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
  - 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.

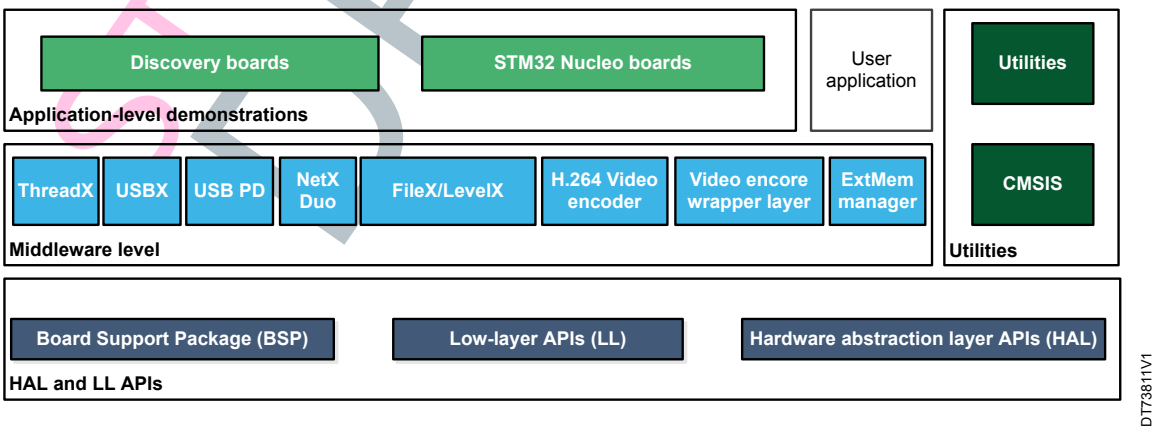
They come with free user-friendly license terms:

- 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 USB Power Delivery library
- ST external memory manager

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.

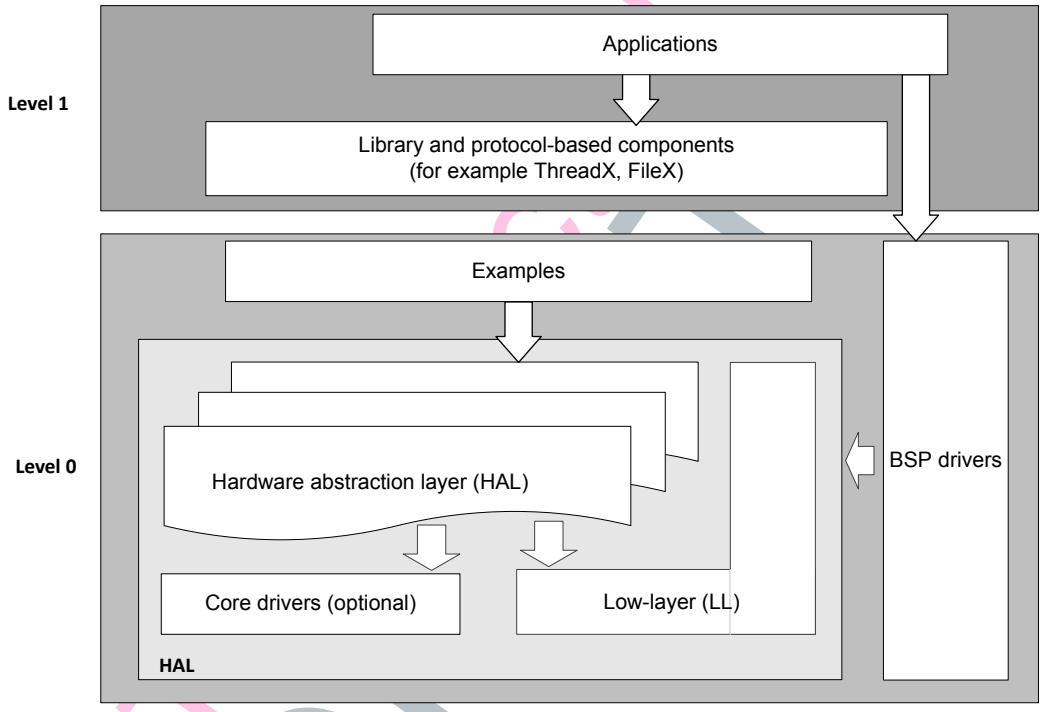
Figure 1. STM32CubeN6 MCU Package components



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 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 HAL

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 (I^2S, 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
- External Memory Manager

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

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

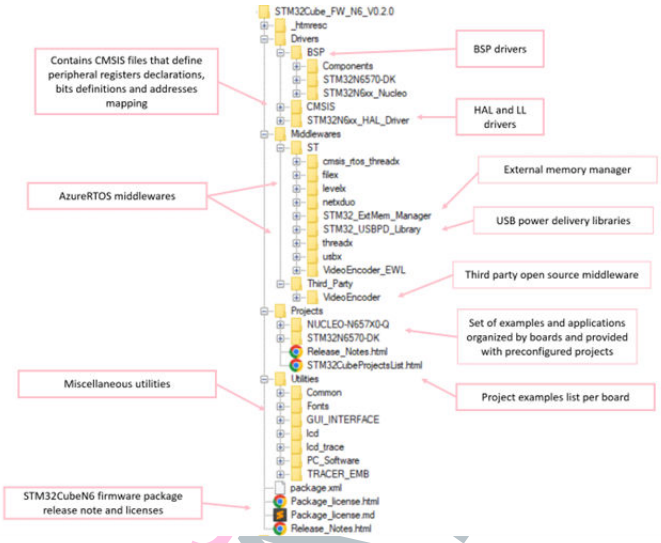
Board	Board STM32N6 supported devices
NUCLEO-N657X0-Q	STM32N657X0H3QU
STM32N6570-DK	STM32N657X0H3QU

The STM32CubeN6 MCU Package can run on any compatible hardware. The user simply updates the BSP drivers to port the provided examples on his 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



DT73813V1

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 EWARM toolchains.

4.2.1 Templates projects structure

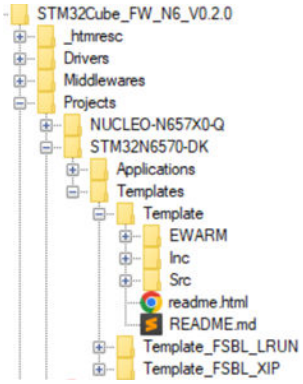
Several templates are provided for each board, but all templates follow the same project structure. Some of them are just simplified, depending on the targeted memory for code execution.

4.2.1.1 Template project

Template project is described by Figure 4.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 4. STM32CubeN6 project template



DT73814V1

- \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.

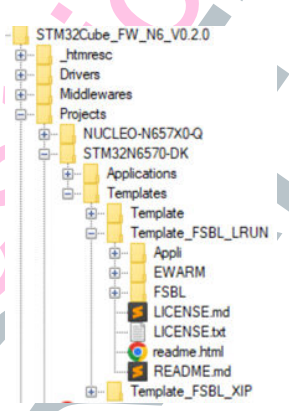
4.2.1.2 *Template_FSBL_LRUN project*

Template FSBL LRUN (LRUN for Load & Run) project provides a template for slightly more complex use cases, described in Figure 5.

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 5. STM32CubeN6 project FSBL_LRUN template



DT73815V1

There are two 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 Appli in external memory when done
- Appli (runs from external flash memory)
  - Lighter system init (I/D-caches)
  - LED toggling (via BSP)

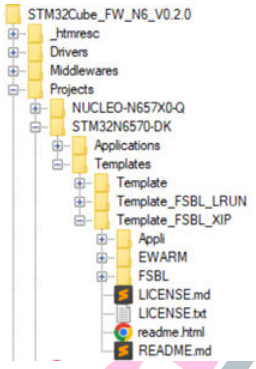
4.2.1.3 *Template_FSBL_XIP project*

Template FSBL XIP (XIP for eXecute In Place) project described in Figure 6 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 Execution mode. When the FSBL is done, the application in turn executes in external memory.

Figure 6. **STM32CubeN6 project FSBL_XIP template**



DT73816V1

There are two 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 Appli in external memory when done
- Appli (runs from external flash memory)
  - Lighter system init (I/D-caches)
  - LED toggling (via BSP)

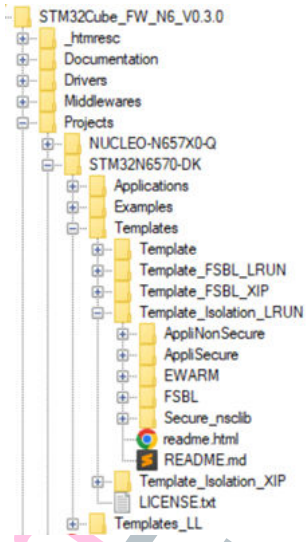
4.2.1.4 *Template_Isolation_LRUN project*

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 7.

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 7. **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 nonsecure internal SRAM)
  - LED toggling (via BSP)

4.2.1.5 *Template_Isolation_XIP project*

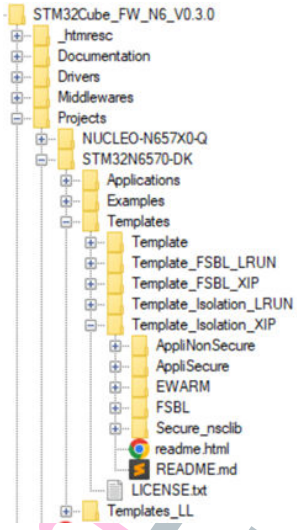
Template Isolation_XIP (XIP for eXecute In Place) project described in Figure 8 provides a template allowing to run a secure and nonsecure applications in external memory.

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, jump into the nonsecure application which executes at its turn.

Figure 8. **STM32CubeN6 project Isolation_LRUN 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)

4.2.2 Memory files description

Depending on project/board and external memories available, several memory descriptor files are provided, listed in Table 3.

Table 3. **Memory scatter files**

Nucleo template or DK template	FSBL	stm32n657xx_axisram2_fsbl.icf
Nucleo FSBL_LRUN or DK FSBL_LRUN	FSBL	stm32n657xx_axisram2_fsbl.icf
	Appli	stm32n657xx_LRUN.icf
Nucleo XIP or DK XIP	FSBL	stm32n657xx_axisram2_fsbl.icf
	Appli	stm32n657xx_XIP_XSPI2.icf
DK Isolation_LRUN	FSBL	stm32n657xx_axisram2_fsbl.icf
	AppliSecure	stm32n657xx_LRUN_s.icf
	AppliNonSecure	stm32n657xx_LRUN_ns.icf
DK Isolation_XIP	FSBL	stm32n657xx_axisram2_fsbl.icf
	AppliSecure	stm32n657xx_XIP_XSPI2_s.icf
	AppliNonSecure	stm32n657xx_XIP_XSPI2_ns.icf

5 Getting started with STM32CubeN6

5.1 Running a template

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. 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.

5.2 Running the Template project

Prior to load and run a template in internal SRAM, 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.
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 with EWARM, follow the steps below:

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)**]

If no debugging is required, follow the header addition and binaries loading steps described above starting from step 4.

5.3 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 `0x7002'0000`.
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 with EWARM, follow the steps below:

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 `0x7002'0000`.
5. Load FSBL project image: [**Project**]>[**Debug**].
6. Run the program: [**Debug**]>[**Go (F5)**].

If no debugging is required, follow the header addition and binaries loading steps described above starting from step 5.

5.4 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 `0x7002'0000`.
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 with EWARM, follow the steps below:

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 `0x7002'0000`.
5. Load FSBL project image: [**Project**]>[**Debug**].
6. Run the program: [**Debug**]>[**Go (F5)**].

If no debugging is required, follow the header addition and binaries loading steps described above starting from step 5.

5.5 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.
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 starting from step 6.

5.6 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`.
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 externalfFlash 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 starting from step 6.

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 is coming in the next releases.

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 4. **Document revision history**

Date	Revision	Changes
22-Sep-2023	0.1	Initial draft release.
28-Nov-2023	0.2	Modified topics • Section 4.2 Firmware package overview • Section 4.2.1.2 Template_FSBL_XIP project • Section 4.2.2 Memory files description • Section 5.2 Running the template project • Section 5.3 Running the FSBL_LRUN (Load&Run) template project • Section 5.4 Running the FSBL_XIP (eXecute In Place) template project Added topics: • Section 4.2.1.4 Template_Isolation_LRUN project • Section 4.2.1.5 Template_Isolation_XIP project • Section 5.3 Running the FSBL_LRUN (Load&Run) template project • Section 5.6 Running the Isolation_XIP (eXecute In Place) template project

Contents

List of tables

List of figures

Disclaimer

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.