Getting started with STM32CubeN6 for STM32N6 series

Introduction

STM32Cube is an STMicroelectronics original initiative to improve designer productivity significantly by reducing development effort, time, and cost. STM32Cube covers the whole STM32 portfolio.

STM32Cube includes:

- A set of user-friendly software development tools to cover project development from conception to realization, among which are:
  - STM32CubeMX, a graphical software configuration tool that allows the automatic generation of C initialization code using graphical wizards
  - STM32CubeIDE, an all-in-one development tool with peripheral configuration, code generation, code compilation, and debug features
  - STM32CubeCLT, an all-in-one command-line development toolset with code compilation, board programming, and debug features
  - STM32CubeProgrammer (STM32CubeProg), a programming tool available in graphical and command-line versions
  - STM32CubeMonitor (STM32CubeMonitor, STM32CubeMonPwr, STM32CubeMonRF, STM32CubeMonUCPD), powerful monitoring tools to fine-tune the behavior and performance of STM32 applications in real time
- STM32Cube MCU and MPU Packages, comprehensive embedded-software platforms specific to each microcontroller and microprocessor series (such as STM32CubeN6 for the STM32N6 series), which include:
  - STM32Cube hardware abstraction layer (HAL), ensuring maximized portability across the STM32 portfolio
  - STM32Cube low-layer APIs, ensuring the best performance and footprints with a high degree of user control over hardware
  - A consistent set of middleware components such as ThreadX, FileX, LevelX, NetX Duo, USBX, USB PD, video encoder API, and OpenBL
  - All embedded software utilities with full sets of peripheral and applicative examples
- STM32Cube Expansion Packages, which contain embedded software components that complement the functionalities of the STM32Cube MCU and MPU Packages with:
  - Middleware extensions and applicative layers
  - Examples running on some specific STMicroelectronics development boards

This user manual describes how to get started with the STM32CubeN6 MCU Package.

Section 2: STM32CubeN6 main features describes the main features of the STM32CubeN6 MCU Package.
Section 3: STM32CubeN6 architecture overview provides an overview of the STM32CubeN6 architecture and the MCU Package structure.

1 General information

The STM32CubeN6 MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M55 processor with Arm® TrustZone® and FPU.

Note: Arm and TrustZone are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 STM32CubeN6 main features

The STM32CubeN6 MCU Package runs on STM32 32-bit microcontrollers based on the Arm® Cortex®-M55 processor with TrustZone® and FPU.

The STM32CubeN6 gathers, in a single package, all the generic embedded software components required to develop an application for the STM32N6 series microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32N6 series microcontrollers but also to other STM32 series.

The STM32CubeN6 MCU Package also contains a comprehensive middleware component constructed around Microsoft® Azure® RTOS middleware and other in-house and open source stacks, with the corresponding examples.

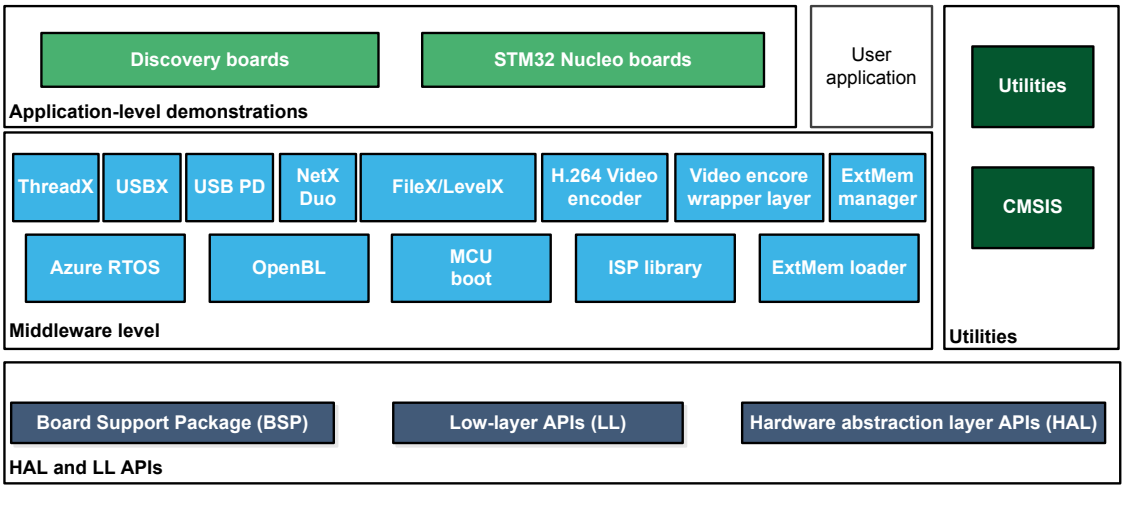
They come with free user-friendly license terms:

- Integrated and full featured Azure® RTOS: Azure® RTOS ThreadX
- Advanced file system and flash translation layer: FileX / LevelX
- CMSIS-RTOS implementation with Azure® RTOS ThreadX
- USB Host and Device stacks coming with many classes: Azure® RTOS USBX
- Industrial grade networking stack: optimized for performance coming with many IoT protocols: NetX Duo
- VeriSilicon® H.264 video encoder software stack
- ST Image Signal Processing (ISP) Library
- ST USB Power Delivery library
- ST external memory manager
- MCUboot
- OpenBootloader

Several applications and demonstration implementing all these middleware components are also provided in the STM32CubeN6 MCU Package.

The STM32CubeN6 MCU Package component layout is illustrated in Figure 1.

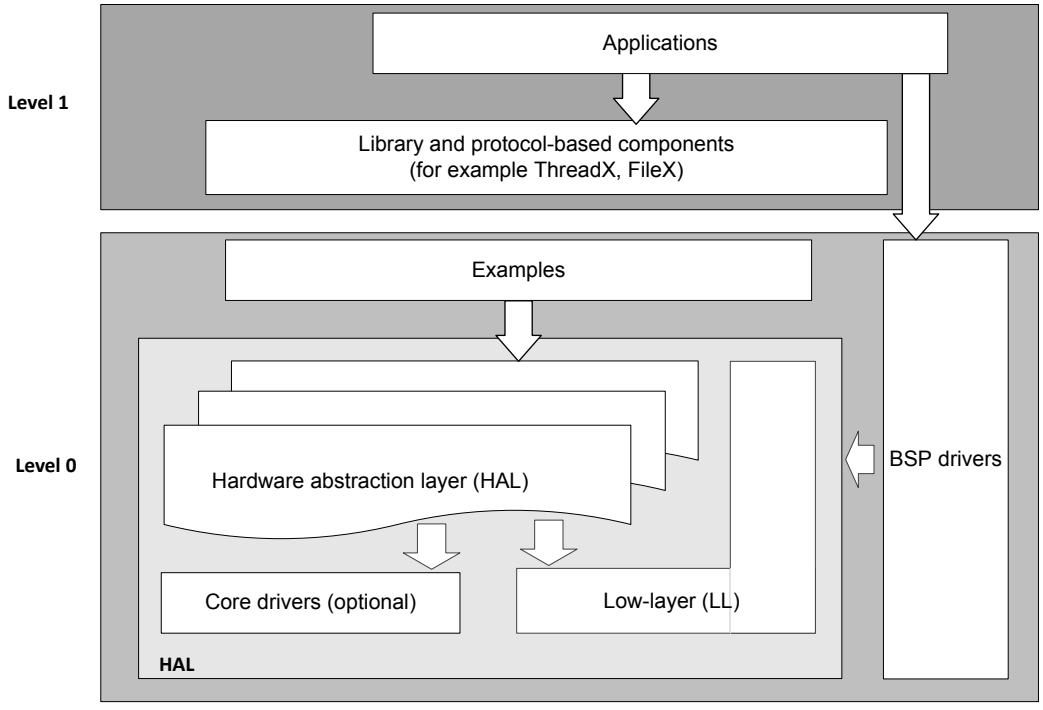
Figure 1. STM32CubeN6 MCU Package components



3 STM32CubeN6 architecture overview

The STM32CubeN6 MCU Package solution is built around three independent levels that easily interact as described in Figure 2.

Figure 2. STM32CubeN6 MCU Package architecture



3.1 Level 0

This level is divided into three sublayers:

- Board support package (BSP).
- Hardware abstraction layer (HAL):
  - HAL peripheral drivers
  - Low-layer drivers
- Basic peripheral usage examples.

3.1.1 Board support package (BSP)

This layer offers a set of APIs relative to the hardware components in the hardware boards (such as LCD, Audio, microSD™...). It is composed of two parts:

- Component:
  This is the driver relative to the external device on the board and not to the STM32. The component driver provides specific APIs to the BSP driver external components and could be portable on any other board.
- BSP driver:
  It allows linking the component drivers to a specific board and provides a set of user-friendly APIs. The API naming rule is `BSP_FUNCT_Action()`.
  Example: `BSP_LED_Init()`, `BSP_LED_On()`

BSP is based on a modular architecture allowing easy porting on any hardware by just implementing the low-level routines.

3.1.2 Hardware abstraction layer (HAL) and low-layer (LL)

The STM32CubeN6 HAL and LL are complementary and cover a wide range of applications requirements:

- The HAL drivers offer high-level function-oriented highly portable APIs. They hide the MCU and peripheral complexity to the end-user.
  The HAL drivers provide generic multi-instance feature-oriented APIs which simplify user application implementation by providing ready-to-use processes. For example, for the communication peripherals (I2S, UART, and others), it provides APIs allowing initializing and configuring the peripheral, managing data transfer based on polling, interrupting, or DMA process, and handling communication errors that may arise during communication. The HAL driver APIs are split into two categories:
  1. Generic APIs which provide common and generic functions to all the STM32 series microcontrollers.
  2. Extension APIs which provide specific and customized functions for a specific family or a specific part number.
- The low-layer APIs provide low-level APIs at the register level, with better optimization but less portability. The LL drivers are designed to offer a fast lightweight expert-oriented layer that is closer to the hardware than the HAL. Contrary to the HAL, LL APIs are not provided for peripherals where optimized access is not a key feature, or for those requiring heavy software configuration or complex upper-level stack.
  The LL drivers feature:
  – A set of functions to initialize peripheral main features according to the parameters specified in data structures.
  – A set of functions to fill initialization data structures with the reset values corresponding to each field.
  – Function for peripheral de-initialization (peripheral registers restored to their default values).
  – A set of inline functions for direct and atomic register access.
  – Full independence from HAL and capability to be used in standalone mode (without HAL drivers).
  – Full coverage of the supported peripheral features.

3.1.3 Basic peripheral usage examples

This layer encloses the examples built over the STM32 peripherals using only the HAL and BSP resources.

3.2 Level 1

This level is divided into two sublayers:

- Middleware components
- Examples based on the middleware components

3.2.1 Middleware components

The middleware is a set of libraries covering Microsoft® Azure® RTOS, VeriSilicon® H.264 video encoder software stack, external memory manager, USBPD library.

Horizontal interaction between the components of this layer is done by calling the featured APIs.

Vertical interaction with the low-layer drivers is done through specific callbacks and static macros implemented in the library system call interface.

The main features of each middleware component are as follows:

- Microsoft® Azure® RTOS
  – Azure® RTOS ThreadX: A real-time operating system (RTOS), designed for embedded systems with two functional modes.
    ◦ Common mode: Common RTOS functionalities such as thread management and synchronization, memory pool management, messaging, and event handling.
    ◦ Module mode: An advanced user mode that allows loading and unloading of prelinked ThreadX modules on the fly through a module manager.
  – NetX Duo
  – FileX
  – USBX
- VeriSilicon® H.264 video encoder
  – Software stack offering H.264 video encoding feature based on the STM32CubeN6 video encoder hardware peripheral.

- USB Power Delivery middleware
  - USB Type-C® power delivery service, implementing a dedicated protocol for power management in this evolution of the USB.org specification (refer to http://www.usb.org/ for more details.)
  - PD3 specifications (support for source/sink/dual roles)
  - Fast role swap
  - Dead battery
  - Use of configuration files to change the core and library configuration without changing the library code (read only)
  - RTOS and standalone operation
  - A link with the low-level driver through an abstraction layer using the configuration file to avoid any dependency between the library and the low-level drivers
- Image Signal Processing middleware
  - Library hosting 2A algorithms (Auto Exposure and Auto White Balance) and mechanisms to control the ISP and load sensor tuning file
- External memory manager
  - This middleware component provides a software solution to facilitate external memory integration.
- External memory loader
  - This middleware component provides a software solution for building the loader for external SFDP NOR memories.
- OpenBootloader
  - This middleware component provides an open source bootloader with exactly the same features and tools as the STM32 system bootloader.
- MCUboot

3.2.2 Utilities

Alike all STM32CubeN6 MCU Package, the STM32CubeN6 provides a set of utilities that offer miscellaneous software and additional system resources services that can be used by either the application or the different STM32Cube firmware intrinsic middleware and components.

4 STM32CubeN6 Firmware package overview

4.1 Supported STM32N6 series devices and hardware

STM32Cube offers a highly portable hardware abstraction layer (HAL) built around a generic architecture. It allows the build-upon layers principle, such as using the middleware layer to implement their functions without knowing, in-depth, what MCU is used. This improves the library code reusability and guarantees easy portability to other devices.

In addition, thanks to its layered architecture, the STM32CubeN6 offers full support of all STM32N6 series.

Table 1 shows the macro to define depending on the STM32N6 series device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32N6 series

Macro defined in stm32n6xx.h	STM32N6 series devices
STM32N657xx	STM32N657X0, STM32N657L0, STM32N657B0, STM32N657I0, STM32N657Z0, STM32N657A0
STM32N647xx	STM32N647X0, STM32N647L0, STM32N647B0, STM32N647I0, STM32N647Z0, STM32N647A0
STM32N655xx	STM32N655X0, STM32N655L0, STM32N655B0, STM32N655I0, STM32N655Z0, STM32N655A0
STM32N645xx	STM32N645X0, STM32N645L0, STM32N645B0, STM32N645I0, STM32N645Z0, STM32N645A0

STM32CubeN6 features a rich set of examples and applications at all levels making it easy to understand and use any HAL driver or middleware components. These examples run on the STMicroelectronics boards listed in Table 2.

Table 2. Boards for STM32N6 series

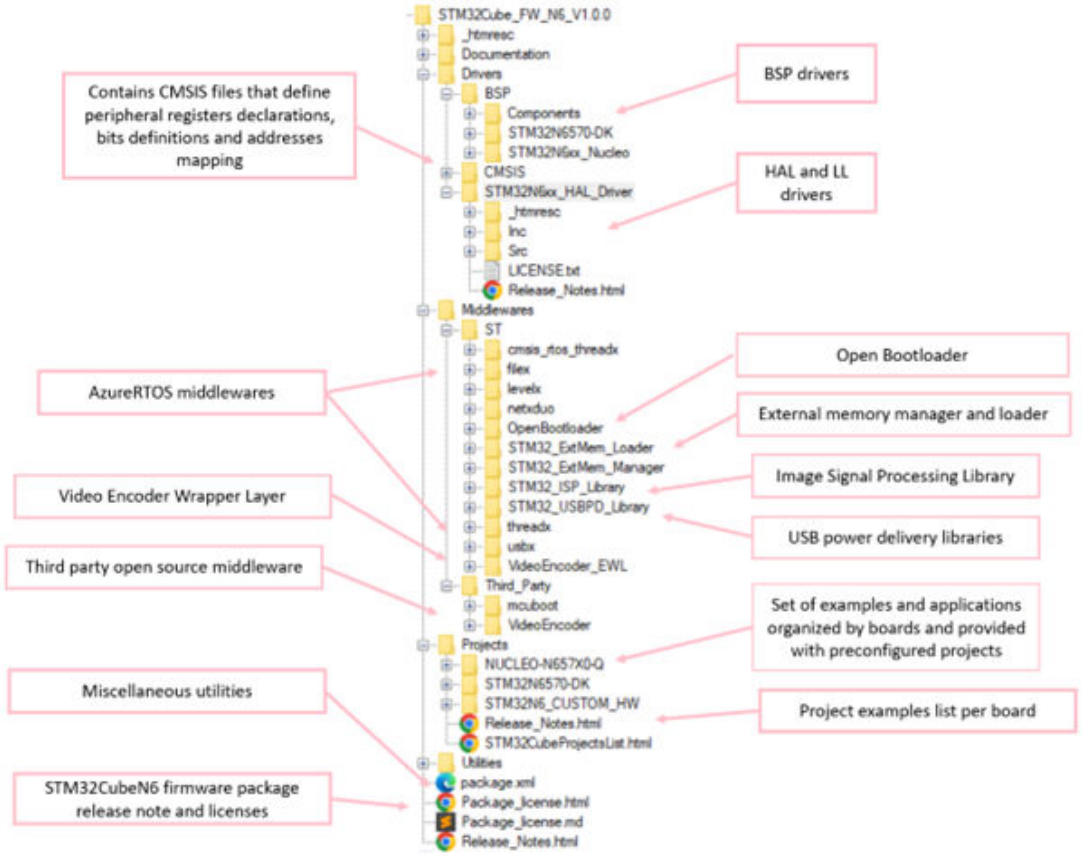
Board	Board STM32N6 supported devices
NUCLEO-N657X0-Q	STM32N657X0H3QU
STM32N6570-DK	STM32N657X0H3QU

The STM32CubeN6 MCU Package can run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on their board if the latter has the same hardware features (such as LED, LCD display, and buttons).

4.2 Firmware package overview

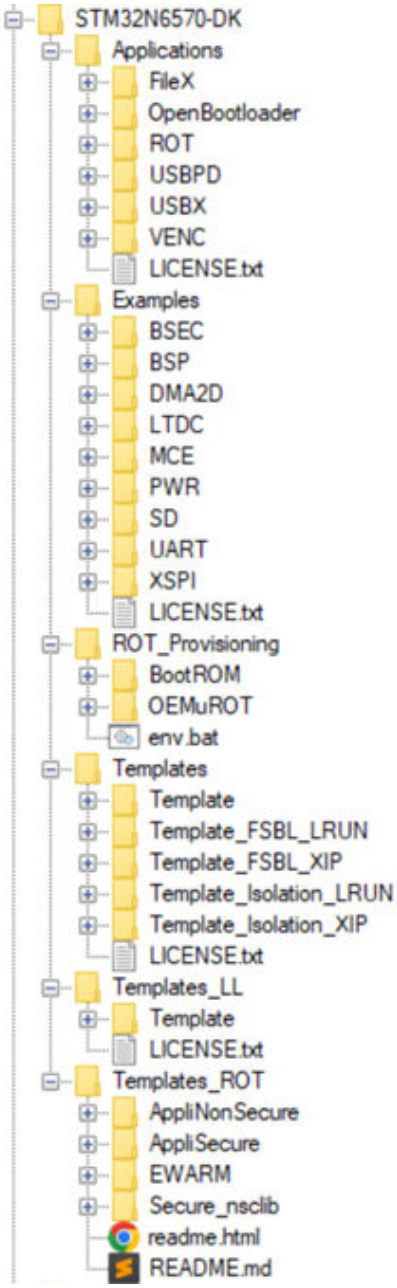
The STM32CubeN6 Package solution is provided in one single zip package having the structure shown in Figure 3.

Figure 3. STM32CubeN6 firmware package structure



Caution: The user must not modify the components files. Only the `\Projects` sources can be edited by the user. For each board, a set of examples is provided with preconfigured projects for IDE toolchains.

Figure 4. Overview of STM32N6570-DK examples



The examples are classified depending on the STM32Cube level they apply to, and are named as follows:

- Level 0 examples are called "Examples" and "Examples_LL". They use respectively HAL drivers, LL drivers without any middleware components.
- Level 1 examples are called Applications. They provide typical use cases of each middleware component.

Any firmware application for a given board can be built quickly with the template projects available in the Templates and Templates_LL directories.

A complete list of supported templates, examples, and applications on each board is available in the *STM32Cube ProjectsList.html* file.

In the STM32CubeN6 MCU Package, most of the HAL examples are executed from internal RAM, directly after the bootROM execution. This is why they are described as FSBL (First Stage Boot Loader) projects and their structure is described below in Section 4.2.1.1: Template project.

4.2.1 Templates projects structure

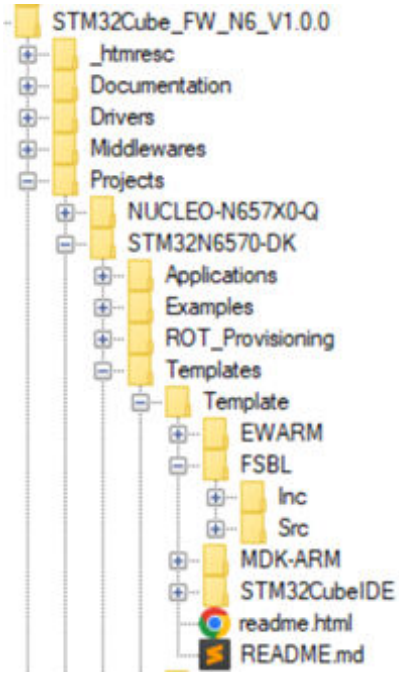
Several templates are provided for each board, but all templates follow the same project structure. Some of them are just simplified, depending on the targeted memory for code execution.

4.2.1.1 Template project

Template project is described by Figure 5.Similarly to other MCUs firmware package project templates, it provides a set of source and include files allowing to start any project. Such template project limits itself to an FSBL project (First Stage Boot Loader) meaning it runs immediately after the execution.

The FSBL binary is initially stored in STM32CubeN6 board external memory. It is copied by the bootROM at power-on in the internal SRAM and is executed in that memory as soon as the bootROM execution is over.

Figure 5. STM32CubeN6 project template



- `\Inc` folder contains all header files for the FSBL.
- `\Src` folder contains all the sources code for the FSBL.
- `\EWARM` folder contains the preconfigured project files & startup_files for EWARM.
- `\MDK-ARM` folder contains the preconfigured project files & startup_files for KEIL.
- `\STM32CubeIDE` folder contains the preconfigured project files & startup_files for STM32CubeIDE.

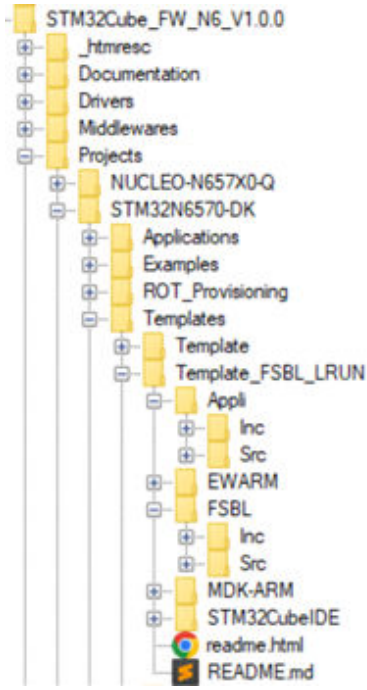
4.2.1.2 Template_FSBL_LRUN project

Template FSBL LRUN (LRUN for Load & Run) project provides a template for slightly more complex use cases, described in Figure 6.

The template yields two binaries, that of the FSBL and that of the application (Appli) both initially stored in STM32N6 board external memory.

At power on, the bootROM copies the FSBL binary in internal SRAM. Once the bootROM task is over, the FSBL project in turn executes: after clock and system settings, it copies the application binary in internal SRAM. When done, the application itself starts up and runs.

Figure 6. **STM32CubeN6 project FSBL_LRUN template**



DT73815V1

There are two sub-projects (basic project structures)

- FSBL (runs from internal flash memory)
  - System init (MPU, I/D-caches, System clock)
  - External memory init via External Memory MW
    - EXTMEM_Init()
    - EXTMEM_MemoryMapped()
    - Copy Appli from external memory to internal SRAM
  - Jump to Appli in external memory when done
- Appli (runs from internal flash memory)
  - Lighter system init (I/D-caches)
  - LED toggling (via BSP)

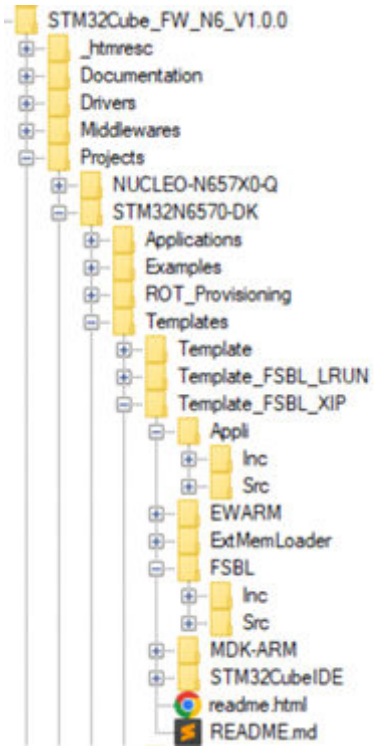
4.2.1.3 *Template_FSBL_XIP project*

Template FSBL XIP (eXecute In Place) project described in Figure 7 provides a template allowing the application to run in external memory.

The template yields two binaries, that of the FSBL and that of the application (Appli) both initially stored in STM32N6 board external memory.

At power on, the bootROM copies the FSBL binary in internal SRAM. Once the bootROM task is over, the FSBL project in turn executes: after clock and system settings, it configures the external memory containing the application binary in memory mapped mode. When the FSBL is done, the application in turn executes in external memory.

Figure 7. **STM32CubeN6 project FSBL_XIP template**



DT73816V1

There are two sub-projects (basic project structures)

- FSBL (runs from internal flash memory)
  - System init (MPU, I/D-caches, System clock)
  - External memory init via External Memory MW
    - EXTMEM_Init()
    - EXTMEM_MemoryMapped()
  - Jump to Appli in external memory when done
- Appli (runs from internal flash memory)
  - Lighter system init (I/D-caches)
  - LED toggling (via BSP)

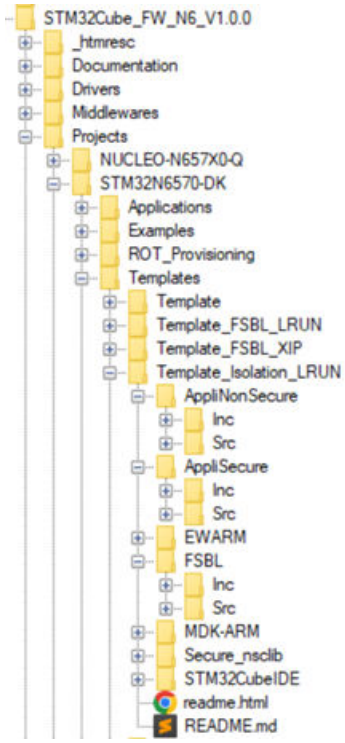
4.2.1.4 *Template_Isolation_LRUN project*

Template Isolation_LRUN (Load & Run) project provides a template for combining a secure application with a nonsecure application both running in internal RAM. The project is described in Figure 8.

The template yields three binaries, that of the FSBL, that of the secure application (AppliSecure) and that of the nonsecure application (AppliNonSecure); the three of them initially stored in STM32N6 board external memory.

At power on, the bootROM copies the FSBL binary in internal SRAM. Once the bootROM task is over, the FSBL project in turn executes. After clock and system settings, it copies the secure and nonsecure application binaries in internal SRAM. When done, the secure application itself starts up and sets the secure and nonsecure configurations. Finally, the nonsecure application is executed.

Figure 8. **STM32CubeN6 project Isolation_LRUN template**



DT73841V1

There are three subprojects (basic project structures)

- FSBL (runs from internal SRAM)
  - System init (MPU, I/D-caches, System clock)
  - External memory init via External Memory MW
    - EXTMEM_Init()
    - EXTMEM_MemoryMapped()
    - Copy secure application from external memory to internal SRAM
    - Copy nonsecure application from external memory to internal SRAM
  - Jump to secure application in internal memory when done
- Secure Application (runs from internal SRAM)
  - Lighter system init (I/D-caches)
  - Secure and nonsecure isolation setting
  - Jump to nonsecure application in internal memory when done
- Nonsecure application (runs from non-secure external flash memory)
  - LED toggling (via BSP)

4.2.1.5 *Template_Isolation_XIP project*

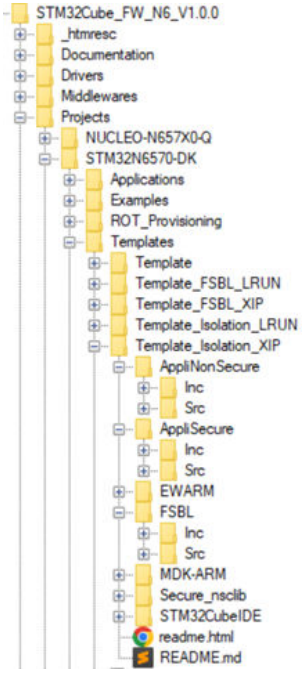
Template Isolation_XIP (eXecute In Place) project described in Figure 9 provides a template allowing to run a secure and nonsecure applications in external memory.

The template yields three binaries, that of the FSBL, that of the secure application, and that of the nonsecure application. All of them initially stored in STM32N6 board external memory.

At power-on, the bootROM copies the FSBL binary in internal SRAM. Once the bootROM task is over, the FSBL project in turn executes: after clock and system settings, it configures the external memory containing the application binary in execution mode. When the FSBL is done, the secure application in turn executes in external memory.

The secure application configures the secure and nonsecure isolation and when done, jumps into the nonsecure application.

Figure 9. STM32CubeN6 project Isolation_XIP template



DT73842V1

There are three sub-projects (basic project structures)

- FSBL (runs from internal SRAM)
  - System init (MPU, I/D-caches, System clock)
  - External memory init via External Memory MW
    - EXTMEM_Init()
    - EXTMEM_MemoryMapped()
  - Jump to secure application in external memory when done
- Secure application (runs from external flash memory)
  - Lighter system init (I/D-caches)
  - Secure and nonsecure isolation setting
  - Jump to nonsecure application in external flash memory when done
- Nonsecure application (runs from nonsecure external flash memory)
  - LED toggling (via BSP)

5 Getting started with STM32CubeN6

5.1 Running an example or an application

This section explains how simple is to run a template within STM32CubeN6.

1. Download the STM32CubeN6 MCU package.
2. Unzip it into a directory of your choice.
3. Make sure not to modify the package structure shown in Figure 1. STM32CubeN6 MCU Package components. Note that it is also recommended to copy the package at a location close to your root volume (meaning `C:\ST` or `G:\Tests`), as some IDEs encounter problems when the path length is too long.

As most of the examples or the applications, follow the basic template structure, the latter is used to describe the steps required to run a project.

For the small number of examples or applications that are based on a more complex structure, the reader is invited to follow the steps provided for the applicable template among the others delivered in the package (FSBL_LRUN, FSBL_XIP, FSBL_Isolation_LRUN, FSBL_Isolation_XIP).

Note: ***Remark regarding EWARM environment****: to monitor a variable in the live watch window, proceed as follow:*

1. *Start a debugging session.*
2. *Open the View > Images.*
3. *Double-click to deselect the second instance of project.out.*

5.1.1 Running the Template project

Prior to load and run a template in internal flash memory, it is mandatory to read the template readme file for any specific configuration.

The lines below picks `\Projects\STM32N6570-DK\Templates\Template` to describe the steps to follow. The steps are applicable to other templates of FSBL or to examples developed on FSBL templates.

1. Browse to `\Projects\STM32N6570-DK\Templates\Template`.
2. Select your preferred toolchain and open the project. A quick overview on how to open, build and run an example with the supported toolchains is given below.
3. Rebuild all files from the project.

Note: At this point, code can be executed in 'boot in development mode' for debugging purposes (settings described in the template readme file) as explained below for each of the IDE. Otherwise, follow the next steps to run the template in booting from external flash memory.

4. Resort to CubeProgrammer tool STM32MP_SigningTool_CLI.exe to add the header to the .bin binary generated by the rebuild done at step 3. Exact command and options to use are indicated in the readme file.
5. Set the board in boot development mode (setting described in the template readme file).
6. Load the image into the board external memory thanks to CubeProgrammer.
7. Set the board in boot from external flash memory mode (all information available in the readme file).
8. Press the reset button, the template automatically executes.

The readme file provides more information if need of template debug when in boot development mode.

To open, build and run an example, follow the steps below:

EWARM

1. Under the example folder, open `\EWARM` subfolder.
2. Launch the Project.eww workspace
3. Rebuild all files: [**Project**]>[**Rebuild all**]

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Load project image: [**Project**]>[**Debug**]
5. Run program: [**Project**]>[**Go (F5)**].

MDK-ARM:

1. Open the `\MDK-ARM` subfolder in the example folder.
2. Launch the Project.uvmpw workspace.

3. The workspace name may differ from one example to another.
4. Rebuild all files: [**Project**]>[**Rebuild all target files**]
   For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.
5. Load the project image: [**Project**]>[**Start/Stop Debug Session**].
6. Run the program: [**Debug**]>[**Run (F5)**].

STM32CubeIDE:

1. Open the STM32CubeIDE toolchain.
2. Click [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
3. Click [**File**]>[**Import**], select [**General**]>[**Existing Projects Into Workspace**] and click [**Next**].
4. Browse to the STM32CubeIDE workspace directory and select the project.
5. Rebuild all project files: select the project in the Project explorer window, then click on [**Project**]>[**Build project**].

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

6. Run the program: [**Run**]>[**Debug (F11)**].

If no debugging is required, follow the header addition and binaries loading steps described above.

5.1.2 Running the FSBL_LRUN (Load&Run) template project

Prior to load and run the FSBL_LRUN template in internal SRAM, it is mandatory to read the template readme file for any specific configuration.

The lines below pick `\Projects\STM32N6570-DK\Templates\Template_FSBL_LRUN` to describe the steps to follow.

1. Browse to `\Projects\STM32N6570-DK\Templates\Template_FSBL_LRUN`.
2. Select your preferred toolchain and open the project. A quick overview on how to open, build and run an example with the supported toolchains is given below.
3. Rebuild all files from the project FSBL.
4. Rebuild all files from the project Appli.
5. Resort to CubeProgrammer tool STM32MP_SigningTool_CLI.exe to add a header to each of the .bin binaries generated by the rebuild done at steps 3 and 4.
6. Set the board in boot development mode (setting described in the template readme file).
7. Load both image into the board external memory thanks to CubeProgrammer
   a. FSBL binary (including the header) at address `0x7000'0000`.
   b. Appli binary (including the header) at address `0x7010'0000`.

Note: At this point, code can be executed in 'boot in development mode' for debugging purposes (settings described in the template readme file) as explained below for each of the IDE. Otherwise, follow the next steps to run the template in booting from external flash memory.

8. Set the board in boot from external flash memory mode (all information available in the readme file).
9. Press the reset button, the template automatically executes.

To open, build and run an example, follow the steps below:

EWARM

1. Under the example folder, open `\EWARM` subfolder.
2. Launch the Project.eww workspace (the workspace name may change from one example to another).
3. Rebuild all files: [**Project**]>[**Rebuild all**].

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Copy the Appli .bin file extended with its header in external flash memory at address `0x7010'0000`.
5. Load FSBL project image: [**Project**]>[**Debug**].
6. Run the program: [**Debug**]>[**Go (F5)**].

MDK-ARM

1. Open the `\MDK-ARM` subfolder in the example folder.

2. Launch the Project.uvmpw workspace.
3. Rebuild all files for both projects: [Project: FSBL] and [Project: Appli]

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Add the header to the .bin file generated by the Appli project compilation (refer to step 5 above).
5. Copy the Appli.bin file extended with its header in external flash memory at address `0x7010'0000`.
6. Load the project image: [**Project**]>[**Debug**].
7. Run the FSBL program: [**Debug**]>[**Run (F5)**].

STM32CubeIDE:

1. Open the STM32CubeIDE toolchain.
2. Click [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
3. Click [**File**]>[**Import**], select [**General**]>[**Existing Projects Into Workspace**] and click [**Next**].
4. Browse to the STM32CubeIDE workspace directory and select the project.
5. Rebuild all project files for both sub-projects

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

6. Add the header to the .bin file generated by the Appli project compilation (refer to step 5 above).
7. Copy the Appli.bin file extended with its header in external flash memory at address `0x7010'0000`.
8. Run the FSBL program: [**Run**]>[**Debug (F11)**].

If no debugging is required, follow the header addition and binaries loading steps described above.

5.1.3 Running the FSBL_XIP (eXecute In Place) template project

Prior to load and run the FSBL_XIP template, it is mandatory to read the template readme file for any specific configuration.

The lines below pick `\Projects\STM32N6570-DK\Templates\Template_FSBL_XIP` to describe the steps to follow.

1. Browse to `\Projects\STM32N6570-DK\Templates\Template_FSBL_XIP`.
2. Select your preferred toolchain and open the project. A quick overview on how to open, build and run an example with the supported toolchains is given below.
3. Rebuild all files from the project FSBL.
4. Rebuild all files from the project Appli.
5. Resort to CubeProgrammer tool STM32MP_SigningTool_CLI.exe to add a header to each of the .bin binaries generated by the rebuild done at steps 3 and 4.
6. Set the board in boot development mode (setting described in template readme file).
7. Load both image into the board external memory thanks to CubeProgrammer
   a. FSBL binary (including the header) at address `0x7000'0000`.
   b. Appli binary (including the header) at address `0x7010'0000`.

Note: At this point, code can be executed in 'boot in development mode' for debugging purposes (settings described in the template readme file) as explained below for each of the IDE. Otherwise, follow the next steps to run the template in booting from external flash memory.

8. Set the board in boot from external flash memory mode (all information available in the readme file).
9. Plug back the board, the template automatically executes.

To open, build and run an example, follow the steps below:

EWARM

1. Under the example folder, open \EWARM subfolder.
2. Launch the Project.eww workspace.
3. Rebuild all files: [**Project**]>[**Rebuild all**].

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Copy the Appli .bin file extended with its header in external flash memory at address 0x7010'0000.
5. Load FSBL project image: [**Project**]>[**Debug**].
6. Run the program: [**Debug**]>[**Go (F5)**].

MDK-ARM

1. Open the \MDK-ARM subfolder in the example folder.
2. Launch the Project.uvmpw workspace.
3. Rebuild all files for both Projects : [Project: FSBL] and [Project: Appli]

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Add the header to the .bin file generated by the Appli project compilation (refer to step 5 above).
5. Copy the Appli .bin file extended with its header in external flash memory at address 0x7010'0000.
6. Load the project image: [**Project**]>[**Start/Stop Debug Session**].
7. Run the FSBL program: [**Debug**]>[**Run (F5)**].

STM32CubeIDE:

1. Open the STM32CubeIDE toolchain.
2. Click [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
3. Click [**File**]>[**Import**], select[**General**]>[**Existing Projects Into Workspace**] and click [**Next**].
4. Browse to the STM32CubeIDE workspace directory and select the project.
5. Rebuild all project files for both sub-projects:

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

6. Add the header to the .bin file generated by the Appli project compilation (refer to step 5 above).
7. Copy the Appli .bin file extended with its header in external flash memory at address 0x7010'0000.
8. Run the FSBL program: [**Run**]>[**Debug (F11)**].

If no debugging is required, follow the header addition and binaries loading steps described above.

5.1.4 Running the Isolation_LRUN (Load&Run) template project

Prior to load and run the Isolation_LRUN template in internal ram, it is mandatory to read the template readme file for any specific configuration. This template illustrates how to jointly run a secure and a nonsecure code in Load&Run mode.

The lines below pick \Projects\STM32N6570-DK\Templates\Template_Isolation_LRUN to describe the steps to follow.

1. Browse to \Projects\STM32N6570-DK\Templates\Template_Isolation_LRUN.
2. Select your preferred toolchain and open the project. A quick overview on how to open, build and run an example with the supported toolchains is given below.
3. Rebuild all files from the project FSBL.
4. Rebuild all files from the secure application project.
5. Rebuild all files from the nonsecure application project.
6. Resort to STM32CubeProgrammer tool STM32MP_SigningTool_CLI.exe to add a header to each of the .bin binaries generated by the rebuild done at steps 3, 4 and 5.
7. Set the board in boot development mode (setting described in template readme file).

8. Load the three images into the board external memory thanks to STM32CubeProgrammer
   a. FSBL binary (including the header) at address `0x7000'0000`.
   b. Secure application binary (including the header) at address `0x7010'0000`.
   c. Nonsecure application binary (including the header) at address `0x7018'0000`.

Note: At this point, code can be executed in 'boot in development mode' for debugging purposes (settings described in the template readme file) as explained below for each of the IDE. Otherwise, follow the next steps to run the template in booting from external flash memory.

9. Set the board in boot from external flash memory mode.
10. Press the reset button. The code then executes in boot from flash mode.

To open, build and run an example, follow the steps below:

EWARM

1. Under the example folder, open `\EWARM` sub-folder.
2. Launch the Project.eww workspace.
3. Rebuild all files: [**Project**]>[**Rebuild all**] for both subprojects.

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Add the header to the .bin file generated by the secure application project compilation (refer to step 5 above).
5. Add the header to the .bin file generated by the nonsecure application project compilation (refer to step 5 above).
6. Copy the secure application .bin file extended with its header in external flash memory at address `0x7010'0000`.
7. Copy the nonsecure application .bin file extended with its header in external flash memory at address `0x7018'0000`.
8. Load FSBL project image: [**Project**]>[**Debug**]
9. Run program: [**Debug**]>[**Go (F5)**]

If no debugging is required, follow the header addition and binaries loading steps described above.

MDK-ARM

1. Open the `\MDK-ARM` subfolder in the example folder.
2. Launch the Project.uvmpw workspace.
3. Rebuild all files for the three sub-projects: [Project: FSBL], [Project: AppliSecure] and [Project: AppliNonSecure]

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

4. Add the header to the .bin file generated by the secure application project compilation (refer to step 6 above).
5. Add the header to the .bin file generated by the nonsecure application project compilation (refer to step 6 above).
6. Copy the secure application .bin file extended with its header in external flash memory at address `0x7010'0000`.
7. Copy the nonsecure application .bin file extended with its header in external flash memory at address `0x7018'0000`.
8. Load the project image: [**FSBL Project**]>[**Start/Stop Debug Session**].
9. Run the FSBL program: [**Debug**]>[**Run (F5)**].

STM32CubeIDE:

1. Open the STM32CubeIDE toolchain.
2. Click [**File**]>[**Switch Workspace**]>[**Other**] and browse to the STM32CubeIDE workspace directory.
3. Click [**File**]>[**Import**], select[**General**]>[**Existing Projects Into Workspace**] and click [**Next**].
4. Browse to the STM32CubeIDE workspace directory and select the project.
5. Rebuild all project files for the three sub-projects:

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

6. Add the header to the .bin file generated by the secure application project compilation (refer to step 6 above).

7. Add the header to the .bin file generated by the nonsecure application project compilation (refer to step 6 above).
8. Copy the secure application .bin file extended with its header in external flash memory at address `0x7010'0000`.
9. Copy the nonsecure application .bin file extended with its header in external flash memory at address `0x7018'0000`.
10. Run the FSBL program: [**Run**]>[**Debug (F11)**].

5.1.5 Running the Isolation_XIP (eXecute In Place) template project

Prior to load and run the Isolation_XIP template in external flash memory, it is mandatory to read the template readme file for any specific configuration. This template illustrates how to jointly run a secure and a nonsecure code in execute-in-place mode.

The lines below pick `\Projects\STM32N6570-DK\Templates\Template_Isolation_XIP` to describe the steps to follow.

1. Browse to `\Projects\STM32N6570-DK\Templates\Template_Isolation_XIP`.
2. Select your preferred toolchain and open the project. A quick overview on how to open, build and run an example with the supported toolchains is given below.
3. Rebuild all files from the project FSBL.
4. Rebuild all files from the secure application project.
5. Rebuild all files from the nonsecure application project.
6. Resort to STM32CubeProgrammer tool STM32MP_SigningTool_CLI.exe to add a header to each of the .bin binaries generated by the rebuild done at steps 3, 4 and 5.
7. Set the board in boot development mode (setting described in template readme file).
8. Load the three images into the board external memory thanks to STM32CubeProgrammer
   a. FSBL binary (including the header) at address `0x7000'0000`.
   b. Secure application binary (including the header) at address `0x7010'0000`.
   c. Nonsecure application binary (including the header) at address `0x7018'0000`.

Note: At this point, code can be executed in 'boot in development mode' for debugging purposes (settings described in the template readme file) as explained below for each of the IDE. Otherwise, follow the next steps to run the template in booting from external flash memory.

9. Set the board in boot from external flash memory mode.
10. Press the reset button. The code then executes in boot from flash mode.

To open, build and run an example with EWARM, follow the steps below:

1. Under the example folder, open `\EWARM` sub-folder.
2. Launch the Project.eww workspace.
3. Rebuild all files: [**Project**]>[**Rebuild all**] for both subprojects.

Note: For debugging purposes, the template can be run in 'boot development mode'. In that case, carry on the following steps.

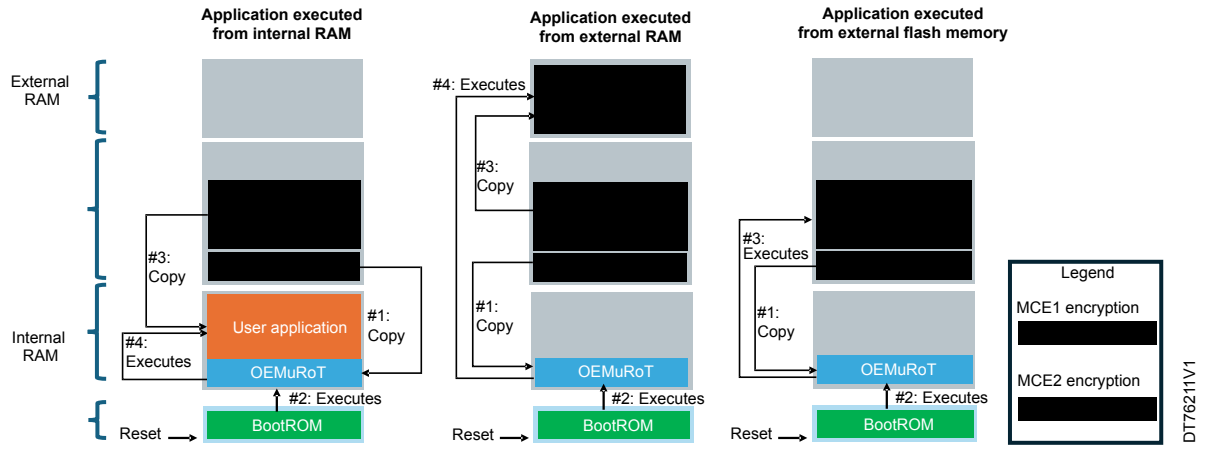
4. Add the header to the .bin file generated by the secure application project compilation (refer to step 5 above).
5. Add the header to the .bin file generated by the nonsecure application project compilation (refer to step 5 above).
6. Copy the secure application .bin file extended with its header in external flash memory at address `0x7010'0000`.
7. Copy the nonsecure application .bin file extended with its header in external flash memory at address `0x7018'0000`.
8. Load FSBL project image: [**Project**]>[**Debug**]
9. Run program: [**Debug**]>[**Go (F5)**]

If no debugging is required, follow the header addition and binaries loading steps described above.

5.2 Running a first Root of Trust (RoT) Example

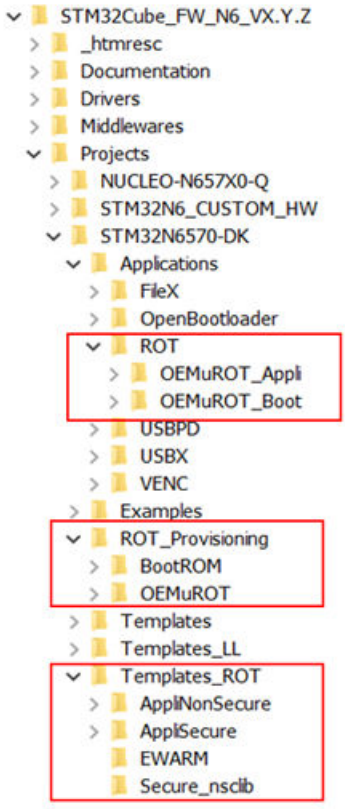
The STM32N6 devices support secure boot through OEMuRoT. OEMuRoT controls the integrity and the authenticity of the user application. If successful, OEMuRoT executes it from external flash memory or from external/internal RAM (load and run configurations).

Figure 10. Root of Trust



The RoT projects can be found in `\Projects\STM32N6570-DK\Applications\ROT`.
They are organized as shown in the figure below.

Figure 11. RoT applications



To run an application through the OEMuRoT bootpath, proceed as follows:

1. First, configure the user environment using the env script available in the ROT_Provisioning folder.

Caution: Make sure the STM32 Trusted Package Creator option is selected during the STM32CubeProgrammer installation, as it is used by the provisioning script.

2. In OEMuRoT folder, launch the provisioning script.
3. Follow the instructions displayed by the terminal. They guide the user through the following steps:
   a. Configuration management: BootROM keys configuration, OEMuRoT keys configuration and OEMuRoT DA password configuration.
   b. Images generation: OEMuRoT image generation, application firmware image generation then data images generation.
   c. Provisioning: OTP programming and images programming in external flash memory
4. Once the steps above have been executed, reset the target and connect the terminal emulator via the STLINK virtual communication port to get the application menu.

5.3 Developing a custom application

Recommendations for application development

To build a firmware application more easily, consider these recommendations:

L1-cache management: The instruction and data cache system integrated into the Arm® Cortex®-M55 processor should be used as a booster for the application and is recommended to enable them.

Buffer in RAM updated by hardware: When the data cache is enabled, the application's behavior may be impacted, especially when the data is hardware-related. For more details, refer to the application note Level 1 cache on STM32F7 series and STM32H7 series (AN4839). In the STM32CubeN6 MCU Package, all template scatter files include a section used as a non-cacheable area.

5.3.1 Using STM32CubeMX to develop or update an application

In the STM32CubeN6 MCU Package, nearly all example projects are generated with the STM32CubeMX tool to initialize the system, peripherals, and middleware. The direct use of an existing example project from the STM32CubeMX tool requires STM32CubeMX 6.13.0 or higher.

Follow these steps to update an application:

- After the installation of STM32CubeMX, open and if necessary update the proposed project. The quickest way to open an existing project is to double-click on the *.ioc file so STM32CubeMX automatically opens the project and its source files.
- The initialization source code of these projects is generated by STM32CubeMX. The main application source code is contained within the comments USER CODE BEGIN and USER CODE END. If the peripheral selection and settings are modified, STM32CubeMX updates the initialization part of the code but preserves the main application source code.

For developing a custom project with STM32CubeMX for any STM32N6 devices with external memory usage, follow these steps:

1. Select the STM32N6 device that matches the required set of peripherals.
2. Select the context to generate code for. Choose either a fully secure project or a joint secure and nonsecure project.
3. Set the FSBL context, which manages the start-up of the application and is always executed from the internal RAM memory after bootROM execution. At a minimum, configure the following items:
   – System clock. This is done as usual through the STM32CubeMX interface and applies for the FSBL project.
   – Peripheral interface to the external memory. The user selects the peripheral connected to the external memory. On the STM32N6 Discovery or Nucleo boards, the XSPI2 instance is the peripheral interface connected by default to the flash serial NOR SFDP memory.
   – If needed, external memory manager middleware. The user must configure the middleware EXTMEM_MANAGER, which provides solutions to simplify the use of the external memory: Select and configure the EXTMEM_MANAGER middleware for the FSBL context.
4. Configure all required embedded software using a pinout-conflict solver, a clock-tree setting helper and the utility performing MCU peripheral configuration (such as GPIO or USART) and middleware stacks (such as USB).

5. If required, set the application context. The application context is the end-user application stored in the external memory and its execution is initiated by the FSBL context. This process causes the application to inherit from the configurations done by the boot context. The application can be loaded in internal RAM to be executed there (Load & Run case) or the external memory can be configured so that the application runs in the latter (eXecute In Place [XIP] case).
   - The system clock. The clock is ready to use and the system clock value is retrieved by the application. This operation is handled by STM32CubeMX when the application code is generated.
   - I/D cache management: level 1 caches are disabled by the boot context before jumping to the application context, so the application must re-enable the I/D caches if necessary.
   - If the MPU was configured in the FSBL context, the configuration is inherited, but it is recommended to perform a new configuration of the MPU aligned with the application needs.
6. Generate the initialization C code based on the selected configuration. This code is ready to use within several development environments. The user code contained within the comments USER CODE BEGIN and USER CODE END is kept at the next code generation.

For more information about, refer to STM32CubeMX for STM32 configuration and initialization C code generation (UM1718).

5.3.2 Developing an application

This chapter describes the steps required to create a custom HAL application using STM32CubeN6.

Create a project

To create a new project, either start from the template project, provided for each board in `\Projects\<BOARDNAME>\Templates`, or from any available project in `\Projects\<BOARDNAME>\Examples` or `\Projects\<BOARDNAME>\Applications` (where <BOARDNAME> refers to the board name, such as STM32N6570-DK). The template project provides only an empty main loop function, which is a good starting point for understanding the STM32CubeN6 project settings. The template has the following characteristics:

- It contains the HAL source code and CMSIS, and BSP drivers that form the minimum set of components required to develop code on a given board.
- It contains the included paths for all firmware components.
- It defines the supported STM32N6 devices, allowing the configuration of the CMSIS and HAL drivers.
- It provides ready-to-use user files that are preconfigured as shown below:
  - HAL initialized with the default time base with Arm® core SysTick.
  - SysTick ISR implemented for HAL_Delay() purposes.

Note: When copying an existing project to another location, make sure to update all include paths.

Add the necessary middleware to the project (optional)

To identify the source files to be added to the project file list, refer to the documentation provided for each middleware component. Refer to the applications in `\Projects\<BOARDNAME>\Applications\<MW_Stack>` (where <MW_Stack> refers to the middleware stack, such as USBX) to know which source files and include paths to add.

Configure the firmware components

The HAL and middleware components offer a set of build-time configuration options, using macros (#define) declared in a header file. A template configuration file is provided with each component that has to be copied to the project folder (usually the configuration file is named xxx_conf_template.h, and the word "_template" needs to be removed when copying it to the project folder). The configuration file provides enough information to understand the impact of each configuration option. More detailed information is available in the documentation provided for each component.

Configure the I/D-caches

To improve performance, in the main() program, the application code must first enable I-cache and D-cache by calling SCB_EnableICache() and SCB_EnableDCache().

Start the HAL library

After jumping to the main program, the application code must call the HAL_Init() API to initialize the HAL library, which carries out the following tasks:

1. Configuration of the SysTick interrupt priority (through TICK_INT_PRIO defined in stm32n6xx_hal_conf.h).
2. Configuration of the SysTick to generate an interrupt every millisecond, which is clocked by the HSI (at this stage, the clock has not been configured yet and the system is running from the internal 64-MHz HSI).
3. Setting the NVIC group priority to 0.
4. Calling the HAL_MspInit() callback function defined in the stm32n6xx_hal_msp.c user file to perform global low-level hardware initializations.

Configure the MPU

After the HAL initialization, the application code must configure the MPU to define a background region with default attributes for all regions, and a region of RAM to store the non-cacheable buffer (recommended for hardware transfers).

Configure the system clock

The system clock configuration done by the FSBL project ensures a known clock setting after the bootROM execution.

HSI clock is enabled and set as source for the CPU and the system clocks by HAL_RCC_OscConfig() API.

Next PLL1 is configured by default by HAL_RCC_OscConfig() to be used as a source for the internal clocks IC2, IC6, and IC11 in setting the appropriate clock selection and clock divider parameters.

Finally, HAL_RCC_ClockConfig() is used to set CPU, AHB and APB bus clocks, in configuring the AHB prescalers, and the APB prescalers.

Initialize the peripheral

1. First, write the peripheral HAL_PPP_MspInit() function by following these steps:
   a. Enable the peripheral clock.
   b. Configure the peripheral GPIOs.
   c. Configure the DMA channel and enable DMA interrupt (if needed).
   d. Enable peripheral interrupt (if needed).
2. Edit stm32n6xx_it.c to call the required interrupt handlers (peripheral and DMA), if necessary.
3. Write the process complete callback functions if peripheral interrupt or DMA is going to be used.
4. In the user main.c file, initialize the peripheral handle structure, then call the HAL_PPP_Init() function to initialize the peripheral.

Develop an application

At this stage, the system is ready and the user application code development can start.

The HAL provides intuitive and ready-to-use APIs to configure the peripheral. It supports polling, interrupts, and a DMA programming model, to accommodate any application requirements. For more details on how to use each peripheral, refer to the rich example set provided in the STM32CubeN6 MCU Package.

Caution: In the default HAL implementation, a SysTick timer is used as the timebase; it generates interrupts at regular time intervals. If HAL_Delay() is called from the peripheral ISR process, make sure that the SysTick interrupt has a higher priority (numerically lower) than the peripheral interrupt. Otherwise, the caller ISR process is blocked. Functions affecting timebase configurations are declared as __weak to make an override possible in case of other implementations in the user file (using a general-purpose timer or other time source). For more details, refer to the HAL_TimeBase example.

5.3.3 ExtMemLoader subproject

"ExtMemLoader" is a subproject that is used to create a binary library capable of downloading an application to external memory. This binary is referred to as a "loader" and can be used by the IDE or STM32CubeProgrammer.

This project relies on the two middleware components: STM32_ExtMem_Manager and STM32_ExtMem_Loader. It does not have a main function, but STM32CubeMX generates an extmemloader_init() function that is responsible for initializing the system.

This initialization function performs the following operations:

- Initialize the system
  - IRQ disabling (interrupt are not used by the loader)
  - Enable the cache for performance purpose
  - Initialize the HAL
  - Disable any ongoing MPU configuration
  - Clock configuration at the maximum speed
- Initialize the memory
  - Initialize the peripheral associated with the memory
  - Call the ExtMem_Manager middleware component to initialize the memory

5.4 Getting STM32CubeN6 releases updates

The new STM32CubeN6 MCU Package releases and patches are available from www.st.com/stm32n6. They can also be retrieved from the [CHECK FOR UPDATE] button in STM32CubeMX. For more details, refer to section 3 of the user manual STM32CubeMX for STM32 configuration and initialization C code generation (UM1718).

6 FAQs

6.1 What is the licensing scheme for the STM32CubeN6 MCU Package?

Refer to the *Package_license* file at the root of the STM32CubeN6 package to retrieve the license terms of all the software elements.

6.2 Which boards are supported by the STM32CubeN6 MCU Package?

The STM32CubeN6 MCU Package provides BSP drivers and ready-to-use examples for the following STM32N6 series board:

- NUCLEO-N657X0-Q
- STM32N6570-DK

6.3 Are any examples provided with the ready-to-use toolset projects?

Yes. STM32CubeN6 provides examples and applications coming with the preconfigured project for IAR Embedded Workbench®, Keil®, and GCC IDEs.

6.4 Are there any links with standard peripheral libraries?

The STM32CubeN6 HAL and LL drivers are the replacement of the standard peripheral library:

- The HAL drivers offer a higher abstraction level compared to the standard peripheral APIs. They focus on the features that are common to the peripherals rather than hardware. A set of user-friendly APIs allows a higher abstraction level which in turn makes them easily portable from one product to another.
- The LL drivers offer low-layer registers level APIs. They are organized more simply and clearly, avoiding direct register accesses. LL drivers also include peripheral initialization APIs, which are more optimized compared to what is offered by the SPL, while being functionally similar. Compared to HAL drivers, these LL initialization APIs allows an easier migration from the SPL to the STM32CubeN6 LL drivers, since each SPL API has its equivalent LL API.

6.5 Does the HAL layer take advantage of interrupts or DMA? How can this be controlled?

Yes. The HAL layer supports three API programming models: Polling, interrupt, and DMA (with or without interrupt generation).

6.6 How are the product or peripheral specific features managed?

The HAL drivers offer extended APIs, which are specific functions provided as add-ons to the common API to support features only available on some products or lines.

6.7 When should the HAL be used versus LL drivers?

HAL drivers offer high-level and function-oriented APIs, with a high level of portability. Product/IPs complexity is hidden for end users.

LL drivers offer low-layer register level APIs, with a better optimization but less portable. They require in depth knowledge of product/IPs specifications.

6.8 How can LL drivers be included in an existing environment? Is there any LL configuration file as for HAL?

There is no configuration file. Source code must directly include the necessary `stm32n6xx_ll_ppp.h` files.

6.9 Can HAL and LL drivers be used together? If yes, what are the constraints?

It is possible to use both HAL and LL drivers. Use the HAL for the IP initialization phase and then manage the I/O operations with LL drivers.

The major difference between HAL and LL is that HAL drivers require to create and use handles for operation management while LL drivers operate directly on peripheral registers.

6.10 Why are SysTick interrupts not enabled on LL drivers?

When using LL drivers in standalone mode, there is no need to enable SysTick interrupts because they are not used in LL APIs, while HAL functions require SysTick interrupts to manage timeouts.

6.11 How are LL initialization APIs enabled?

The definition of LL initialization APIs and associated resources (Structures, literals, and prototypes) is conditioned by the `USE_FULL_LL_DRIVER` compilation switch.

To be able to use LL initialization APIs, add this switch in the toolchain compiler preprocessor.

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
24-Oct-2024	1	Initial release.

Contents

List of tables

List of figures