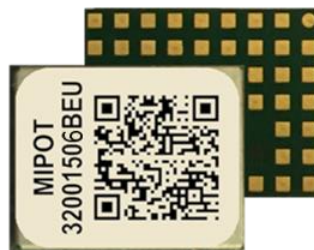# Wirelesxs Dual Core MCU Modules 32001506xxx/32001552xxx

Application note

Dual Core Programming Guide

Description

The 32001506xxx is a multi-protocol sub-GHz radio module based on STM32WL55JC asymmetric dual core arm®Cortex®-M4/Cortex®-M0+ microcontroller. It keeps arm®Cortex®-M4 core, flash and RAM memory resources and most of the internal peripherals free for the user to integrate his own application without an additional host microcontroller.

Current programming guide targets application developers. It gives an overview of the 32001506xxx module's architecture and provides information about how to create a new project or import an existing one to start developing a user application for the arm®Cortex®-M4 core. Furthermore it shows how to interact with preloaded RF radio stack fully controlled by arm®Cortex®-M0+ core from user application residing on arm®Cortex®-M4 core side.

Contents

1. Prerequisites

Several software development tools may be used to develop applications based on 32001506xxx module.

Current guide will refer to STM32CubeIDE version 1.10.1 or newer which is part of the STM32Cube ecosystem and provides a development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers.

A knowledgebase of STM32WL55JC internal architecture and basic microcontroller programming concepts are required to start developing applications with 32001506xxx module.

For detailed information about STM32WL55JC microcontroller architecture and peripherals please refer to datasheet *"DS13293 Multiprotocol LPWAN dual core 32-bit arm®Cortex®-M4/M0+ LoRa®, (G)FSK, (G)MSK, BPSK, up to 256KB flash, 64KB SRAM"* and to *"RM0453 Reference manual - STM32WL5x advanced arm®-based 32-bit MCUs with sub-GHz radio solution"*.

_____

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

STM32Cube is an STM trademark.

2. Architecture overview

The following image highlights the STM32WL55JC internal main parts involved in current guide.
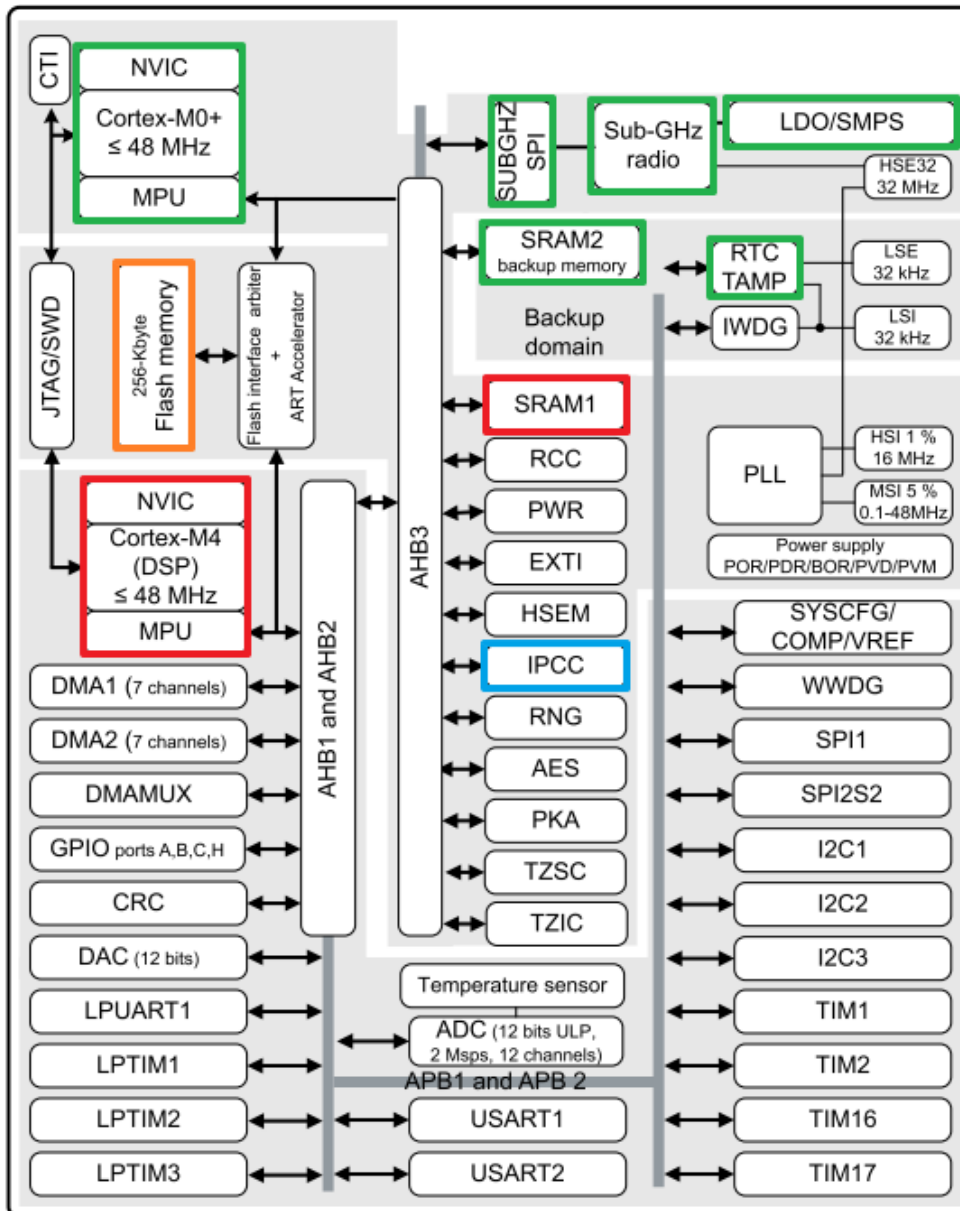
32001506xxx module's architecture requires that green bordered parts are exclusively managed by the arm®Cortex®-M0+ core and can be considered as a "black box" driven by arm®Cortex®-M4 core (red part) through IPCC (Inter-Processor Communication Controller, blue part).

Of course internal flash memory (orange part) is shared among two cores meaning that part of it is reserved for arm® Cortex®-M0+ radio application and all remaining is for user application running on arm®Cortex®-M4. Each core uses its flash memory section in an exclusive mode and is not allowed to operate outside.

3. Features

3.1. arm®Cortex®-M0+ core

The arm®Cortex®-M0+ is reserved for sub-GHz radio management and implements the radio stack required by the specific version of the 32001506xxx module. It uses the RTC for stack timing requirements and 32 kB SRAM2 block as volatile working memory; flash memory is shared with arm®Cortex®-M4 core in the terms already mentioned.

3.2. arm®Cortex®-M4 core

The arm®Cortex®-M4 is fully available for user application together with all the peripherals not used by the radio part (all except sub-GHz radio components and RTC). It uses 32 kB SRAM1 block as volatile working memory and shares flash memory with the arm®Cortex®-M0+ core.

3.3. IPCC – Inter-Processor Communication Controller

IPCC is used to perform bidirectional communication between cores. It is an ST Microcontroller proprietary inter-core communication controller. For details please refer to *"RM0453 Reference manual - STM32WL5x advanced arm®-based 32-bit MCUs with sub-GHz radio solution"*.

IPCC communication operates on a common RAM memory area shared between arm®Cortex®-M4 and arm®Cortex®-M0+. In 32001506xxx a 1 kB area is reserved starting from address 0x20008000 to address 0x200083FF. The IPCC shared memory is totally inside SRAM2 block, so it does not affect arm®Cortex®-M4 available RAM.

IPCC architecture is based on 12 communication channels, 6 of them in the direction from arm®Cortex®-M4 to arm®Cortex®-M0+ and remaining 6 in the opposite direction.

32001506xxx module uses 2 pairs of channels in half-duplex mode. The first pair, named *communication channel*, is used for main communication protocol messages, e.g. configuration, transmit or receive commands. The second pair, named *service channel*, is used for internal service messages, e.g. low power management. Each channel operates on its own defined buffer.

The communication over the communication channel between the arm®Cortex®-M4 and arm®Cortex®-M0+ cores is performed with the same communication protocol defined for the physical SPI/I^{2}C/UART channels used into 32001505xxx module and is internally managed through transmit/receive interrupts and notifications. For details please refer to *"32001505xxx_Command_Reference_revx.x.pdf"* document and to the following paragraph *"9.1. Communication messages"*.

Protocol messages are written/read to/from IPCC buffers located inside shared RAM area.

A simple arm®Cortex®-M4 application template may be provided as basic reference to implement IPCC communication with arm®Cortex®-M0+ core.

4. Creating a new STM32 project for arm®Cortex®-M4

If not already done, please download and install STM32CubeIDE development platform. It can be found at following link *https://www.st.com/en/development-tools/stm32cubeide.html* where also user manual and installing guide are available.
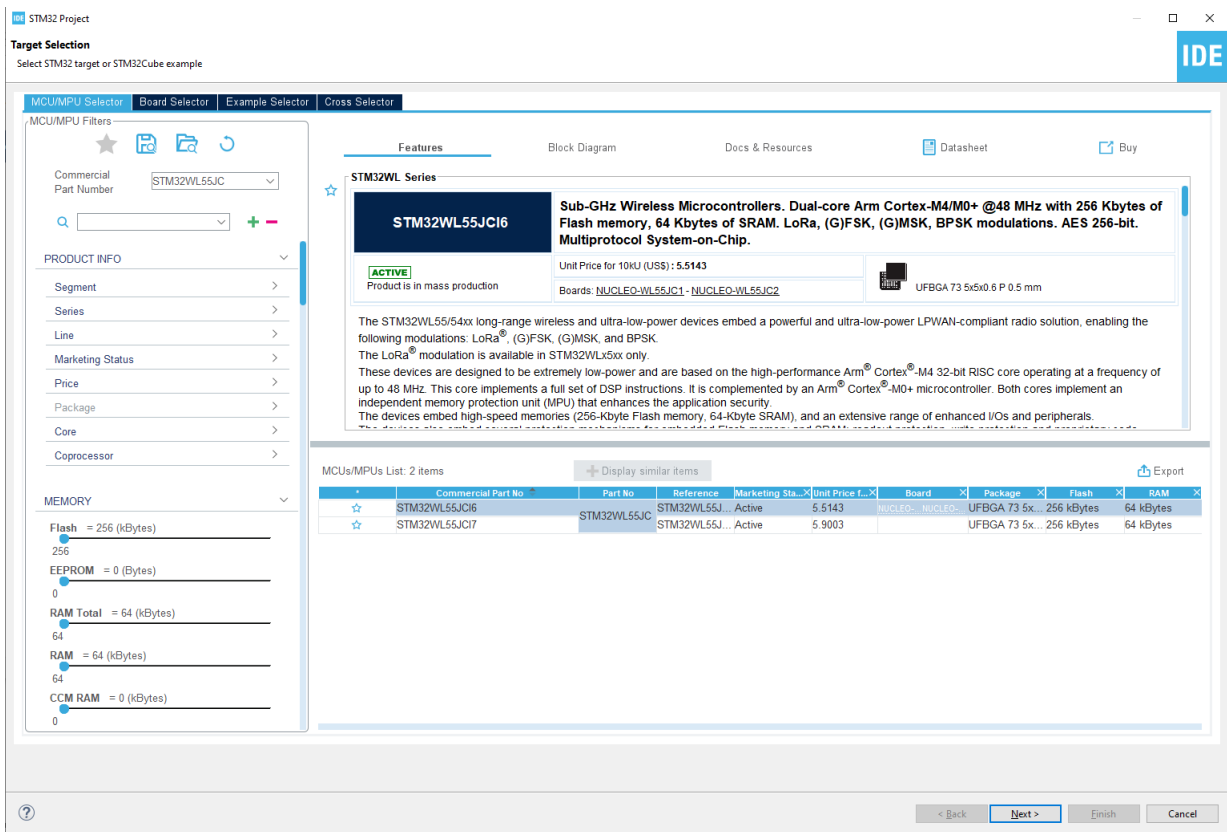
After launching STM32CubeIDE select a workspace path. It is the working folder pathname where the project will be created.
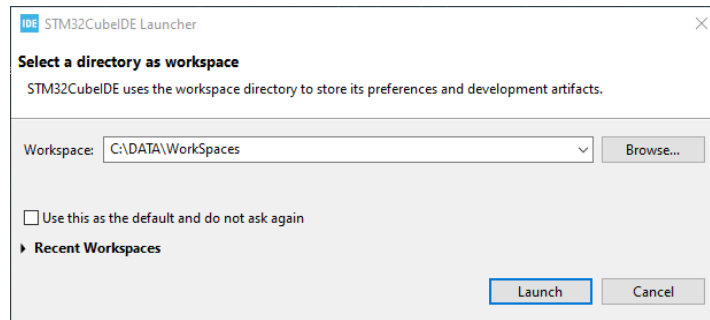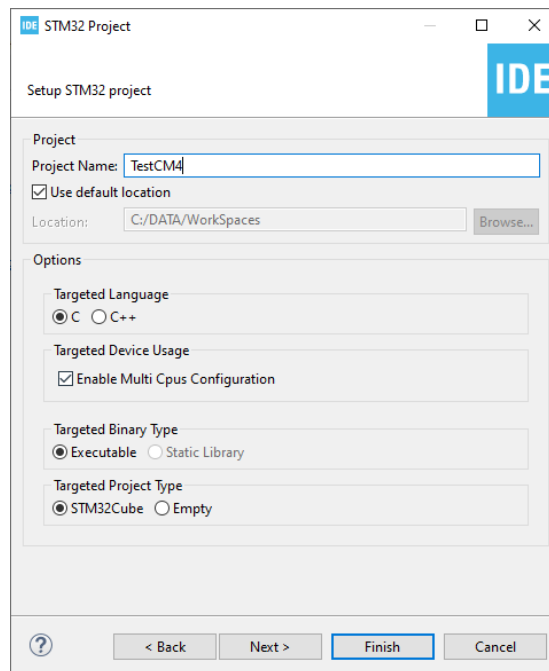

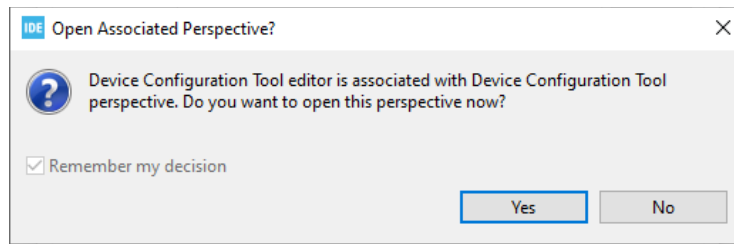To create a new project press *Start new STM32 project* button from the *Information Center* tab:
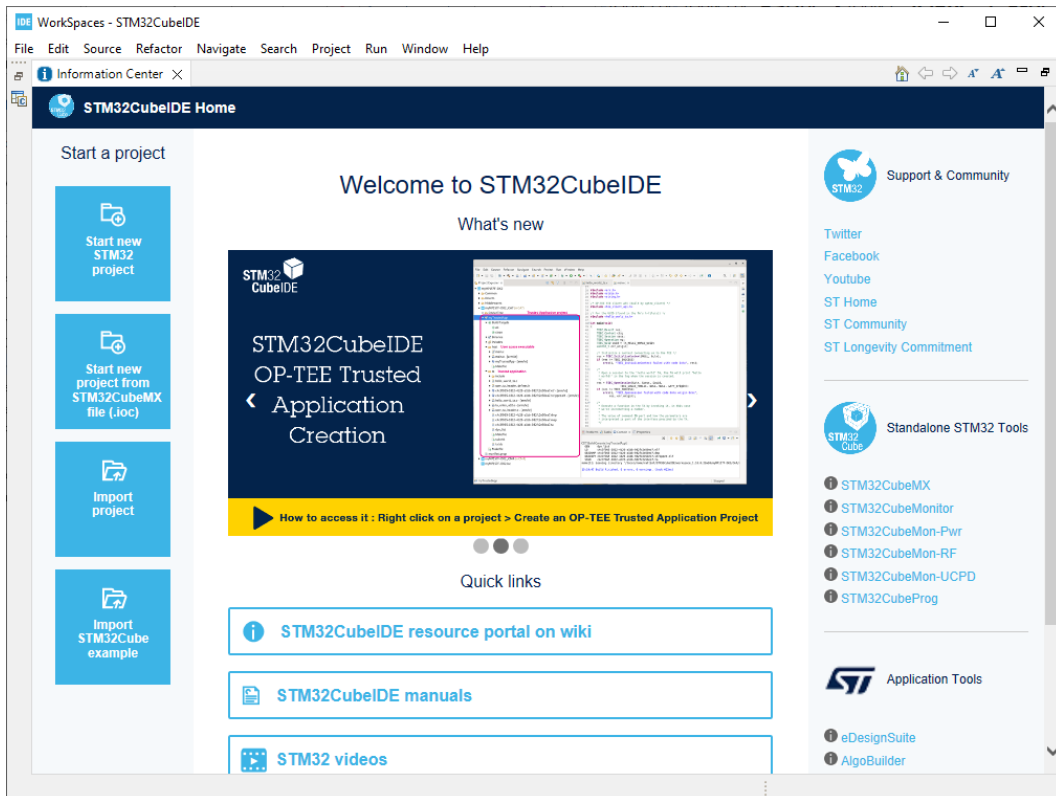
Then select the part number for which create the new project and press *Next* button:

Enter a name for the new project, let the *"Enable Multi Cpus Configuration"* flag selected and then press *Finish* button:

Press Yes button to the request *"Open Associated Perspective?"*:

The project is then created and the STM32CubeIDE opens the Device Configuration Tool to allow device hardware configuration:
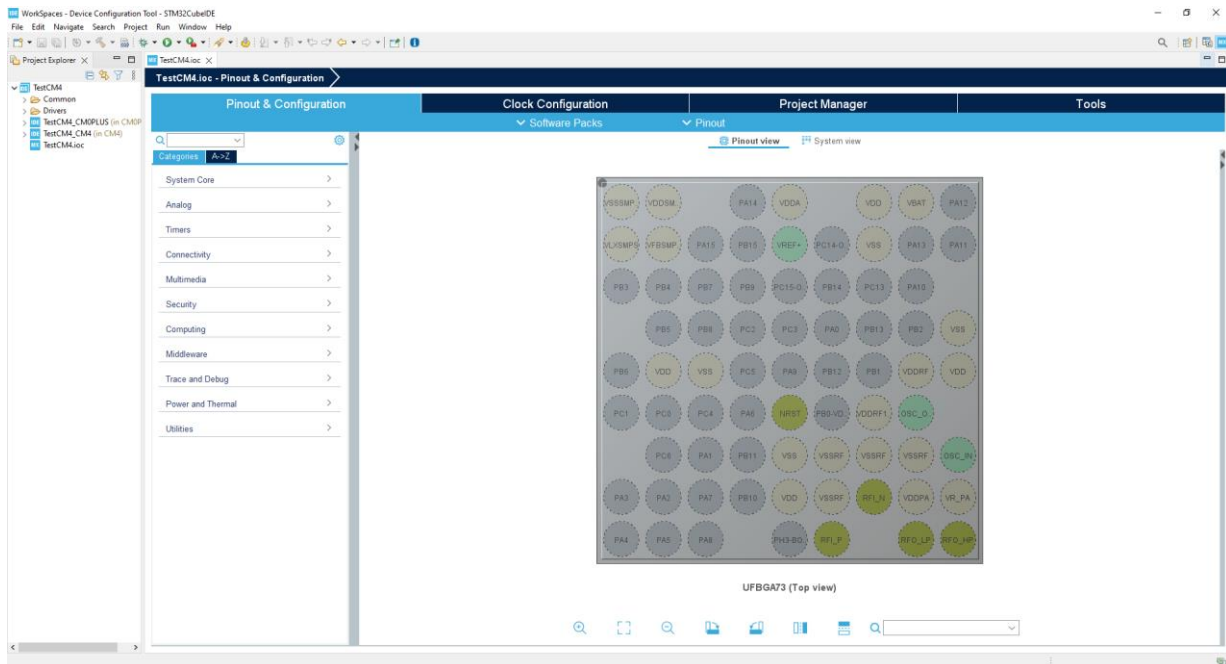
Here it is possible to set pin functionalities, select the peripherals to enable and configure them, configure the clocks for all the peripherals, enable middlewares and so on.

Configuring the clocks is quite trivial because the oscillators are the same for both cores, so no any choices are allowed. The module contains a 32 MHz TCXO as HSE (High Speed External oscillator) main oscillator and a 32.768 kHz crystal used as LSE (Low Speed External oscillator) for the RTC. In the *System Core* category, into *RCC* section select TCXO for HSE and Crystal/Ceramic Resonator for LSE.

In *Timers* category, into *RTC* section enable the RTC for Cortex®-M0+ core and set *Activate Clock Source* flag to enable LSE for the RTC.
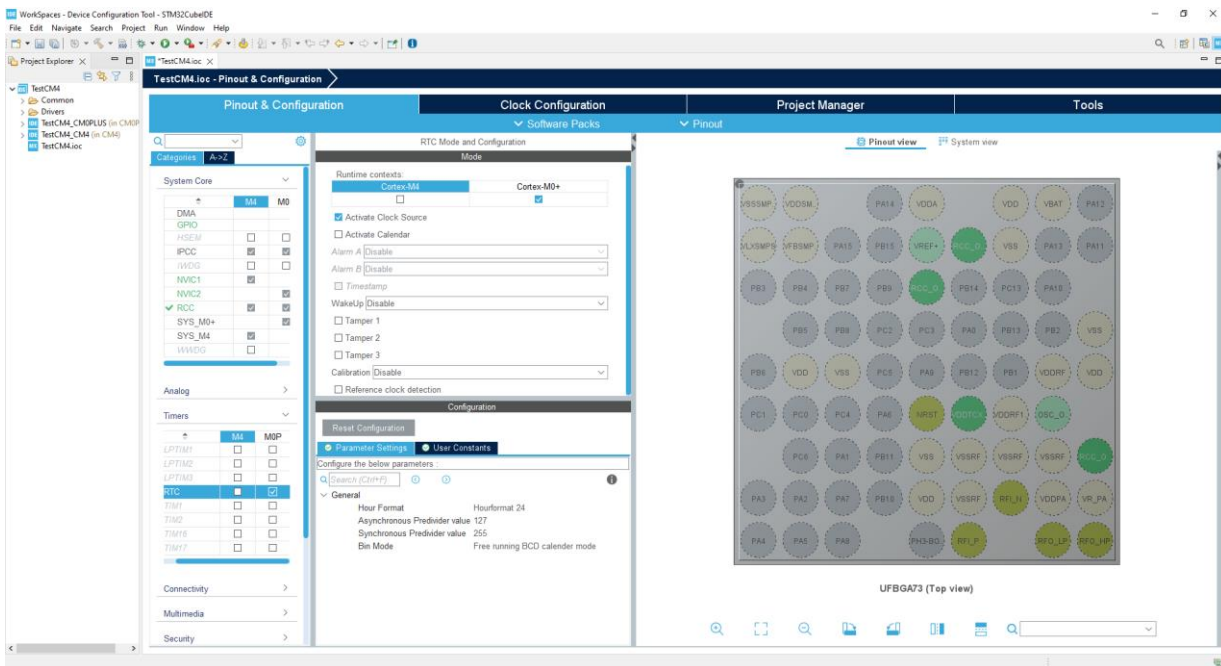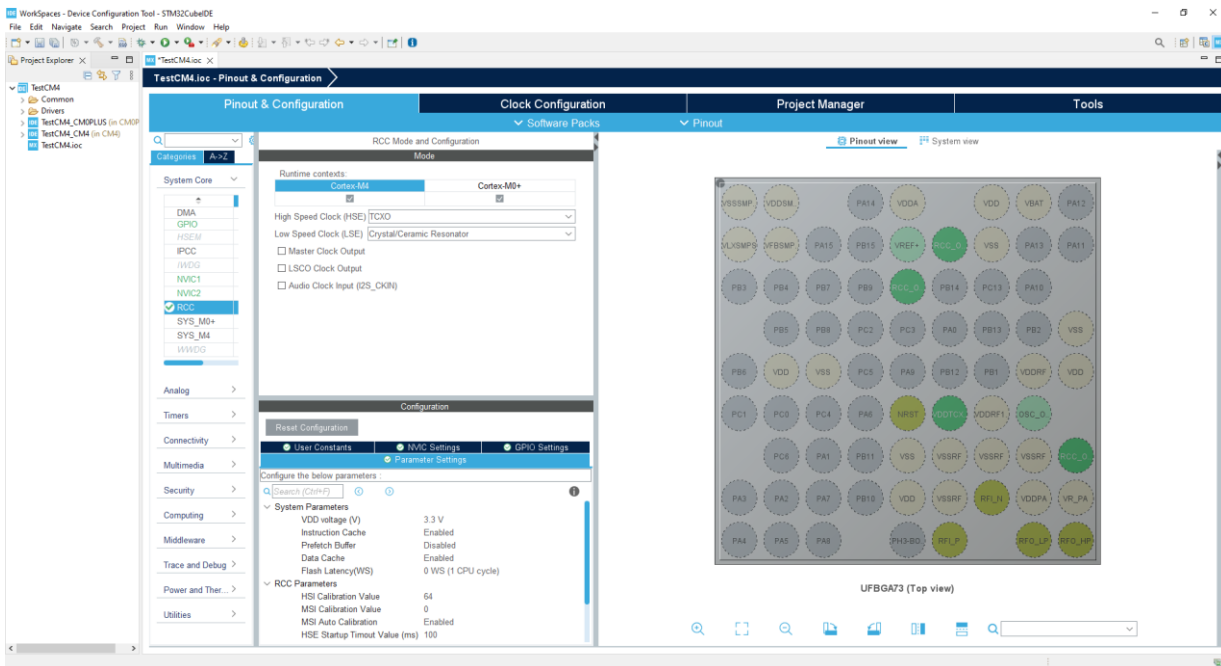
Then, in the Clock Configuration tab select the LSE source clock for the RTC, select the HSE as main oscillator source clock, enable the PLLCLK and set the following values for PLL multipliers/dividers: PLLM = /4, *N=X12, /R=/2. This way a SYSCLK equal to 48 MHz will be obtained. It is necessary to keep the HCLK2 for CPU2 (Cortex®-M0+) equal to 48 MHz. Other clocks may also be changed based on user application requirements.
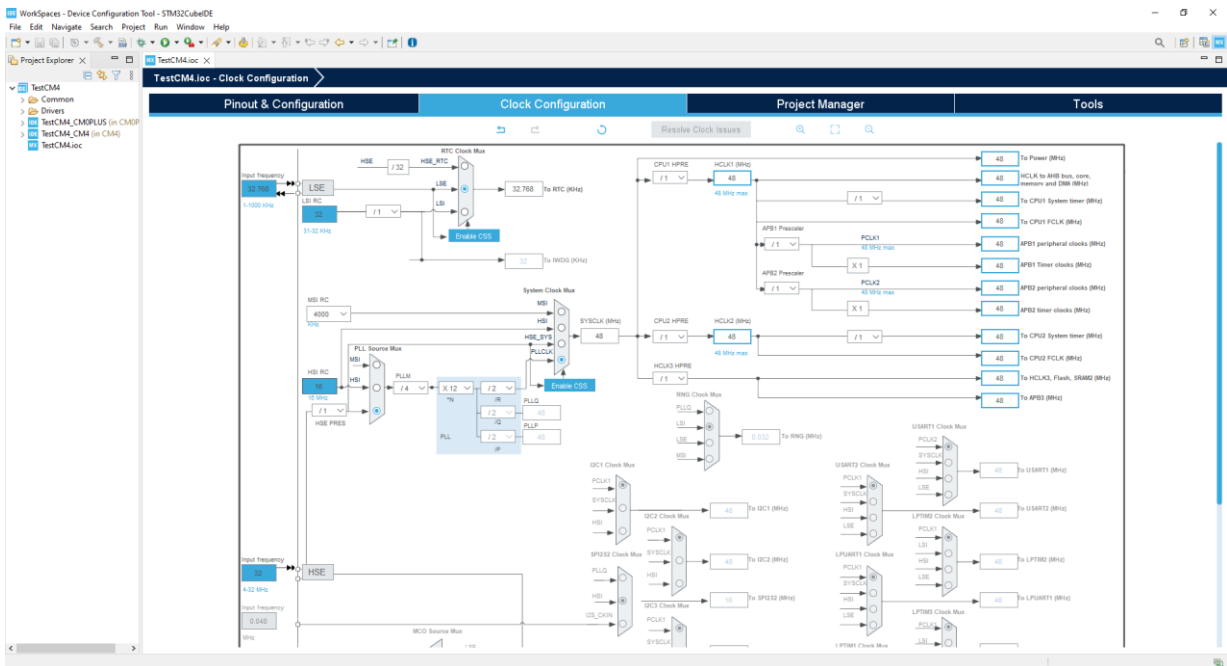
In the *System Core* category, into *IPCC* section set the flag Activated to enable the inter-processor communication controller.
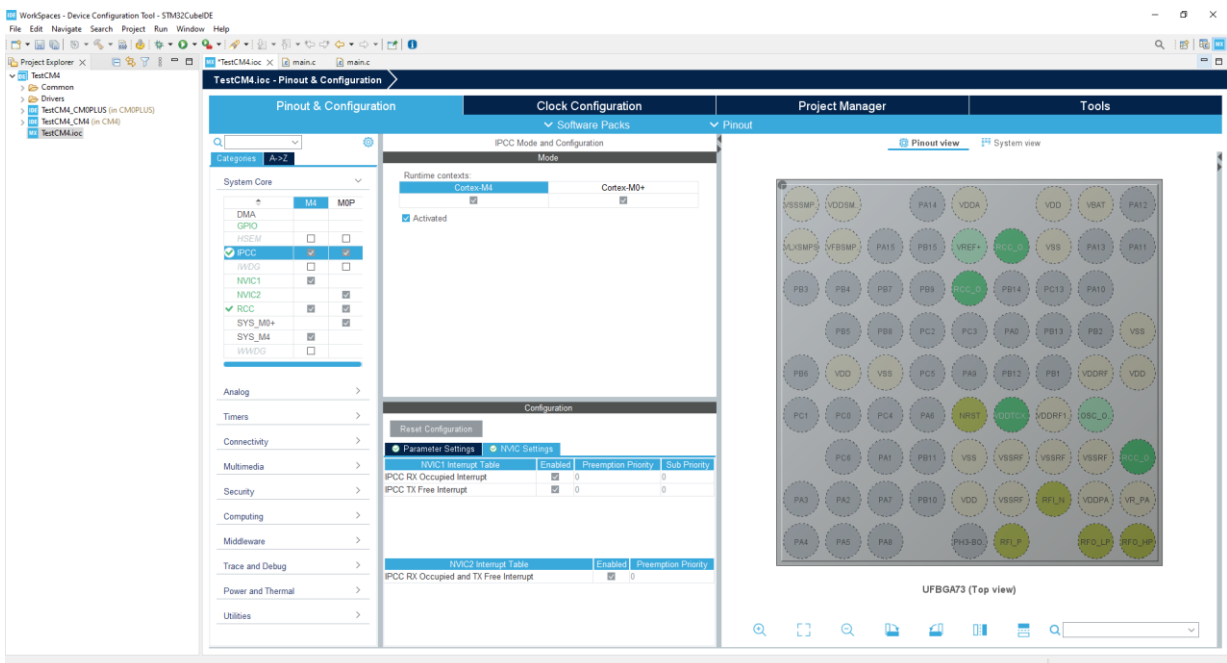
In the *Project Manager* tab, into *Code Generator* section the flag *Generate peripheral initialization as pair of '.c/.h' files per peripheral* may be selected to obtain a separation into multiple pairs of files of the low level code generated for each enabled peripheral.
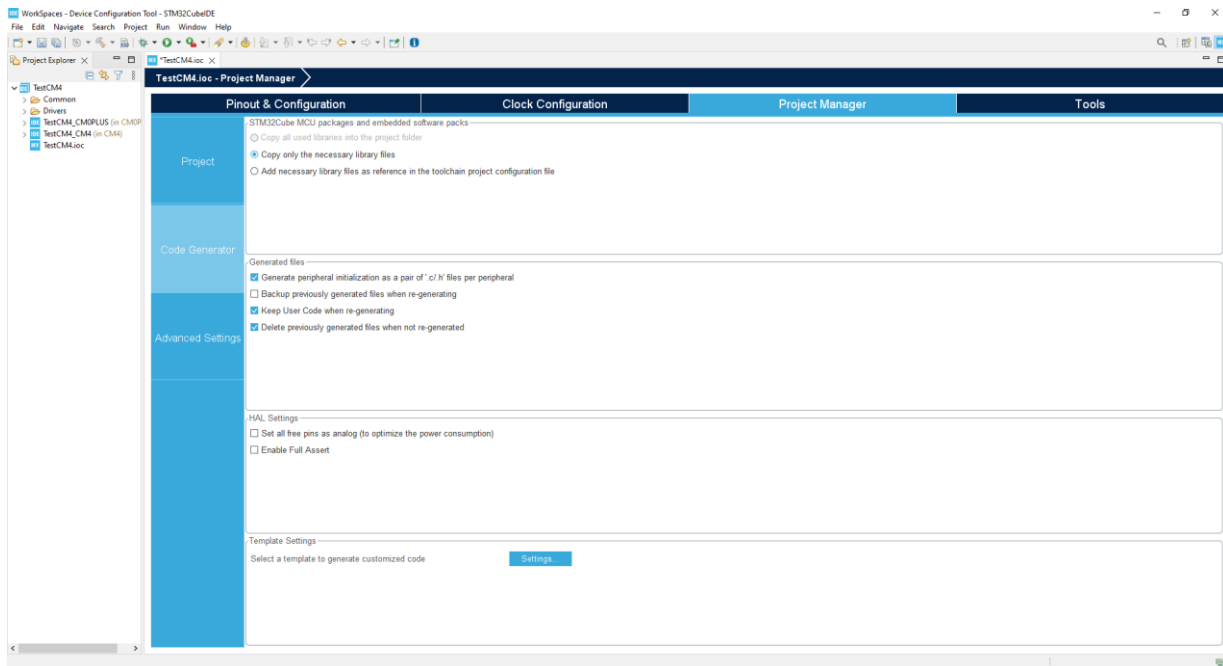
In the *Project Manager* tab, into *Advanced Settings* section, it is possible to choose for each enabled peripheral the library to use for low level code generation. Two choices are available: *HAL* driver library and *LL* driver library which provide a set of low level tools to control peripherals. LL library differs from HAL library in the fact that it provides a complete control of every peripheral's feature allow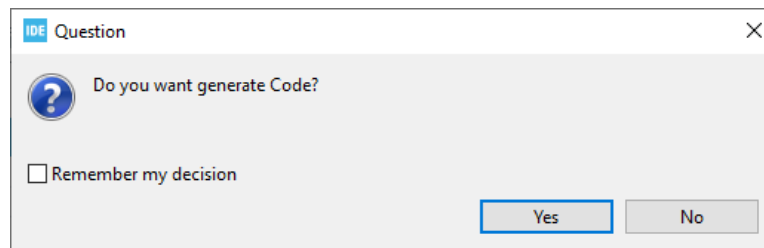ing the user to directly operate on peripheral's registers. HAL library, instead, is a higher level driver and adds an abstraction layer between hardware and application code which introduces some pre-configured management of the peripherals.

When saving the project the device configuration tool asks to generate the low level code:

Answering *Yes*, causes device configuration tool to create a framework project with all enabled peripherals initialized and ready for the user to start developing his own application.
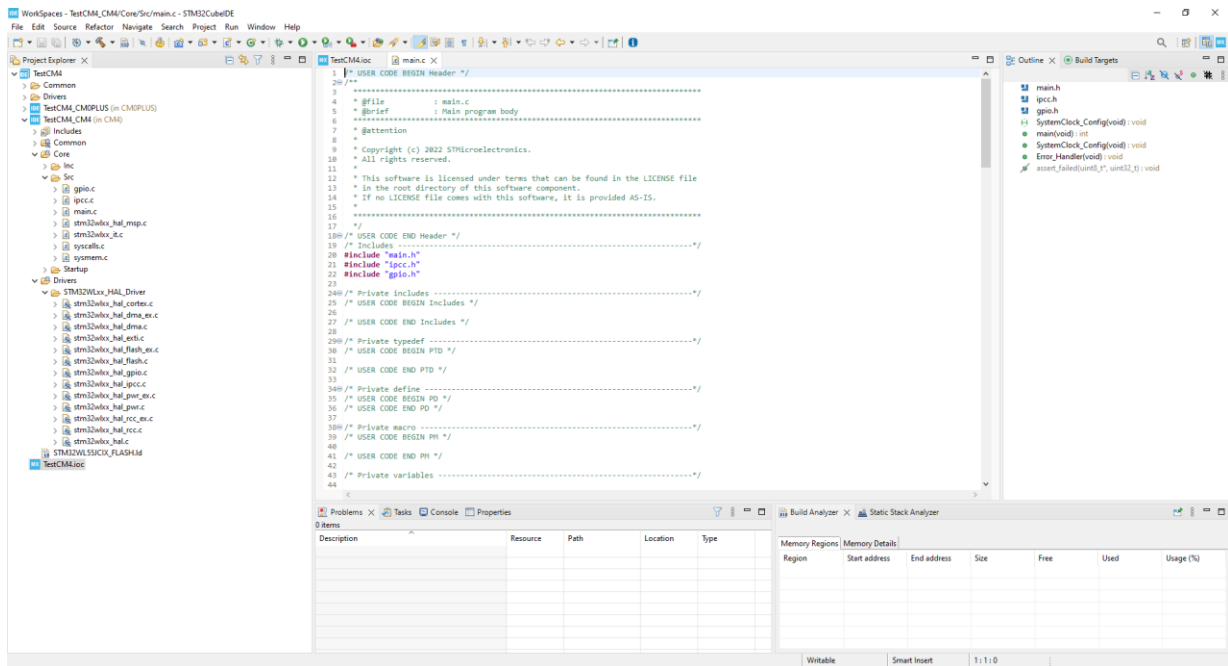
Code generation may be requested at any time using the specific icon , using the *Alt+K* key shortcut or selecting the *Generate Code* item into *Project menu*.
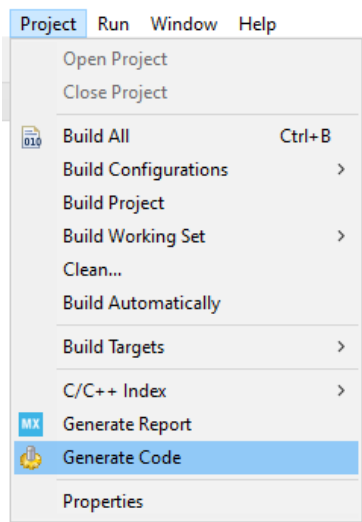
The configuration tool generates the code for both arm®Cortex®-M0+ and arm®Cortex®-M4 cores. The project for the arm®Cortex®-M0+ core is needed just to enable the HCLK2 clock for Cpu2 and the IPCC controller but the user does not have to develop any code for it and must not re-program any code into arm®Cortex®-M0+ reserved flash memory.

5. Importing an existing STM32 arm®Cortex®-M4 project

To import an existing STM32 arm®Cortex®-M4 project press *Import project* button from the *Information Center* tab:
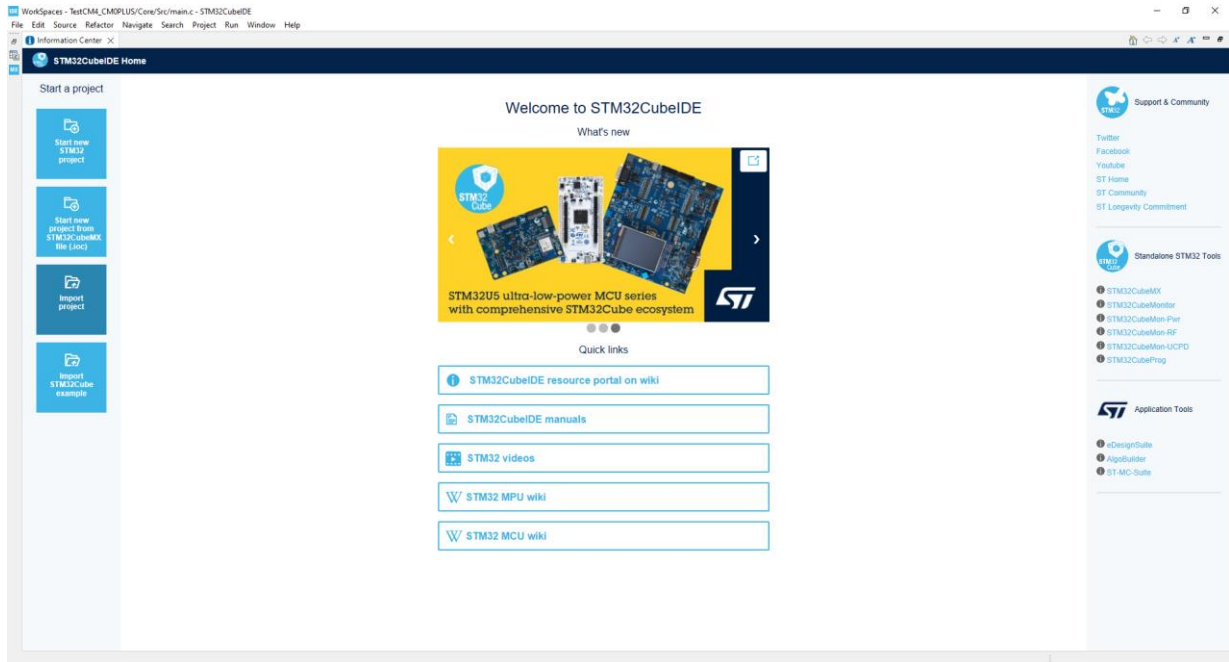
or select *Import* item from *File* menu:
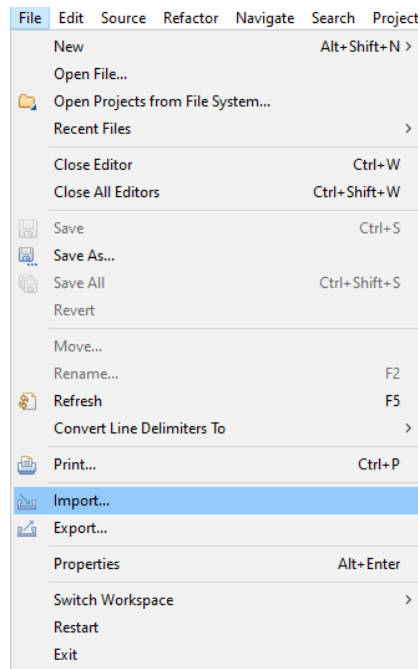
then choose *Projects from Folder or Archive* option from the *Import* dialog and press *Next*
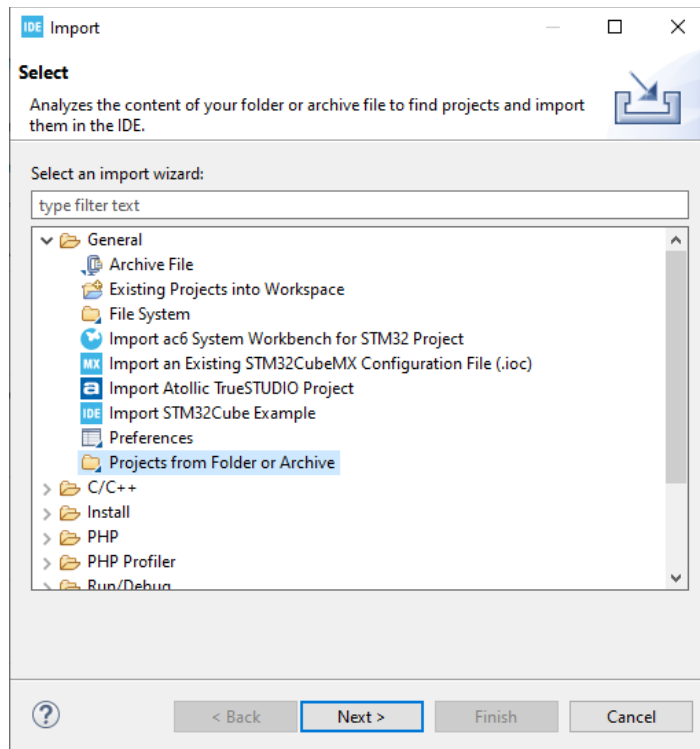
In the following dialog select the folder from which import the existing project and press *Finish*:
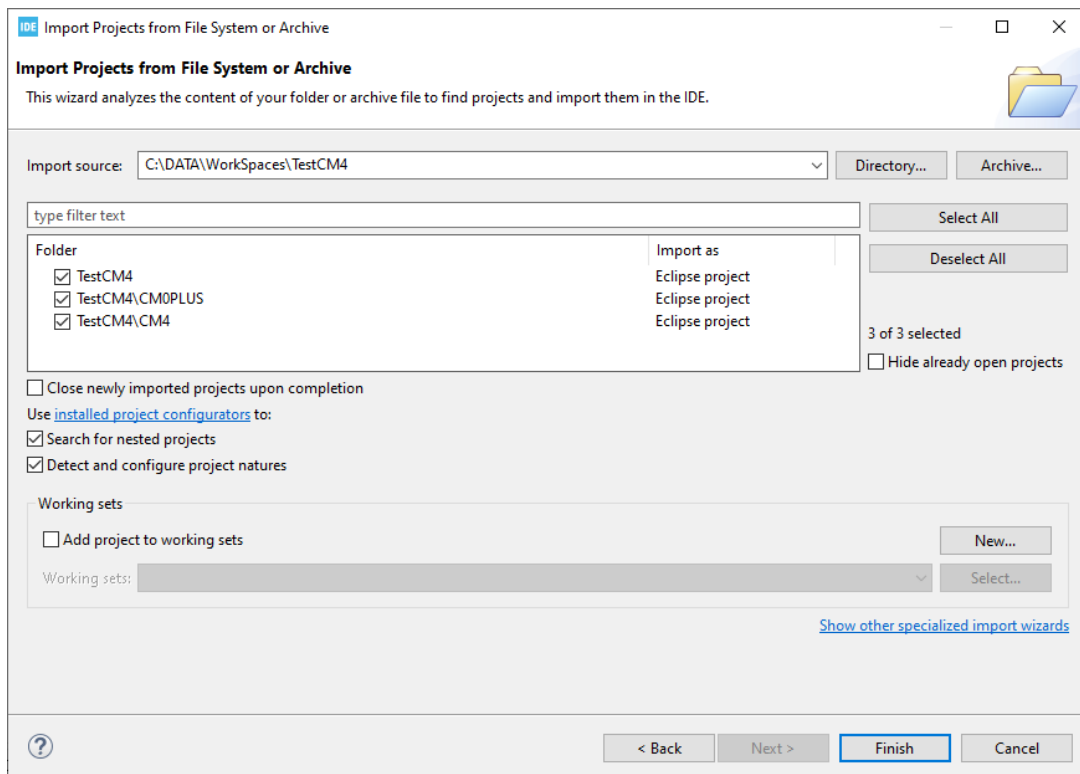
When importing a multiple project the user may choose to import the full project containing both the sub-projects for arm®Cortex®-M0+ and arm®Cortex®-M4 cores or simply the single project for the arm®Cortex®-M4 core.

In the project import case there is no need of the arm®Cortex®-M0+ project because here the assumption is that the arm®Cortex®-M4 project was previously created considering the dual core operation scenario.

In the above example the previously created TestCM4 project has been imported but, of course, any existing project can be imported from its specific folder or archive in the same way.

6. Debugging STM32 arm®Cortex®-M4 code

The arm®Cortex®-M0+ code is protected and not accessible by debug tools. So the only debuggable part is the user code developed for the arm®Cortex®-M4 core.

Before starting a debug session it is required to successfully build almost once the project and create a debug configuration. To do so right click on the project name in *Project Explorer* tab, select *Debug As* and then *Debug Configurations…*. Otherwise press the *Debug* icon and select *Debug Configurations…*.
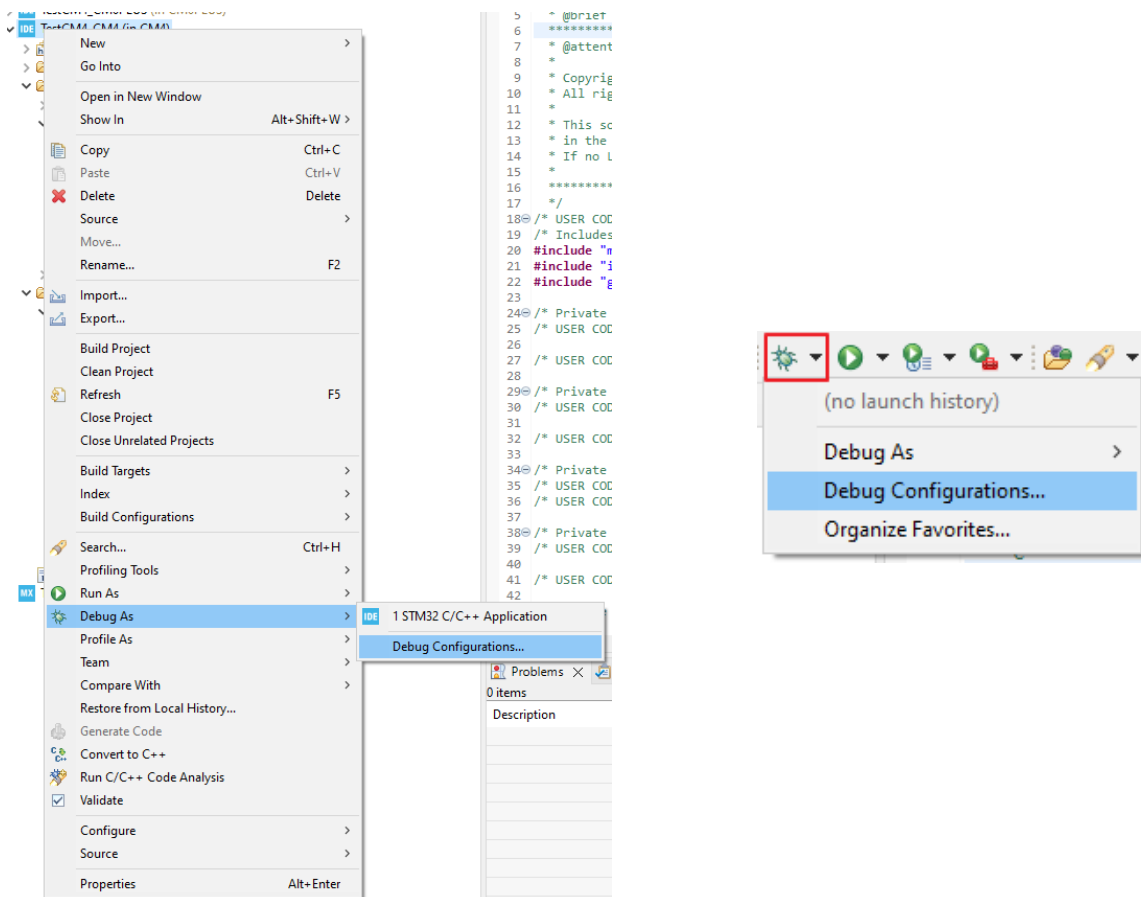
In the *Debug Configuration* dialog double click on STM32 C/C++ Application
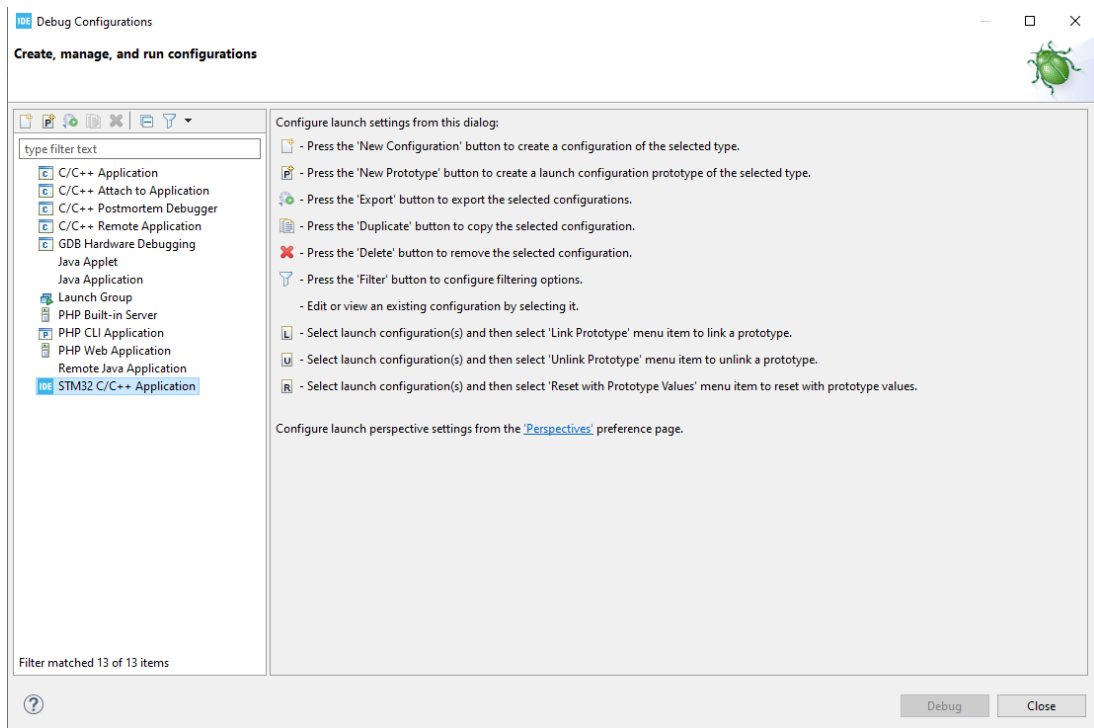
If the project had been successfully built almost once, a new debug configuration will be created with reference to the correct project:
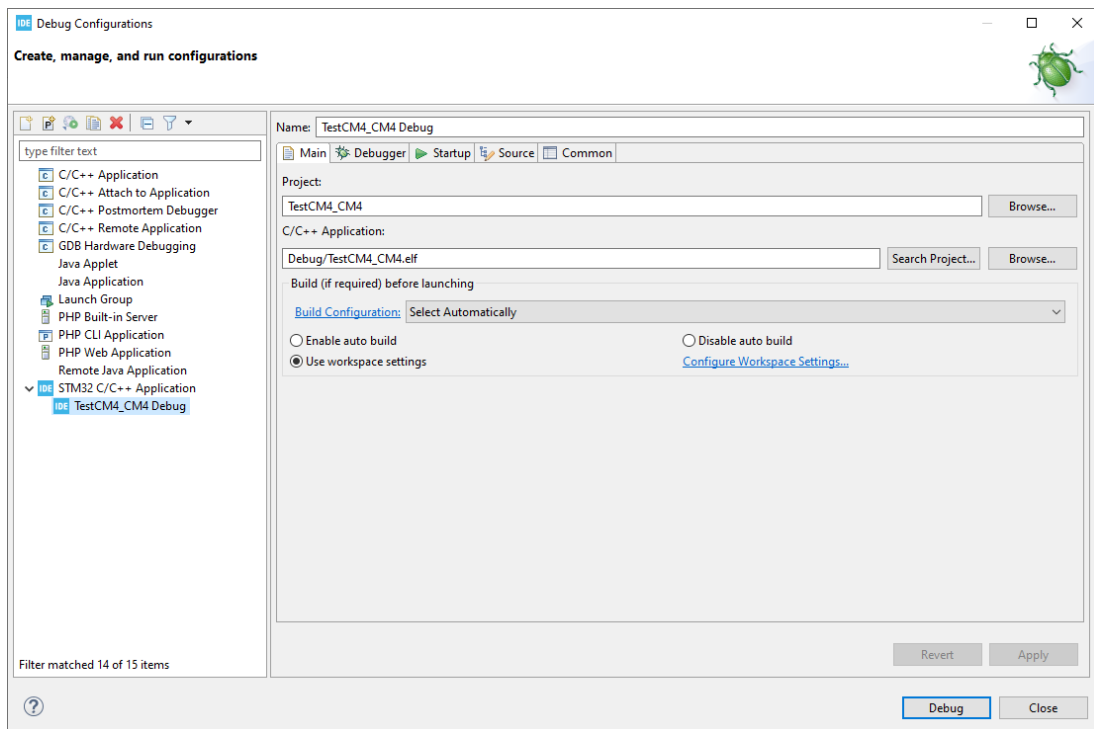
If ST-LINK is used as debug device there is no need of further settings. Otherwise the debug device has to be set into *Debugger* tab:
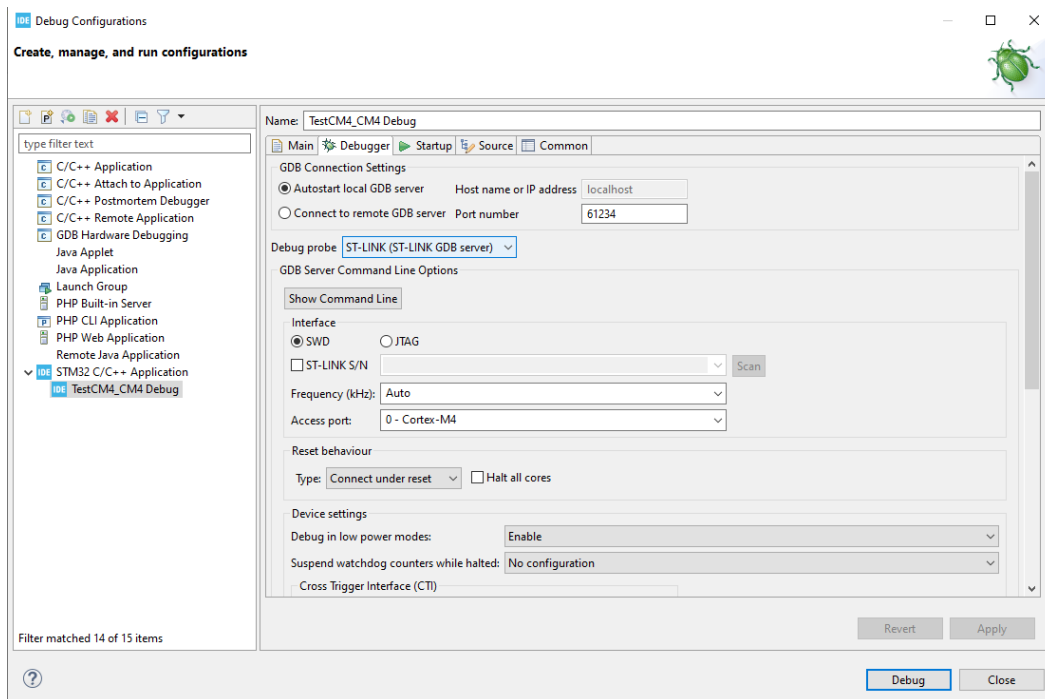
Pressing the *Debug* button the debug session will start and all classic debug tools will be available:
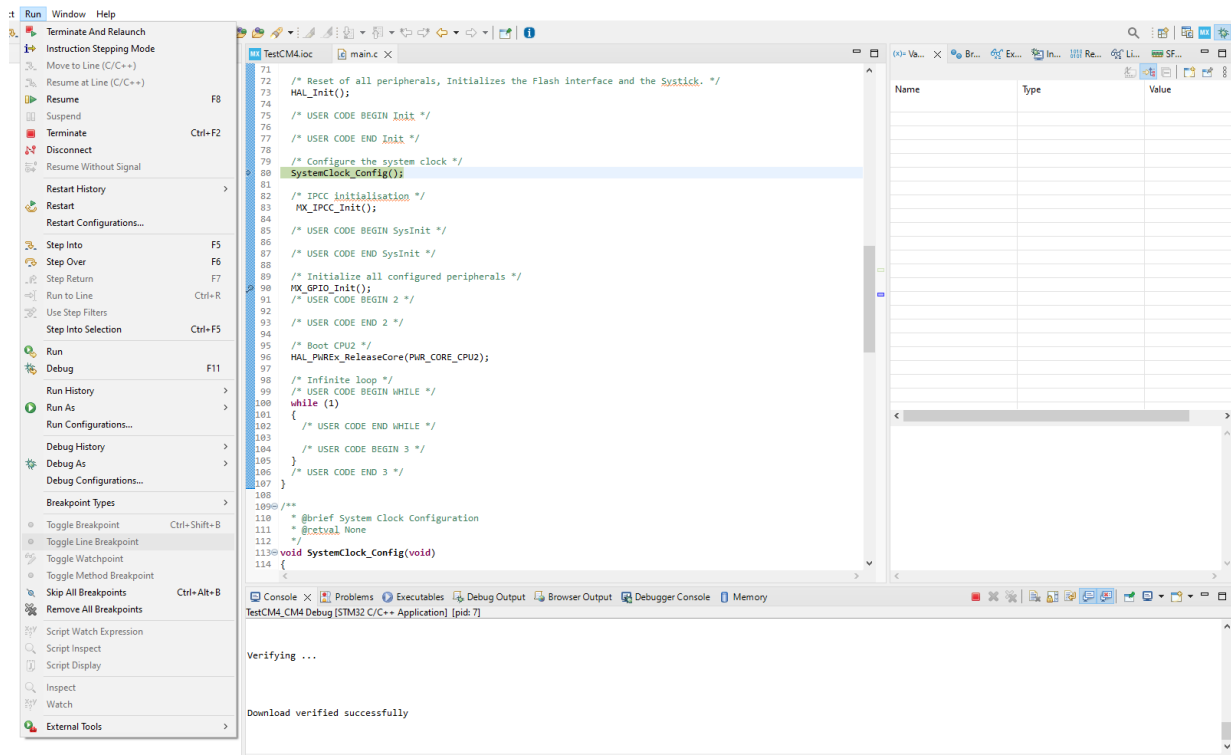
7. Flash memory and RAM considerations

Flash memory and RAM memory are shared resources but they are sharply splitted between cores to avoid situations where a core overflows to the memory domain assigned to the other one.

There is only one exception to this rule regarding the IPCC shared memory area. It is a 1 kB RAM area residing into arm®Cortex®-M0+ volatile memory zone but shared with arm®Cortex®-M4 to allow data exchange between cores.

The amount of flash memory available for the arm®Cortex®-M4 application code depends on the radio protocol stack chosen for the radio part. There are 5 available radio protocol stacks corresponding to as many software versions of the 32001506xxx module. They are listed in the following table:

Module type	Radio stack	Flash start address	Flash end address	Size
32001506Axx	WMBus	0x08000000	0x08031FFF	200 kB
32001506Bxx	LoRaWAN®		0x08029FFF	168 kB
32001506Cxx	LoRa® Mipot		0x0802C7FF	178 kB
32001506Dxx	LoRa® Modem		0x0802FFFF	192 kB
32001506Fxx	Multiprotocol LoRaWAN® + LoRa® Modem		0x08021FFF	136 kB

The amount of RAM memory available for the arm®Cortex®-M4 is the same for all module types as shown in the following table:

Module type	Radio stack	RAM start address	RAM end address	Size
ALL	ALL	0x20000000	0x20007FFF	32 kB

The IPCC shared RAM area is the same for all module types as shown in the following table:

Module type	Radio stack	IPCC start address	IPCC end address	Size
ALL	ALL	0x20008000	0x200083FF	1 kB

Flash and RAM memory areas are defined inside the linker file created for the specific build configuration used. As an example the linker file section defining the memory areas for the 32001506DEU is shown in the following image:

```
  1 /*
  2 ** LinkerScript
  3 */
  4
  5 /* Entry Point */
  6 ENTRY(Reset_Handler)
  7
  8 /* Highest address of the user mode stack */
  9 _estack = ORIGIN(RAM) + LENGTH(RAM); /* end of "SRAM1" Ram type memory */
 10
 11 _Min_Heap_Size  = 0x200; /* required amount of heap  */
 12 _Min_Stack_Size = 0x400; /* required amount of stack */
 13
 14 /* Memories definition */
 15 MEMORY
 16 {
 17   ROM    (rx)   : ORIGIN = 0x08000000, LENGTH = 192K      /* Flash memory dedicated to CM4 */
 18   RAM    (xrw)  : ORIGIN = 0x20000000, LENGTH = 32K       /* Non-backup SRAM1 dedicated to CM4 */
 19 }
 20
 21 /* Sections */
 22 SECTIONS
 23 {
 24   /* The startup code into "ROM" Rom type memory */
 25   .isr_vector :
 26   {
 27     . = ALIGN(8);
 28     KEEP(*(.isr_vector)) /* Startup code */
 29     . = ALIGN(8);
 30   } >ROM
 31
 32   /* The program code and other data into "ROM" Rom type memory */
 33   .text :
 34   {
 35     . = ALIGN(8);
 36     *(.text)           /* .text sections (code) */
 37     *(.text*)          /* .text* sections (code) */
 38     *(.glue_7)         /* glue arm to thumb code */
 39     *(.glue_7t)        /* glue thumb to arm code */
 40     *(.eh_frame)
 41
 42     KEEP (*(.init))
 43     KEEP (*(.fini))
 44
 45     . = ALIGN(8);
 46     _etext = .;        /* define a global symbols at end of code */
 47   } >ROM
 48
 49   /* Constant data into "ROM" Rom type memory */
 50   .rodata :
 51   {
 52     . = ALIGN(8);
```

8. STM32 arm®Cortex®-M0+ boot

The arm®Cortex®-M0+ can boot only if the arm®Cortex®-M4 enables it by setting the bit 15 *C2BOOT* (*CPU2 boot after reset or wakeup from Stop or Standby mode*) of the PWR control register 4 (*PWR_CR4*).

This can be done calling the function *HAL_PWREx_ReleaseCore(PWR_CORE_CPU2)* if using HAL drivers, calling the function *LL_PWR_EnableBootC2()* if using LL drivers or directly writing to the PWR_CR4 register (*SET_BIT(PWR->CR4, PWR_CR4_C2BOOT)*).

In the following image the Cpu2 boot is performed using the HAL function call.

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

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

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

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* IPCC initialisation */
  MX_IPCC_Init();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

  /* Boot CPU2 */
  HAL_PWREx_ReleaseCore(PWR_CORE_CPU2);

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

9. Inter-core communication

Inter-core communication is performed through IPCC (Inter-Processor Communication Controller) which is based on the 1 kB shared volatile memory area physically located at the beginning of the arm®Cortex®-M0+ RAM domain.

Communication is logically separated into two kinds of messages: main communication protocol messages used for radio module control (e.g. working parameters configuration, transmit and receive commands) and internal service messages used for inter-processor coordination (e.g. low power management).

Messages are internally routed through different physical channels; so a communication channel and a service channel are implemented, each of them based on a specific communication buffer residing at different locations into shared memory.

There is also a further shared memory location used for extra-IPCC information exchange; it is used e.g. to share the arm®Cortex®-M0+ status (started or not) or to transfer to the arm®Cortex®-M0+ data or commands asynchronous with respect to the normal IPCC flow (e.g. a core reset request).

The above mentioned buffers are listed in the following table:

Buffer type	Usage	RAM start address	RAM end address	Size
Communication	Communication protocol messages	0x200082FC	0x200083FF	260 Bytes
Service	Internal service messages	0x200082EC	0x200082FB	16 Bytes
Extra-data	Asynchronous messages	0x200082E8	0x200082EB	4 Bytes

9.1. Communication messages

The communication over the communication channel between the arm®Cortex®-M4 and arm®Cortex®-M0+ cores is performed with the same communication protocol defined for the physical SPI/I²C/UART channels used into 32001505xxx. For details please refer to *"32001505xxx_Command_Reference_revx.x.pdf"* document.

The core that initiates a data exchange over the communication channel must:

- verify that the channel is free;
- write the protocol message into the communication buffer;
- notify to the other core that the channel is busy and a message is waiting to be read;
- wait for the receiving core to notify that the message has been read and the channel is free again.

The receiving core can react in two ways:

- if no reply is needed for the received message the core clears the communication buffer and simply notifies that the channel is free;
- if a reply is required the core must clear the communication buffer, write its answer into the communication buffer and then notify that the channel is free.

The sending core, on its side, when receives a channel free notify must always check the communication buffer for a valid message present. No further notification has to be sent and the IPCC communication cycle is anyway considered to be completed.

It is mandatory to perform the complete IPCC communication cycle for every sent message otherwise the inter-core communication stops and waits for the missing step.

9.2. Service messages

The communication over the service channel between the arm®Cortex®-M4 and the arm®Cortex®-M0+ cores is based on a very small set of commands using the following minimal protocol syntax:

> *<cmd>[<arg>]*

where

- *<cmd>* is the command code;
- *<arg>* is the optional argument associated to the command.

Both *<cmd>* and *<arg>* are 1 byte fields.

Service commands are listed in the following table:

Command code	Argument	Description	Direction
0x01	none	Request low power mode	CM4 ←→ CM0+
0x02	none	Request wake-up from low power	CM4 ←→ CM0+
0x05	baud rate	Request UART baud rate setting	CM4 ← CM0+
0xA5	none	End of message processing	CM4 → CM0+

Commands *0x01 'Request low power mode'* and *0x02 'Request wake-up from low power'* can be sent from both cores depending on the module's working mode. Please refer to paragraph *"11. Module's pins usage"* to gather more information about how module pins are managed.

If the module uses pins to change and signal its status then commands 0x01 and 0x02 are sent from arm®Cortex®-M0+ to arm®Cortex®-M4 because in this case low power is managed by arm®Cortex®-M0+ and propagated to arm®Cortex®-M4.

Otherwise low power is managed by arm®Cortex®-M4 and propagated to arm®Cortex®-M0+ through commands 0x01 and 0x02, so they follow the opposite direction.

Command *0x05 'Request UART baud rate setting'* is used only in the case of stand-alone module emulation mode with UART communication. In this working mode the module behaves as a stand-alone module with the arm®Cortex®-M4 routing messages from the physical UART to the arm®Cortex®-M0+ core and vice versa. As the baud rate value is a parameter contained into the arm®Cortex®-M0+ configuration memory and the UART is managed by the arm®Cortex®-M4, the arm®Cortex®-M0+ uses this command to inform the arm®Cortex®-M4 about the baud rate to set. Admitted values for baud rate are:

Baud rate value	Baud rate
0x00	9600
0x01	19200
0x02	38400
0x03	57600
0x04	115200

Command *0xA5 'End of message processing'* is a sort of acknowledge that the arm®Cortex®-M4 sends to arm®Cortex®-M0+ after receiving a communication message from it. Its aim is to prevent the arm®Cortex®-M0+ sending further communication messages before the arm®Cortex®-M4 completes processing the previous one. arm®Cortex®-M4 must send it otherwise arm®Cortex®-M0+ will not send any new messages.

Service messages comply to the same IPCC rules already mentioned for communication messages, except for the fact that they use the service channel and the related buffer.

The core that initiates a data exchange over the service channel must:

- verify that the channel is free;
- write the message into the service buffer;
- notify to the other core that the channel is busy and a message is waiting to be read;
- wait for the receiving core to notify that the message has been read and the channel is free again.

On the service channel commands do not require a reply. So the receiving core reacts in the following way:

- the core clears the communication buffer and simply notifies that the channel is free;

No further notification has to be sent and the IPCC communication cycle is anyway considered to be completed.

It is mandatory to perform the complete IPCC communication cycle for every sent message otherwise the inter-core communication stops and waits for the missing step.

9.3. Asynchronous messages

Asynchronous messages are commands or signals without arguments that are exchanged out of the regular IPCC communication flow. They are listed in the following table:

Command code	Description	Direction
0xAA	arm®Cortex®-M0+ not started	CM4 → CM0+
0xBB	arm®Cortex®-M0+ started	CM4 ← CM0+
0xCC	Request IPCC interface re-initialization	CM4 → CM0+
0xDD	Request sub-GHz radio re-initialization	CM4 → CM0+
0xEE	Request arm®Cortex®-M0+ restart	CM4 → CM0+

10. Low power management

Details about power management can be found into chapter 6 *Power control (PWR)* of the STM32WL55JC reference manual.

Here we just recap the concept that three power domains have to be considered: arm®Cortex®-M4 power domain, arm®Cortex®-M0+ power domain and System power domain. When talking about low power management it is necessary to clarify at which power domain it is referred to. It may happen that at a specific time arm®Cortex®-M4 core is in a particular low power state while arm®Cortex®-M0+ is in a different one. The System low power condition corresponds to the lowest (in terms of deepness in sleep condition) low power state between the ones of each core's power domain. If arm®Cortex®-M4 is in Sleep mode while arm®Cortex®-M0+ is in Stop 2 mode the System is considered to be in Sleep mode.

When arm®Cortex®-M0+ is set to low power it reaches Stop 2 mode.

Module 32001506xxx provides two ways to manage low power depending on the pin usage set during configuration.

If pin usage is enabled then the module activates signalling through pins and low power management is performed through NWAKE pin level checking:

- NWAKE pin high → module to low power;
- NWAKE pin low → wake up module.

NWAKE pin level checking is performed by arm®Cortex®-M0+ core. So when a change is detected the arm®Cortex®-M0+ behaves as follows:

- if NWAKE becomes high, it sends the service command 0x01 to arm®Cortex®-M4 to signal it that a low power request has arised and then goes to Stop 2 low power mode;
- if NWAKE becomes low, it wakes up from Stop 2 mode and then sends service command 0x02 to arm®Cortex®-M4 to signal it that a wake up request has arised.

If pin usage is disabled then low power management has to be performed by the user application controlling the arm®Cortex®-M4 core.

When the user application requests a low power condition it sends the service command 0x01 to arm®Cortex®-M0+ core which goes to Stop 2 low power mode.

When the user application requests a wake up from low power condition it sends the service command 0x02 to arm®Cortex®-M0+ core which wakes up from Stop 2 low power mode.

11. Module's pins usage

Module's pins usage is the feature that allows the arm®Cortex®-M0+ core to manage low power condition based on the level of NWAKE pin and to signal radio data availability through NDATA_INDICATE pin.

Pins usage is kept for compatibility with the stand-alone version of the module (32001505xxx product) but is normally disabled by default.

To enable pins usage the following configuration command has to be sent to the arm®Cortex®-M0+ core through the IPCC interface mechanism:

0xAA 0x32 0x02 0x91 0x01 0x90

To disable pins usage the command is:

0xAA 0x32 0x02 0x91 0x00 0x91

To read the pins usage status issue the command:

0xAA 0x33 0x01 0x91 0x91

Above commands correspond to configuration memory write/read commands at address 0x91 which is the location of the parameter that defines the pins usage. Admitted values are 0x00 → pins disabled and 0x01 → pins enabled.

12. Revision History

Revision	Date	Description
0.1	08.11.2022	Preliminary