Getting started with STM32CubeH7RS for STM32H7Sx/7Rx MCUs

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 STM32CubeH7RS for the STM32H7Rx/7Sx MCUs), 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 RTOS, FAT file system, TCP/IP, USB Host and Device, USB-PD, OpenBL, external memory loader and manager, and MCUboot
  - 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 STM32CubeH7RS MCU Package.

Section 2 describes the main features of the STM32CubeH7RS MCU Package. Section 3 and Section 4 provide an overview of the STM32CubeH7RS architecture and MCU Package structure.

1 General information

The STM32H7Rx/7Sx products come in different lines, mainly graphic and general-purpose lines, all based on the Arm® Cortex®-M7 processor.

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

2 STM32CubeH7RS main features

STM32CubeH7RS gathers, in a single package, all the generic embedded software components required to develop an application for the STM32H7Rx/7Sx MCUs microcontrollers. In line with the STM32Cube initiative, this set of components is highly portable, not only within the STM32H7Rx/7Sx MCUs microcontrollers but also to other STM32 series.

STM32CubeH7RS is fully compatible with the STM32CubeMX code generator for generating initialization code. The package includes low-layer (LL) and hardware abstraction layer (HAL) APIs that cover the microcontroller hardware, together with an extensive set of examples running on STMicroelectronics boards. The HAL and LL APIs are available in an open-source BSD license for user convenience. They are compliant with the MISRA C:2012 guidelines and have been reviewed with a static analysis tool to eliminate possible runtime errors. Reports are available on demand.

STM32H7Rx/7Sx products are "boot flash" products, embedding a small flash memory for the initial boot step. The application is located in an external memory.

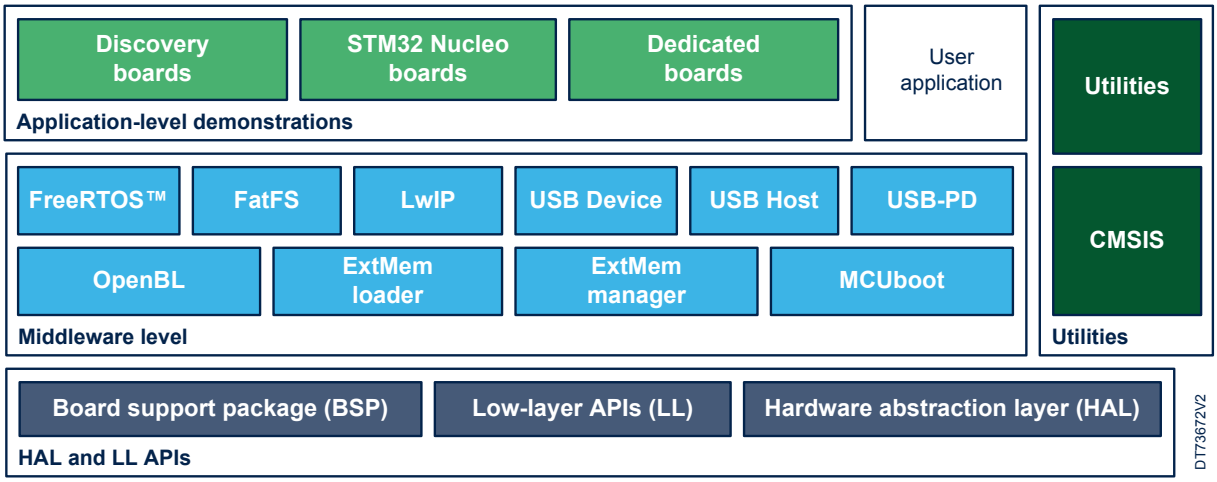
The STM32CubeH7RS MCU Package also contains a comprehensive middleware components with the corresponding examples. These come with very permissive, user-friendly license terms:

- Full USB Host and Device stacks, supporting many classes:
  - USB Host classes: HID, MSC, CDC, Audio, MTP
  - USB Device classes: HID, MSC, CDC, Audio, DFU
- External memory manager
- External memory loader
- CMSIS-RTOS implementation with the FreeRTOS™ open-source solution. This RTOS solution comes with dedicated communication primitives (stream and message buffers), allowing data to pass from an interrupt service routine to a task, or from one core to another in STM32H7 dual-core lines.
- FAT file system based on the open-source FatFS solution
- TCP/IP stack based on the open-source LwIP solution
- USB-PD library
- OpenBL
- MCUboot

Several applications and demonstration implementing these middleware components are provided in the STM32CubeH7RS MCU Package.

The STM32CubeH7RS MCU Package component layout is illustrated in the figure below.

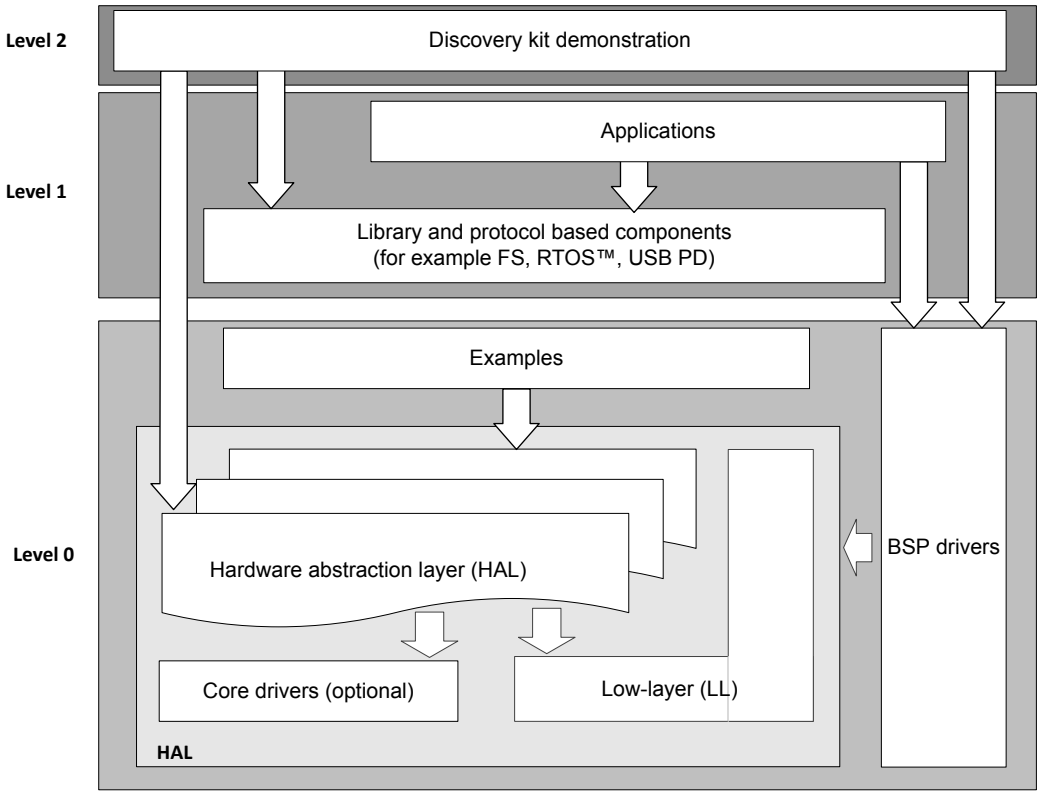
Figure 1. STM32CubeH7RS MCU Package components



3 STM32CubeH7RS architecture overview

The STM32CubeH7RS MCU Package solution is built around three independent levels that easily interact as described in the figure below.

Figure 2. STM32CubeH7RS 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 audio codec, touchscreen, SRAM or LCD drivers). It is composed of two parts:

- Component:
  This driver is related 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 can be ported onto any other board.
- BSP driver:
  This drivers links 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()`

The BSP is based on a modular architecture allowing for easy porting onto any type of hardware by implementing the low-level routines.

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

The STM32CubeH7RS HAL and LL are complementary and cover a wide range of application requirements:

- The HAL drivers offer high-level, function-oriented, and highly-portable APIs. They hide the MCU and peripheral complexity from 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 the initialization and configuration of the peripheral, managing data transfer based on the polling, interrupt, or DMA process, and the handling of communication errors that may arise during communication. The HAL driver APIs are split into two categories:
  - Generic APIs, providing common and generic functions to all STM32 series.
  - Extension APIs, providing specific and customized functions for a specific device family or a specific part number.
- The low-layer APIs provide low-level APIs at register level, with better optimization but less portability. They require a deep knowledge of the MCU and peripheral specifications. The LL drivers are designed to offer a fast, light-weight, expert-oriented layer, which is closer to the hardware than the HAL. Contrary to the HAL, LL APIs are not provided for peripherals without optimized access as a key feature, or for those requiring heavy software configuration and/or complex upper-level stacks.
  The LL drivers feature:
  - A set of functions to initialize the peripheral main features according to the parameters specified in data structures
  - A set of functions used to fill the initialization data structures with the reset values corresponding to each field
  - A function for peripheral de-initialization (where the peripheral registers are restored to their default values)
  - A set of inline functions for direct and atomic register access
  - Full independence from the HAL and the possibility of standalone-mode usage (without any HAL drivers)
  - Full coverage of the supported peripheral features

3.1.3 Basic peripheral usage examples

This layer encloses the examples built with 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 FreeRTOS™, FatFS, LwIP, and other in-house and open-source librairies (such as USB Host and Device, USB-PD, external memory manager, external memory loader, and MCUboot). All are integrated and customized for STM32 MCU devices and enriched with application examples based on STM32 evaluation boards. Horizontal interactions between the components of this layer are done by calling the feature APIs, while the vertical interaction with the low-layer drivers happens through specific callbacks and static macros implemented in the library system call interface.

The main features of each middleware component are as follows:

- FreeRTOS™
  - Open-source standard
  - CMSIS compatibility layer
  - Tickless operation during low-power mode
  - Integration with all STM32Cube middleware modules

- FAT file system
  - FatFS open-source library
  - Long file name support
  - Dynamic multi-drive support
  - RTOS and standalone operation
  - Examples with microSD™ and USB Host mass-storage class
- LwIP TCP/IP stack
  - Open-source standard
  - RTOS and standalone operation
- USB Host and Device libraries
  - Support for several USB classes (mass-storage, HID, CDC, DFU, AUDIO, MTP)
  - Support for multipacket transfer features for sending large amounts of data without splitting them into max packet size transfers
  - Use of configuration files to change the core and library configuration without changing the library code (read-only)
  - 32-bit aligned data structures to handle DMA-based transfers in high-speed modes
  - Support for multi-USB OTG core instances from user level through configuration file (that allows an operation with more than one USB Host/USB Device peripheral)
  - 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
- USB-PD Device and Core libraries
  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/developers/power delivery/ 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
- 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 Examples based on the middleware components

Each middleware component comes with one or more examples (also called applications) showing how to use it. Integration examples that use several middleware components are provided as well.

3.3 Level 2

This level is composed of a single layer, which consists of a global real-time and graphical demonstration based on the middleware service layer, the low-level abstraction layer, and the basic peripheral usage applications for board-based features.

3.4 Utilities

Like all STM32Cube MCU Packages, STM32CubeH7RS provides a set of utilities that offer miscellaneous software and additional system resource services that can be used either by the application or the different STM32Cube firmware-intrinsic middleware and components.

- Common
- CPU
- Fonts
- GUI
- JPEG
- LCD
- lcd_trace
- log
- ROT_Appli_Config
- Tracer_EMB

4 STM32CubeH7RS MCU package overview

4.1 Supported STM32H7Rx/7Sx MCUs 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 in-depth knowledge of which MCU is used. This improves the library code re-usability and portability to other devices.

Additionnally, the layered architecture of STM32CubeH7RS offers full support for all STM32H7Rx/7Sx MCUs. The user only needs to define the right macro in the `stm32h7rsxx.h` file.

Table 1 lists which the macro to define depending on the STM32H7Rx/7Sx device used. This macro must also be defined in the compiler preprocessor.

Table 1. Macros for STM32H7Rx/7Sx MCUs

Macro defined in stm32h5xx.h	STM32H5 part numbers
STM32H7R3xx	STM32H7R3R8V6, STM32H7R3V8T6, STM32H7R3Z8T6, STM32H7R3I8T6, STM32H7R3V8Y6, STM32H7R3V8H6, STM32H7R3Z8J6, STM32H7R3I8K6, STM32H7R3L8H6, STM32H7R3L8H6H
STM32H7R7xx	STM32H7R7I8T6, STM32H7R7Z8J6, STM32H7R7I8K6, STM32H7R7L8H6, STM32H7R7L8H6H
STM32H7S3xx	STM32H7S3R8V6, STM32H7S3V8T6, STM32H7S3Z8T6, STM32H7S3I8T6, STM32H7S3V8Y6, STM32H7S3V8H6, STM32H7S3Z8J6, STM32H7S3I8K6, STM32H7S3L8H6, STM32H7S3L8H6H
STM32H7S7xx	STM32H7S7I8T6, STM32H7S7Z8J6, STM32H7S7I8K6, STM32H7S7L8H6, STM32H7S7L8H6H

STM32CubeH7RS features a rich set of examples and applications at all levels, making it easy to understand and use any HAL driver and/or middleware components. These examples run on the STMicroelectronics boards listed in Table 2 below.

Table 2. Boards for STM32H7Rx/7Sx MCUs

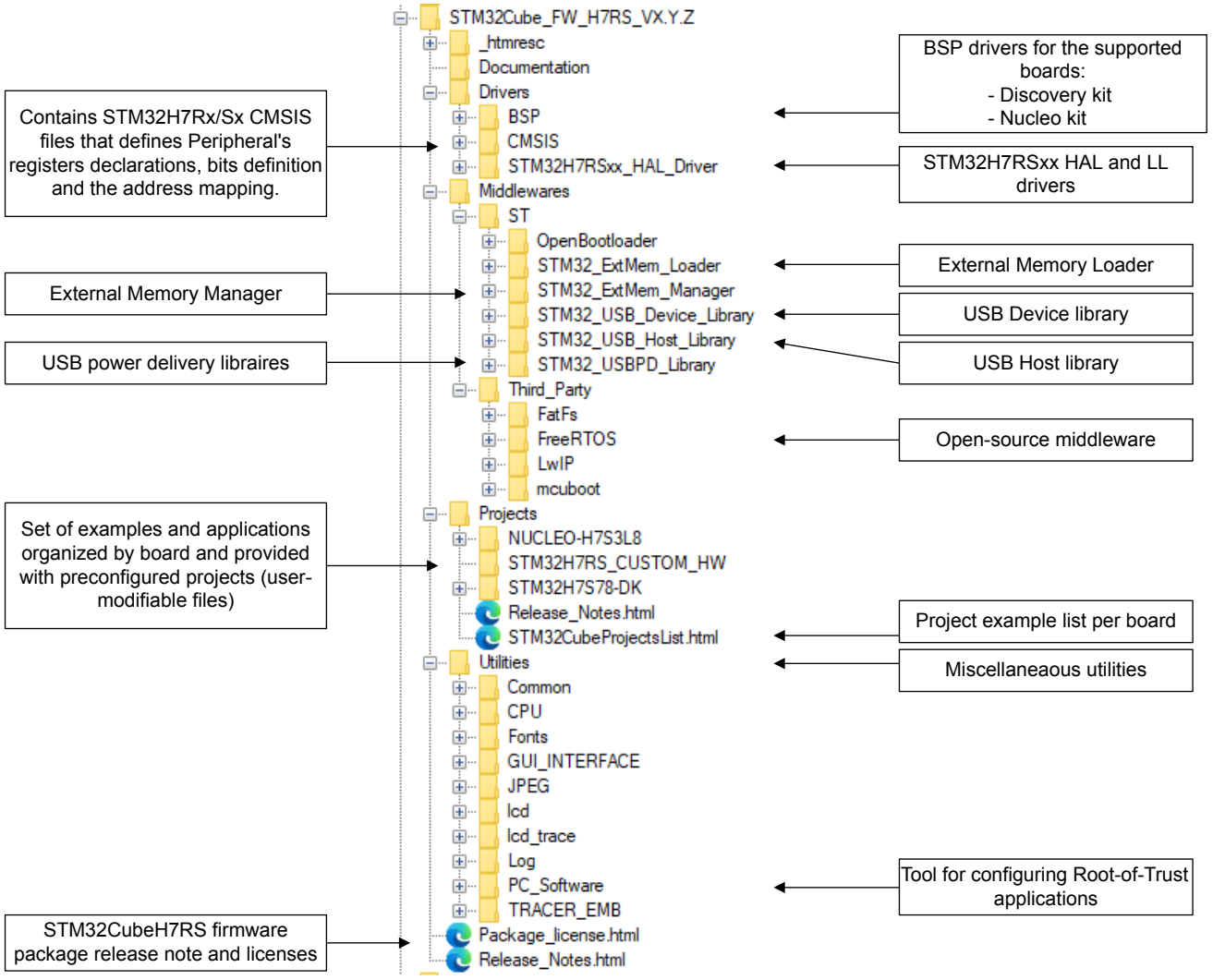
Board	Supported STM32H7Rx/7Sx part numbers
STM32H7S78-DK	STM32H7S7L8H6H
NUCLEO-H7S3L8	STM32H7S3L8H6

The STM32CubeH7RS MCU Package can run on any compatible hardware. The user updates the BSP drivers to port the provided examples onto their own board, if the latter has the same hardware features (such as LED, LCD display, and buttons).

4.2 MCU Package overview

The STM32CubeH7RS MCU Package is provided in one single zip package with the structure shown in Figure 3 below.

Figure 3. STM32CubeH7RS MCU Package structure

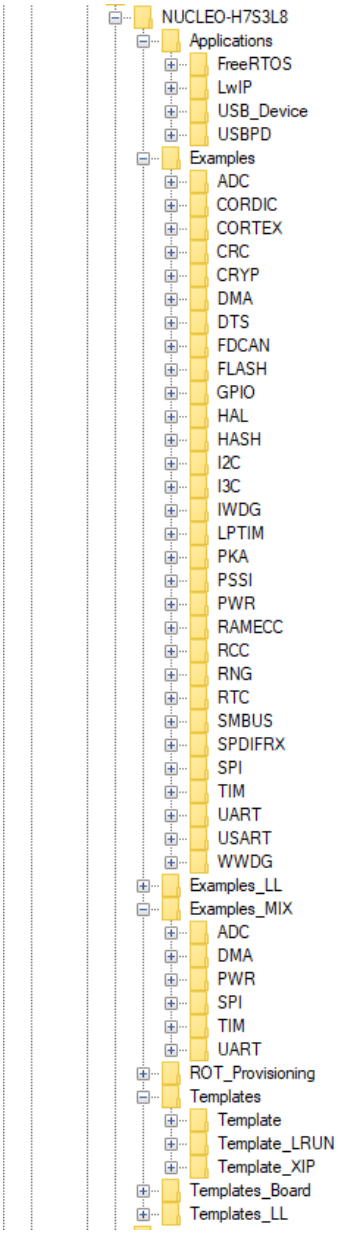


The component files must not be modified by the user. Only the *\Projects* sources are editable by the user.

For each board, a set of examples and templates is provided with preconfigured projects for EWARM, MDK-ARM, and STM32CubeIDE toolchains.

Figure 4 below shows the project structure for the STM32H7S78-DK board.

Figure 4. Overview of STM32CubeH7RS examples



DT73675V1

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

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

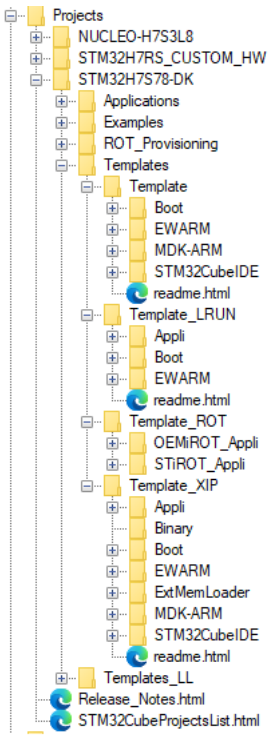
Any firmware application for a given board can be built quickly with the template projects available in the `Templates` and `Templates_LL` directories.

4.3 Templates project structure

Several templates are provided for each board, but all templates follow the same project structure:

1. `Templates`**`\Templates_Board`**: a reference template using the STM32Cube HAL and BSP API to demonstrate the STM32CubeMX "Start my project from ST Board": easy access to the features of the STMicroelectronics board.
2. `Templates`**`\Template`**: a reference template based on the STM32Cube HAL API that can be used to build any firmware to execute from the internal flash memory.
3. `Templates`**`\Template_LL`**: a reference template based on the STM32Cube LL API that can be used to build any firmware to execute from the internal flash memory.
4. `Templates`**`\Template_XIP`**: a reference template based on the STM32Cube HAL API that can be used to build any firmware application to execute code from an external flash memory (`\Appli` subproject). It boots from the internal flash memory and jumps to the application code in an external flash memory (`\Boot` subproject).
5. `Templates`**`\Template_LRUN`**: a reference template based on the STM32Cube HAL API that can be used to build any firmware application to execute from RAM an application stored in an external memory (`\Appli` subproject). It boots from the internal flash memory, copies the application from the external memory to an external/internal memory and jumps to the application code in the external/internal memory (`\Boot` subproject).
6. `Templates`**`\Template_ROT`**: contains the template for building a secure application.

Figure 5. Templates project structure



DT73676V1

These project folders can contain up to three subprojects or contexts:

- **\Boot**: manages the startup of the application (and runs from the internal flash memory).
  - System initialization (MPU, I/D-caches, system clock)
  - Configuration of the external memory interface peripheral
  - External memory initialization via external memory middleware:
    - EXTMEM_Init()
    - EXTMEM_MemoryMapped()
  - Jump to external memory
- **\Appli**: end user code (runs from an external flash memory).
  - Lighter system initialization (I/D-caches)
  - LED toggling (via BSP)
- **\ExtMemLoader**
  - Build specific external memory loader for targeted tools or IDE

The project folders contain the following subfolders:

- \Boot:
  - \Inc: contains all header files for the "Boot" part.
  - \Src: contains the source code for the "Boot" part.
- \Binary: contains the default Boot_XIP.hex file to be loaded in the internal flash memory. It is built with the \Inc and \Src files of the \Boot folder.
- \Appli
  - \Inc: contains all header files for the user application part.
  - \Src: contains the source code for user application part.
- \ExtMemloader: contains the files used to generate a binary to download an application to an external memory.
- \EWARM: contains the preconfigured project, startup, and linker files for EWARM.
- \MDK-ARM: contains the preconfigured project, startup, and linker files for MDK-ARM.
- \STM32CubeIDE: contains the preconfigured project, startup, and linker files for STM32CubeIDE.
- *.ioc file that allows the user to open most of firmware examples with STM32CubeMX.

Table 3 provides the type of projects available for each board.

Table 3. Project availability for each board

Board	Templates_LL	Templates	Examples	Examples_MIX	Examples_LL	Applications
STM32H7S78-DK	Available	Available	Available	Not available	Not available	Available
NUCLEO-H7S3L8	Available	Available	Available	Available	Available	Available

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

In the STM32CubeH7RS MCU Package, most of the HAL examples are executed from an external flash memory (XIP: execute in place).

Unless there are specific needs in the initial configuration of the external memory or system clock (the same as the one provided in the Boot part of \Template_XIP), only the Appli subproject is present and the extra image in the Boot part from the Template_XIP project (of the selected hardware board) is loaded automatically from the IDE.

If a specific configuration of the external memory or system clock frequency is needed, both the Boot and Appli subprojects are present, since the Boot part implements the specific configuration.

All LL examples are executed from the internal flash memory and exclusively from the Boot subproject.

5 Getting started with STM32CubeH7RS

5.1 Running a first example

This section explains how to run a first example with STM32CubeH7RS.

Follow these steps before running an example:

1. Download the STM32CubeH7RS MCU Package.
2. Unzip it into an appropriate directory. Make sure not to modify the package structure shown in Figure 3. STM32CubeH7RS MCU Package structure.
3. *Recommended:* copy the package as close as possible to the root volume (for example `C\ST` or `G:\Tests`), because some IDEs encounter problems when the path length is too long.

5.1.1 Running an example in the internal flash memory

Prior to loading and running an example in the internal flash memory, it is mandatory to read the example readme file for any specific configurations.

1. Browse to `\Projects\NUCLEO-H7S3L8\Examples_LL`.
2. Open `\CRC`, then `\CRC_CalculateAndCheck`.
3. Open the project with the preferred toolchain. A quick overview on how to open, build, and run an example with the supported toolchains is given below.
4. Rebuild all files from the `Boot` subproject and load the image into the target memory using the preconfigured load feature in the IDE. This step is detailed below depending on the used IDE.
5. Run the example: the calculated CRC code is stored in a variable. Once calculated, the CRC value is compared to the expected CRC value, and if both are equal, LD1 is turned on. If there are any errors, LD1 blinks (in 1-second intervals).

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

- EWARM:
  1. Open the `\EWARM` subfolder in the example folder.
  2. Launch the `Project.eww` workspace.

Note: The workspace name may differ from one example to another.

  3. Rebuild all files: [**Project**]>[**Rebuild all**].
  4. Load the project image: [**Project**]>[**Debug**].
  5. Run the program: [**Debug**]>[**Go (F5)**].
- MDK-ARM:
  1. Open the `\MDK-ARM` subfolder in the example folder.
  2. Launch the `Project.uvprojx` workspace.

Note: The workspace name may differ from one example to another.

  3. Rebuild all files: [**Project**]>[**Rebuild all target files**].
  4. Load the project image: [**Project**]>[**Start/Stop Debug Session**].
  5. 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**].
  6. Run the program: [**Run**]>[**Debug (F11)**].

5.1.2 Running an example in the external flash memory

Prior to loading and running an example in the external flash memory, it is mandatory to read the example readme file for any specific configurations.

1. Browse to `\Projects\STM32H7S78-DK\Examples`.
2. Open `\GPIO`, then `\GPIO_IOToggle`.
3. Open the project with the preferred toolchain. A quick overview on how to open, build, and run an example with the supported toolchains is given below.
4. Rebuild all files from the `Appli` subproject and load the image into the target memory using the preconfigured load feature in the IDE. This subproject first loads the template `Boot_XIP.hex` to the internal flash memory and then loads the *Appli* part to the external memory available on the STM32H7S78-DK board.
5. Run the example: LD1, LD2, LD3, and LD4 (connected to PO.01, PO.05, PM.02, and PM.03 on the STM32H7S78-DK board) toggle in an infinite loop.

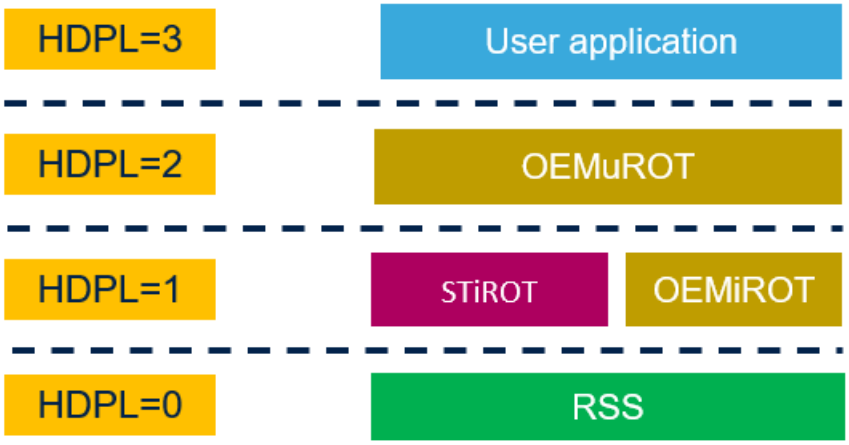
Note: If the `Boot` subproject is present in an example, compile this before compiling the `Appli` subproject.

5.1.3 Running a first Root of Trust (ROT) example

5.1.3.1 Bootpath overview

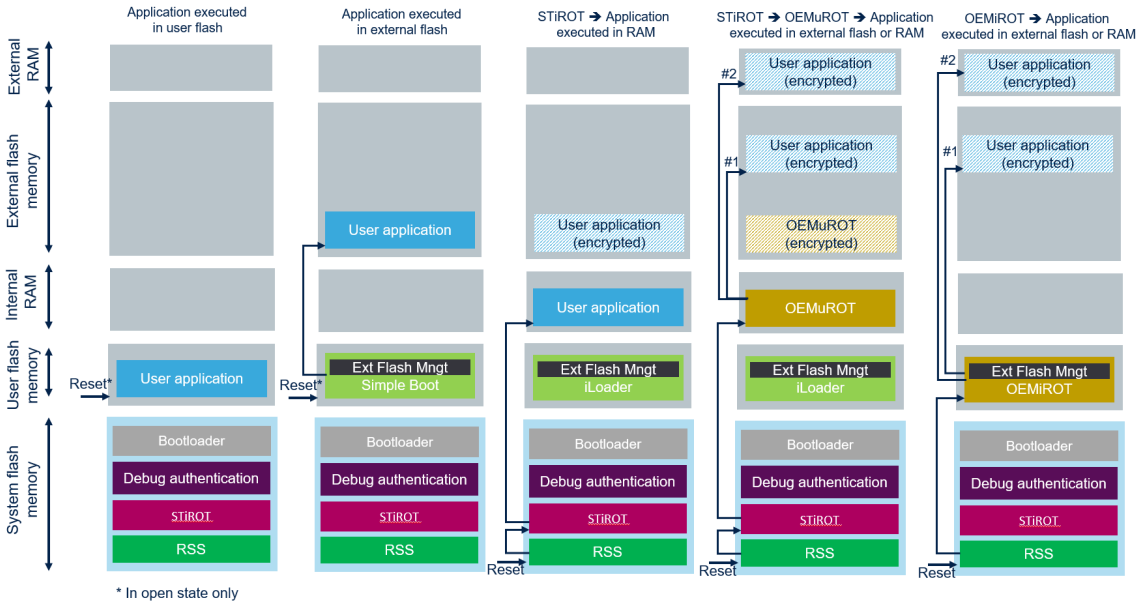
The STM32H7Rx/7Sx devices support temporal isolation through hide protection level (HDPL).

Figure 6. Temporal isolation levels on STM32H7Rx/7Sx MCUs



Several bootpaths are demonstrated on STM32H7Rx/7Sx devices. They consist of one or two boot stages provided by STMicroelectronics or implemented by original equipment manufacturers (OEMs).

Figure 7. Security bootpath supported on STM32H7Rx/7Sx MCUs

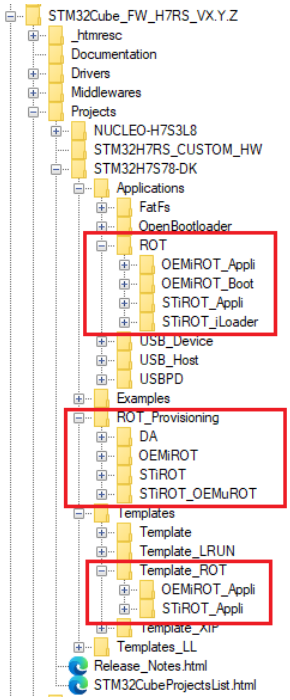


5.1.3.2 *ROT applications*

Prior to loading and running an ROT application, check the application readme file for any specific configurations that ensure that the related bootpath is enabled.

The ROT applications can be found in `\Projects\STM32H7S78-DK\Applications\ROT`. They are organized as shown in the figure below.

Figure 8. ROT application structure



DT73679V1

5.1.3.3 *OEMiROT, STiROT, and STiROT_OEMuROT bootpaths*

To run the OEMiROT, STiROT, or STiROT_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 STM32TrustedPackageCreator option is selected during the STM32CubeProgrammer installation, as it is used by the provisioning script.

2. Select the required bootpath, then launch the provisioning script located in each bootpath folder.
3. Once the provisioning script for the desired bootpath has started, follow the instructions displayed by the terminal. They guide the user through the following steps:
   a. Configuration management: option byte key (OBK) generation (STiROT or OEMiROT and debug authentication configuration).
   b. Image generation: image build (Secure Boot and application).
   c. Provisioning: image programming and OBK provisioning.
4. Once the steps above have been executed, reset the target and connect the terminal emulator via the ST-LINK virtual communication port to get the application menu.

Caution: Do not change the product state from "open" to a higher state without having provisioned the debug authentication (certificate and permissions). Otherwise, the MCU becomes unusable. The provisioning script ensures that the provisioning of the debug authentication is performed before modifying the product state, so the device can be reinitialized.

5.1.3.4 *Debug authentication (DA) regression*

After having run an ROT application, the device can be erased and reinitialized by erasing the flash memory and by switching the product back to an "open" state. This can be done by running the regression script located in the `ROT_Provisioning\DA` folder.

5.2 Developing a custom application

Recommendations for application development

To help you build a firmware application, consider these recommendations:

1. **MPU configuration**: to prevent speculative access on an Arm® Cortex®-M7 processor, it is recommended to use the memory protection unit (MPU) to control the accessible address ranges. This can be done by using a background region restricting the address ranges, so that any access outside the range generate a memory management fault error.
2. **L1-cache management**: the instruction and data cache system integrated into the Arm® Cortex®-M7 processor should be used as a booster for the application and is recommended to enable them.
3. **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 STM32CubeH7RS MCU Package, all template scatter files include a "buffer" section, which is used as a non-cacheable area through an MPU configuration with a non-cacheable attribute. This section must be adapted to the application to position data in relation to the hardware.
4. **External memory scatter file**: the STM32CubeH7RS MCU Package provides several scatter file templates for applications using external memory over a serial memory interface. The table below lists the available files with a brief description of their intent.

Table 4. Scatter files for STM32H7Rx/7Sx MCUs

Scatter files	Description
`stm32h7rsxx_ROMxspi1.xxx` `stm32h7rsxx_ROMxspi2.xxx`	System with one external memory: • Code exec from the address of XSPI1 mapped memory. • Code exec from the address of XSPI2 mapped memory.
`stm32h7rsxx_ROMxspi1_RAMxspi2.xxx` `stm32h7rsxx_RAMxspi1_ROMxspi2.xxx`	System with two external memories: • Code exec from the address of XSPI1 mapped memory and data from the address of XSPI2 mapped memory. • Code exec from the address of XSPI2 mapped memory and data from the address of XSPI1 mapped memory.

5.2.1 Using STM32CubeMX to develop or update an application

In the STM32CubeH7RS 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 6.11.0 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 by the comments `USER CODE BEGIN` and `USER CODE END`. If the IP 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 STM32H7Rx/7Sx devices with external memory usage, follow these steps:

1. Select the STM32H7Rx/7Sx microcontroller that matches the required set of peripherals.
2. Select the context to generate code for. (When a product is selected, different contexts become available for selection.)

3. Set the boot context, which manages the startup of the application and is executed from the internal flash memory. At a minimum, configure the following:
   - System clock. This is done as usual through the STM32CubeMX interface and applies for the boot and external memory loader (if selected) contexts.
   - Peripheral interface to the external memory. The user selects the peripheral connected to the external memory. On the STM32H7Rx/7Sx Discovery board, the XSPI2 instance corresponds to the peripheral interface to the serial NOR SFDP memory.
   - 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 boot context.
     - Configure the external memory.
     - Configure the boot use case to execute the application from the external flash memory (execute in place) or from the internal memory (load and run).
4. Configure all required embedded software using a pinout-conflict solver, a clock-tree setting helper, a power consumption calculator, and the utility performing MCU peripheral configuration (such as GPIO or USART) and middleware stacks (such as USB).
5. Set the application context. The application context is the end user application stored in the external memory and its execution is initiated by the boot context. This process causes the application to inherit from the configurations done by the boot context.
   - The system clock. The clock is ready to use and just the system clock value is known to 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 boot 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 by the comments `USER CODE BEGIN` and `USER CODE END` is kept at the next code generation.

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

For a list of the available example projects for the STM32CubeH7RS, refer to the STM32Cube firmware examples for STM32CubeH7RS application note.

5.2.2 Developing an application in the internal flash memory

This section describes the steps needed to create a custom LL or HAL application using STM32CubeH7RS.

5.2.2.1 LL project (boot subproject)

This chapter describes the steps required to create a custom LL application using STM32CubeH7RS.

Creating a new project

To create a new project, either start from the Templates_LL project provided for each board in `\Projects\ <BOARDNAME>\Templates_LL` or from any available project in `\Projects\<BOARDNAME>\Examples_LL` (where "<BOARDNAME>" refers to the board name, such as NUCLEO-H7S3L8).

This Template_LL project is a simple application with only the boot subproject, which demonstrates the usage of LL drivers.

This project provides an empty main loop function, which is a good starting point to understand the project settings for STM32CubeH7RS. The main characteristics of the template are the following:

- Source codes of the LL and CMSIS drivers, which are the minimum set of components needed to develop code on a given board.
- Include paths for all required firmware components.
- Selection of the supported STM32H7Rx/7Sx device and correct configuration of the CMSIS and LL drivers.

- Ready-to-use user files, which are preconfigured as follows:
  - `main.h`: `LED` and `USER_BUTTON` definition abstraction layer.
  - `main.c`:
    - Power supply configuration of the board
    - MPU configuration (recommended on Arm® Cortex®-M7)
    - I/D-cache enabling
    - System clock configuration for maximum frequency

Porting an existing project to another board

1. Start from the Templates_LL project provided for each board, available in the `\Projects\<BOARDNAME>\Templates_LL` folder.
2. Select an LL example.

Note: To find the board on which LL examples are deployed, refer to the list of LL examples in `STM32CubeProjectsList.html`.

Porting the LL example

- Copy and paste the `Templates_LL` folder to keep the initial source, or directly update an existing `Templates_LL` project.
- Replace `Templates_LL` files with the `Examples_LL` targeted project files.
- Keep all board-specific parts. For clarity reasons, board-specific parts have been flagged with the following specific tags:

```
/* ============== BOARD SPECIFIC CONFIGURATION CODE BEGIN ============== */
/* ============== BOARD SPECIFIC CONFIGURATION CODE END ============== */
```

The main porting steps are the following:

1. Replace the `stm32h7rsxx_it.h` file.
2. Replace the `stm32h7rsxx_it.c` file.
3. Replace the `main.h` file and update it. Keep the LED and user button definitions from the LL template within the `BOARD SPECIFIC CONFIGURATION` tags.
4. Replace the `main.c` file and update it:
   - Keep the clock configuration of the `SystemClock_Config()` LL template function within the `BOARD SPECIFIC CONFIGURATION` tags.
   - Depending on the LED definition, replace each LEDx occurrence with another LEDy available in the `main.h` file.

With these modifications, the example is now ready to run on the targeted board.

5.2.2.2 *HAL project (boot subproject)*

This chapter describes the steps required to create a custom HAL application using STM32CubeH7RS.

1. **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 NUCLEO-H7S3L8).
   The template project provides only an empty main loop function, which is a good starting point for understanding theSTM32CubeH7RS 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 include paths for all firmware components.
   – It defines the supported STM32H7Rx/7Sx MCUs 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 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.

2. **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 USB) to know which source files and include paths to add.
3. **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.
4. **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_EnableICache()`.
5. **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:
   a. Configuration of the SysTick interrupt priority (through `TICK_INT_PRIO` defined in `stm32h7rsxx_hal_conf.h`).
   b. Configuration of the SysTick to generate an interrupt every millisecond, which is clocked by the MSI (at this stage, the clock has not been configured yet and the system is running from the internal 64-MHz MSI).
   c. Setting the NVIC group priority to 0.
   d. Calling the `HAL_MspInit()` callback function defined in the `stm32h7rsxx_hal_msp.c` user file to perform global low-level hardware initializations. In this function, the power configuration on the board (SMPS or LDO) especially needs to be configured.
6. **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).
7. **Configure the system clock**
   The system clock configuration is done by calling the two APIs described below:
   – `HAL_RCC_OscConfig()`: this API configures the internal and/or external oscillators, as well as the PLL sources and factors. The user chooses to configure one or all oscillators. They can skip the PLL configuration if there is no need to run the system at a high frequency.
   – `HAL_RCC_ClockConfig()`: this API configures the system clock source, the flash memory latency, the AHB prescalers, and the APB prescalers.

8. **Initialize the peripheral**
   a. First, write the peripheral `HAL_PPP_MspInit` function by following these steps:
      i. Enable the peripheral clock.
      ii. Configure the peripheral GPIOs.
      iii. Configure the DMA channel and enable DMA interrupt (if needed).
      iv. Enable peripheral interrupt (if needed).
   b. Edit `stm32xxx_it.c` to call the required interrupt handlers (peripheral and DMA), if necessary.
   c. Write the process complete callback functions if peripheral interrupt or DMA is going to be used.
   d. In the user `main.c` file, initialize the peripheral handle structure, then call the `HAL_PPP_Init()` function to initialize the peripheral.
9. **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 STM32CubeH7RS 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.

5.2.3 Developing an application in the external flash memory

This section describes the steps needed to create a custom HAL execute-in-place (XIP) or load-and-run (LRUN) application using STM32CubeH7RS.

To create a new project, either start from the XIP project provided for each board in `\Projects\ <BOARDNAME> \Templates_XIP` or the LRUN project template in `\Projects\<BOARDNAME>\Examples_LRUN` (where "<BOARDNAME>" refers to the board name, such as NUCLEO-H7S3L8).

Both templates are composed of a project structure with three subprojects: Boot, Appli and ExtMemLoader. They are described in the following sections.

5.2.3.1 Boot subproject

In the case of external memory usage by the application, the boot subproject is built using the STM32_ExtMem_Manager middleware component. This project mainly performs these three operations:

- System initialization (caches, MPU, clocks, peripherals)
- Initialization of external memory/memories
- Application start (based on an execute in place (XIP) or load and run (LRUN) boot use case).

System initialization and external memory configuration

In this stage, the user can change following :

- MPU configuration
- I/D-cache enabling.
- `HAL_Init()`
- System clock configuration:
  - Hardware voltage scaling configuration
  - PLLx configuration for external memories (for instance, up to 200 MHz for serial interface, and up to 133 MHz for parallel interface)
- Clock configuration for peripherals:
  - `HAL_RCCEx_PeriphCLKConfig()`, with:
    - Desired peripheral clock selection
    - Clock source for this peripheral
- Initialize all configured peripherals, especially the external memory interface (for example, `MX_XSPI1_Init()`)

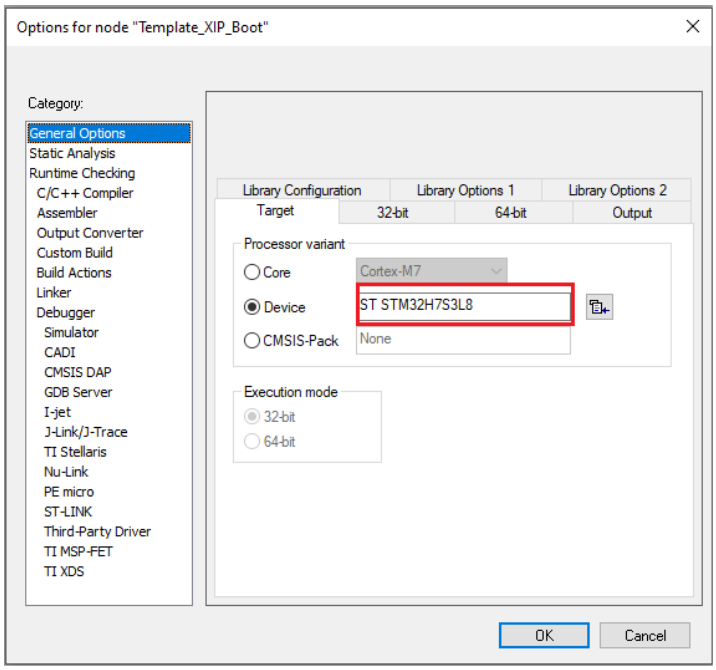
- Initialization of the STM32_ExtMem_Manager middleware component with a call to `MX_EXTMEM_MANAGER_Init()`
- Launch the application depending on the boot choice (XIP or LRUN) selected in the STM32_ExtMem_Manager middleware component.

Internal memory loader selection

As with other STM32 products, select the targeted device to select the associated internal flash memory loader automatically.

In IAR Embedded Workbench, go to [**General Options**]>[**Target**]>[**Device**] to select the device, as shown in the figure below.

Figure 9. IAR: device selection for the boot subproject



DT73680V1

5.2.3.2 *Appli subproject*

The application part being executed following the boot, it inherits part of the configuration programmed by the application subproject. For efficiency purposes, the template considers the boot as part of the startup of the application, so the configurations are aligned as much as possible with the needs of the application. Thus, the configuration of the clock system is consistent with the needs of the application.

There is only one exception to consider: the MPU. The application subproject uses a memory mapping that is very different from the memory mapping of the boot subproject, so it is recommended to set an MPU configuration adapted to the needs of the application. The template configures the MPU in the user code section.

Note: The scatter file of the application depends on the memories present in the system. This file is provided as a template and can be modified by the application.

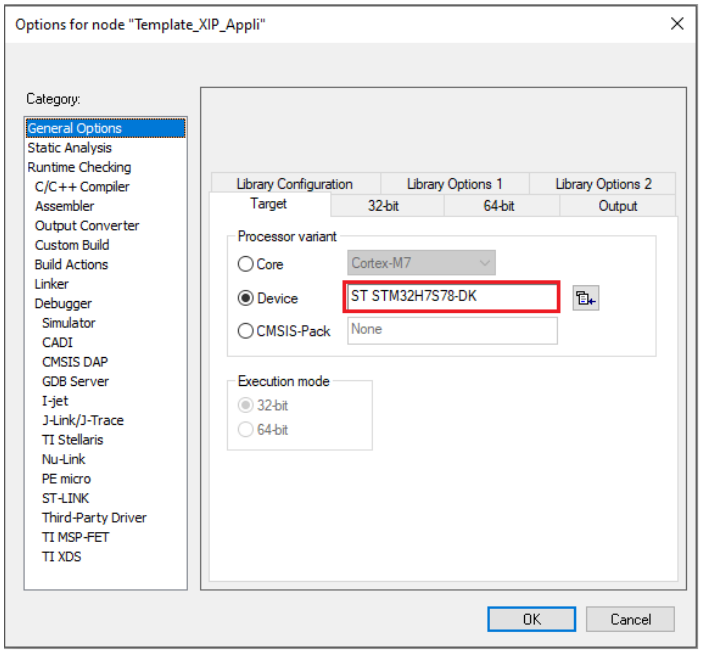
When working on a supported hardware board, the external memory loaders for supported memories are provided or supported natively by the IDE.

External memory loader selection in IAR Embedded Workbench®

In IAR Embedded Workbench® (EWARM), two solutions exist to select the native external memory loaders for the STM32H7S78-DK or NUCLEO-H7S3L8 boards. The user can either replace the selected device by the board device depending on the selected board.

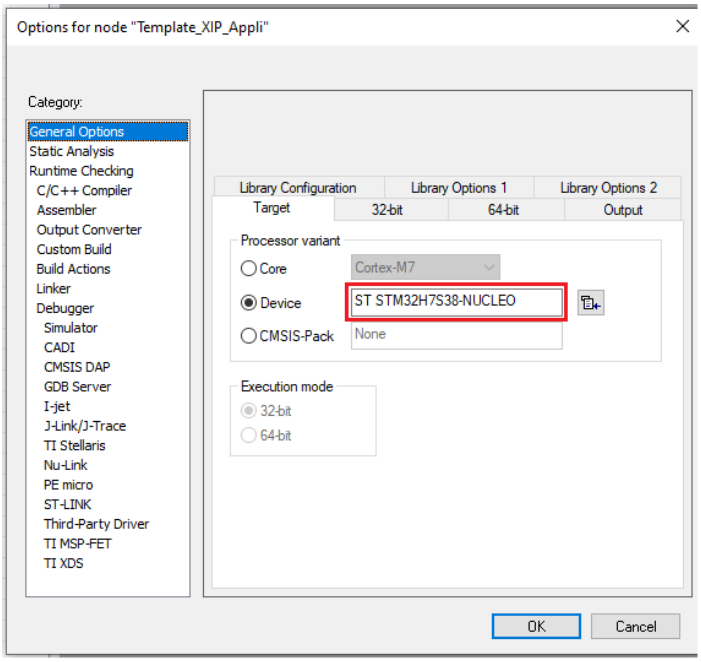
Go to [**General Options**]>[**Target**]>[**Device**], as shown below.

Figure 10. IAR: device selection for the appli subproject (STM32H7S78-DK)



DT73681V1

Figure 11. IAR: device selection for the appli subproject (NUCLEO-H7S3L8)

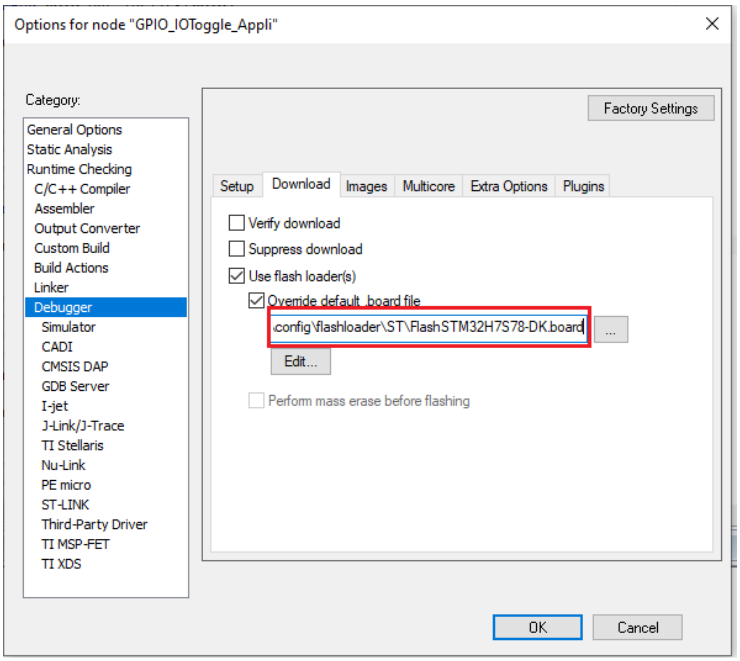


DT73682V1

Or they can directly override and select the proper external flash loader in the [**Debugger**]>[**Dowload**]:

- `FlashSTM32H7S38-NUCLEO.board` for NUCLEO-H7S3L8
- `FlashSTM32H7S78-DK.board` for STM32H7S78-DK

Figure 12. **IAR: device selection for the appli subproject via Debugger options**



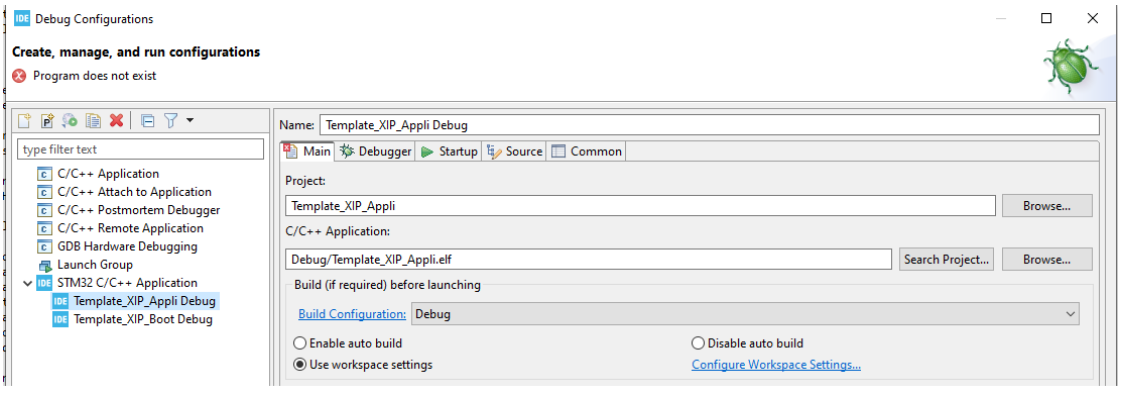
DT73683V1

External memory loader selection in STM32CubeIDE

STM32CubeIDE associates the use of the external loader with the debug configuration. The user must therefore create a debug configuration for his application and specify the loader corresponding to the STM32 board, as detailed in the figures below.

1. First, go to [**Run**]>[**Debug Configuration**] and double-click on [**STM32 C/C++ Application**].
2. Select the subproject and choose a name for the debug configuration.

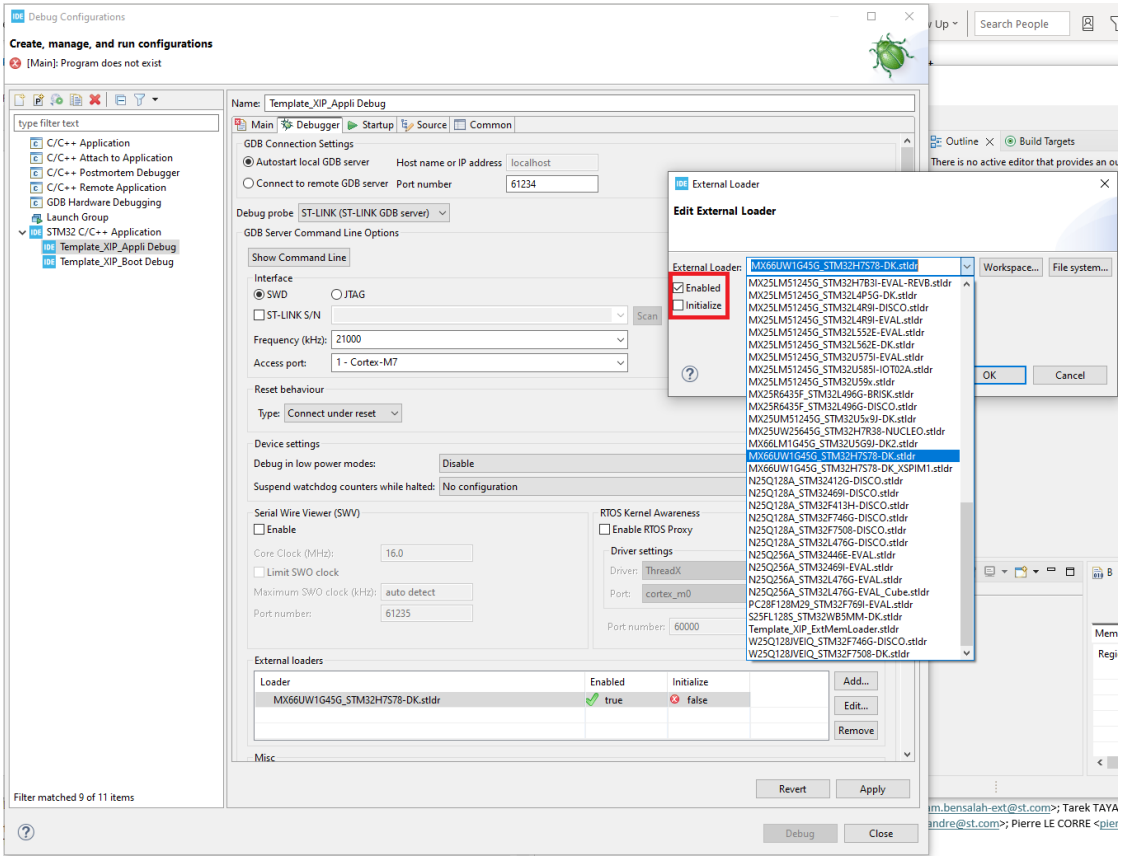
Figure 13. **STM32CubeIDE: selecting the subproject and naming the debug configuration**



DT73684V1

3. Select the external loader: [**Debugger**]>[**External Loader**]>[**Add**]

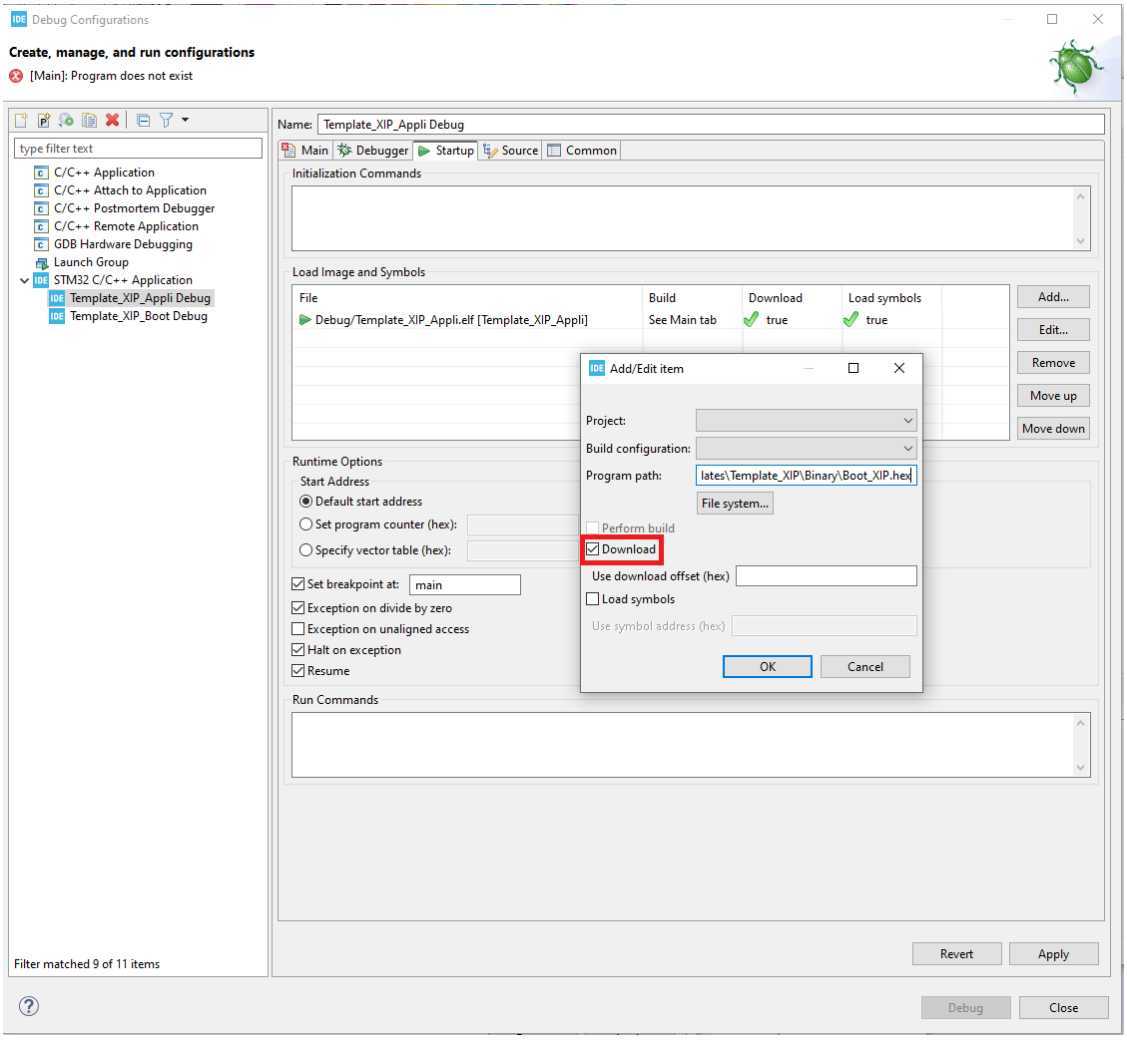
Figure 14. STM32CubeIDE: selecting the external loader



4. The user can select [**Enabled**] in the popup window. The [**Initialize**] checkbox must remain unchecked.

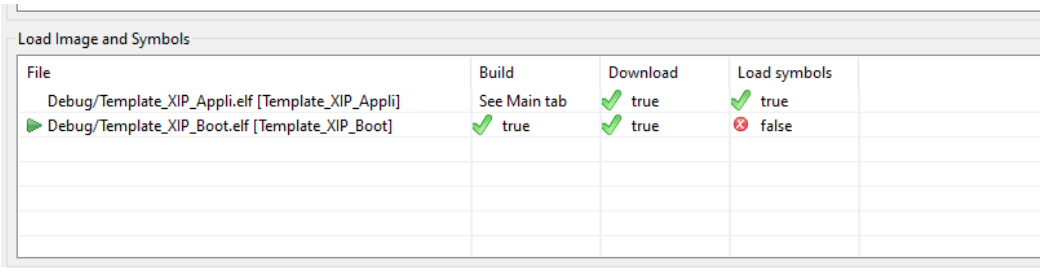
5. Inside the Startup window:
   – In the Load Image and Symbols window, click on the [**Add**] button.
   – Using the [**File system**] button, force the selection of the hex file of the template_XIP (corresponding to your board).
   – Select only the [**download**] option and uncheck [**load symbols**].

Figure 15. STM32CubeIDE: loading images



6. Change the order with the [**Move up**] and [**Move down**] buttons to put "template_XIP_Boot" in the second position.

Figure 16. STM32CubeIDE: change image order



7. Apply the configuration and launch the debug.

5.2.3.3 *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

5.3 Getting STM32CubeH7RS release updates

The new STM32CubeH7RS MCU Package releases and patches are available from www.st.com/stm32h7rs. 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 FAQ

6.1 What is the license scheme for the STM32CubeH7RS MCU Package?

The HAL is distributed under a non-restrictive BSD (Berkeley Software Distribution) license.

The middleware stacks made by STMicroelectronics (for example, the USB Host and Device libraries) come with a licensing model allowing easy reuse, provided it runs on an STMicroelectronics device. The middleware based on well-known open-source solutions (FreeRTOS™, FatFS, LwIP) have user-friendly license terms. For more details, refer to the license agreement of each middleware component.

6.2 What boards are supported by the STM32CubeH7RS MCU Package?

The STM32CubeH7RS MCU Package provides BSP drivers and ready-to-use examples for the following STM32H7Rx/7Sx boards:

- STM32H7S78-DK
- NUCLEO-H7S3L8

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

Yes. STM32CubeH7RS provides a rich set of examples and applications. They come with the preconfigured projects for IAR Embedded Workbench®, Keil®, and GCC.

6.4 Are there any links with standard peripheral libraries?

The STM32CubeH7RS 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 the hardware. A set of user-friendly APIs allows a higher abstraction level, making them easily portable from one product to another.
- The LL drivers offer low-layer registers level APIs. They are organized in a simpler and clearer way, avoiding direct register access. 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 allow easier migration from the SPL to the STM32H7Rx/7Sx LL drivers, since each SPL API has its equivalent LL API(s).

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/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 available on some products/lines only.

6.7 When should HAL drivers 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 an LL configuration file, like for HAL drivers?

There is no configuration file. The source code must include the `stm32h7rsxx_ll_ppp.h` file(s).

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. How to mix HAL and LL driver is illustrated in the "Examples_MIX" example.

6.10 Are there any LL APIs which are not available with HAL?

Yes, there are. Some Cortex® APIs have been added to `stm32h7rsxx_ll_cortex.h`, for instance for accessing the SCB or SysTick registers.

6.11 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 by the LL APIs, while HAL functions require SysTick interrupts to manage timeouts.

6.12 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 to the toolchain compiler preprocessor.

6.13 How can STM32CubeMX generate code based on embedded software?

STM32CubeMX has built-in knowledge of STM32 microcontrollers, including their peripherals and software, allowing it to provide a graphical representation to the user and generate `*.h` and `*.c` files based on the user configuration.

6.14 How to get regular updates on the latest STM32CubeH7RS MCU Package releases?

Refer to Section 5.3: Getting STM32CubeH7RS release updates.

6.15 Why put the DMA buffer in a non-cacheable area?

Since both the CPU and DMA are controllers on the AXI bus and CPU access may be cached, to avoid dealing with data coherency, a simple solution is to declare a non-cacheable area with the memory protection unit (MPU) for data exchanged by DMA. This ensures that both the CPU and DMA can access the same data content.

6.16 How to ensure the integrity of internal flash memory content?

A CRC hardware module is present in the embedded flash memory. This module allows the calculation of a CRC on a specific area using a dedicated HAL flash API. Then a comparison can be made at application level with the expected value, ensuring the integrity of the data.

6.17 What is the main difference between DCMIPP and DCMI?

Contrary to the DCMI peripheral, the DCMIPP peripheral is the controller on the bus and increases the performance for acquisition (no need to use DMA).

6.18 How to synchronize graphic operations with the display events?

The GFXTIM peripheral is a dedicated timer for graphic operations and events.

6.19 How to protect the data in the external memory in memory mapped mode?

In memory mapped mode, data content in the external memory can be protected using the MCE peripheral.

6.20 How to use different MCE algorithms on one external memory connected to the XSPI port?

The AES algorithm (available on MCE1) is used with the XSPI1 instance, whereas the Noekeon algorithm (available on MCE2) is used with the XSPI2 instance. By default, the XSPI1 instance is connected to port 1 and the XSPI2 instance to port 2. By configuring the XSPI I/O manager in swapped mode, the XSPI1 instance can be connected to port 2 and the XSPI2 instance to port 1.

Thus, with the MCE peripheral and in direct mode, a memory connected to XSPI port 1 uses the AES algorithm, whereas it uses the Noekeon algorithm in swapped mode

6.21 How to increase or decrease the size of ITCM/DTCM?

The ITCM and DTCM memories can be set to varied sizes by using some parts of the SRAM1/SRAM3 memories. This configuration is done through the flash memory user option bytes. It is recommended to do this with specific tools like STM32CubeProgrammer and not during code execution, as it may lead to unexpected behavior due to the memory mapping in use.

Linker files must be updated to reflect this mapping change. Refer to the *Embedded SRAM* chapter in RM0477.

Attention: *Modifying the ITCM size changes the SRAM1 start address.*

6.22 What does XiP mean?

XiP stands for "execution in place". This is the method of executing code directly from the external flash memory.

For each board, the STM32CubeH7RS MCU Package provides a *Template_XIP* project to demonstrate it and most of other HAL-based projects are developed for the XiP method.

6.23 What does LRUN mean?

LRUN stands for "load and run". This is the method consisting of first copying code from the external storage to the RAM (internal or external) and then executing the code from that RAM.

For each board, the STM32CubeH7RS MCU Package provides a *Template_LRUN* project to demonstrate it and this method can be reused to build applications.

6.24 Why split the code into two parts on STM32CubeH7RS ("Boot"/"Appli")?

To simplify the development of a boot flash memory application, it is recommended to split the application development into two subprojects: "Boot" and "Appli".

The boot part is responsible for configuring the system (such as the CPU clock, MPU, and external memory interfaces) and manages the startup of the application.

The application part is for the applicative code present in the external memory.

6.25 How to develop applications that are executed from the external flash memory on NUCLEO-H7S3L8H or STM32H7S78-DK boards?

There are several ways to do this:

- Develop a custom application in the `Appli` folder in the "Template_XIP" project.
- Develop a custom application in the `Appli` folder in any other project showing how to use HAL drivers.
- Use STM32CubeMX to select the board and peripherals, and configure the peripherals for the application context prior to code initialization generation and application code completion.

The user can always reuse the default `Boot_XIP.hex` image from the "Template_XIP" project as the boot stage prior to jumping to their own application code

6.26 How to develop applications that are executed from from the external flash memory on a custom board?

The easiest way to do this is to use STM32CubeMX to:

- Select the STM32H7Rx/7Sx device.
- Define the boot context with the power, clock, external memory interface peripheral, IOs, and external memory manager configuration in XiP mode.

- Generate code and validate the boot context code and memory initialization.
- Associate and configure the resources needed in the application context dedicated to the external flash memory.
- Generate the source code and complete the application code in the `Appli` folder of the project.
- Download and execute the code composed of the initial boot sequence, running from the internal flash memory, that calls the application, running from the external flash memory.

6.27 What are the external memories supported by the middleware external memory manager?

This information is available from the external memory manager release note in `Middlewares\ST\STM32_ExtMem_Manager`.

6.28 What to do in case of an error when using the SFDP-compliant external flash memory with the external memory manager?

After ensuring that the board has no hardware issues and that the different IOs are correctly configured, the user can enable the debug trace in the project (`DEBUG_TRACE`) and set debug levels in the `stm32_extmem_conf.h` file of the external memory manager middleware component of the application. This creates output logs that can be shares with STMicroeletronics for support through forums (STMicroelectronics community or github).

6.29 Can the application use the available space of the internal flash memory?

There are no restrictions on sharing the internal flash memory between the boot and the application. If the boot case is XIP, the internal memory is shared between the boot and the application. This must be configured in the scatter file. If the boot case is LRUN, this is more complex and requires putting in place the appropriate mechanisms to copy code parts to each internal flash and RAM memory.

6.30 Can the application update the CPU clock?

Yes, this is possible, but if the application runs on an external memory initialized at the boot, the clock associated with the memory interface must be preserved to avoid crashing the application.

STM32H7Rx/7Sx devices provide a protection mechanism in the external memory interface clock path to prevent this.

6.31 Can the middleware STM32_ExtMem_Manager be used for the low-level file system?

Yes, it is possible. However, currently the middleware does not provide the option to read or write using DMA mode.

6.32 Can the middleware STM32_ExtMem_Manager be used with LL drivers?

No, there is no plan to support such a case.

6.33 What is the recommendation before changing a device from an OPEN to a CLOSE state?

Use the provisioning scripts provided in the firmware package (`Projects\STM32H7S78-DK\ROT_Provisioning\DA`).

6.34 Why is an external memory loader needed?

The external memory loader is needed to program external memories.

STMicroelectronics provides a new way of building an external memory loader, based on STM32_ExtMem_Loader and STM32_ExtMem_Manager through the STM32CubeMX configuration.

The STM32CubeH7RS MCU Package provides a template project with a specific workspace named "ExtMemLoader" (`HW_Board\Templates\Template_XIP\ExtMemLoader`) for each board to demonstrate it and this method can be reused to build a loader for other memories.

6.35 Can the external memory loader be executed from a different SRAM address?

Yes, it is possible, but each case must be treated separately.

- With STM32CubeProgrammer, the load address can be modified by accessing the database in this file:
  `STM32CubeProgrammer_installation_folder/Data_Base/STM32_Prog_DB_0x485.xml`

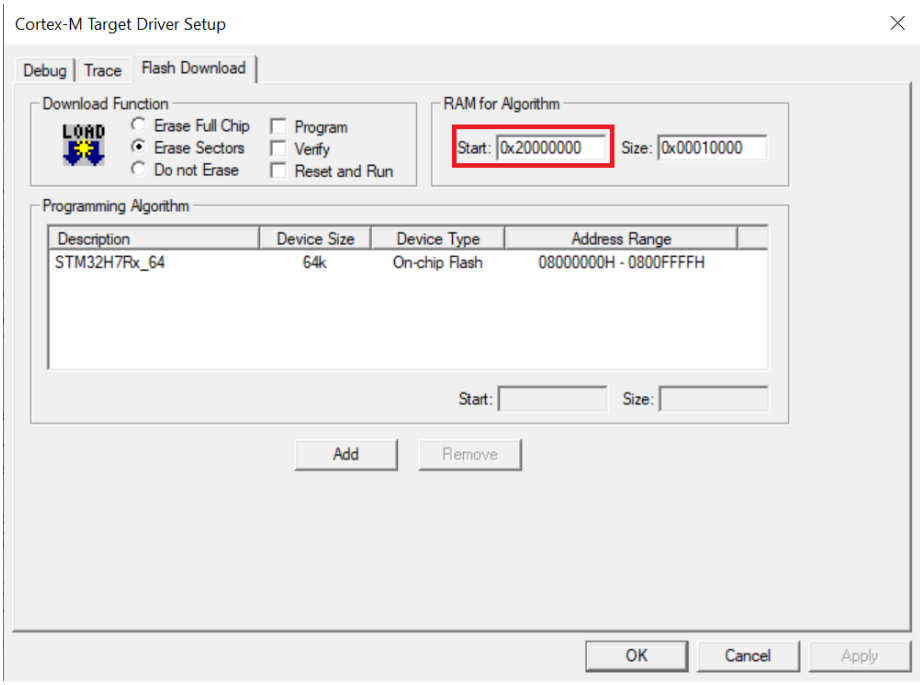
Figure 17. STM32CubeIDE: changing the SRAM loading address for the external memory loader

```
<!-- Peripherals -->
<Peripherals>
    <!-- Embedded SRAM -->
    <Peripheral>
        <Name>Embedded SRAM</Name>
        <Type>Storage</Type>
        <Description/>
        <ErasedValue>0x00</ErasedValue>
        <Access>RWE</Access>
        <!-- 1024 KB -->
        <Configuration>
            <Parameters address="0x20000000" name="SRAM" size="0x10000"/>
            <Description/>
            <Organization>Single</Organization>
            <Bank name="Bank 1">
                <Field>
                    <Parameters address="0x20000000" name="SRAM" occurrence="0x1" size="0x10000"/>
                </Field>
            </Bank>
        </Configuration>
    </Peripheral>
```

DT73686V1

- For MDK-ARM, the necessary changes can be made through the application project settings: **[Options]>[Debug]>[Settings]>[Flash Download]**.

Figure 18. MDK-ARM: changing the SRAM loading address for the external memory loader



DT73687V1

- For EWARM, it is possible to make the change directly from the linker file of the ExtMemLoader project.

Revision history

Table 5. **Document revision history**

Date	Revision	Changes
22-Feb-2024	1	Initial release.

Contents

List of tables

List of figures