Introduction to external memory manager and external memory loader middleware for boot flash MCU

Introduction

This application note details the external memory manager and loader middleware for boot flash MCUs, outlining how to establish a boot system that can launch applications from external memory. It covers supported boot modes, including execute in place (XIP) and Load and run (LRUN).

Table 1. Applicable products

Type	Product
Microcontrollers	STM32H7R3/7S3, STM32H7R7/7S7 lines

1 General information

This application note applies to STM32H7Rx/7Sx series microcontrollers that are Arm® Cortex® M7 core-based devices.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Boot flash application with the STM32 ecosystem description

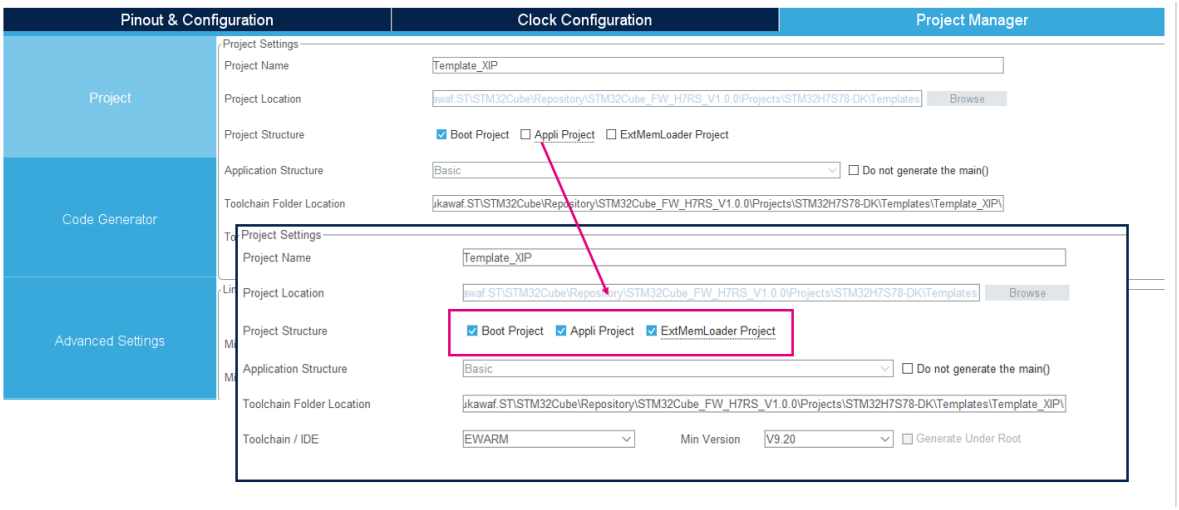
2.1 Boot flash application structure

The STM32CubeMX provides the ability to create a boot flash application, which uses the external memory manager and the external memory loader middleware to create a boot system that can launch an application stored on external memory.

The project structure of a boot flash application is divided into three different contexts:

- **Boot**: Boot code to run from the internal flash memory.
- **Application**: Application code to run from the external flash memory.
- **ExtMemLoader**: Code dedicated to generate the external loader.

Figure 1. Boot flash application structure



2.2 Summary of supported external memories and interfaces

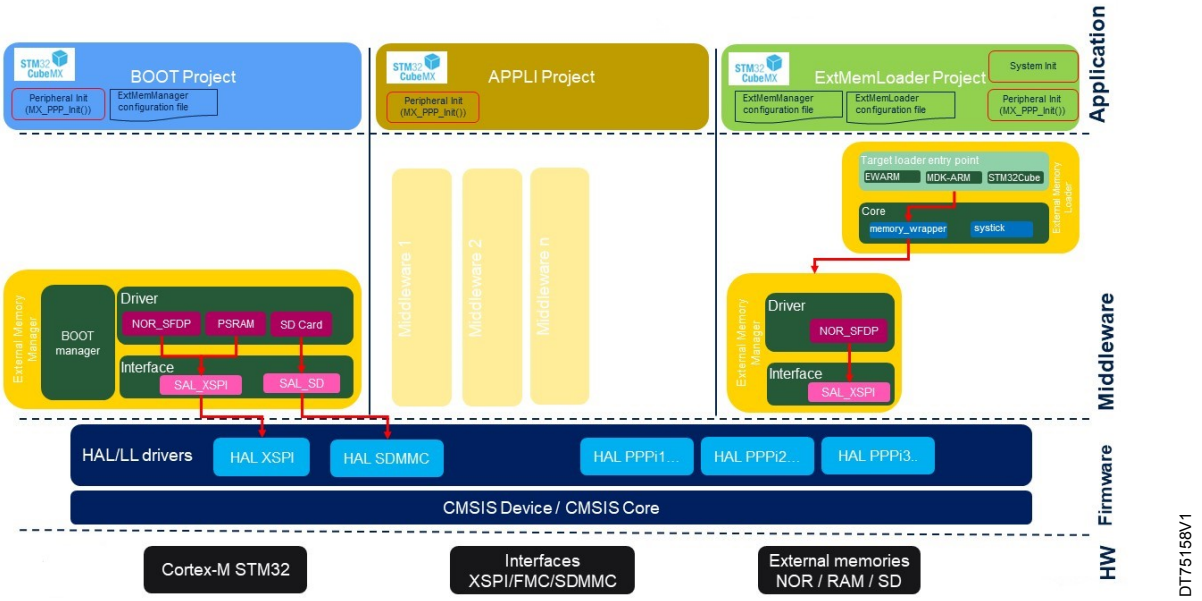
External memory manager is designed to support a variety of memory types, including:

- NOR_SFDP (Serial Flash Discovery Protocol)
- PSRAM
- SDCARD

2.3 Boot flash application architecture

The boot flash application architecture is structured into several layers, with an application layer dedicated to user applications, and a middleware layer that introduces two innovative components: the external memory manager and the external memory loader. Both of them streamline the development of boot flash applications and ensure compatibility with various platforms. Within the middleware for STM32, users benefit from a unified API that provides seamless access to different memory types, alongside a BOOT system designed to initiate applications from external memory. The firmware layer is replete with HAL/LL drivers, including XSPI and SDMMC. On the hardware front, the architecture incorporates the Cortex-M STM32 core, interfaces such as XSPI, SDMMC, and supports a range of external memory options like NOR, RAM, and SD.

Figure 2. Boot flash application architecture



Note: This application does not require middleware services because it mounts a physical address through a service called 'map mode' at boot time. Then, it executes on this memory, which contains firmware compatible with this address range.

3 External memory manager and loader middleware: an overview

3.1 External memory manager

3.1.1 Overview

The external memory manager module has been implemented to assist in the development of boot flash applications, but of can also be used on other platforms.

It is an STM32 middleware providing two services:

- A unique API to access all types of memory.
- A BOOT system to launch an application stored on an external memory.

3.1.2 Supported boot mode

The application supports the execution model Execute in place support (XIP).

- Execute in place support (XIP)
- Load and run support (LRUN)

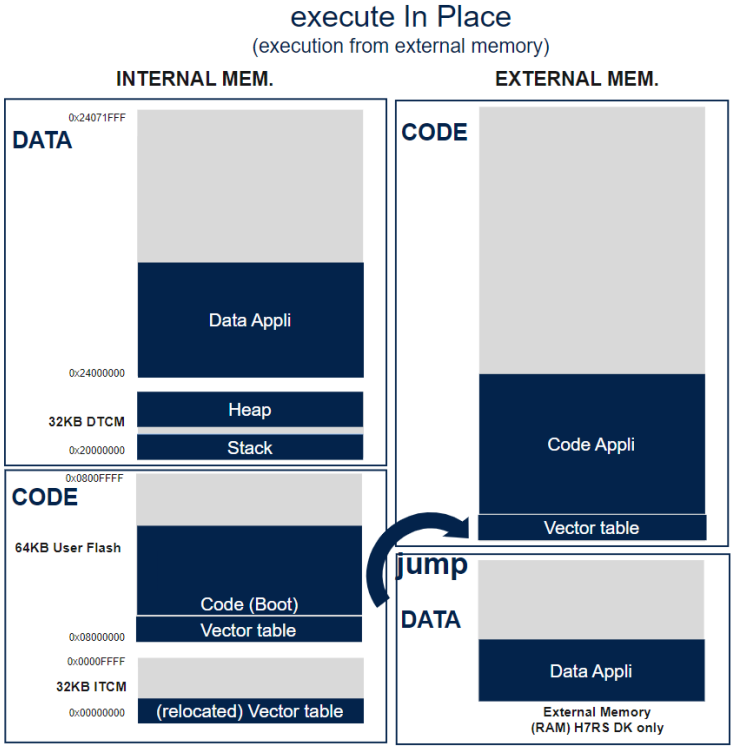
The users must select the configuration that matches their needs by choosing the appropriate mode in the external memory manager middleware within the STM32CubeMX.

3.1.2.1 Execute in place (XIP)

The XIP model is based on code execution directly from the external nonvolatile memory that is used for code storage.

This execution model requires memory-mapped support to grant the CPU with direct access to the executed-code user application.

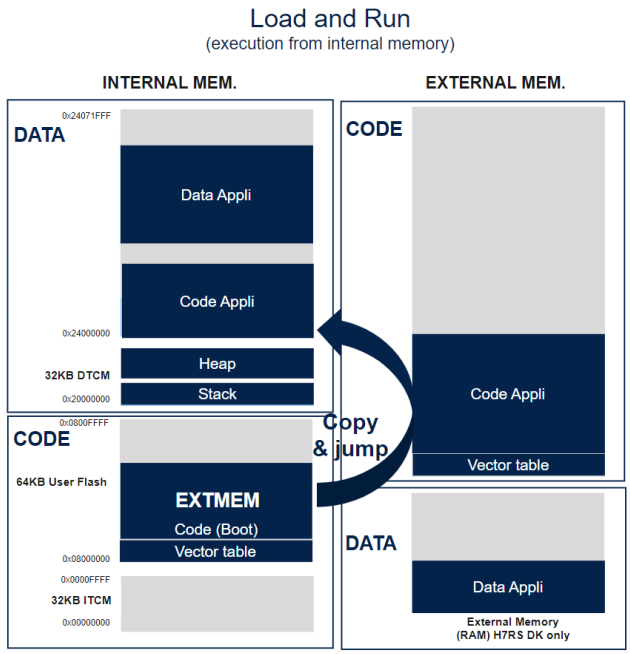
Figure 3. Execute in place



3.1.2.2 *Load and run (LRUN)*

The application is loaded from an external memory into the internal or the external memory, and runs from that memory. In this process, the application code is first loaded into RAM. Once the code is loaded into RAM, the processor can execute it.

Figure 4. Load and run process



3.1.3 External memory manager driver

The external memory manager provides a unique interface for all supported memories. It is the driver part that is responsible for adapting the request to the context of the physical memory as well as the hardware used.

Depending on the type of memory, it is possible that certain services may not make sense or may not be of interest. In this case, the APIs return an error code EXTMEM_ERROR_NOTSUPPORTED.

The table below provides an overview of various memory management functions, their purposes, and the parameters they require.

Table 2. External memory manager API function

API name	Description	Parameters
`EXTMEM_Init`	This function manages the memory initialization	• `uint32_t MemId` : Memory ID • `uint32_t ClockInput` : clock value applies on the hardware link used by the memory
`EXTMEM_DeInit`	This function manages the memory uninitialization	`uint32_t MemId` : Memory ID
`EXTMEM_Read`	This function manages the memory reading	• `uint32_t MemId` : Memory ID • `uint32_t Address` • `uint8_t* Data`

API name	Description	Parameters
		• `uint32_t Size`
`EXTMEM_Write`	This function manages the memory writing	• `uint32_t MemId` : Memory ID • `uint32_t Address` • `uint8_t* Data` • `uint32_t Size`
`EXTMEM_WriteInMappedMode`	This function manages the memory writing in mapped mode	• `uint32_t MemId` : Memory ID • `uint32_t Address` • `uint8_t* Data` • `uint32_t Size`
`EXTMEM_EraseSector`	This function manages the memory sector erase	• `uint32_t MemId` : Memory ID • `uint32_t Address` • `uint32_t Size`
`EXTMEM_EraseAll`	This function performs the complete erase of the memory	`uint32_t MemId` : Memory ID
`EXTMEM_GetInfo`	This function manages the memory uninitialization	`uint32_t MemId` : Memory ID
`EXTMEM_MemoryMappedMode`	This function enables/ disables the map mode	• `uint32_t MemId` : Memory ID • `EXTMEM_StateTypeDef State`
`EXTMEM_GetMapAddress`	This function returns the map address of the memory	• `uint32_t MemId` : Memory ID • `uint32_t *BaseAddress` : returned value containing the map address.

3.2 External memory loader

3.2.1 Overview

The external memory loader is a middleware for STM32 that helps in developing various target loader entry points. It includes three target entry point definitions for building a loader compatible with different IDEs:

- **EWARM:** Contains the target entry points definition to build a loader compatible with the IAR Embedded Workbench IDE.
- **MDK-ARM:** Contains the target entry points definition to build a loader compatible with the Keil® µVision 5 IDE.
- **STM32Cube:** Contains the target entry points definition to build a loader compatible with the STM32Cube tools (STM32CubeProgrammer and STM32CubeIDE).

The external memory loader relies on the external memory manager services to interface with the memory and IDEs entry points to perform standard operations like initialization, reading, writing, erasing, mass erasing, and memory mapping.

3.2.2 Loader definition

A "loader" is a set of functions grouped together in a binary that are individually called by a tool to perform actions on dedicated memory. The loader is loaded by the tools inside a RAM space and its execution is managed by the tool. The loader specificities are:

- **Binary:** The loader is a binary using a continuous RAM area to store code and data.
- **Loader file:** The loader file is an ELF file (contains symbol information).
- **Interrupt:** Interrupts are disabled. This is a recommendation for the tool to maintain control over the execution flow.
- **Stack pointer:** The loader does not allocate memory space to the stack.
- **RAM buffer:** There is no space allocated to store the data to be written.
- **Map mode:** This is the preferred mode for checking written data.

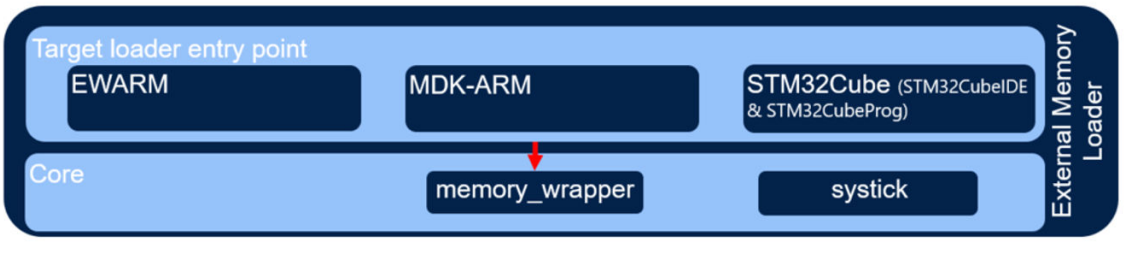
3.2.3 Architecture of external memory loader

The external memory loader is a middleware for STM32 that consists of a core component and target entry points for the loader.

The core contains two components:

- **Memory wrapper:** an abstraction layer providing basic memory services based on the usage of the middleware external memory manager.
- **SysTick management:** a SysTick management wrapper to make SysTick management functional without interrupt, the main reason for this module is to keep functional HAL management of timing.

Figure 5. Architecture of external memory loader



3.2.4 Loader mechanism in IDEs

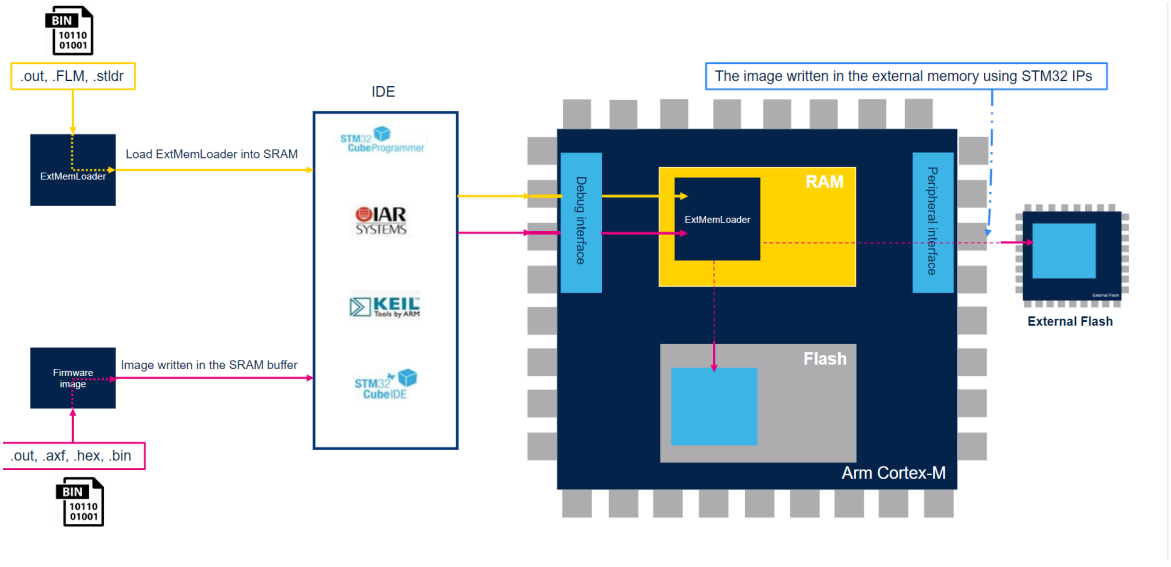
The main loader operations include initializing external memory, erasing memory by sector or by mass erase, writing buffers to external memory, and verifying the downloaded content of external memory.

To run a loader, the user needs three essential components: a tool such as an IDE (IAR, KEIL, CubeIDE) or a loader program (STM32CubeProgrammer), a binary-format loader containing code and symbol information, and an STM32 board with embedded external flash memory.

The loader tool operates in the following steps:

1. Read the loader file to retrieve function and memory information.
2. Load the code into the internal RAM of the STM32.
3. Allocate space in the RAM for RAM buffer and stack.
4. Use the loader functions to load firmware into the external memory.

Figure 6. **Loader mechanism in IDEs**



3.2.5 Target loader entry point

3.2.5.1 EWARM (IAR IDE)

IAR loader configuration file

1. **Flash memory configuration file (.flash):** is an XML file that describes the memory elements to the debugger. These include, for example, the flash memory base address, the flash loader name, and flash memory characteristics, such as block and page size. The file also specifies which flash loader to use.

Figure 7. **Flash memory configuration file**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<flash_device>
  <exe>$PROJ_DIR$/../Eloader/template_xip_Eloader/Exe/template_xip_Eloader.out</exe>
  <page>256</page>
  <block>200 0x10000</block>
  <flash_base>0x70000000</flash_base>
  <macro>$PROJ_DIR$/../../ExtMemLoader/config/ExternalFlasherVerify.mac</macro>
  <aggregate>1</aggregate>
</flash_device>
```

DT75163V1

2. **Flash memory system configuration file (.board):** is an XML file that specifies the information needed to perform flash loading for a specific board. You can prepare this file in advance for various development boards and create or modify the file in the IAR Embedded Workbench IDE.

Figure 8. **Flash memory system configuration file**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<flash_board>
  <pass>
    <range>CODE 0x70000000 0x7FFFFFFF</range>
    <loader>$PROJ_DIR$/../../ExtMemLoader/config/EXTMEM_LOADER.flash</loader>
    <rel_offset>0x70000000</rel_offset>
  </pass>
  <pass>
    <range>CODE 0x08000000 0x0800FFFF</range>
    <loader_offset> 0x20000000</loader_offset>
    <loader>$TOOLKIT_DIR$\config\flashloader\ST\FlashSTM32H7R-Sxx8.flash</loader>
  </pass>
</flash_board>
```

DT75164V1

3.2.5.2 *MDK-ARM (Keil IDE)*

MDK-ARM loader configuration file:

The generated *.FLM file needs to be added to the pack with device support, so that it is available to the tool user for programming their device. Usually, a directory flash is created and the algorithm is saved in this directory. The algorithm is specified within the /package/devices/family level:

Figure 9. MDK-ARM loader configuration file

```
<devices>
    <family Dfamily="STM32H7 Series" Dvendor="STMicroelectronics:13">
        <processor Dcore="Cortex-M7" DcoreVersion="r0p1" Dmpu="MPU" Dfpu="DP_FPU" Dendian="Little-endian" Dclock="550000000"/>
        <book name="Documentation/DUI0646B_cortex_m7_dgug.pdf" title="Cortex-M7 Generic User Guide"/>
        <description>
        </description>
        <algorithm name="CMSIS/Flash/Template_XIP_ExtMemLoader.FLM"    start="0x70000000" size="0x08000000" RAMstart="0x20000000" RAMsize="0xFFF4"  default="0" />
        <!-- ************************  Subfamily 'STM32H7R3'  ***************************** -->
        <subFamily DsubFamily="STM32H7R3">
            <debug svd="CMSIS/SVD/STM32H7RSxx.svd"/>
```

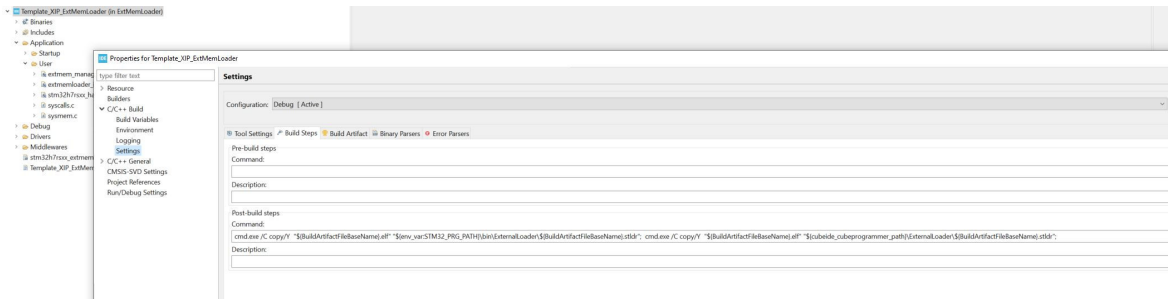
DT75165V1

- The argument start specifies the base address for the flash memory programming algorithm.
- The argument size specifies the size covered by the flash memory programming algorithm. End address = start + size - 1.
- The argument default specifies whether a flash memory programming algorithm is set as the default algorithm in a project (when true).

3.2.5.3 *STM32Cube*

The ExtMemLoader does not need configuration and is copied automatically under STM32CubeProgrammer using post build.

Figure 10. STM32CUBE loader configuration file



DT75166V1

Note: It is possible to use STM32Cubeprogrammer to create a loader for it through three different Integrated Development Environments (IDEs). It is important to note that for each IDE, the corresponding IDE should be used.

4 Running and building boot flash projects

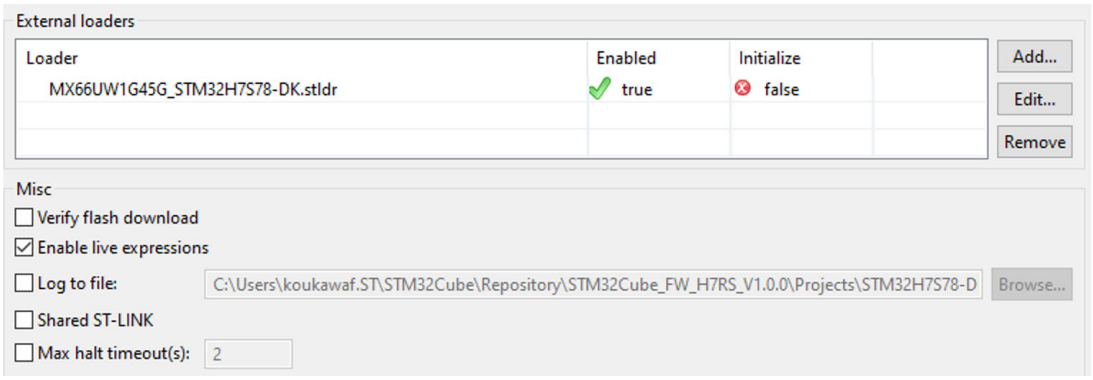
EWARM and MDK-ARM

1. Rebuild all files from subproject appli and load your images into memories. This subproject first loads Boot_XIP.hex in the internal flash memory, then load the appli part in the external memory available on NUCLEO-H7S3L8 board.
2. Run the example.

STM32CubeIDE

1. Compile the application. Ensure that the "active configuration" is set to "debug" to compile the example/ application. This is necessary because without the debug configuration, only assembly-level debugging is available.
2. Open the debug configuration. Navigate to the menu [Run] >[Debug configuration] and double-click on [STM32 C/C++ Application]. This action creates a default debug configuration for the currently selected project.
3. Configure the external loaders:
   a. Click on [Add] to include a new external loader.
   b. From the presented list, select the appropriate loader for your Board/Memory, such as **MX25UW25645G_NUCLEO-H7S3L8.stldr** or **MX66UW1G45G_STM32H7S78-DK.stldr**.
   c. Ensure that the "Enabled" option is checked and the "Initialize" option remains unchecked.
4. Miscellaneous settings: Within the same [Debugger] tab, navigate to the "Misc" section and uncheck the "Verify flash download" option.

Figure 11. External loaders parameters



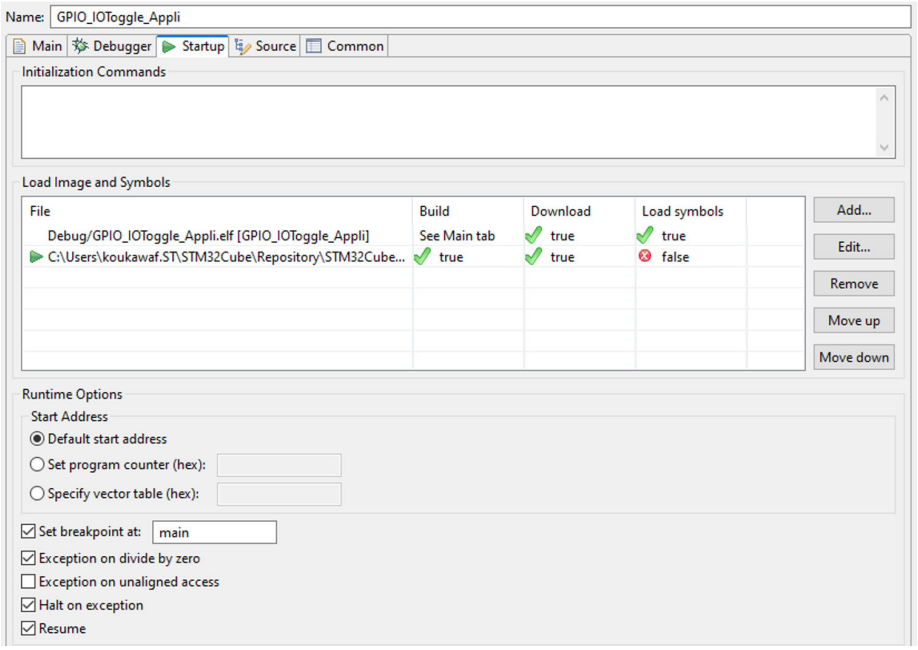
DT75167V1

5. Load Image and Symbols:
   a. Switch to the [Startup] tab and locate the "Load Image and Symbols" section.
   b. Click on "Add" to proceed with the following steps based on your project type:

- **If your project contains a boot project:**
  1. Click on "Project" and select the boot project.
  2. Click on "Build configuration" and select "Use active".
  3. Ensure the following options are configured:
     - "Perform build" checked.
     - "Download" checked.
     - "Load symbols" unchecked.

- **If your project does not contain a boot project:**
  1. Click on [File System] and select the Boot HEX file corresponding to your board. The Boot_XIP.hex is typically located in the [Binary] folder of each Template_XIP project.
  2. To select a .hex file, you may need to type "*" and press the "Enter" key in the file name dialog.
  3. Configure the following options:
     - "Download" checked.
     - "Load symbols" unchecked.
  4. Click Ok to confirm.
  5. Back in the [Startup] tab, adjust the order so that the boot project is in the second position.

Figure 12. Startup settings



DT75168V1

5 Boot flash application configuration

This section details how to use the two middleware components: the external memory manager and the external memory loader. It also provides step-by-step examples for implementation to build a boot flash application, including a use case example for STM32H7Rx/7Sx devices.

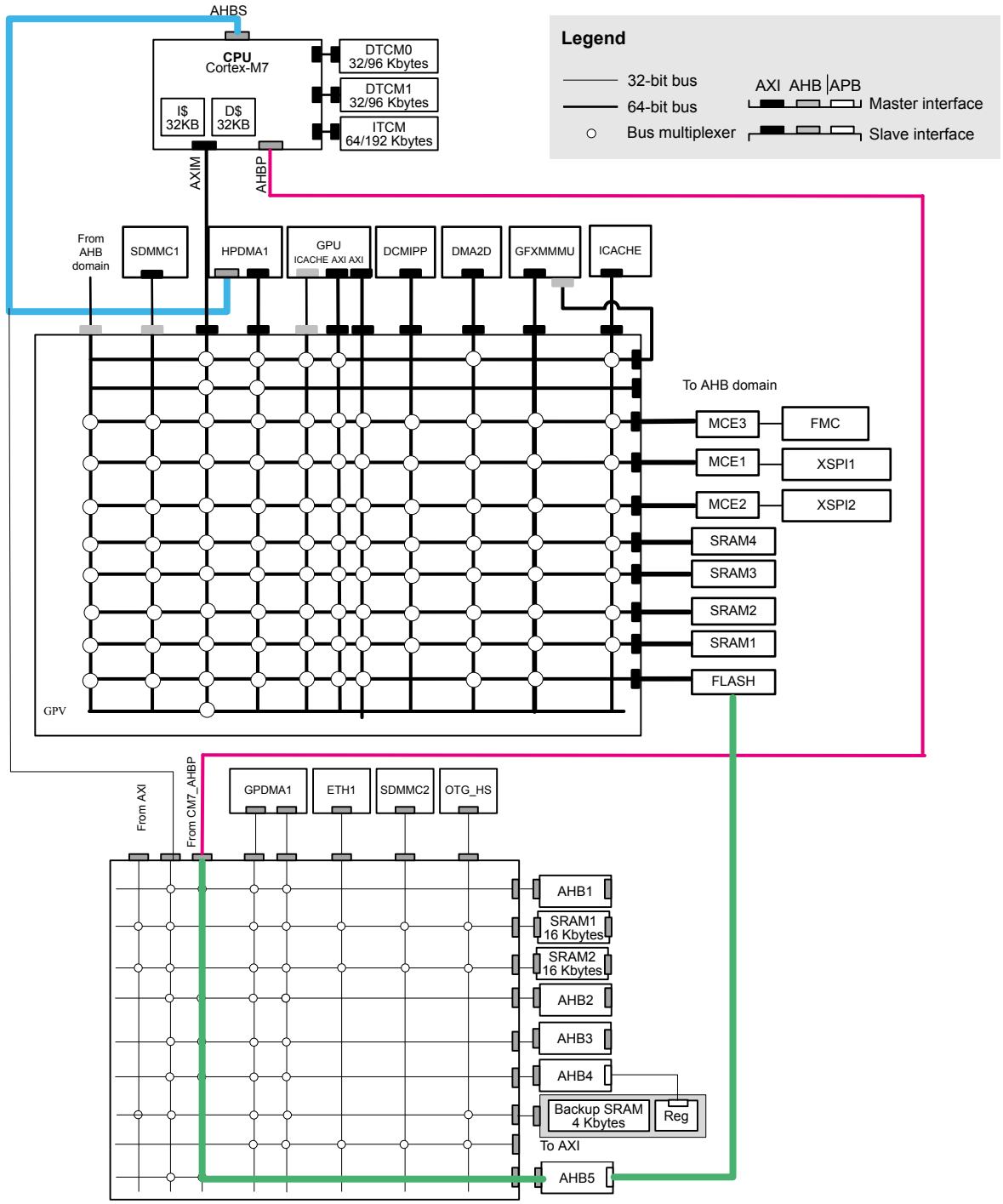
5.1 STM32H7Rx/7Sx system architecture

The STM32H7Rx/7Sx devices embed two bus matrices: one 64-bit AXI matrix and one 32-bit AHB matrix. This configuration allows efficient and simultaneous operation of high-speed peripherals,and removes bus congestion when several masters are simultaneously active (different masters located in separated bus matrices). Each bus matrix ensures and arbitrates concurrent accesses from multiple masters to multiple slaves. This allows efficient simultaneous operation of high-speed peripherals. The arbitration uses a round-robin algorithm. The maximum bus matrix frequency is half the maximum CPU frequency.

Only the Cortex-M7, the ITCM-RAM, and the DTCM-RAM can run at the CPU frequency. All bus matrices are connected by means of inter-domain buses to enable a master located in a given matrix to access a slave located in another matrix.

Figure 13 shows the overall system architecture of the STM32H7Rx/7Sx with the connections of the interconnect matrix.

Figure 13. **STM32H7Rx/7Sx system architecture**



5.2 Boot flash application based on STM32CubeMX

5.2.1 Hardware description

The STM32H7S78-DK Discovery kit is a complete demonstration and development platform for the Arm Cortex-M7 core-based STM32H7S7L8 microcontroller. The STM32H7S78-DK Discovery kit includes a full range of hardware features that helps the user to evaluate many peripherals, such as:

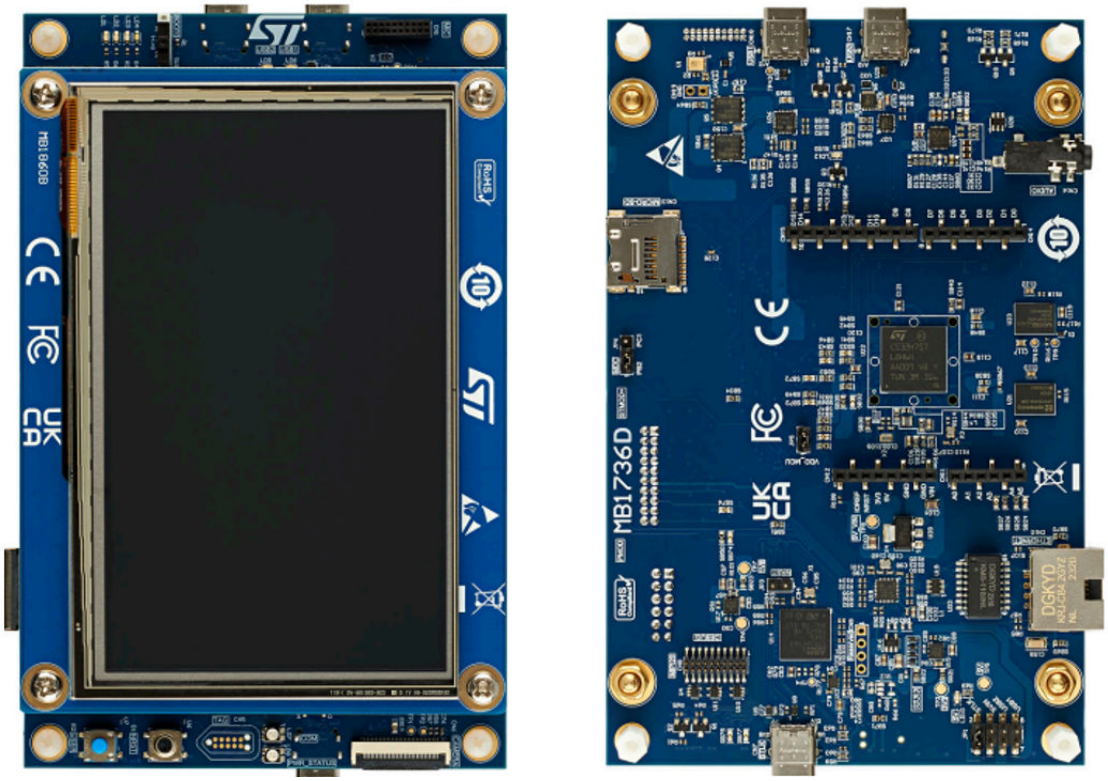
- USB Type-C®
- OCTOSPI flash memory and hexadeca-SPI PSRAM devices
- Audio codec
- Digital microphones
- ADC
- Flexible extension connectors
- User button
- Four flexible extension connectors that allow easy and unlimited expansion capabilities for specific applications such as wireless connectivity, analog applications, and sensors

The STM32H7S7L8 microcontroller features:

- Three I2C buses
- Six SPI ports
- Three USART ports
- Two SDMMC ports
- Two CAN ports
- Ethernet port
- Two SAI ports
- Two 12-bit ADCs
- Embedded step-down converter
- Two OCTOSPI memory interfaces
- One hexadeca-SPI interface
- USB OTG_HS port with power delivery
- LCD-TFT controller
- Flexible memory controller (FMC)
- 8 to 14-bit DCMI interface
- JTAG
- SWD debugging support

The STM32H7S78-DK Discovery kit integrates an STLINK-V3EC embedded in-circuit debugger and programmer for the STM32 MCU, with a USB Virtual COM port bridge and comes with the comprehensive MCU Package.

Figure 14. **STM32H7S78-DK top and bottom view**

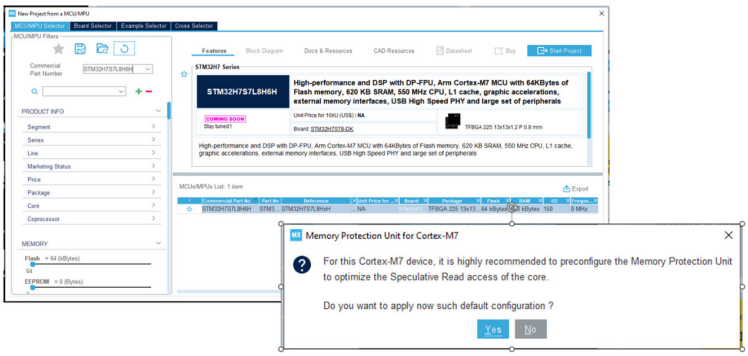


DT75169V1

5.2.2 Configuration of boot flash example

This section shows example of basic boot flash project configurations based on the STM32H7S78-DK Discovery kit. Refer to Figure 15.

Figure 15. **STM32H7S78 MCU select**

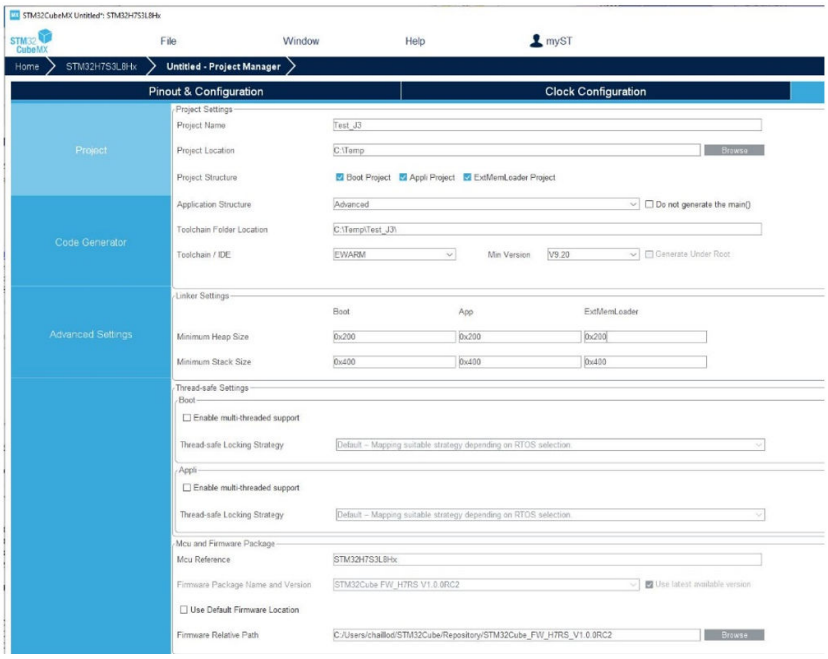


DT75171V1

When beginning with STM32CubeMX, the second step involves configuring the project structure and selecting the corresponding toolchain or IDE via the Project Manager menu, after choosing the MCU.

To establish the project structure in the Project Manager, ensure that you name the project and confirm the selection of all three subprojects, as shown in Figure 16.

Figure 16. **Project selection**



DT75172V1

5.2.2.1 *Boot subproject settings*

The boot flash boot subproject is mainly built using the external memory manager middleware and this project is responsible for three operations:

1. Set the clock of the system.
2. Configure the peripheral interface to the external memory.
3. Configure the external memory manager middleware.

Note: The part of the application being executed following the BOOT inherits the configuration programmed by the boot subproject. In order to gain efficiency, this template considers the boot as a part of the startup of the application. The configurations are therefore 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.

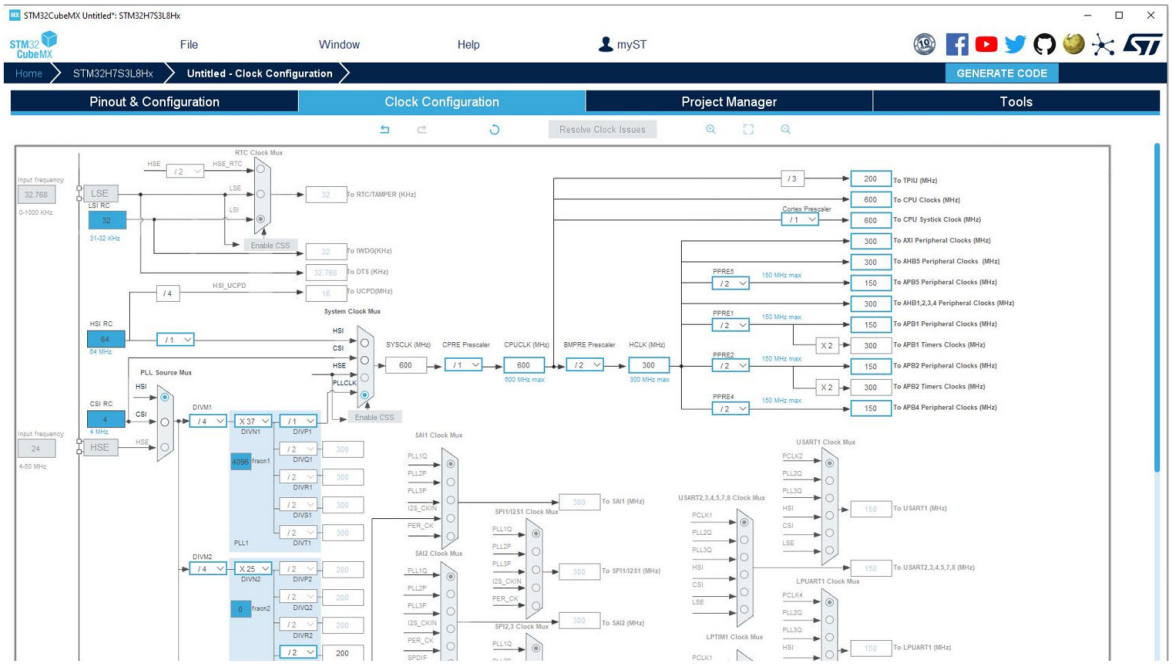
5.2.2.1.1 STM32CubeMX - System clock configuration

The clock configuration is done as usual via STM32CubeMX interface and applies for the BOOT/FSBL and EXTMEM subprojects.

In this example the system clock is configured as follow:

- Use of internal HSI clock, where the main PLL is used as system source clock
- HCLK at 300 MHz

Figure 17. Clock configuration



5.2.2.1.2 STM32CubeMX - Cortex-M7 configuration

In the system core settings, check the Cortex-M7 configuration in the boot and application subprojects to define two regions within the MPU settings refer to Figure 18:

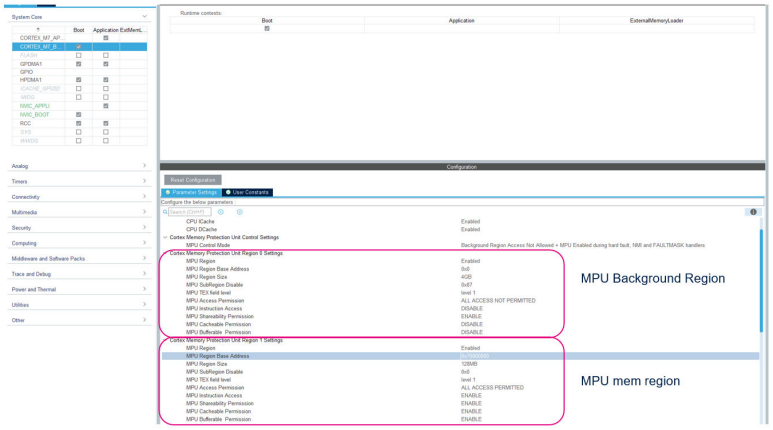
- **MPU background region**: Configure a background region to set default attributes for all regions and to prevent speculative accesses on the normal memory type.
- **MPU memory region**: Allocate a region of RAM to store a noncacheable buffer.

Enable the following options in the parameter settings:

1. Enable the Instruction Cache (ICACHE).
2. Enable the Data Cache (DCACHE).
3. Enable the Memory Protection Unit (MPU) by selecting "Background Region Access Not Allowed + MPU Enabled during hard fault, NMI, and FAULTMASK handlers."

Note: The configuration of the MPU is inherited if it was set during the BOOT context. However, it is recommended to perform a new configuration of the MPU aligned with the application needs, and to define a region for each external memory present in the application. By default, all external memories are disabled. Then, only the access to memories which are really there and will be used are enabled.

Figure 18. MPU configuration

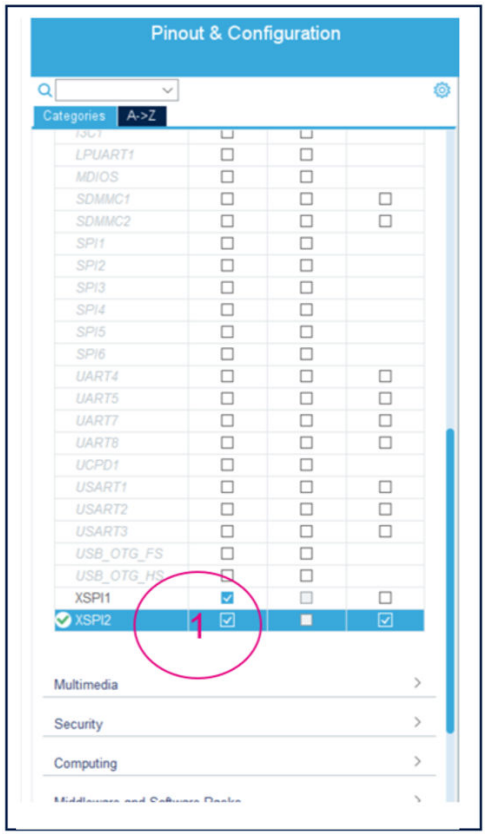


5.2.2.1.3 STM32CubeMX - the serial NOR SFDP memory interface configuration

To configure the external memory peripheral, select on the peripheral connected to the external memory and choose the boot context.

In this example, select the XSPI2 instance in the connectivity tab, which corresponds to the interface for the serial NOR SFDP memory on the STM32H7Rx/7Sx discovery board (MB1736-HS7S78-DK).

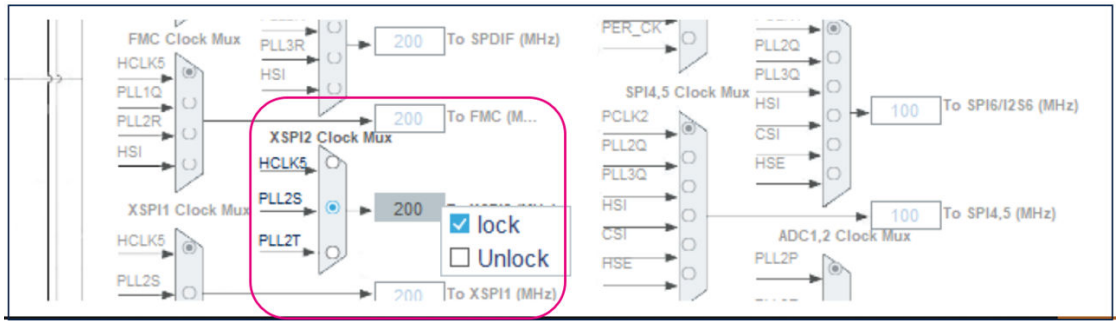
Figure 19. STM32CubeMX - XSPI Instance selection



DT74350V1

- XSPI clock source configuration: In the clock configuration tab, select the PLL2S clock source to get 200 MHz as shown in Figure 20.

Figure 20. STM32CubeMX – XSPI clock configuration



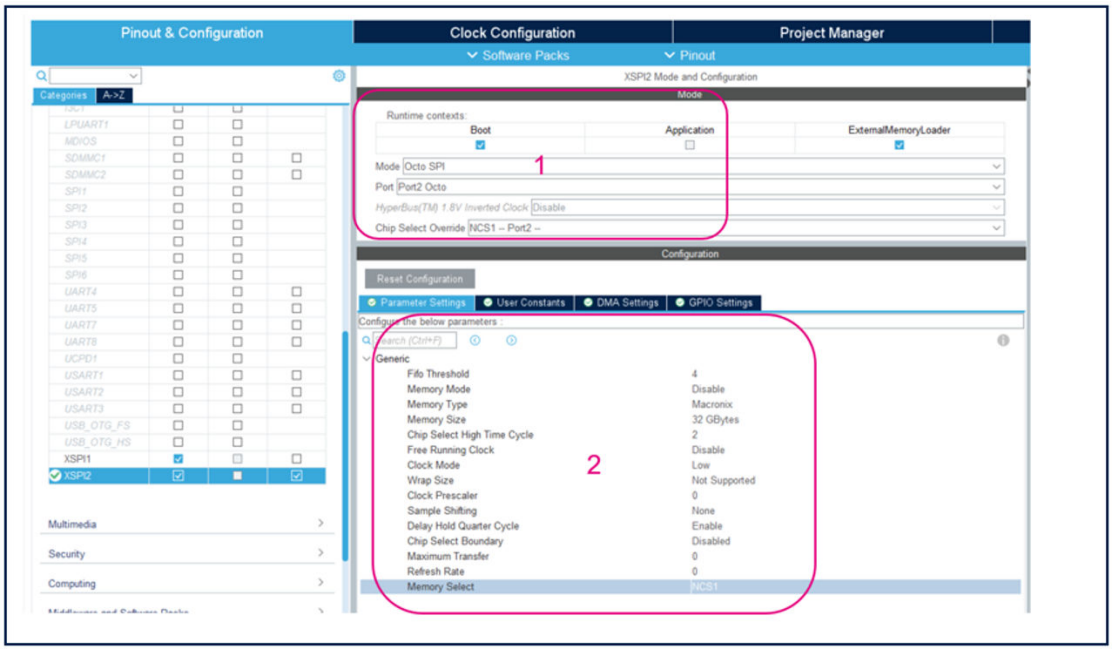
DT74351V1

Once the XSPI2 clock configuration is done, the user must configure the XSPI instance depending on the external memory used.

- In the runtime contexts tab, select octo-SPI mode, and then select the port and the chip select.

- In the parameter settings tab, set the parameters of the XSPI2 instance as shown below in Figure 21 to connect to the NOR SFDP memory.

Figure 21. STM32CubeMX – XSPI configuration



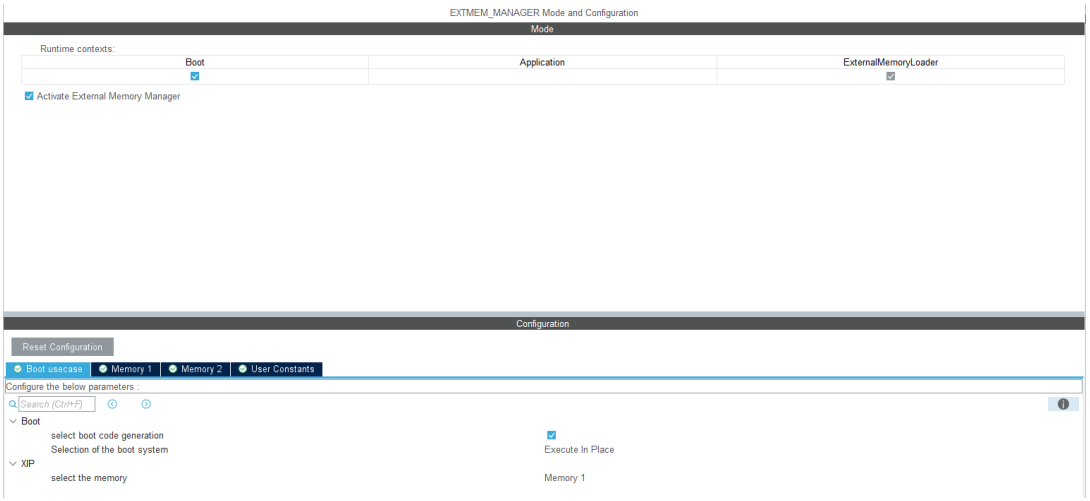
5.2.2.1.4 STM32CubeMX - EXTMEM_MANAGER middleware configuration

The user must configure the middleware EXTMEM_MANAGER that provides solutions to simplify the use of the external memory.

In the middleware and software packs section, select the EXTMEM_MANAGER middleware within the boot subproject and activate it.

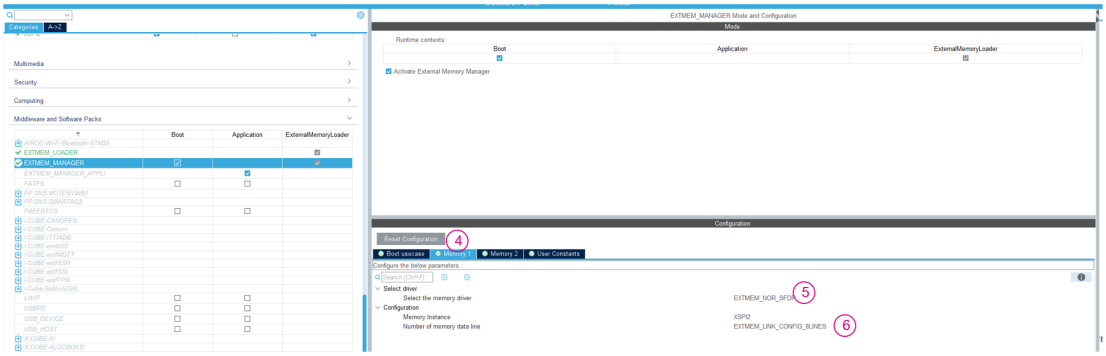
- In the boot use case tab, configure the number of memories to 1, and configure the boot use case to execute in place (XIP) to run application code directly from NOR SFDP memory.

Figure 22. External memory manager activation



- In the Memory 1 tab (number 4 in Figure 23), options to configure the memory settings are available.

Figure 23. Memory boot system selection



DT75180V1

- Select the Driver Within the Memory 1 tab, locate the section labeled "Select driver" (number 5 in Figure 23). Here, choose the driver that enables SDR_SFDP support. In this example, the driver selected is "EXTMEM_NOR_SFDP".
- Configure Interface After selecting the driver, move to the "Configuration" section (number 6 in Figure 23). Enable the XSPI interface by selecting "XSPI2".
- Configure the number of memory data lines to eight lines by selecting "EXTMEM_LINK_CONFIG_8LINES".
- Enable IO HSLV.

Figure 24. Enable HSLV

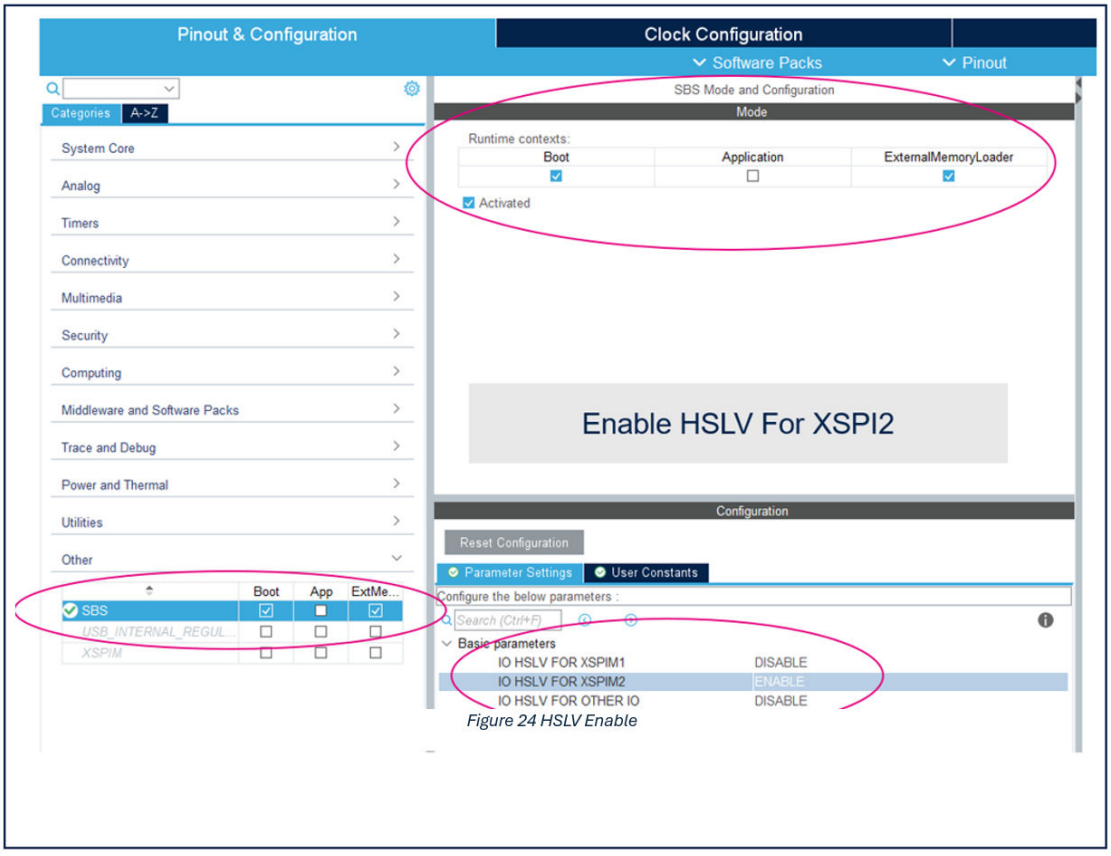


Figure 24 HSLV Enable

DT74355V1

Note: The HSLV (High Speed Low Voltage) feature allows certain I/Os to increase their maximum speed at low-voltage. Enabling the SBS IO HSLV has no effect if the option bytes are not correctly set.

The I/O HSLV configuration bit must not be set if the I/O supply (VDD) is above 2.7 V. Setting it while the voltage is higher than 2.7 V can damage the device. There is no hardware protection associated to this feature, so it is recommended to use it only as a static configuration for fixed I/O supply.

5.2.2.2 *ExtMemLoader subproject settings*

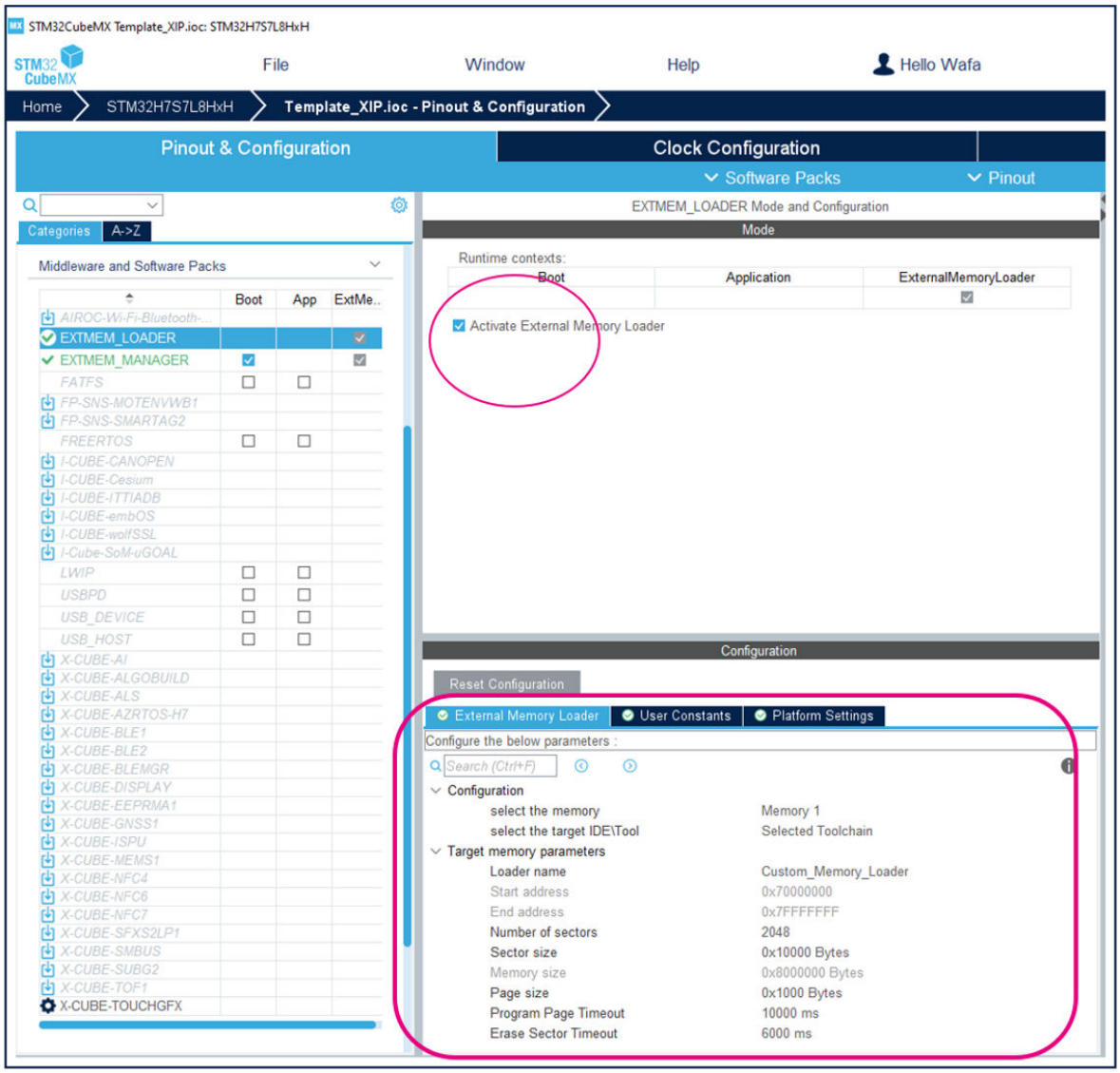
ExtMemLoader is a subproject used to generate a binary library capable of downloading an application to the external memory. This binary is called a "Loader" and can be used by the IDE or the STM32CubeProgrammer.

In the middleware and software packs section, select the EXTMEM_LOADER middleware within the ExtMemLoader subproject and activate it.

- In the EXTMEM Loader tab, configure the target memory parameters as shown in Figure 25.

Note: To define the target memory parameters, check the datasheet of the external memory used to execute your application.

Figure 25. External memory loader configuration



5.2.2.3 *Application subproject configuration*

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 made by the BOOT context.

1. The system clock: The clock is ready to use, but the application must call SystemCoreClockUpdate() to inherit the system clock value for its own HAL time base management.
2. I/D cache management: L1 caches are disabled by the boot before jumping into the application context, so the application must re-enable the I/D caches if necessary.
3. If the MPU was configured on the BOOT context, the configuration is inherited, but it is recommended to perform a new configuration of the MPU aligned with the application needs.

Find below, the code generated by STM32CubeMX for items 1 and 2. For item 3, it must be managed using user code sections as STM32CubeMX does not manage MPU configuration per context.

```
/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */
  MPU_Config();
  /* USER CODE END 1 */

  /* Enable I-Cache---------------------------------------------------------*/
  SCB_EnableICache();

  /* Enable D-Cache---------------------------------------------------------*/
  SCB_EnableDCache();

  /* MCU Configuration--------------------------------------------------------*/

  /* Update SystemCoreClock variable according to current clock configuration.  */
  SystemCoreClockUpdate();

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
-------------------------------------------------
```

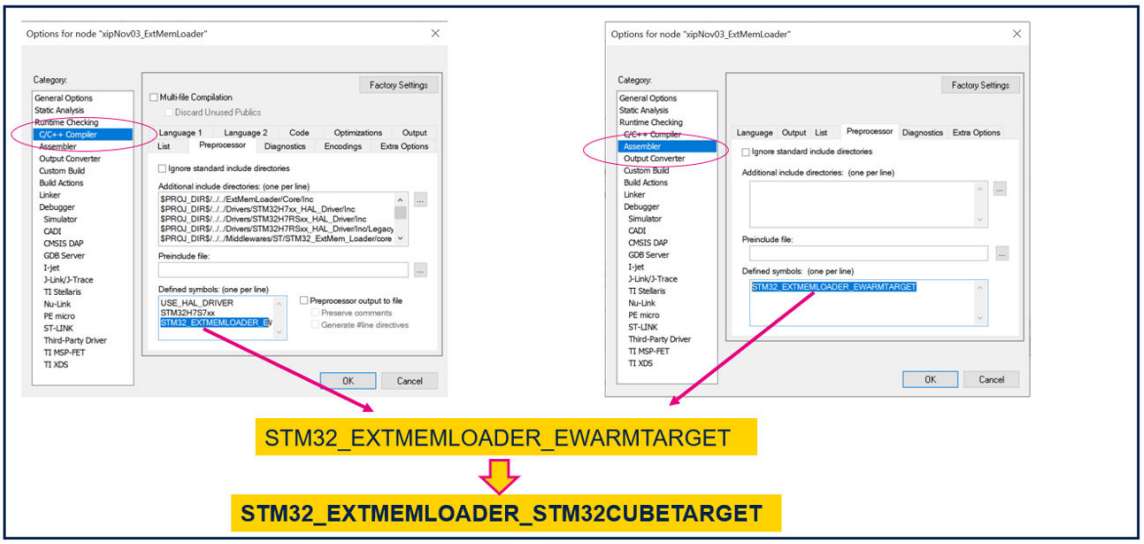
Note: The scatter file of the application depends on the memories present in the system, STM32CubeMX embeds an algorithm that selects the file according to the EXTMEM_MANAGER configuration done in the boot subproject.

5.2.3 Generating loader files with STM32CubeProgrammer

To generate the STM32CubeProgrammer loader file from the IAR ExtMemLoader project, follow the steps below:

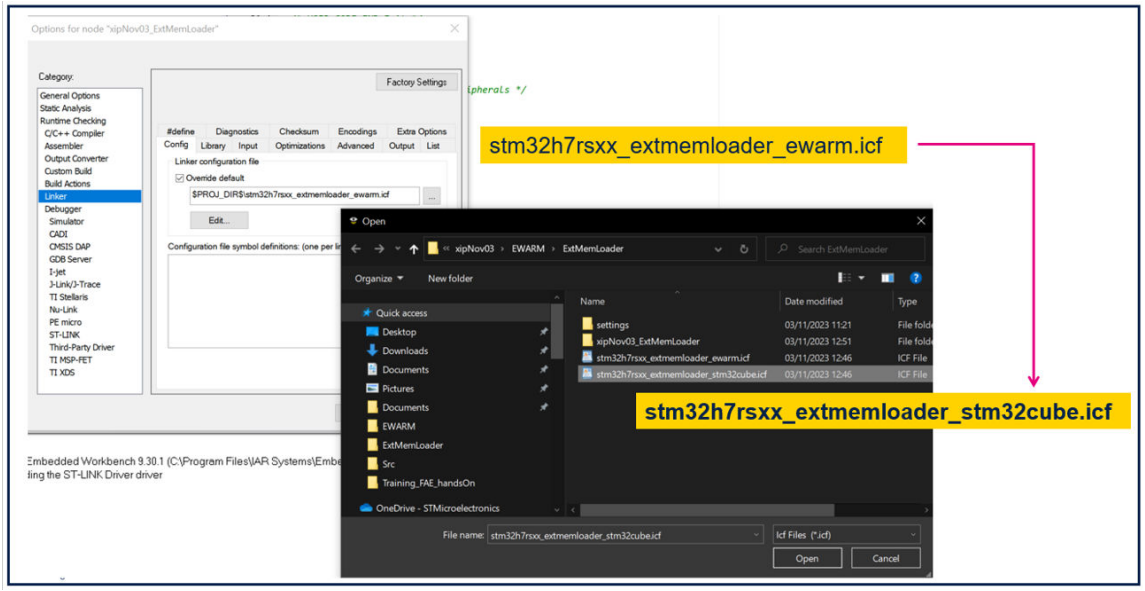
1. In the project options, rename the C/C++ and assembler preprocessor directive from `STM32_EXTMEMLOADER_EWARMTARGET` to `STM32_EXTMEMLOADER_STM32CUBETARGET` as shown in Figure 26.

Figure 26. STM32Cube/EWARM Loader



2. In the project options, select the linker category to switch from `stm32h7rsxx_extmemloader_ewarm.icf` to `stm32h7rsxx_extmemloader_stm32cube.icf`.

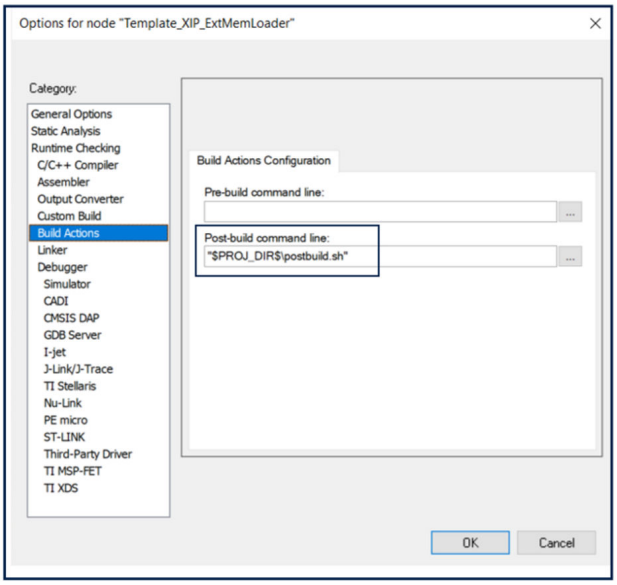
Figure 27. Linker selection



3. Add a postbuild operation to set up the generated loader within the STM32CubeProgrammer. To achieve this, include the `postbuild.sh` script in the project's build settings, which is located in the `EWARM\ExtMemLoader` directory as shown in Figure 28.

Note: The user is required to run the IDE in administrator mode.

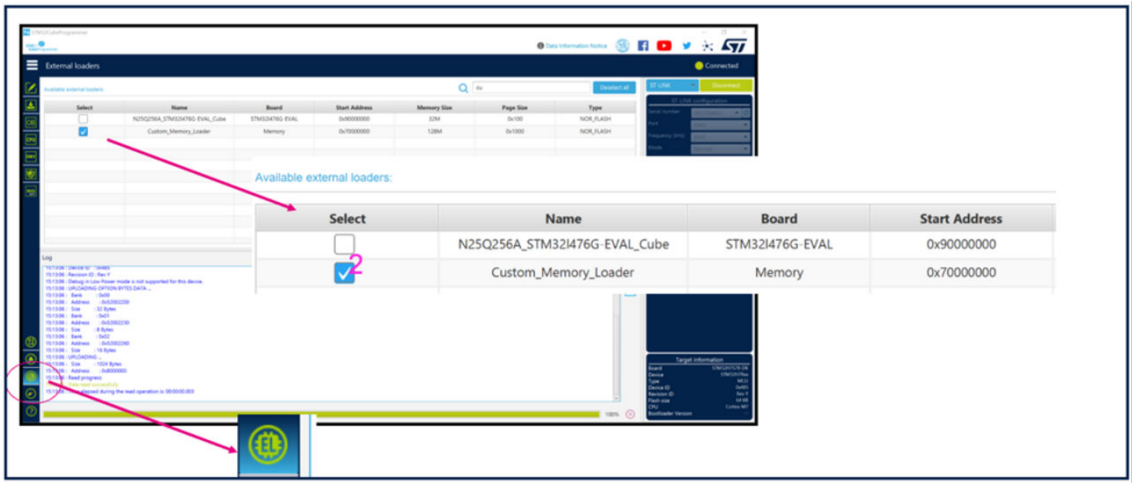
Figure 28. EWARM postbuild configuration



DT74359V1

4. Start the STM32CubeProgrammer and connect to the board. Then, choose the newly added loader from within the STM32CubeProgrammer.

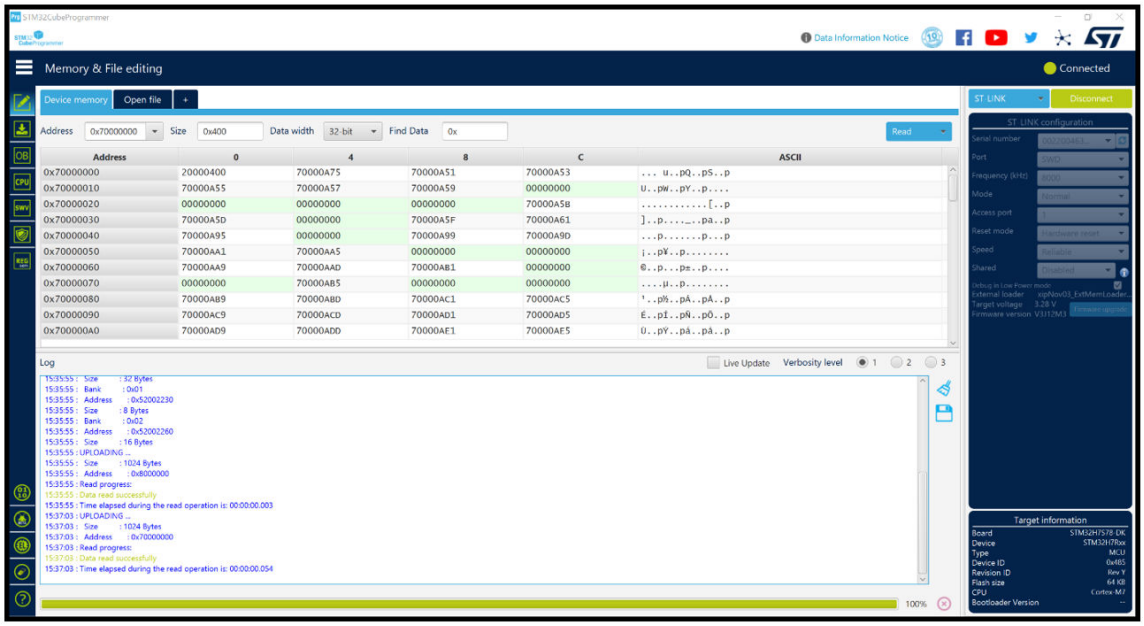
Figure 29. STM32CubeProgrammer - Custom loader selection



DT74360V1

5. Check the application code using the newly added loader by entering the external memory address in the device memory tab.

Figure 30. STM32CubeProgrammer - XSPI memory view



Revision history

Table 3. **Document revision history**

Date	Version	Changes
01-Oct-2024	1	Initial release.

Contents

List of tables

List of figures