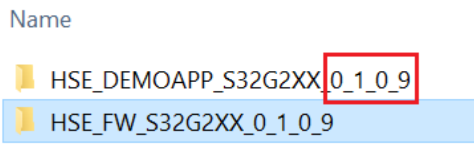


**Figure 38. HSE version verification**

4. The next check is the HSE FW's execution status and can be done by executing *HSE_Status()*. Since, the *Program Counter* must be on line 90, either set a breakpoint on line 92 and click on *Go* or right-click on that line and select *Go Till*. Then, verify the printed parameters against the corresponding status, as shown in the following figure.

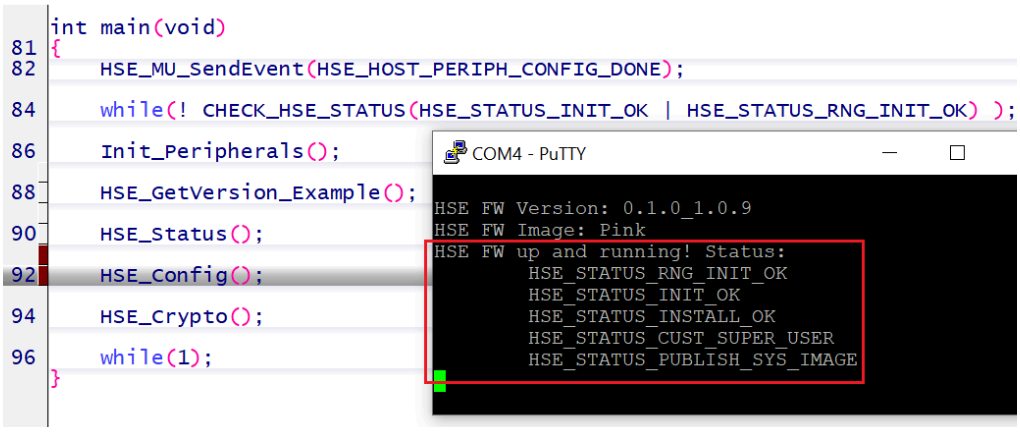


**Figure 39. Code snippet displaying HSE_config()**

5. Although, *HSE_Config()* does not play an active role in realising the objectives of this app-note, the function has some dependencies on *HSE_Crpto()* and therefore must be executed as well. Furthermore, the missing parameters *HSE_STATUS_PRIMARY_SYS_IMAGE* and *HSE_STATUS_BACKUP_SYS_IMAGE* in the above figure, indicate that as intended, the sys_img was not generated. Additionally, after this step, it is advised to leave the *PuTTY* window open, as it is used to verify the successful execution of step 8.

# 9 Encryption and decryption

The following steps shows how to perform encryption and decryption:

1. After stepwise debugging until line 92, the next step is to verify the HSE cryptographic services on line 94 of HSE_Status. Moreover, as described in Section 4 and seen in the following figure, *HSE_Crpto()* contains only one function i.e. *HSE_AES_Example()*. The function tests both encryption and decryption of the same plain-text; where, *aesEcbPlaintext* is the plain-text, *aesEcbKey* is the key and *testOutput_encrypt* and *testOutput_decrypt* is where the cipher-text and deciphered-text will be stored after encryption and decryption, respectively.

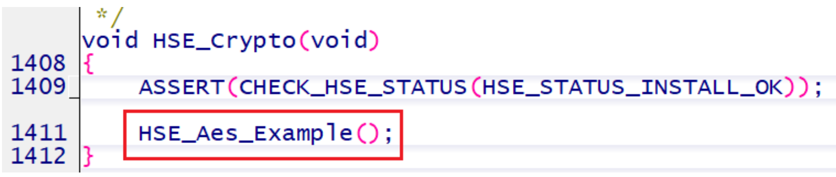


**Figure 40. HSE AES example**

2. The *aesEcbKey* and *aesEcbPlaintext* can be modified in the *hse_crpyto.c* source file. So, for simplicity, the key and the plain text were filled with *0x33* refer to the following figure. The user, however, may choose to insert a different combination.

```c
/* AES ECB Data */
static const uint8_t aesEcbKey[] =
{
    0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x33,
    0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x33
};

static const uint8_t aesEcbPlaintext[] =
{
    0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x33,
    0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x33

};
```

**Figure 41. HSE AES example two**

3. In the encryption stage, *aesEcbPlaintext* is encrypted using *aesEcbKey* and the result is stored in *testOutput_encrypt*, as shown in the following figure.

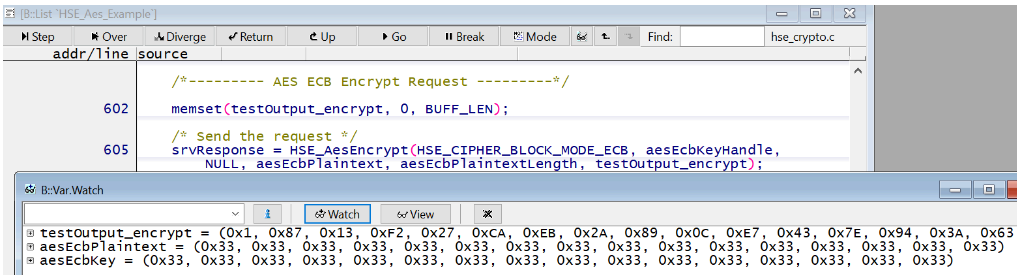


**Figure 42. HSE AES example three**

4. Moreover, the *Watch* function of the *Trace32 Debugger* facilitates effortless monitoring of variables from the loaded *.elf*. So, to add a variable to the *Watch Window*, right-click on the variable and from the options select *Add to Watch Window* refer to the following figure.

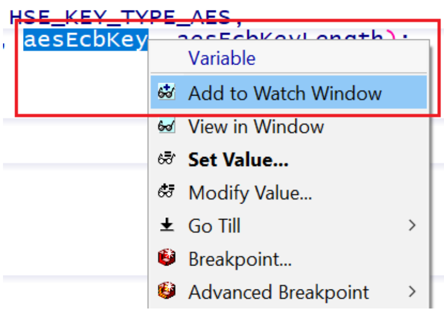


**Figure 43. HSE AES example four**

5. The number 51, shown in the following figure, is the decimal value for 0x33.

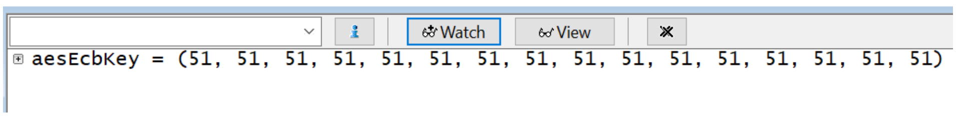


**Figure 44. HSE AES example five**

6. Note that in order to change the variable's format in the *Watch Window*, right-click on the variable and select *Format* from the list of options, as shown in the following figure.

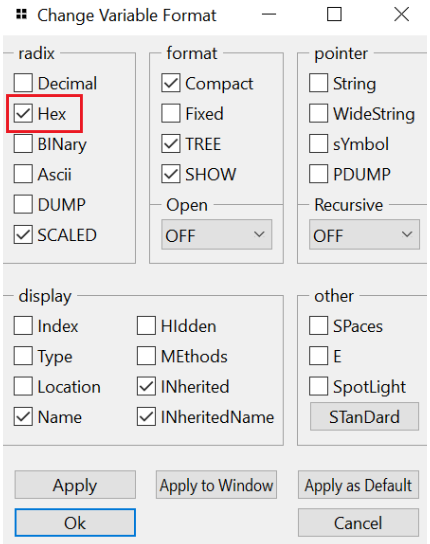


**Figure 45. Configuring output window setting**

7. In the decryption stage, the *testOutput_encrypt* is first copied to *aesEcbCiphertext*. Next, the cipher-text, using the same *aesEcbKey*, is decrypted and stored in *testOutput_decrypt*, as shown in the following figure.

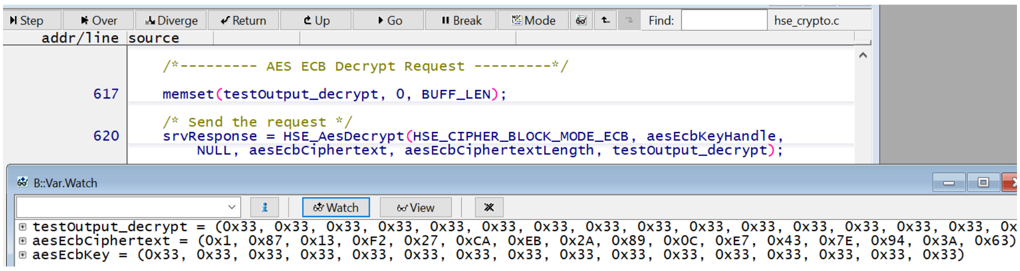


**Figure 46. HSE AES example six**

8. The parameter *HSE_Aes_Example* in the following figure indicates the successful execution of the *HSE_AES_Example()*.

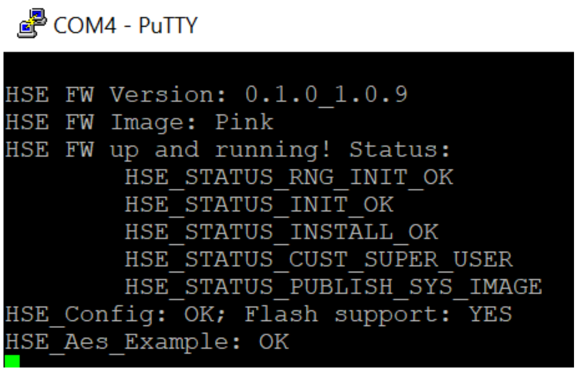


**Figure 47. HSE Demo execution successful**

## 10 Verifying encryption and decryption results

The following steps shows how to verify the encrypted and decrypted results using an online *AES Conversion Tool*:

1. The result from the encryption and decryption still needs to be verified. Moreover, this step will reveal HSE's capability in performing accurate AES based cryptographic conversions.
2. Online AES Encryption and Decryption Tool (javainuse.com) is one such online tool that is capable of executing cryptographic checks and can, therefore, be used for comparing the results. However, NXP does not endorse using any such tool.
3. Furthermore, for test purpose, *aesEcbKey* and *aesEcbPlaintext* were entered as key and plain-text, respectively refer to testOutput_encrypt. Moreover, the encrypted value of the cipher-text matched the *testOutput_encrypt* compare testOutput_encrypt with the below figure. Hence, it can be concluded that encryption carried out by HSE is accurate.

```
018713f227caeb2a890ce7437e943a63
```

**Figure 48. Final result verification**

4. However, the verification of the decryption process was skipped, as the *aesEcbCiphertext* i.e. the cipher-text, for simplicity, was copied from the *testOutput_encrypt* and therefore, the decryption must return the original plain-text i.e. 0x33, 0x33,.., which it did.
5. Moreover, it is evident from figure 9.1 that the *testOutput_decrypt* matches *aesEcbPlaintext*. Therefore, proving that the decryption was also successful.

# 11 Legal information

## 11.1 Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

## 11.2 Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Suitability for use in automotive applications** — This NXP product has been qualified for use in automotive applications. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

**NXP B.V.** - NXP B.V. is not an operating company and it does not distribute or sell products.

## 11.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.

**AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μVision, Versatile** — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

## Contents





**Date of release: 5 October 2023**
**Document identifier: AN14070**
**Document number: AN14070**