AFBR-S50 SDK

Porting Guide to a Cortex-M4

Programming Guide
Version 1.0

Table of Contents

Chapter 1: Introduction

The API for the AFBR-S50 sensor family is not bound to specific features of one microcontroller and the software can therefore be ported to a variety of microcontroller units (MCUs). However, the primary features need to be adapted to the specific hardware of a different MCU.

This document explains the necessary steps to allow the example application from the AFBR-S50 SDK to run on a different MCU model and manufacturer, with the Nucleo-F401RE board carrying the STM32F401RETx MCU with one Cortex-M4 core as an example.

The document describes one way to include the software to a different vendor's IDE, including the setup of the project structure on a step-by-step approach, and the code changes required to access the required hardware on the new MCU.

It does not cover the aspects of building without an IDE, and it relies on platform abstractions provided by the platform manufacturer.

NOTE: Refer to the API Reference Manual for more information on the API usage. The manual can be accessed by starting the AFBR-S50 Explorer > Help > API Reference Manual.

ATTENTION: The figures and illustrations are specific to the STMCubeIDE and the Nucleo-F401RE board, while the steps and basic procedure should be similar on other boards.

1.1 AFBR-S50MV85G-EK Evaluation Kit Software

The software that is part of the official AFBR-S50MV85G-EK evaluation kit consists of two parts:

- A static library containing the logic required to control the AFBR-S50 device and perform measurements.
- Applications that operate the device, like the simple ExampleApp as a starting point for your application development.

Note that the library with the device logic is designed to be independent from the MCU hardware, so that it can be compiled and distributed independently from the actual MCU model and therefore will be made available to you in binary form.

To allow this independency, the access to the actual hardware is modeled by a hardware abstraction layer (HAL), which provides the functionality to access the device. This is an API that must be implemented on the application side and is used by the AFBR-S50 library.

This document provides a guide on how to implement the required HAL API on your targeted device and to get the example application running.

Chapter 2: Phase 1: Installing and Preparing the IDE

Many compilers and IDEs are available for various microcontrollers, most of which have a commercial license.

However, most manufacturers of Arm-based MCUs offer also a free IDE with an integrated version of the ARM toolchain, which is most frequently a specifically adapted version of the eclipse IDE that has special support for their MCUs.

While the basic steps for porting the software that is part of the official AFBR-S50MV85G-EK evaluation kit are similar, the individual steps in this guide are illustrated using an IDE called STM32CubeIDE by ST Microcontrollers.

Step 1. Downloading and Installing the IDE

The first step is to download and install the IDE from the web site of ST Microcontrollers, currently available from the following link.

https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-ides/stm32cubeide.html

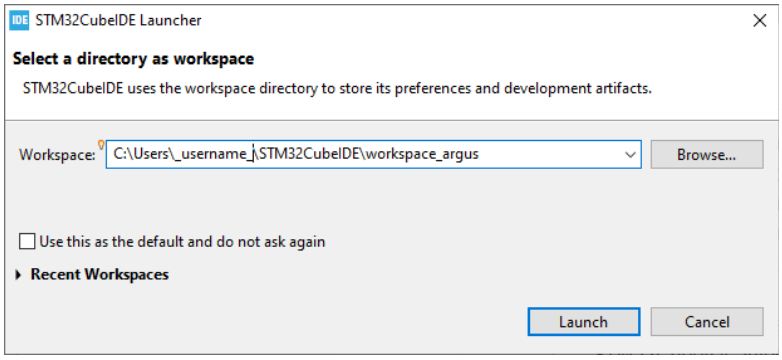
Follow the instructions of the installer.

Step 2. Defining the Workspace

The workspace is the area where all projects regarding the device live.

You can either use the default, or create a new workspace for the project (**File** > **Switch Workspace** > **Other…**).

Figure 1: Switching to a Project Workspace



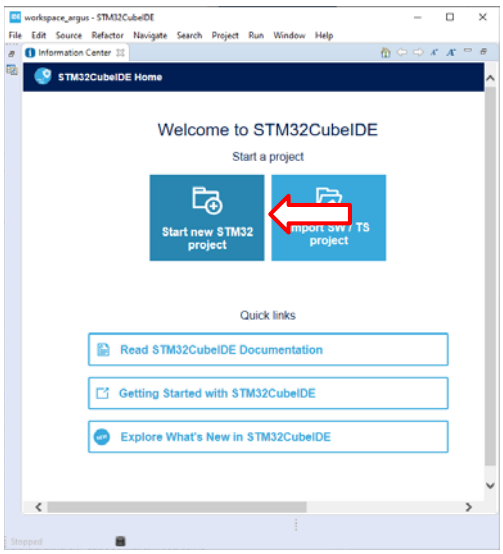
In this example, a new workspace is created with the name "workspace_argus" for the user _username_, which is the name of the user logged in.

Step 3. Creating a Native Project

First, start with a project for the targeted board or processor type.

In the example, this is done by selecting **Start new STM32 project** from the startup menu.

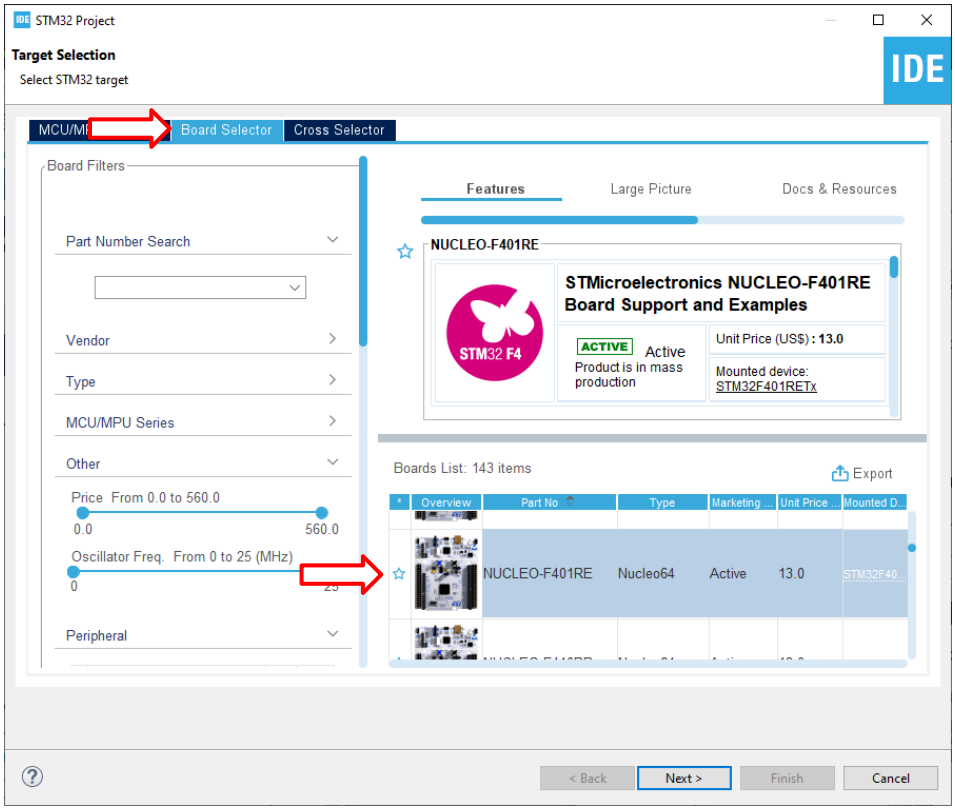
Figure 2: Starting a New Project in the IDE



The advantage of using the manufacturer's IDE to create the project is that it can create an initial setup tailored to the used board or microcontroller. Therefore, the next step is to select the used board or MCU from the selection list.

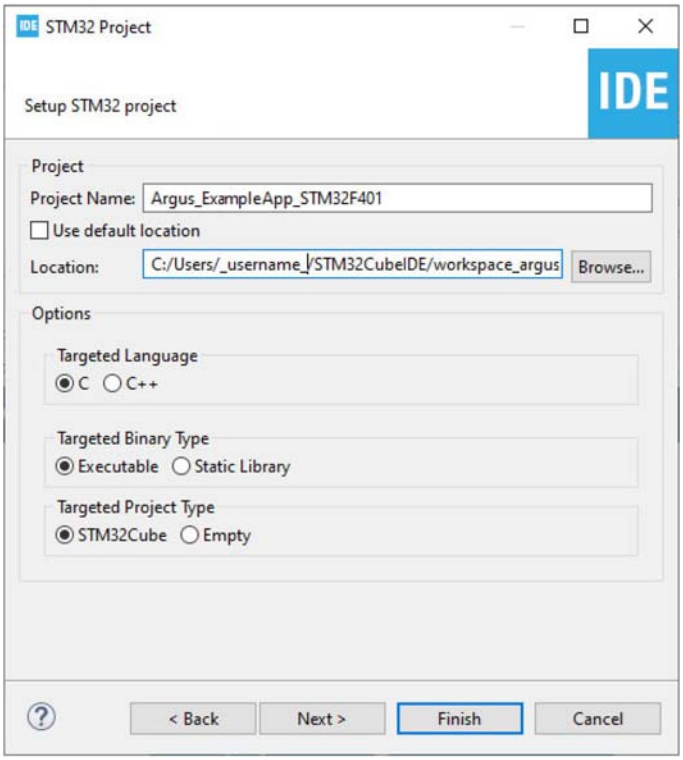
In this example, this is the NUCLEO-F401RE evaluation board, which features the STM32F401RE microcontroller.

Figure 3: Target Board/MCU Selection in the IDE



Next, you must choose the project name and set options.

Figure 4: Target Setup in the IDE



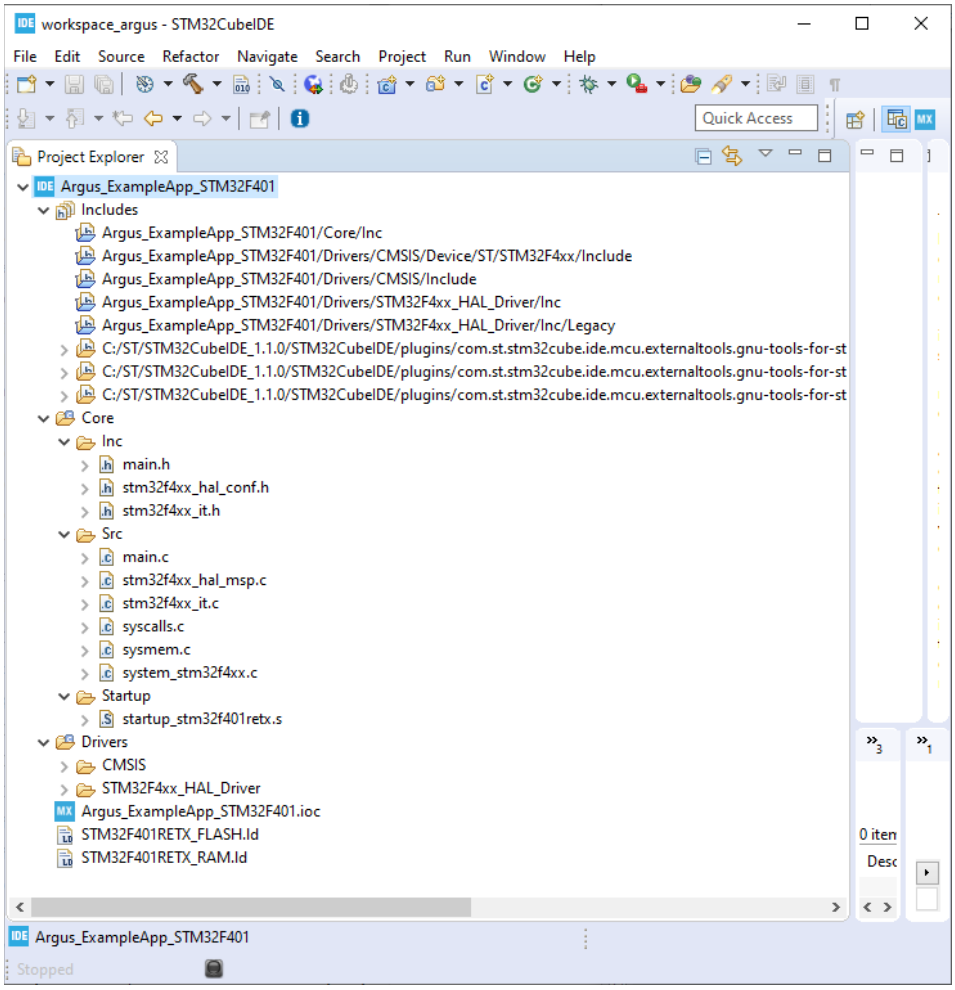
The name can be chosen freely, for example, according to your company's standards, as can be the targeted language. Because the example uses the native type as the targeted project type, the IDE generates a project that already contains specific code for the architecture.

Click **Finish** to create the project.

ATTENTION: You might see a prompt about whether to initialize all peripherals with their default mode. If this occurs, click **No** to follow the instructions.

Because the project is created specifically for the object type, several files containing platform-specific code are generated by default in the project's structure.

Figure 5: Automatically Created Project Structure



2.1 File Structure

Four types of files exist in the project:

- Files that belong to the AFBR-S50 SDK

  There is no need to adapt these files, and they should live outside the workspace.

  They are linked from the project using the include path of the compiler.

- Automatically generated files that belong to the target projects

  These files are those imported or generated by the wizards of the IDE and the files from the device application. They should be part of the workspace.

  The IDE creates them automatically in the `Core` folders for the files that are specific to the project and the `Drivers` folder for the included predefined files from the source distribution.

- Manually generated API hardware layer

  These files implement the hardware layer API required by the AFBR-S50 library. They use the automatically generated files.

  An `API` folder is created for these files.

- Manually generated files that belong to the target projects

  These files are created in the editor and belong to the device application. They should also be part of the workspace.

  An App folder is be created for these files.

Step 4. Creating the File Structure

Create the following folders as source folders in the Project on the top level. This is done using the context menu of the Argus_ExplorerApp project in the Project Explorer (**New** > **Source Folder**).

- `API`
- `App`

ATTENTION: It is essential to create a **Source Folder** rather than **Folder**. Otherwise, the source files are not compiled with the IDE.

Figure 6: Creating the Source Folders in the IDE

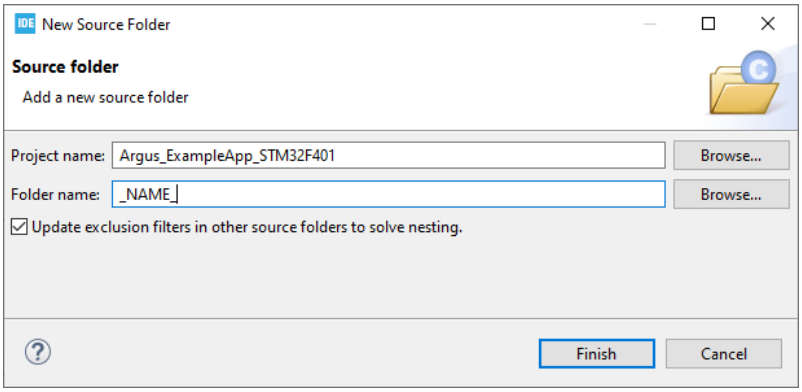
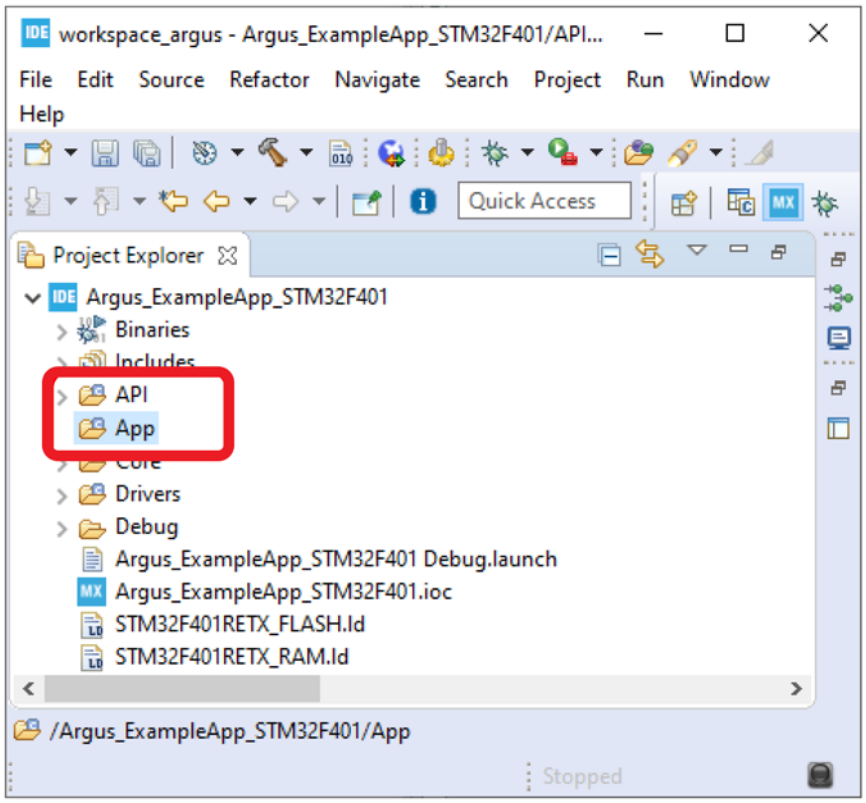


Figure 7: Project Structure with New Folders



Chapter 3: Phase 2: Addition of the MCU Devices with the IDE

Now you need to add the hardware devices on the MCU and their configuration and initialization.

Two options are usually available:

1. Manually create the device configuration.
2. Have the device configuration automatically created by the wizard.

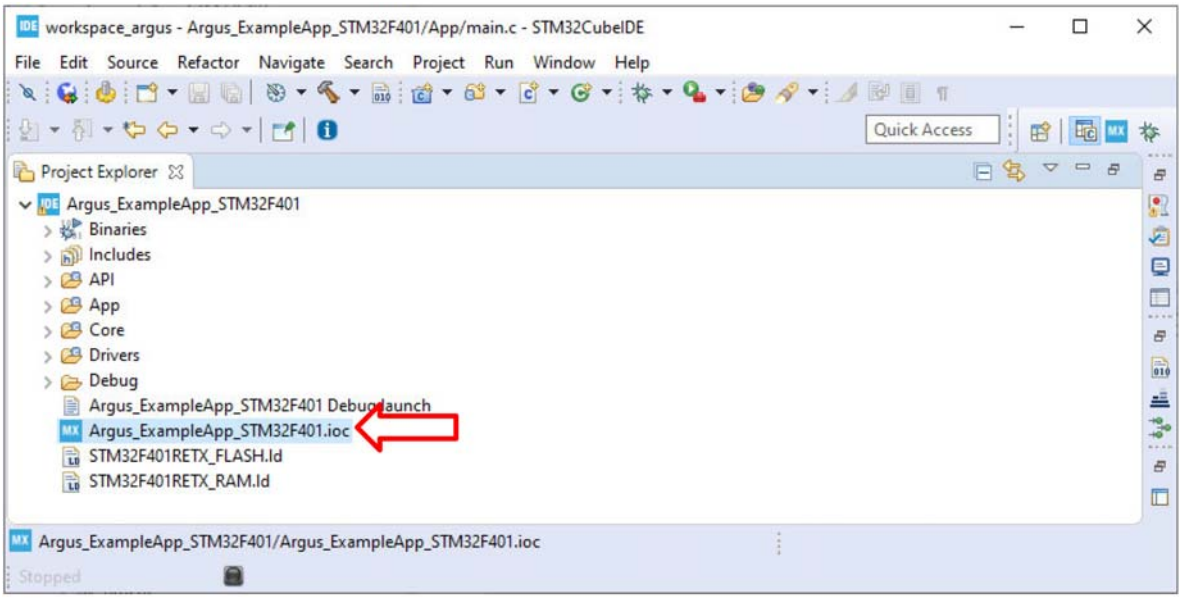
The first option is always possible and does not depend on the IDE. On the other hand, it requires a more detailed knowledge of the MCU type and the I/O hardware registers, on the manufacturer's software that comes with it, or both.

Therefore, the second option is chosen, but it is difficult to explain all the steps in detail, so that they can be reproduced in a similar fashion on a different vendor's hardware. In addition, the focus is on what exactly is set up, so that the description is also helpful if the first approach is chosen.

Step 5. Opening the Device Configuration Tool in the IDE

The STMCube32 IDE has a device configuration tool, in which the setup of the hardware can be defined graphically. This tool can be opened by clicking **Argus_ExampleApp_STM32F401.ioc** in the project folder.

Figure 8: Opening the Device Configuration Tool



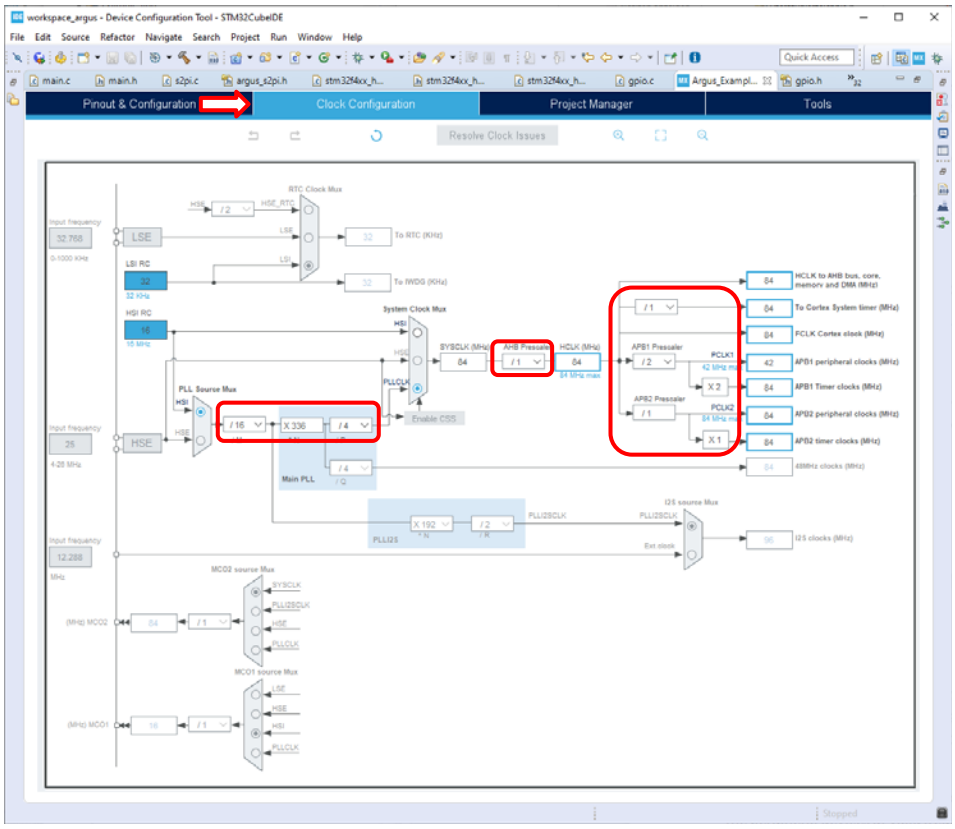
3.1 Clock Configuration

The board must have a valid clock configuration to operate the compute core and the peripheral devices at valid frequencies. Choose the highest valid frequencies to operate at optimal speed.

In the IDE, this configuration can be selected in the Clock Configuration section of the device configuration tool. This includes the configuration of the oscillators for the board and their frequencies. In the predefined board example, the only required setup is the multipliers and dividers to get the correct frequencies for the system clock and the peripheral clocks.

For performance reasons, the device should operate the MCU processing core and the peripheral devices at the maximum speed according to the data sheet, which is 84 MHz for the core and most internal clocks, and 42 MHz for the PCLK1 (that is, APB1 peripheral clocks). The following figure shows the appropriate prescaler values. Make sure that all resulting clock frequencies (the values on the very right of the graph) are correct.

Figure 9: System Clock Configuration in the IDE



3.2 S2PI (= SPI + GPIO) Layer

The S2PI layer is a combination of SPI and GPIO. It concerns all data lines to the ToF sensor.

Step 6. SPI Basic Setup

The first task is to determine or identify the GPIO lines that are connected to the device.

For the SPI connection alone, you need four GPIO pins directly to address the device:

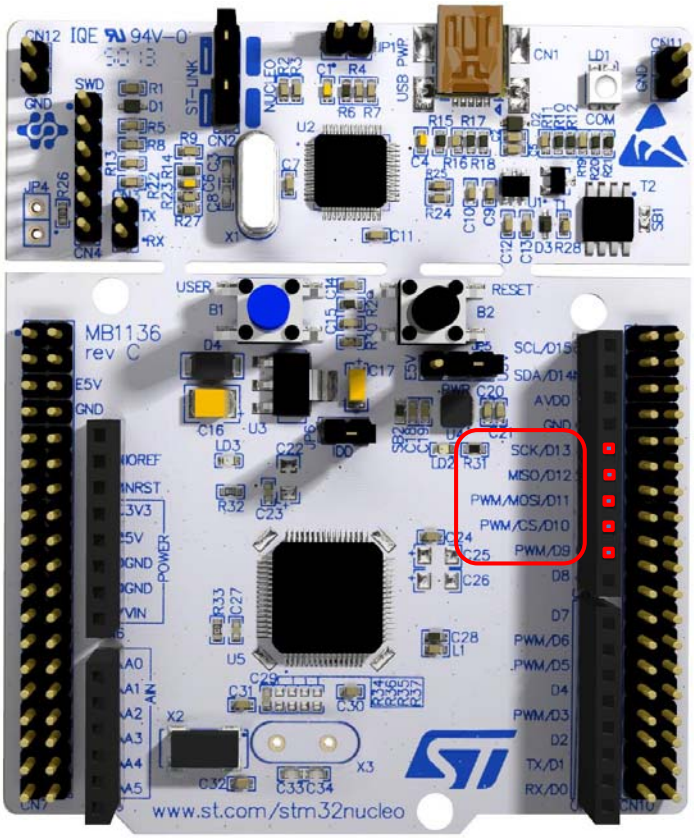
1. SPI clock (SCK)
2. SPI master in/slave out (MISO)
3. SPI master out/slave in (MOSI)

4. SPI slave select (SS or NSS), usually called chip select (CS) by the slave

Because the SPI interface can only be operated by the microcontroller as master, an additional input GPIO line is required to allow the device to signal when the requested data is available. From the board layout, this pin is D9 on the board, which maps to PC7 on the MCU. It is called IRQ in the following section, and is active low.

From the device adapter on the board, these can be mapped to the external names that they correspond to as shown in the following figure.

Figure 10: Nucleo-F401RE Board View



The GPIO lines can be determined from the board specification.

Table 1: SPI GPIO Mappings

Function	Marking on the Board	External Pin	GPIO on the MCU
SCK	SCK/D13	D13	PA5
MISO	MISO/D12	D12	PA6
MOSI	PWM/MOSI/D11	D11	PA7
NSS	PWM/CS/D10	D10	PB6
IRQ	PWM/D9	D9	PC7

Usually, several SPI controllers are on a microcontroller, so the correct one must be chosen. Here, the controller is identified with the vendor's documentation as SPI1. However, although one of the printed names of the board is CS, NSS of SPI1 corresponds to GPIO PA4, not PB6.

Unfortunately, this means that the NSS must be set up and operated manually. On the other hand, you could attach more than one slave to the same SPI interface; for example, another AFBR-S50 device. This, however, is out of the scope of this document.

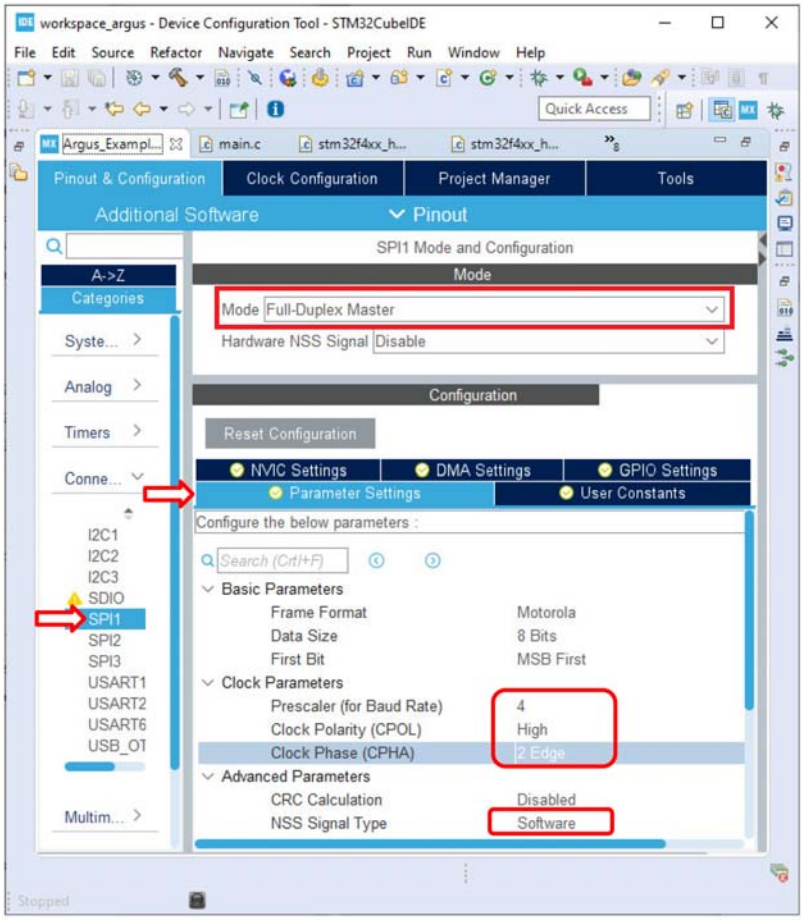
If the device is attached to the native NSS of your board, you can choose this NSS to be operated by hardware. In this case, skip all of the following steps for special handling of NSS using software. On this board, hardware operation must be disabled.

To set up the physical parameters of the SPI interface, you need to know the operation parameters of the device. The user manual of the AFBR-50 describes these as SPI mode 3 (CPOL=1, CPHA=1). This means that the clock polarity should be high (in idle state), and the data is captured on the second (rising) edge of the clock signal.

In addition, the SPI implementation of the STM F401RE only allows the SPI clock speed to be the system clock speed (which is also the MCU clock speed) divided by a power of 2, the prescaler. Because the system clock speed is 84 MHz, choose a prescaler of 4 to yield an SPI speed of 21 MHz.

All of these parameters can be chosen in the IDE as shown in the following figure.

Figure 11: SPI Base Settings in the IDE

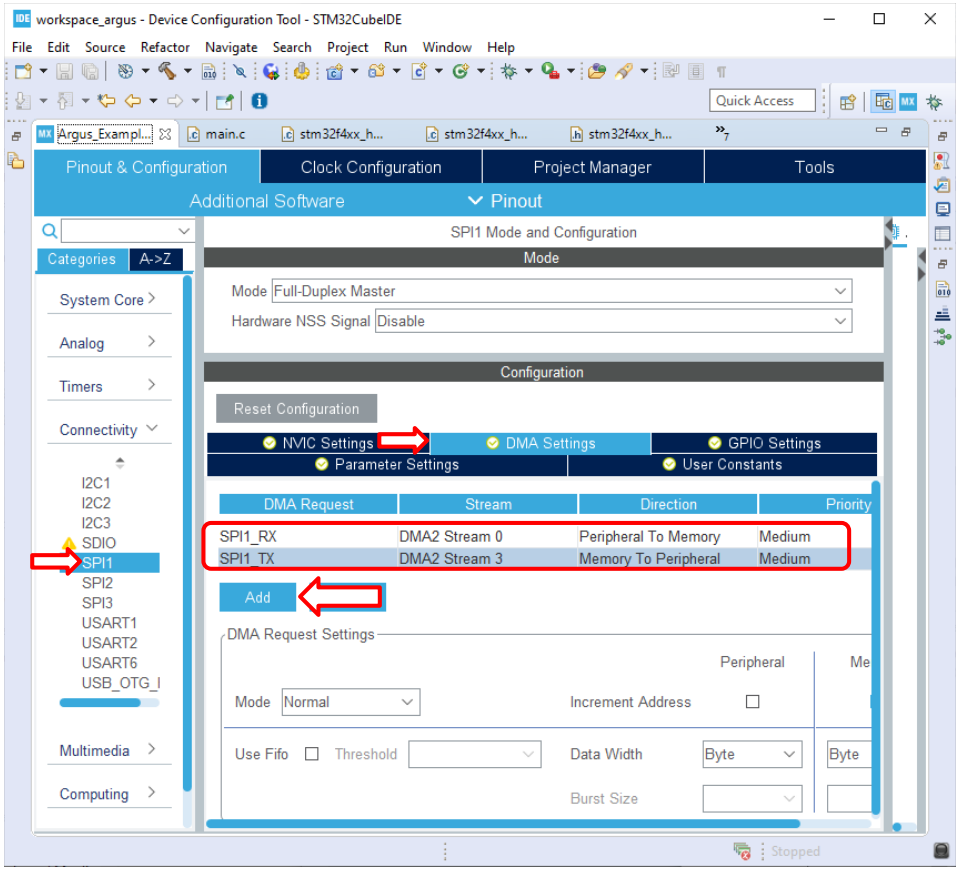


Step 7. SPI DMA Setup

With an SPI speed of 21 MHz, the data transfer rate is very high, and therefore, the transfer mechanism for the SPI data should be direct memory access (DMA).

This requires additional configuration. Usually, two independent channels for data transmission (TX) and reception (RX) must be set up. No special configuration is necessary for these channels, so they can be activated in the IDE by clicking **Add** twice and selecting each channel.

Figure 12: : SPI DMA Settings in the IDE



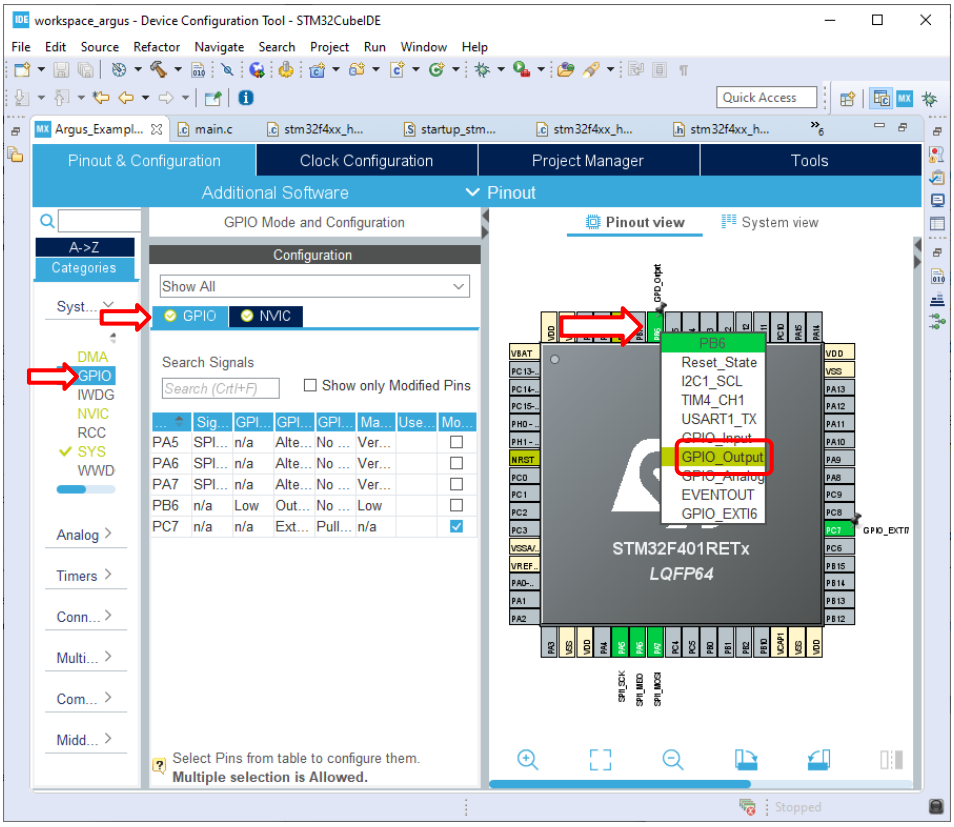
Step 8. NSS/IRQ GPIO Setup

Now, the two remaining GPIOs must be set up manually.

Navigate to the GPIO setup page by selecting **Pinout & Configuration** > **Categories** > **System Core** > **GPIO**.

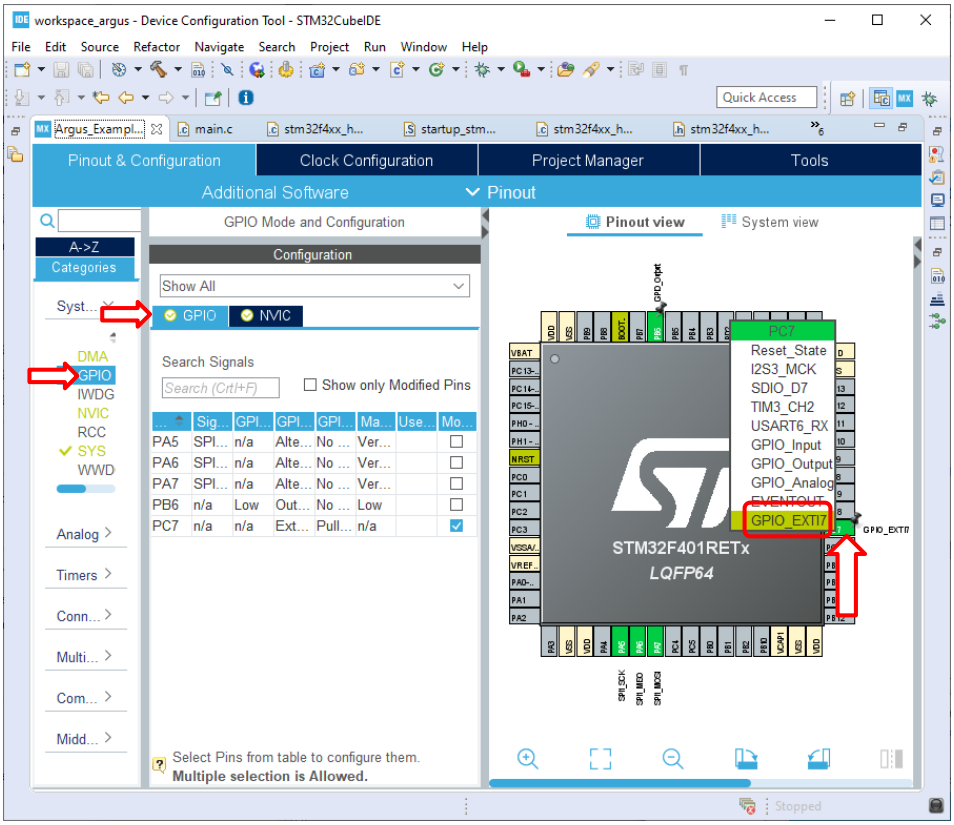
For the NSS GPIO line, the type is set to GPIO output line on the CS pin (PB6) identified by the board review.

Figure 13: NSS GPIO Type Selection in the IDE



For the interrupt GPIO line, the type is set to external interrupt on the IRQ pin (PC7) identified by the board review.

Figure 14: IRQ GPIO Type Selection in the IDE



Now the detailed settings for these two GPIO pins can be modified.

For the NSS GPIO line (PB6), all other parameters are simply selected as for the SPI output lines:

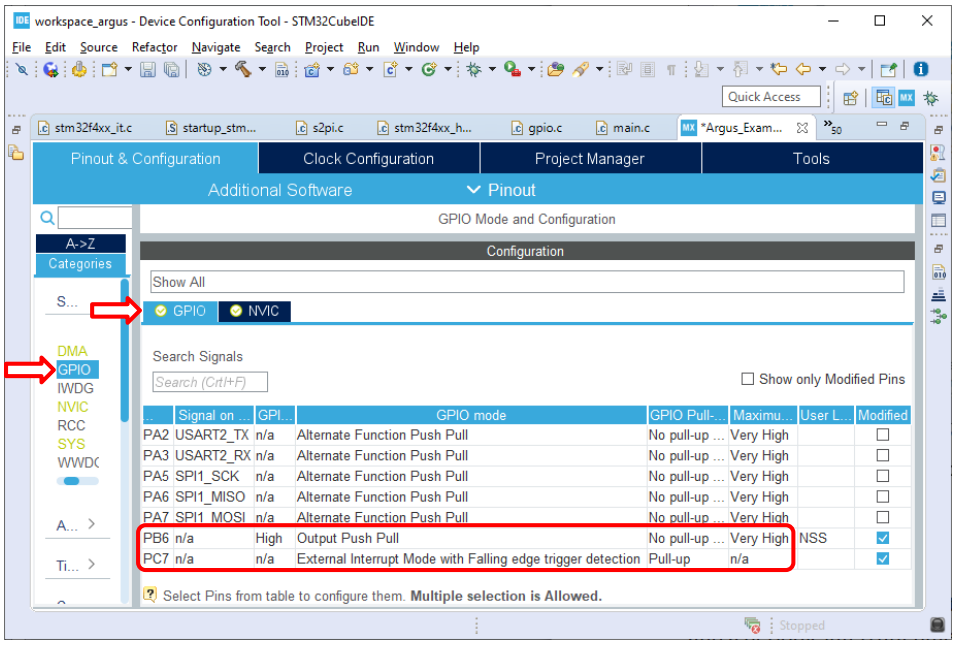
- GPIO output level = **High**
- GPIO Mode = **Output Push Pull**
- GPIO Pull-up/Pull-down = **No pull-up and no pull-down**
- Maximum output speed = **Very High**

The IRQ line (PC7) is an active low input line, so it should be pulled high by default. In addition, this GPIO must be activated and configured as an external input triggering an interrupt on the falling edge. Use the following settings:

- GPIO Mode = **External Interrupt Mode with Falling edge trigger detection**
- GPIO Pull-up/Pull-down = **Pull-up**

The following figure shows the described configuration in the IDE.

Figure 15: IRQ and NSS GPIO Pin Settings in the IDE



3.3 Timer Layer

The timer layer implements two timers: a lifetime counter (LTC) for time measurement duties and a periodic interrupt timer (PIT) for the triggering of measurements on a time-based schedule.

CAUTION! The lifetime counter is mandatory to heed the eye-safety limits. This timer must be set up carefully to guarantee the laser safety to be within Class 1.

3.3.1 Lifetime Counter (LTC)

Set up the lifetime counter to deliver the current time in microseconds, with microsecond resolution.

Because it would not be advisable to trigger the SysTick interrupt that frequently, this counter is based on hardware timers.

However, even if a 32-bit timer is used, it wraps after 4294.967296 seconds, which is a little more than one hour of operation. To avoid this, two 32-bit timers are chained together. In this example, the first timer represents the fractional part of time (that is, microseconds) and wraps after 1,000,000 ticks. Each tick is exactly one microsecond. The second timer represents the integer part of time (that is, seconds). It is triggered by the first timer upon restart.

This platform provides two 32-bit timers, TIM2 and TIM5, that are used for the lifetime counter.

NOTE: If your hardware does not have counters of enough width, or not enough counters, you have other options that require additional code:

- Chain more counters to replace one 32-bit by two 16-bit timers.
- Chain a 32-bit and a 16-bit timer and use the full 32-bit span for the first counter, and determine the seconds in the code (will wrap after more than 9 years).

- If the preceding options are not feasible, use a 32-bit or two chained 16-bit timers, and compare the result to the previously read counter value. Assume that the counter has only wrapped once if the newly read value is smaller than the previous one, and add 4,294,967,296 microseconds to the counter value.

Step 9. Setting Up the First LTC Timer

Because the STM32F401 provides enough timers, you use the first available 32-bit timer for the first LTC timer, which is TIM2.

To achieve this configuration, set up the timer as follows:

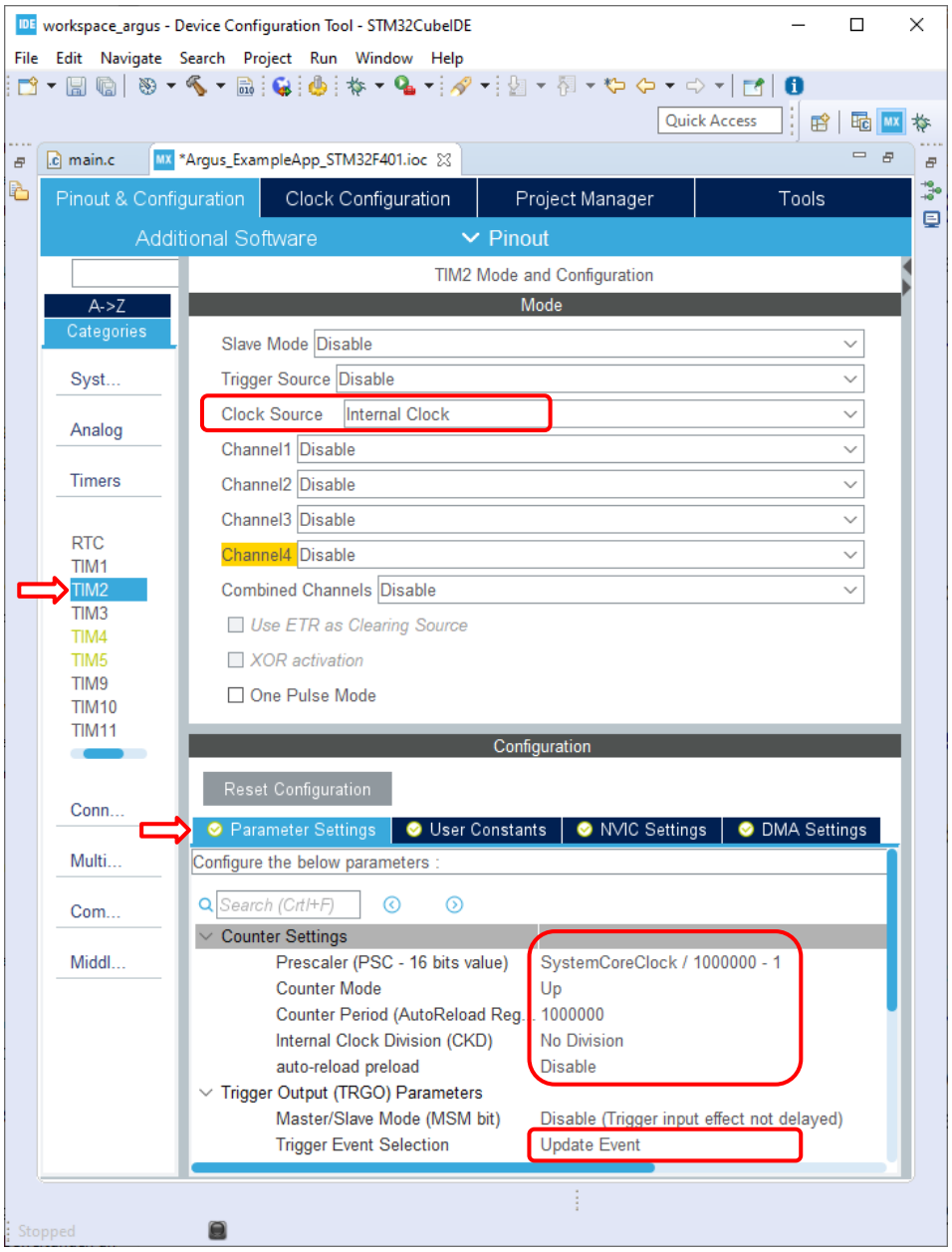
- It should be in normal mode and be triggered by the internal clock.
- The counting direction should be UP.
- The prescaler value is set in a way that generates the counter value in microseconds:

  Because the counter is driven by the system clock, this value is calculated from the system clock frequency:

  SystemCoreClock / 1000000 – 1
- The counter period should be set to 1000000.
- An output trigger is generated when the counter period value is reached.

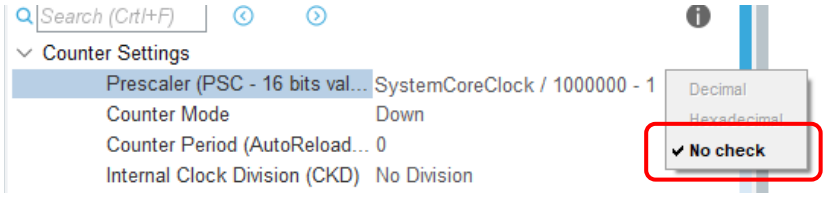
In the IDE, the following figure shows the setting for this counter.

Figure 16: Configuration of the First LTC Timer in the IDE



NOTE: To enter an expression into the **Prescaler** parameter field, the parameter check has to be disabled before. Otherwise, the expression is cleared to 0. To disable the parameter check, click on the Gear-Wheel symbol that appears near the **Value** field and select **No check**.

Figure 17: Disabling the Parameter Check for the Prescaler Setting



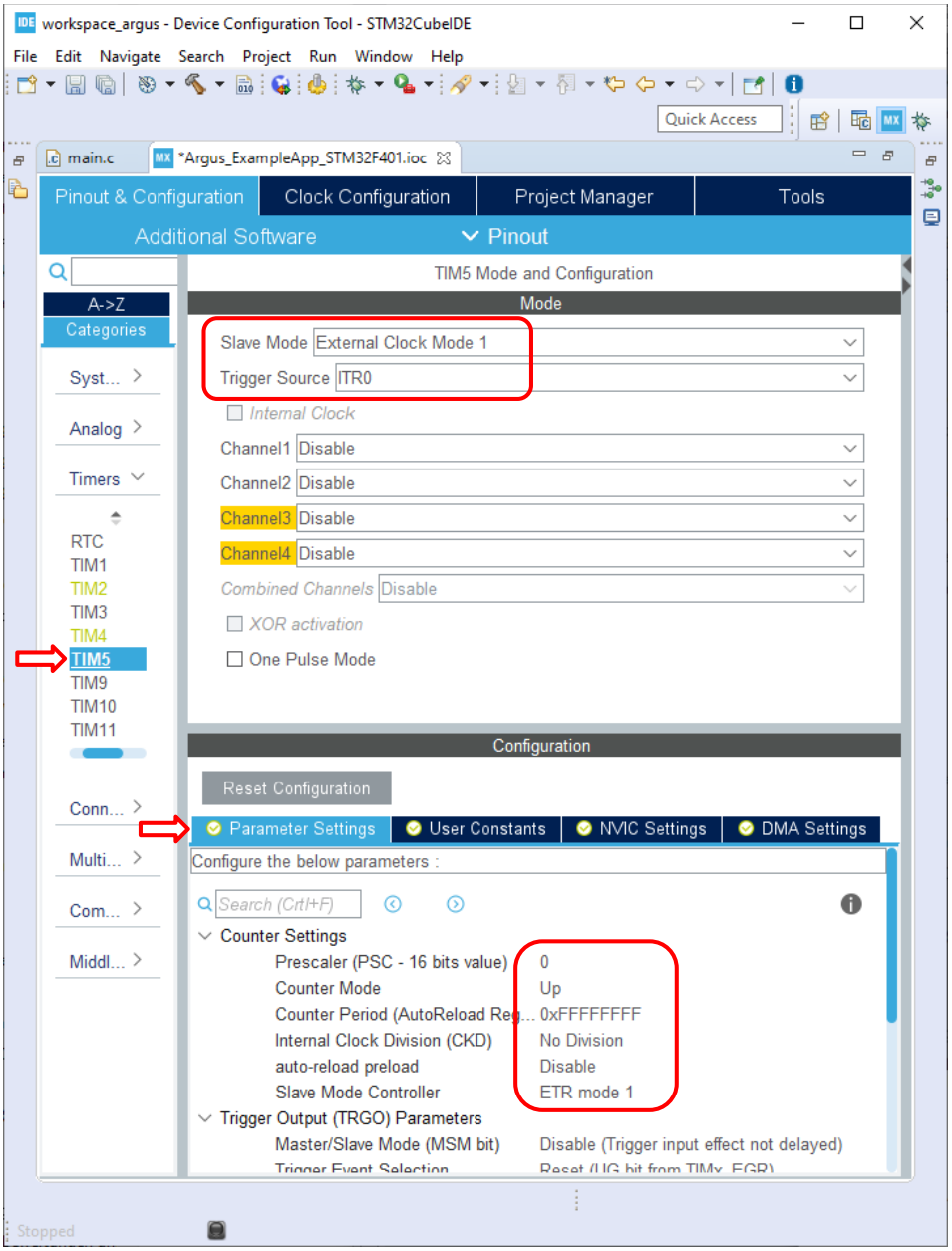
Step 10. Setting Up the Second LTC Timer

Now, you set up the second 32-bit timer for the LTC, which is TIM5.

The second timer is set up as follows:

- It should be in slave mode, with the external source used as a trigger.
- The counting direction should be UP.
- The event generated by the first timer should be used as a trigger (ITR0 according to the manual).
- A prescaler is not required (0).
- The counter should count to the maximum value (0xFFFFFFFF).

Figure 18: Configuration of the Second LTC Timer in the IDE



3.3.2 Periodic Interrupt Timer (PIT)

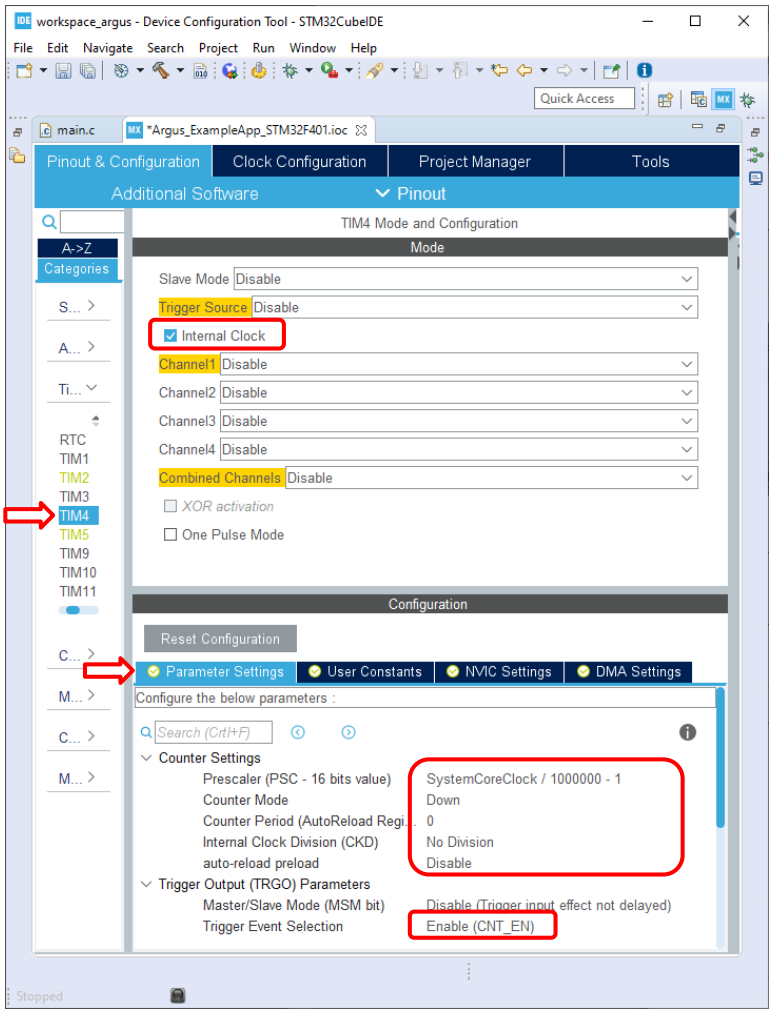
The periodic interrupt timer triggers the ToF measurement periodically. By using a dynamic configuration of the prescaler and the counter value, a 16-bit counter should be sufficient. The example uses the 16-bit counter TIM4.

This timer is set up as follows:

- It should be in normal mode.
- The counting direction does not really matter, DOWN is selected.

- The prescaler and counter values are set dynamically later when activating the counter.
- An interrupt is generated when the counter reaches 0.

Figure 19: Configuration of the PIT Timer in the IDE



3.4 Optional: UART

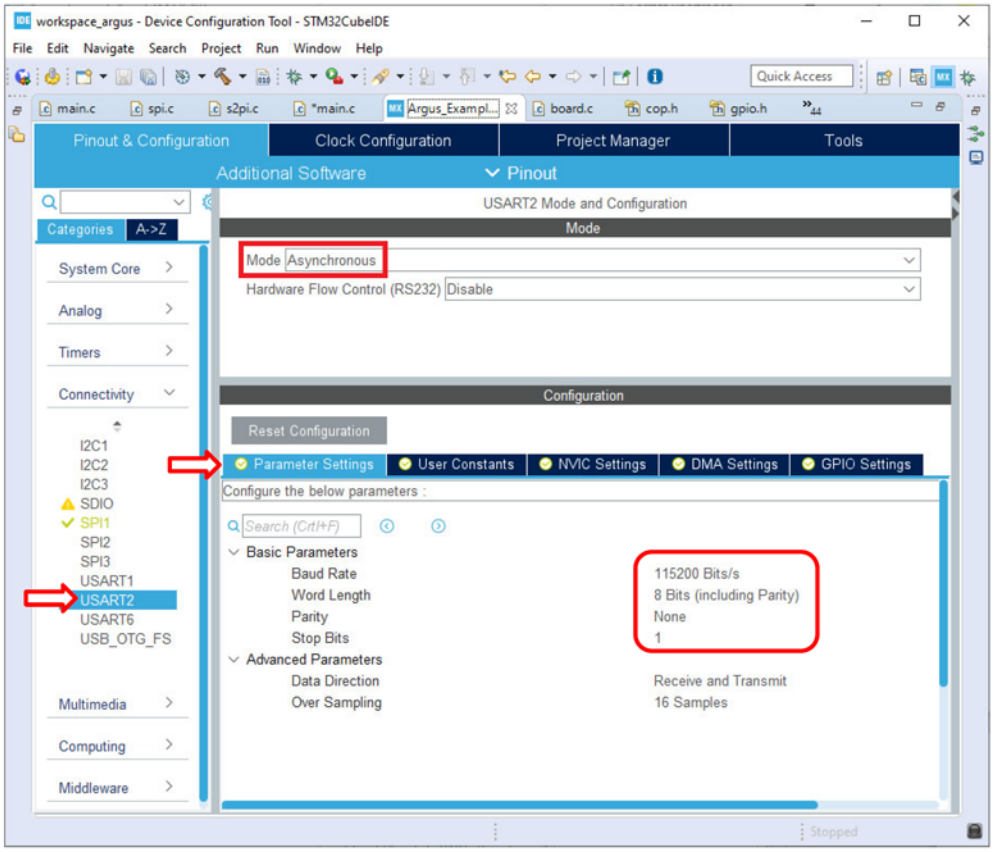
To be able to log from the example application, the UART interface must be set up.

This does not result in an actual serial line, but it is a virtual serial line provided over the USB port.

For the UART, the default settings are a good starting point (8N1). The default baud rate of 115,200 could also be increased to be able to log even at high frame rates.

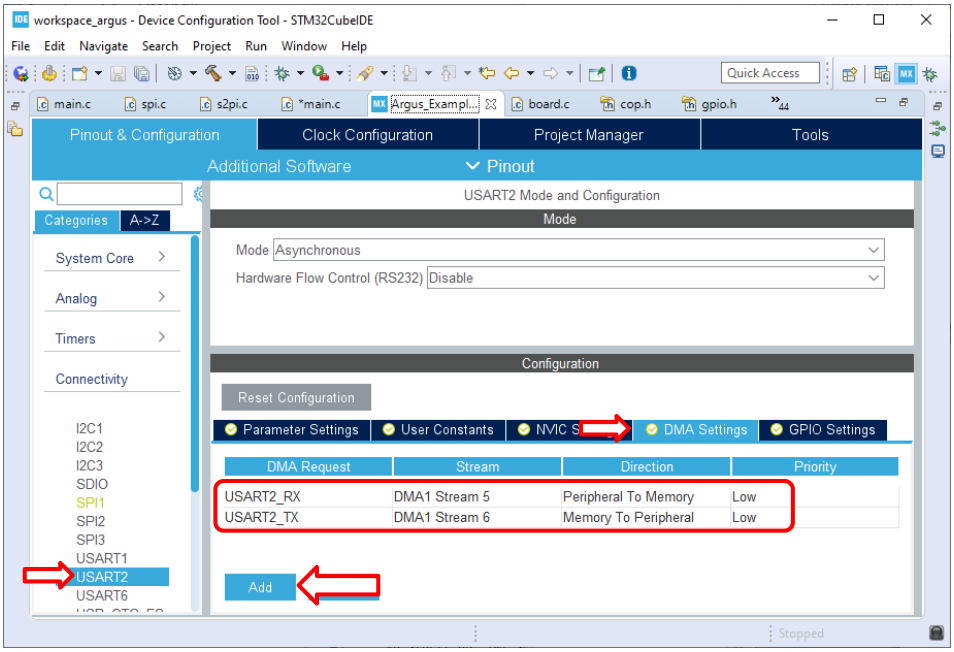
The configuration in the IDE looks like the following figure.

Figure 20: UART Configuration in the IDE



As for SPI, you want to be able to use DMA for sending, so the DMA must also be set up.

Figure 21: DMA Settings for UART in the IDE



3.5 Interrupt Configuration

Now all devices have been set up, but they also depend on interrupts that have to be configured:

- The DMA transmit and receive complete interrupts for the SPI interface
- The external interrupt for the AFBR-S50 device
- The timer interrupt for the Periodic Interrupt Timer
- The DMA transmit complete interrupt for the UART interface

All of these interrupts can be assigned with a preemptive priority: interrupts with a lower priority number have actually a higher priority.

Of the previously mentioned interrupts, the SPI DMA interrupts have the highest priority. Because the complete indication in the STM HAL layer for a SPI transmit and receive operation is connected to the SPI receive DMA, the transmit complete indication should be handled first, and therefore has the highest priority (after crucial system interrupts, which should be left at the maximum priority 0).

Next, you want to have the indications to read data (external interrupts) or start a new measurement (timer interrupt). Finally, you have the UART DMA complete interrupt, which is not critical in timing.

After that, you have the system interrupts that are not critical. Here, this is only the SysTick interrupt for the internal clock with a frequency of 1000 Hz.

Therefore, the following table shows the interrupt configuration.

Table 2: Interrupt Priorities

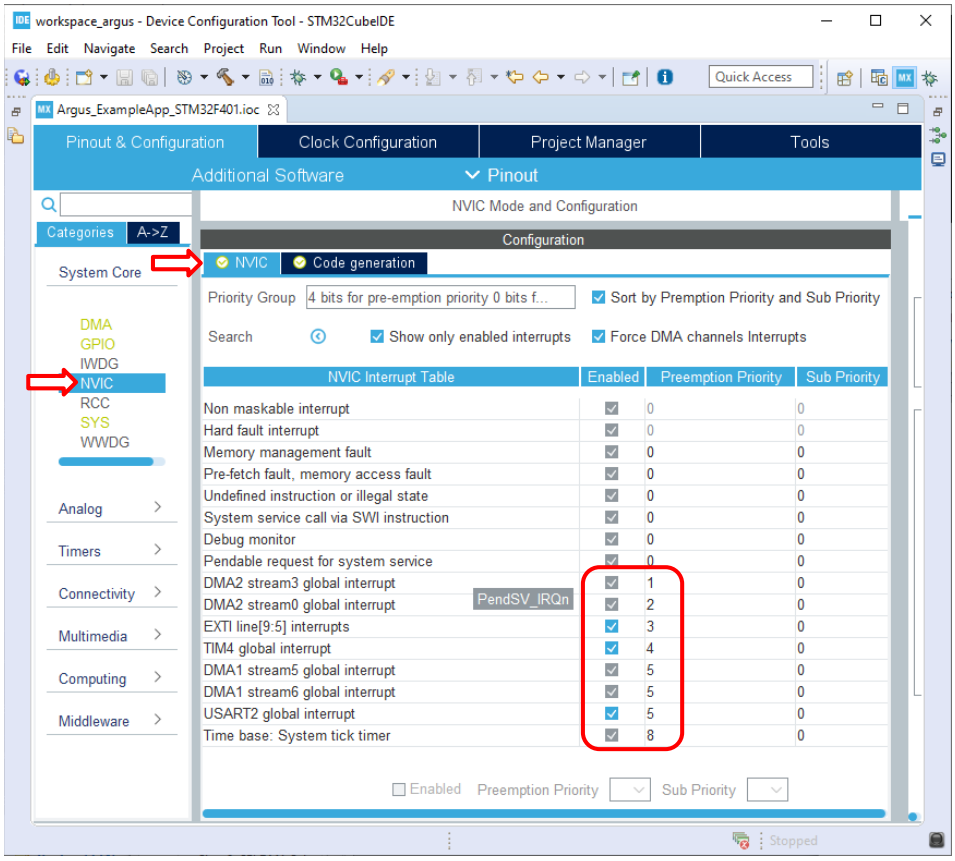
Interrupt	Priority
Critical System Interrupts	0
SPI DMA Transmit Complete	1
SPI DMA Receive Complete	2
External Interrupt	3
Periodic Interrupt Timer	4
UART Interrupts	5
SysTick Timer Interrupt	8

NOTE: If you have less interrupt priorities available on your system, it is not required to use such a differentiated interrupt scheme: External and PIT can share the same priory, UART and SysTick also, and on most platforms, SPI Tx and Rx interrupts can have the same priority, too.

Step 11. Configuring the Interrupts in the IDE

The configuration in the IDE for these values look like those in the following figure.

Figure 22: Interrupt Priority Configuration in the IDE (1)

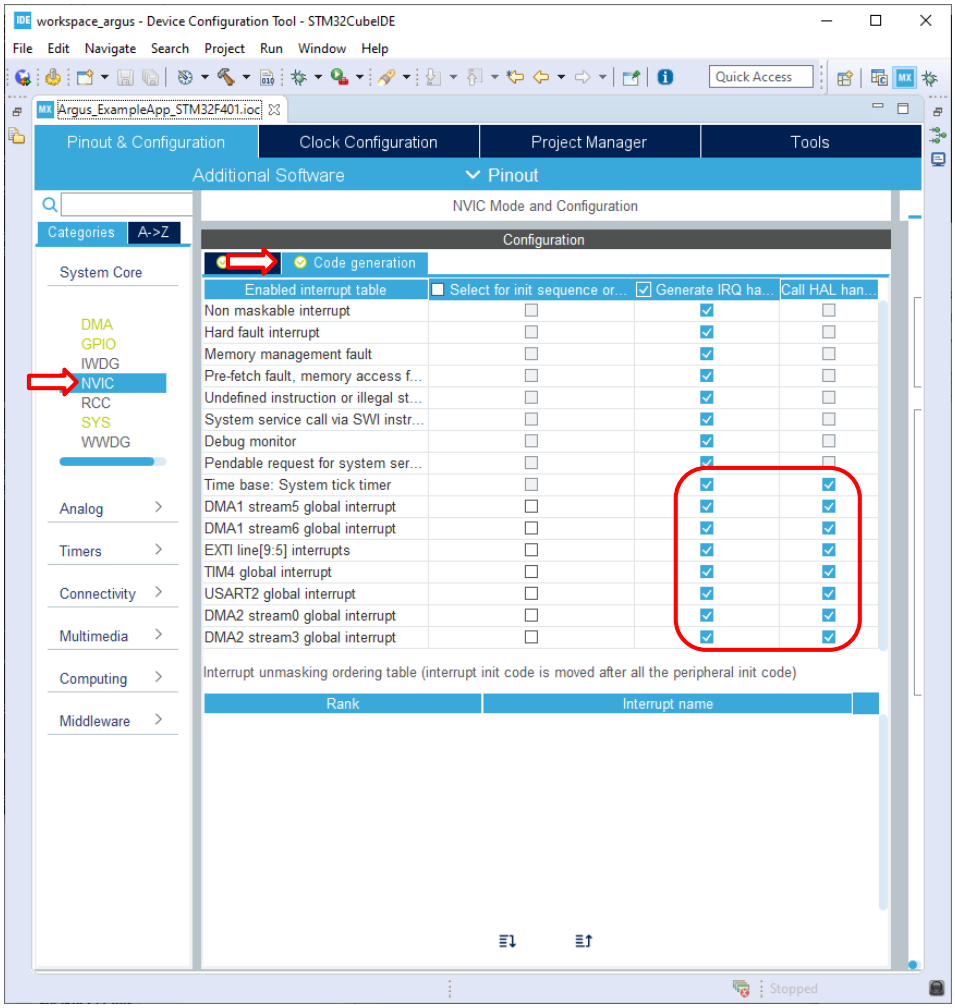


NOTE:

- To enable certain interrupts, you might need to disable the **Show only enabled interrupts** option on top of the list to see all available interrupts.
- The NVIC configuration warning that states **Preemption priorities have been reset to 0 as FREERTOS is deselected.** can be ignored.

Additionally, to simplify the implementation of the following interrupts, the handlers must be automatically generated and added to the project.

Figure 23: Interrupt Priority Configuration in the IDE (2)



3.6 Code Generation

The final step is now to trigger the code generation function from the API.

Step 12. Setting the Code Generation Options

There are two relevant options for the code generation:

- As we want to adapt a different project, the `main()` function should not be generated automatically, because this would be conflicting otherwise.
- The hardware initializations should be generated in separate files to identify them more easily.

These settings can be made in the **Project Manager** tab.

Figure 24: Preventing the Generation of main()

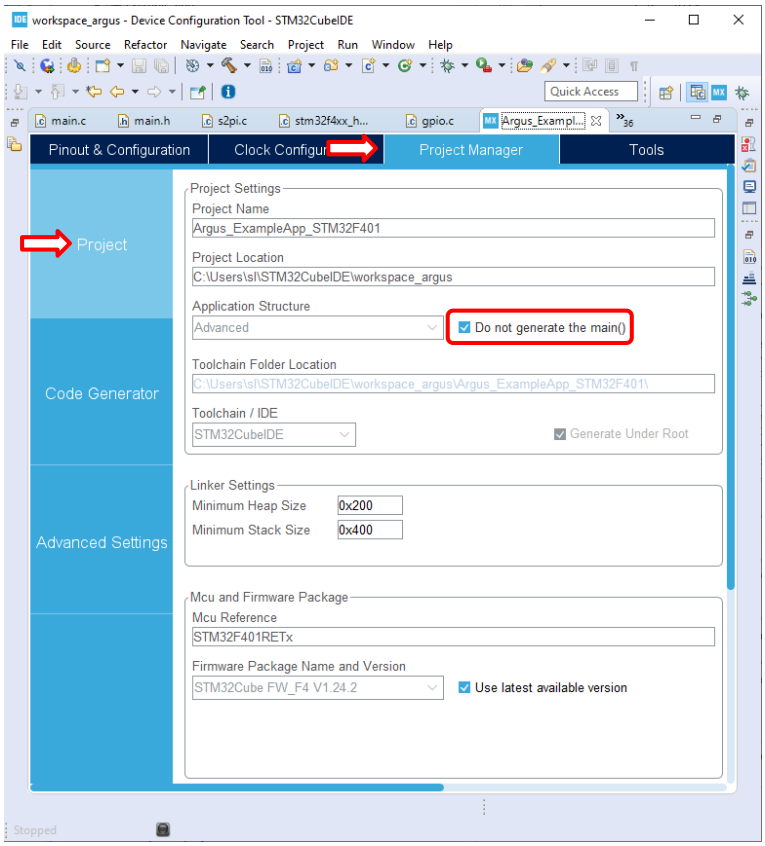
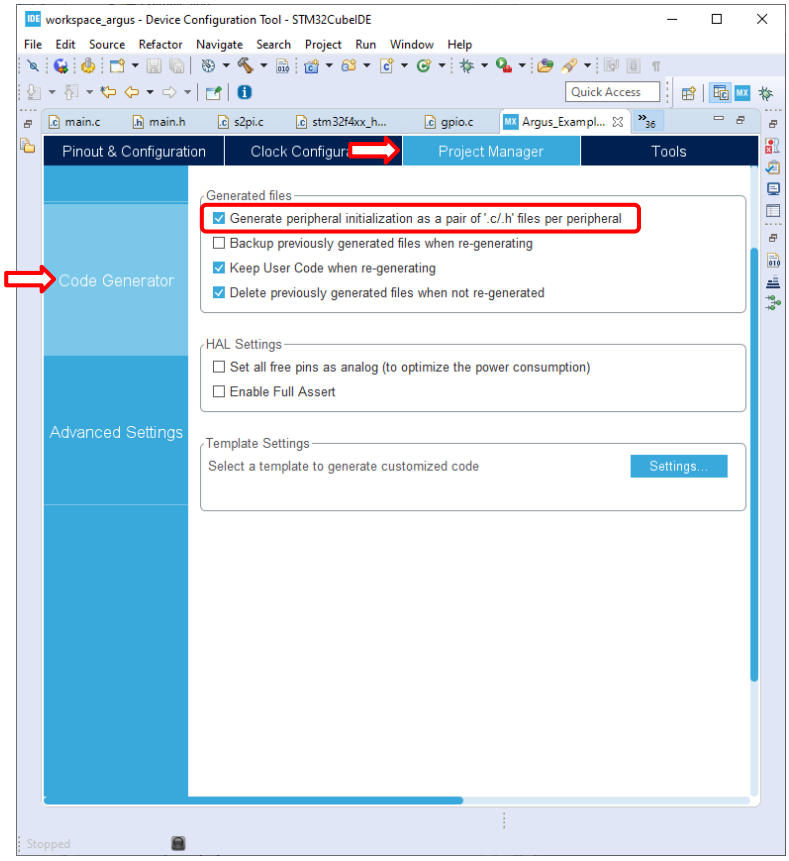


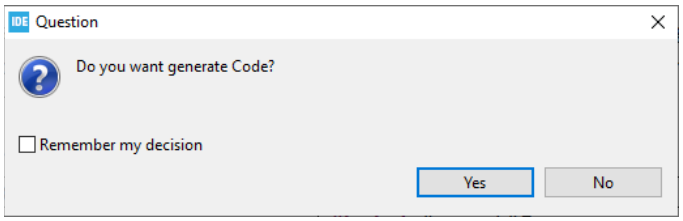
Figure 25: Forcing Separate Initialization Files



Step 13. Performing the Code Generation

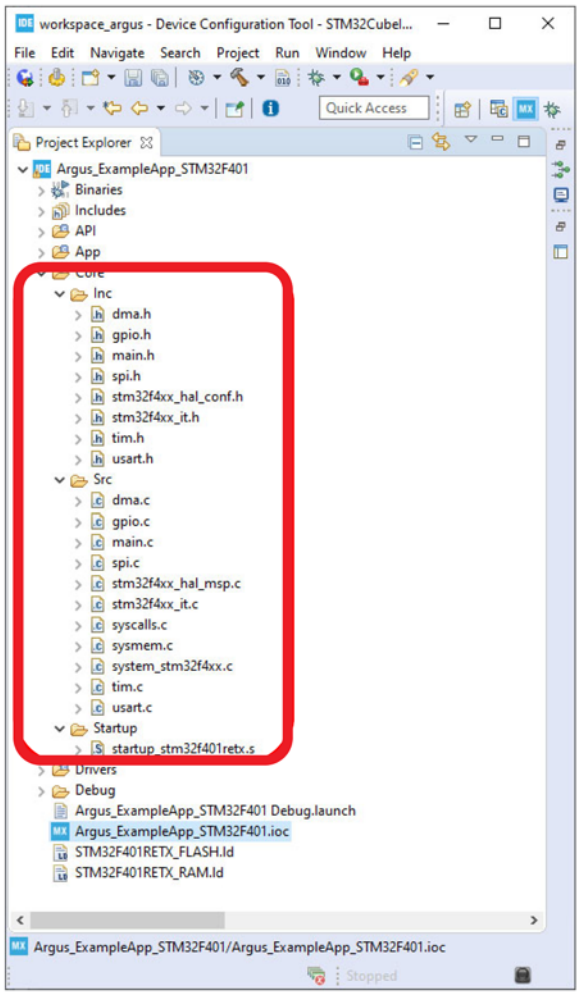
The code generation can simply be started by saving the configured setup and then confirming the code generation in the dialog box, or it can also manually triggered by pressing **Alt** + **K**.

Figure 26: Confirming the Automatic Code Generation



The code generation now creates several files in the `Core` folder.

Figure 27: Generated Files in the IDE



These files contain the configurations shown in the following table.

Table 3: Description of Source Files Generated by the IDE

Path	Functions to Be Called	Description
`Core/Inc/dma.h` `Core/Src/dma.c`	`MX_DMA_Init()`	DMA initialization for all peripherals (UART and SPI), with interrupt settings
`Core/Inc/gpio.h` `Core/Src/gpio.c`	`MX_GPIO_Init()`	Initialization for all GPIO lines not assigned to other controllers (chip select and external interrupt), with interrupt settings for the latter
`Core/Inc/spi.h` `Core/Src/spi.c`	`MX_SPI1_Init()`	Setup of the SPI controller and the used GPIO lines operated by the SPI controller (CLK, MISO, MOSI)

`Core/Inc/tim.h`
`Core/Src/tim.c`
	`MX_TIM2_Init()`
	`MX_TIM4_Init()`
	`MX_TIM5_Init()`
		Setup of the timers used for the lifetime counter (LTC) and the periodic interrupt timer (PIT), including the interrupt settings for the latter

`Core/Inc/usart.h` `Core/Src/usart.c`	`MX_USART2_UART_Init()`	Setup of the UART controller and the used GPIO lines operated by the UART controller

Table 3: Description of Source Files Generated by the IDE (Continued)

Path	Functions to Be Called	Description
`Core/Inc/main.h` `Core/Src/main.c`	`SystemClock_Config()`	Setup of the board clock configurations
`Core/Src/stm32f4xx_it.c`	`MX_DMA_Init()`	Generated interrupt handlers for the configured interrupts, forwarding the interrupts to the STM32 hardware abstraction layer (HAL)
`Core/Src/syscalls.c` `Core/Src/sysmem.c`		Minimum required system calls and system memory calls to support the C standard
`Core/Src/stm32f4xx_hal_msp.c`		Internally called function for the hardware abstraction layer setup.
`Core/Src/system_stm32f4xx.c`		Generated system specific initialization, automatically called from startup code
`Startup/startup_stm32f401retx.s`		Generated board startup code

Chapter 4: Adapting the Generated Data to the Argus API

The next steps are to create the functions required by the API hardware layer interfaces to satisfy the requirements of the AFBR-S50 library.

To simplify the process, as much as possible from the reference implementation in the SDK is reused, but without modification of the original files.

Step 14. Adding the Required Include Paths

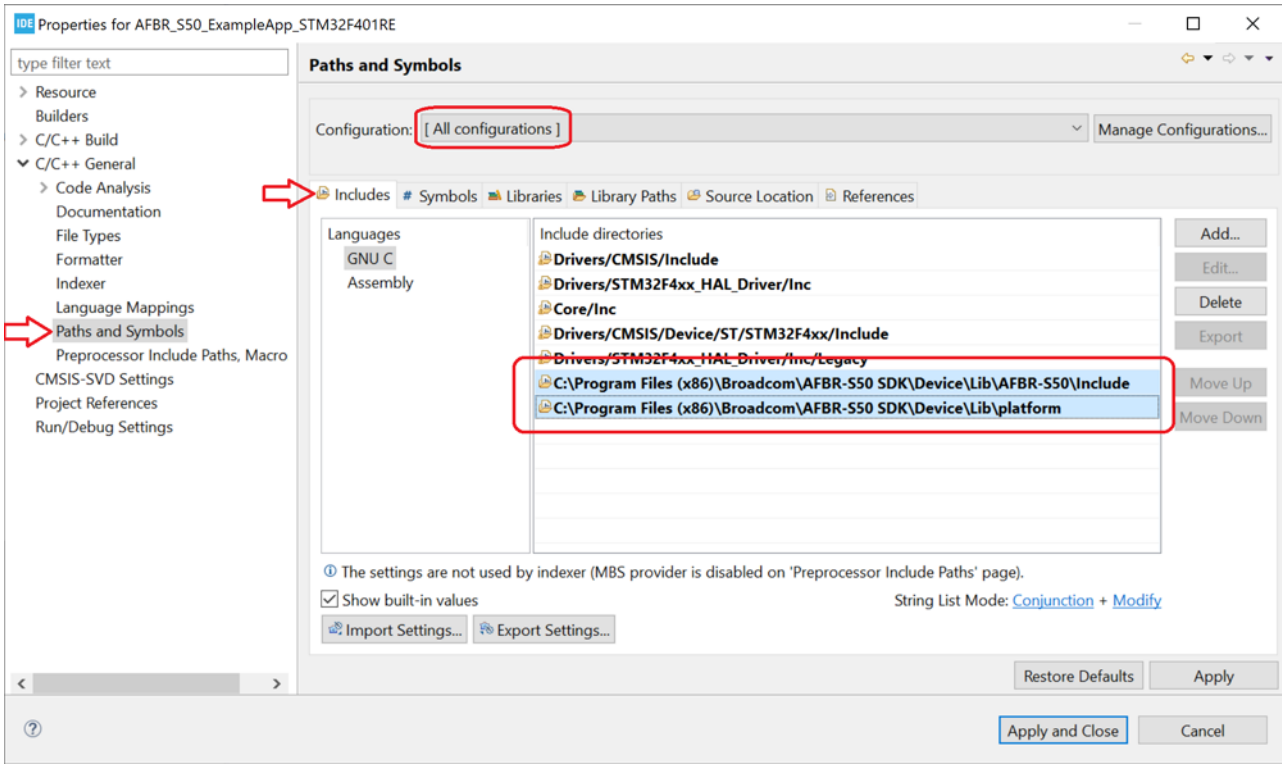
To be able to find the include files, they must be set up in the include path. The following table shows the paths that are to be added.

Table 4: Additional Include Paths

Path	Description
`C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Lib\AFBR-S50\Include`	Path to the API include files
`C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Lib\AFBR-S50\platform`	Path to the platform API files

NOTE: The actual path may change depending on the actual installation directory.

Figure 28: Added Include Paths in the IDE



Access the project configuration menu by right-clicking on the project in the Project Explorer and selecting **Properties**.

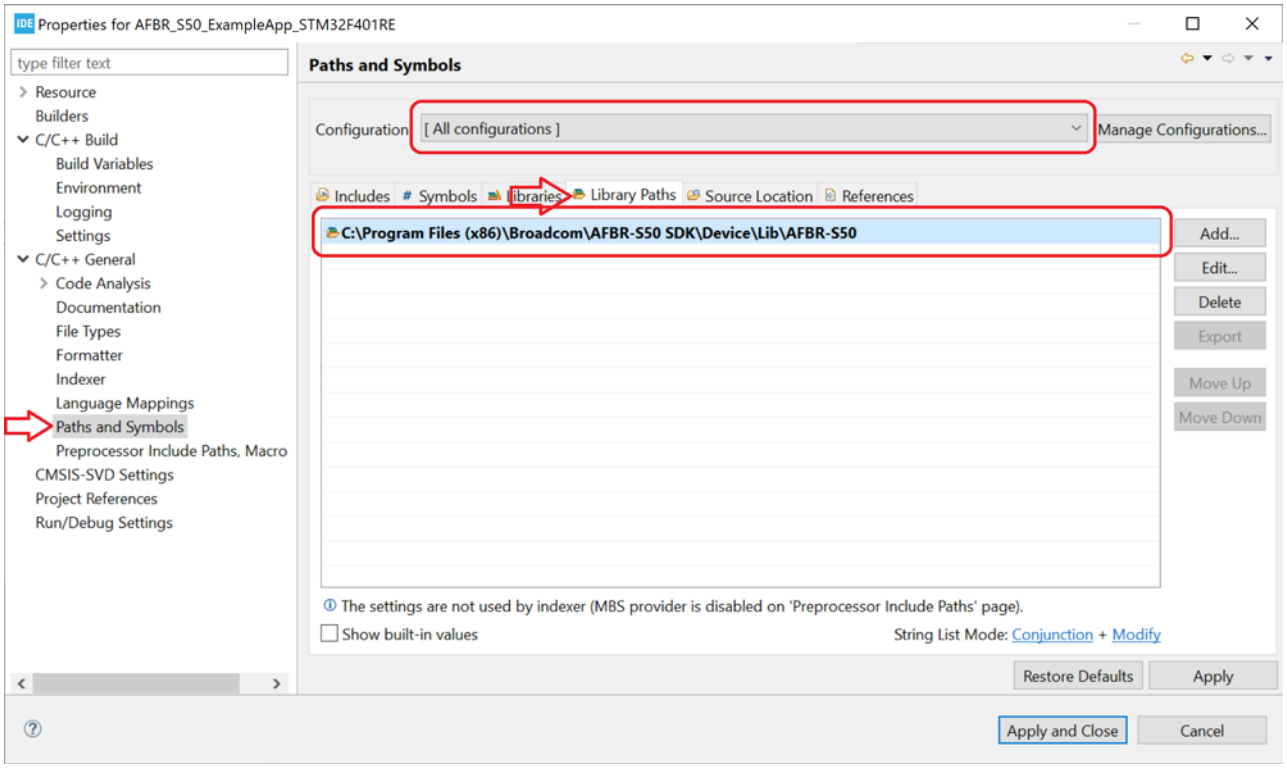
NOTE: Make sure to select **[All configurations]** before setting the path.

Step 15. Adding the AFBR-S50 Library

To be able to link against the AFBR-S50 library, it must be added to the build.

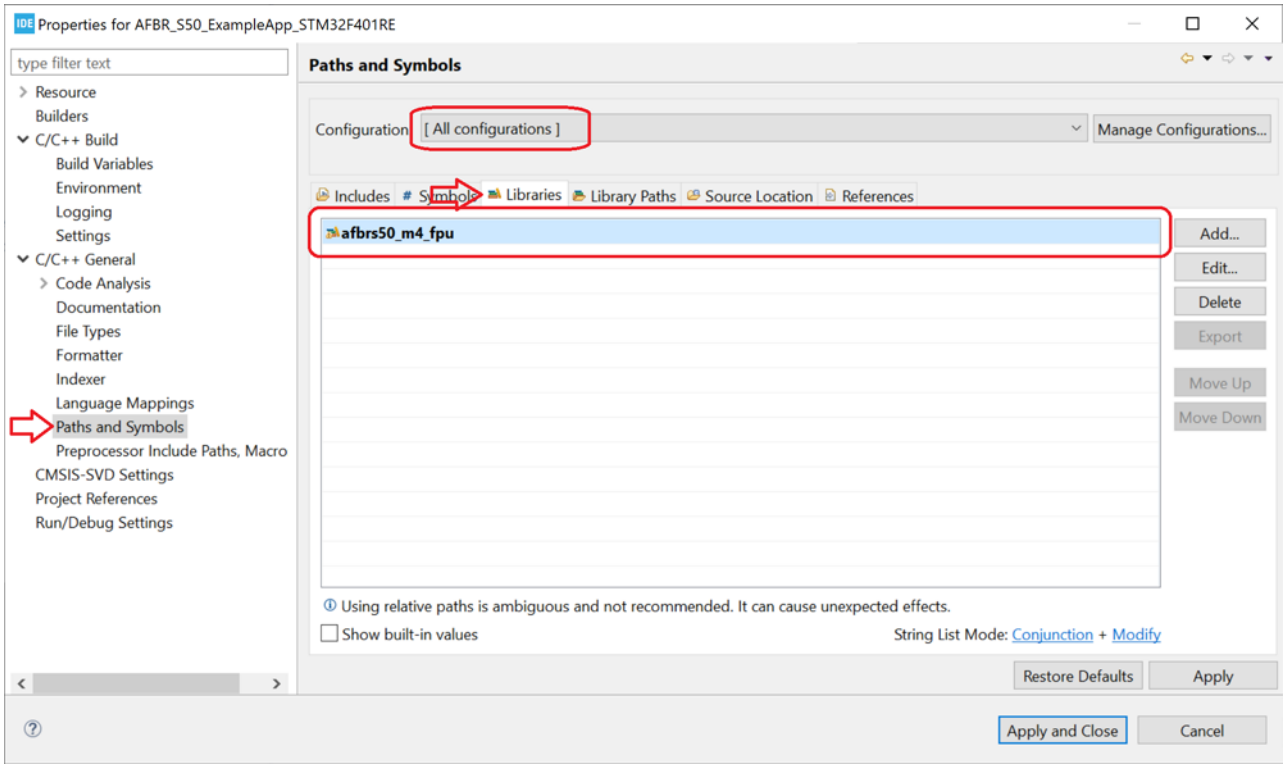
First, add the library path similar to the include path.

Figure 29: Added Library Path in the IDE



Now, add the library itself.

Figure 30: Added AFBR-S50 Library in the IDE



The library name depends on the actual architecture. In case of STM32F401RE, the Cortex-M4 incl. Hardware Floating-Point unit is used. Find more details on the different library variants in the MCU Porting Guide > Architecture Compatibility section in the *API Reference Manual*.

NOTE: In SDK v1.0.x, only a Cortex-M0 library was available. Still all Cortex-Mx variants are supported using the soft floating-point ABI. See Section A.1, Setting Up Floating-Point ABI for Soft Floating Point Usage.

4.1 IRQ API

The IRQ API is simple and only provides the AFBR-S50 library and the following hardware implementations with a way to lock all interrupts temporarily.

Step 16. Creation of the IRQ File

The IRQ API is implemented in a new file, `irq.c`, in the `API` folder. This file should be created as an empty source file.

Step 17. Implementing the IRQ Locking

The implementation is simple and does not have many dependencies.

Listing 1: File "API/irq.c"

```
#include <assert.h>
#include "main.h"

/*! Global lock level counter value. */
static volatile int g_irq_lock_ct;


/*!***************************************************************************
 * @brief    Enable IRQ Interrupts
 *
 * @details    Enables IRQ interrupts by clearing the I-bit in the CPSR.
 *             Can only be executed in Privileged modes.
 *
 * @return    -
 ****************************************************************************/
void IRQ_UNLOCK(void)
{
    assert(g_irq_lock_ct > 0);
    if (--g_irq_lock_ct <= 0)
    {
        g_irq_lock_ct = 0;
        __enable_irq();
    }
}


/*!***************************************************************************
 * @brief    Disable IRQ Interrupts
 *
 * @details    Disables IRQ interrupts by setting the I-bit in the CPSR.
 *             Can only be executed in Privileged modes.
 *
 * @return    -
 ****************************************************************************/
void IRQ_LOCK(void)
{
    __disable_irq();
    g_irq_lock_ct++;
}
```

4.2 S2PI API

Step 18. Creation of the S2PI File

The S2PI API is implemented in a new file, `s2pi.c`, in the `API` folder. This file should be created as an empty source file.

Step 19. Adding the S2PI Includes

First, the headers declaring the API functions to be implemented are included. This means the API header and the header for the generated implementation.

Listing 2: File "API/s2pi.c" – Include statements

```
#include "dma.h"
#include "gpio.h"
#include "spi.h"
#include "driver/irq.h"
#include "driver/gpio.h"
#include "driver/s2pi.h"
```

Step 20. Implementing the S2PI Data Structures

Next, a data structure is defined that holds all the data for one SPI module.

The following information is contained:

- The current status of the device
- The callback function after an SPI transfer
- A parameter for this callback function
- The callback function after an external interrupt from the device
- A parameter for that callback function
- The alternate mode of the GPIOs for SPI mode
- A mapping of all used logical pins to GPIO pins and ports

The first part is a mapping type for the pins and ports.

Listing 3: File "API/s2pi.c" – SPI GPIO pin mapping

```
/*! A structure that holds the mapping to port and pin for all SPI modules. */
typedef struct
{
    /*! The GPIO port */
    GPIO_TypeDef * Port;

    /*! The GPIO pin */
    uint32_t Pin;
}
s2pi_gpio_mapping_t;
```

Then there is the data structure with the parameters mentioned previously.

Listing 4: File "API/s2pi.c" – SPI data structure

```
/*! A structure to hold all internal data required by the S2PI module. */
typedef struct
{
    /*! Determines the current driver status. */
    volatile status_t Status;

    /*! Determines the current S2PI slave. */
    volatile s2pi_slave_t Slave;

    /*! A callback function to be called after transfer/run mode is completed. */
    s2pi_callback_t Callback;

    /*! A parameter to be passed to the callback function. */
    void * CallbackData;

    /*! A callback function to be called after external interrupt is triggered. */
    s2pi_irq_callback_t IrqCallback;

    /*! A parameter to be passed to the interrupt callback function. */
    void * IrqCallbackData;

    /*! The alternate function for this SPI port. */
    const uint32_t SpiAlternate;

    /*! The mapping of the GPIO blocks and pins for this device. */
    const s2pi_gpio_mapping_t GPIOs[ S2PI_IRQ+1 ];
}
s2pi_handle_t;
```

Finally, the SPI settings in this data structure are initialized in an instance containing this data.

NOTE: If multiple devices should be supported, this should be implemented as an array.

The pin names are taken from the included `platform/AFBR-S50_s2pi.h` (using `driver/s2pi.h`). The actual values can be determined from the generated `Core/Src/spi.c` file.

Listing 5: File "API/s2pi.c" – SPI data object

```
s2pi_handle_t s2pi_ = { .SpiAlternate = GPIO_AF5_SPI1,
                        .GPIOs = { [ S2PI_CLK  ] = { GPIOA, GPIO_PIN_5 },
                                   [ S2PI_CS   ] = { GPIOB, GPIO_PIN_6 },
                                   [ S2PI_MOSI ] = { GPIOA, GPIO_PIN_7 },
                                   [ S2PI_MISO ] = { GPIOA, GPIO_PIN_6 },
                                   [ S2PI_IRQ  ] = { GPIOC, GPIO_PIN_7 } } };
```

Step 21. Implementing the S2PI Initialization

Next, the timer initialization routine is implemented. The prototype is found in `driver/s2pi.h`.

The initialization function calls the generated hardware initializations for the GPIO, DMA, and SPI layers. This guide assumes that only a single SPI device is attached, so it returns an error if the requested SPI slave identifier is not the first one. The helper function to set the baud rate (defined as follows) is set to determine the settings for the baud rate.

NOTE: The comments are copied from the prototype in `platform/AFBR-S50_s2pi.h`.

Listing 6: File "API/s2pi.c" – S2PI Initialization Code

```
/*!*****************************************************************************
 * @brief   Initializes the GPIO driver and does pin muxing.
 * @details Does actually nothing, as all GPIO pins are initialized in
 *          S2PI_Init().
 * @return   -
 ******************************************************************************/
void GPIO_Init(void) {}

/*!*****************************************************************************
 * @brief   Initialize the S2PI module.
 * @details Setup the board as a S2PI master, this also sets up up the S2PI
 *          pins.
 *          The SPI interface is initialized with the corresponding default
 *          SPI slave (i.e. CS and IRQ lines) and the default baud rate.
 *
 * @param   defaultSlave The default SPI slave to be addressed right after
 *                       module initialization.
 * @param   baudRate_Bps The default SPI baud rate in bauds-per-second.
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t S2PI_Init(s2pi_slave_t defaultSlave, uint32_t baudRate_Bps)
{
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_SPI1_Init();

    if (defaultSlave != S2PI_S1)
        return ERROR_S2PI_INVALID_SLAVE;

    return S2PI_SetBaudRate(baudRate_Bps);
}
```

Step 22. Implementing the SPI Get Status Function

The current status of the SPI connection and operation mode (see the following) can also be queried.

Listing 7: File "API/s2pi.c" – SPI Status and Mode Query

```
/*!*****************************************************************************
 * @brief   Returns the status of the SPI module.
 *
 * @return  Returns the \link #status_t status\endlink:
 *           - #STATUS_IDLE: No SPI transfer or GPIO access is ongoing.
 *           - #STATUS_BUSY: An SPI transfer is in progress.
 *           - #STATUS_S2PI_GPIO_MODE: The module is in GPIO mode.
 ******************************************************************************/
status_t S2PI_GetStatus(void)
{
    return s2pi_.Status;
}
```

Step 23. Implementing the Helper Functions for the SPI Baud Rate

The baud rate is set or read by helper functions.

As the choice of the baud rate values is limited by the prescaler, the nearest possible value below the requested one is selected.

Note that a calculation based on the system clock is required. The value starts with the highest possible baud rate and decreases the prescaler until the baud rate is smaller than the desired rate or the lowest possible baud rate is reached.

The actual calculation depends on the hardware. On the STM32F401, the prescaler is always a power of two by which the peripheral clock is divided.

Listing 8: File "API/s2pi.c" – Baud Rate setting

```
/*!*************************************************************************
 * @brief   Sets the SPI baud rate in bps.
 * @param   baudRate_Bps The default SPI baud rate in bauds-per-second.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 *          - #STATUS_OK on success
 *          - #ERROR_S2PI_INVALID_BAUD_RATE on invalid baud rate value.
 **************************************************************************/
status_t S2PI_SetBaudRate(uint32_t baudRate_Bps)
{
    uint32_t prescaler = 0;
    /* Determine the maximum possible value not greater than baudRate_Bps */
    for (; prescaler < 8; ++prescaler)
        if (SystemCoreClock >> (prescaler + 1) <= baudRate_Bps)
            break;
    MODIFY_REG(hspi1.Instance->CR1, SPI_CR1_BR, prescaler << SPI_CR1_BR_Pos);
    return STATUS_OK;
}
```

When the baud rate is read, the real baud rate should be returned, not the requested value.

Listing 9: File "API/s2pi.c" – Baud rate readout

```
/*!*************************************************************************
 * @brief   Gets the current SPI baud rate in bps.
 * @return  Returns the current baud rate.
 **************************************************************************/
uint32_t S2PI_GetBaudRate(void)
{
    uint32_t prescaler = (hspi1.Instance->CR1 & SPI_CR1_BR) >> SPI_CR1_BR_Pos;
    return SystemCoreClock >> (prescaler + 1);
}
```

Step 24. Implementing the SPI/GPIO Switch

Next, the switching between SPI and GPIO mode for the PINs is implemented.

As the CS (chip select) is already an ordinary GPIO, only the other pins are affected: CLK, MISO, and MOSI. Switching is does not require too much setup. The mode must be changed from alternate (SPI) mode to push-pull output mode (GPIO). The SPI settings can be taken from the automatically generated `HAL_SPI_MspInit()` in the `Core/Src/spi.c` file, the GPIO setting is similar to the CS line setting in `MX_GPIO_Init()` in the `Core/Src/gpio.c` file.

Listing 10: File "API/s2pi.c" – Setting the mode on the GPIO lines

```
/*!*****************************************************************************
 * @brief   Sets the mode in which the S2PI pins operate.
 * @details This is a helper function to switch the modes between SPI and GPIO.
 * @param   mode The gpio mode: GPIO_MODE_AF_PP for SPI,
 *                              GPIO_MODE_OUTPUT_PP for GPIO.
 *****************************************************************************/
static void S2PI_SetGPIOMode(uint32_t mode)
{
    GPIO_InitTypeDef  GPIO_InitStruct;

    /* SPI CLK GPIO pin configuration  */
    GPIO_InitStruct.Pin       = s2pi_.GPIOs[S2PI_CLK].Pin;
    GPIO_InitStruct.Mode      = mode;
    GPIO_InitStruct.Pull      = GPIO_NOPULL;
    GPIO_InitStruct.Speed     = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = s2pi_.SpiAlternate;
    HAL_GPIO_Init(s2pi_.GPIOs[S2PI_CLK].Port, &GPIO_InitStruct);

    /* SPI MISO GPIO pin configuration  */
    GPIO_InitStruct.Pin       = s2pi_.GPIOs[S2PI_MISO].Pin;
    HAL_GPIO_Init(s2pi_.GPIOs[S2PI_MISO].Port, &GPIO_InitStruct);

    /* SPI MOSI GPIO pin configuration  */
    GPIO_InitStruct.Pin      = s2pi_.GPIOs[S2PI_MOSI].Pin;
    HAL_GPIO_Init(s2pi_.GPIOs[S2PI_MOSI].Port, &GPIO_InitStruct);
}
```

Now, this can be used to implement the capturing of the GPIOs (switching to GPIO mode). Note that the state is checked if it is currently STATUS_IDLE (SPI mode) and changed to STATUS_S2PI_GPIO_MODE while the interrupts are locked.

Listing 11: File "API/s2pi.c" – Switching the GPIO mode

```
/*!*****************************************************************************
 * @brief   Captures the S2PI pins for GPIO usage.
 * @details The SPI is disabled (module status: #STATUS_S2PI_GPIO_MODE) and the
 *          pins are configured for GPIO operation. The GPIO control must be
 *          release with the #S2PI_ReleaseGpioControl function in order to
 *          switch back to ordinary SPI functionality.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t S2PI_CaptureGpioControl(void)
{
    /* Check if something is ongoing. */
    IRQ_LOCK();
    status_t status = s2pi_.Status;
    if (status != STATUS_IDLE)
    {
        IRQ_UNLOCK();
        return status;
    }
    s2pi_.Status = STATUS_S2PI_GPIO_MODE;
    IRQ_UNLOCK();

    /* Note: Clock must be HI after capturing */
    HAL_GPIO_WritePin(s2pi_.GPIOs[S2PI_CLK].Port, s2pi_.GPIOs[S2PI_CLK].Pin, GPIO_PIN_SET);

    S2PI_SetGPIOMode(GPIO_MODE_OUTPUT_PP);

    return STATUS_OK;
}
```

To switch back to SPI mode, there is the reverse status change.

Listing 12: File "API/s2pi.c" – Switching the SPI mode

```
/*!**************************************************************************
 * @brief   Releases the S2PI pins from GPIO usage and switches back to SPI mode.
 * @details The GPIO pins are configured for SPI operation and the GPIO mode is
 *          left. Must be called if the pins are captured for GPIO operation via
 *          the #S2PI_CaptureGpioControl function.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ****************************************************************************/
status_t S2PI_ReleaseGpioControl(void)
{
    /* Check if something is ongoing. */
    IRQ_LOCK();
    status_t status = s2pi_.Status;
    if (status != STATUS_S2PI_GPIO_MODE)
    {
        IRQ_UNLOCK();
        return status;
    }
    s2pi_.Status = STATUS_IDLE;
    IRQ_UNLOCK();

    S2PI_SetGPIOMode(GPIO_MODE_AF_PP);

    return STATUS_OK;
}
```

Step 25. Implementing the GPIO Access

GPIO access is required to access the devices' EEPROM. The EEPROM interface is multiplexed to the SPI pins to reduce the number of physical lines required. However, the interface that is understood by the EEPROM is not compatible with the SPI interface and, thus, the interface is emulated in software using GPIO toggling.

NOTE: The timing requirements for the EEPROM interface might require the GPIO access to be slowed down. This can be achieved by an artificial delay for each GPIO access through the S2PI layer. The default delay is 10 µs to achieve a baud rate of approximately 100 kHz. Because the EEPROM is only read once upon device initialization, the exact timing is not essential for the measurement performance.

Listing 13: File "API/s2pi.c" – Helper macro for the delay

```
/*! An additional delay to be added after each GPIO access in order to decrease
 *  the baud rate of the software EEPROM protocol. Increase the delay if timing
 *  issues occur while reading the EERPOM.
 *  e.g. Delay = 10 µsec => Baud Rate < 100 kHz */
#ifndef S2PI_GPIO_DELAY_US
#define S2PI_GPIO_DELAY_US 10
#endif


#if (S2PI_GPIO_DELAY_US == 0)
#define S2PI_GPIO_DELAY() ((void)0)
#else
#include "utility/time.h"
#define S2PI_GPIO_DELAY() Time_DelayUSec(S2PI_GPIO_DELAY_US)
#endif
```

With this delay, reading and writing can be implemented.

Listing 14: File "API/s2pi.c" – Writing pins in GPIO mode

```
/*!*****************************************************************************
 * @brief   Writes the output for a specified SPI pin in GPIO mode.
 * @details This function writes the value of an SPI pin if the SPI pins are
 *          captured for GPIO operation via the #S2PI_CaptureGpioControl previously.
 * @param   slave The specified S2PI slave.
 * @param   pin The specified S2PI pin.
 * @param   value The GPIO pin state to write (0 = low, 1 = high).
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 *****************************************************************************/
status_t S2PI_WriteGpioPin(s2pi_slave_t slave, s2pi_pin_t pin, uint32_t value)
{
    /* Check if pin is valid. */
    if (pin > S2PI_IRQ || value > 1)
        return ERROR_INVALID_ARGUMENT;

    /* Check if in GPIO mode. */
    if(s2pi_.Status != STATUS_S2PI_GPIO_MODE)
        return ERROR_S2PI_INVALID_STATE;

    HAL_GPIO_WritePin(s2pi_.GPIOs[pin].Port, s2pi_.GPIOs[pin].Pin, value);

    S2PI_GPIO_DELAY();

    return STATUS_OK;
}
```

Reading is similar.

Listing 15: File "API/s2pi.c" – Reading pins in GPIO mode

```
/*!*****************************************************************************
 * @brief   Reads the input from a specified SPI pin in GPIO mode.
 * @details This function reads the value of an SPI pin if the SPI pins are
 *          captured for GPIO operation via the #S2PI_CaptureGpioControl previously.
 * @param   slave The specified S2PI slave.
 * @param   pin The specified S2PI pin.
 * @param   value The GPIO pin state to read (0 = low, 1 = high).
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 *****************************************************************************/
status_t S2PI_ReadGpioPin(s2pi_slave_t slave, s2pi_pin_t pin, uint32_t * value)
{
    /* Check if pin is valid. */
    if (pin > S2PI_IRQ || !value)
        return ERROR_INVALID_ARGUMENT;

    /* Check if in GPIO mode. */
    if(s2pi_.Status != STATUS_S2PI_GPIO_MODE)
        return ERROR_S2PI_INVALID_STATE;

    *value = HAL_GPIO_ReadPin(s2pi_.GPIOs[pin].Port, s2pi_.GPIOs[pin].Pin);

    S2PI_GPIO_DELAY();

    return STATUS_OK;
}
```

Step 26. Implementing the CS Cycling

To cancel integration, the SPI CS line must be cycled. The function performing this is implemented here. Again, it checks if the device is currently idle.

NOTE: The `SPI_WriteGpioPin()` function cannot be reused, because it implements an additional artificial delay, and only works in GPIO mode. You do not need to switch to GPIO mode here, because the CS line is set up as GPIO anyway. If, in your implementation, the CS line is controlled by the SPI, you must switch it to GPIO mode first and back afterwards.

Listing 16: File "API/s2pi.c" – Performing the SPI CS cycling

```
/*!*****************************************************************************
 * @brief   Cycles the chip select line.
 * @details In order to cancel the integration on the ASIC, a fast toggling
 *          of the chip select pin of the corresponding SPI slave is required.
 *          Therefore, this function toggles the CS from high to low and back.
 *          The SPI instance for the specified S2PI slave must be idle,
 *          otherwise the status #STATUS_BUSY is returned.
 * @param   slave The specified S2PI slave.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 *****************************************************************************/
status_t S2PI_CycleCsPin(s2pi_slave_t slave)
{
    /* Check the driver status. */
    IRQ_LOCK();
    status_t status = s2pi_.Status;
    if ( status != STATUS_IDLE )
    {
        IRQ_UNLOCK();
        return status;
    }
    s2pi_.Status = STATUS_BUSY;
    IRQ_UNLOCK();

    HAL_GPIO_WritePin(s2pi_.GPIOs[S2PI_CS].Port, s2pi_.GPIOs[S2PI_CS].Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(s2pi_.GPIOs[S2PI_CS].Port, s2pi_.GPIOs[S2PI_CS].Pin, GPIO_PIN_SET);

    s2pi_.Status = STATUS_IDLE;

    return STATUS_OK;
}
```

Step 27. Implementing the SPI Transfer Start

As previously designed, the SPI transfer is performed by DMA. Therefore, the SPI transfer is only started with the transfer function, and the completion is indicated by an interrupt.

Here, the function to start the SPI transfer is implemented. First, the arguments are checked and the slave. The callback and its data are stored for the interrupts later. Then the SPI CS signal is asserted (set low) and the transfer is started.

NOTE: If the data should be transmitted only, there is no valid receive buffer, so a different function must be triggered to transmit only.

CAUTION! Some of the SPI transmissions are very short, so the completion interrupt comes early with fast SPI speeds. If an interrupt, even a low-priority interrupt like SysTick, delays the setup marginally, the functions `HAL_SPI_Transmit_DMA()` or `HAL_SPI_TransmitReceive_DMA()` may not have unlocked the internal structure in the STM32 HAL before the completion interrupt occurs. Therefore, all interrupts are locked until these functions return by using the `IRQ_Lock()` and `IRQ_Unlock()` methods. This may also be necessary with other vendors' implementations.

Listing 17: File "API/s2pi.c" – Starting the SPI transfer

```
/*!*****************************************************************************
 * @brief   Transfers a single SPI frame asynchronously.
 * @details Transfers a single SPI frame in asynchronous manner. The Tx data
 *          buffer is written to the device via the MOSI line.
 *          Optionally the data on the MISO line is written to the provided
 *          Rx data buffer. If null, the read data is dismissed.
 *          The transfer of a single frame requires to not toggle the chip
 *          select line to high in between the data frame.
 *          An optional callback is invoked when the asynchronous transfer
 *          is finished. Note that the provided buffer must not change while
 *          the transfer is ongoing. Use the slave parameter to determine
 *          the corresponding slave via the given chip select line.
 *
 * @param   slave The specified S2PI slave.
 * @param   txData The 8-bit values to write to the SPI bus MOSI line.
 * @param   rxData The 8-bit values received from the SPI bus MISO line
 *                 (pass a null pointer if the data don't need to be read).
 * @param   frameSize The number of 8-bit values to be sent/received.
 * @param   callback A callback function to be invoked when the transfer is
 *                   finished. Pass a null pointer if no callback is required.
 * @param   callbackData A pointer to a state that will be passed to the
 *                       callback. Pass a null pointer if not used.
 *
 * @return  Returns the \link #status_t status\endlink:
 *           - #STATUS_OK: Successfully invoked the transfer.
 *           - #ERROR_INVALID_ARGUMENT: An invalid parameter has been passed.
 *           - #ERROR_S2PI_INVALID_SLAVE: A wrong slave identifier is provided.
 *           - #STATUS_BUSY: An SPI transfer is already in progress. The
 *                           transfer was not started.
 *           - #STATUS_S2PI_GPIO_MODE: The module is in GPIO mode. The transfer
 *                                     was not started.
 ******************************************************************************/
status_t S2PI_TransferFrame(s2pi_slave_t spi_slave,
                            uint8_t const * txData,
                            uint8_t * rxData,
                            size_t frameSize,
                            s2pi_callback_t callback,
                            void * callbackData)
{
    /* Verify arguments. */
    if (!txData || frameSize == 0 || frameSize >= 0x10000)
        return ERROR_INVALID_ARGUMENT;

    /* Check the spi slave.*/
    if (spi_slave != S2PI_S1)
        return ERROR_S2PI_INVALID_SLAVE;
```

```
    /* Check the driver status, lock if idle. */
    IRQ_LOCK();
    status_t status = s2pi_.Status;
    if (status != STATUS_IDLE)
    {
        IRQ_UNLOCK();
        return status;
    }
    s2pi_.Status = STATUS_BUSY;
    IRQ_UNLOCK();

    /* Set the callback information */
    s2pi_.Callback = callback;
    s2pi_.CallbackData = callbackData;

    /* Manually set the chip select (active low) */
    HAL_GPIO_WritePin(s2pi_.GPIOs[S2PI_CS].Port, s2pi_.GPIOs[S2PI_CS].Pin, GPIO_PIN_RESET);

    HAL_StatusTypeDef hal_error;

    /* Lock interrupts to prevent completion interrupt before setup is complete */
    IRQ_LOCK();
    if (rxData)
        hal_error = HAL_SPI_TransmitReceive_DMA(&hspi1, (uint8_t *) txData, rxData, (uint16_t)
frameSize);
    else
        hal_error = HAL_SPI_Transmit_DMA(&hspi1, (uint8_t *) txData, (uint16_t) frameSize);
    IRQ_UNLOCK();

    if (hal_error != HAL_OK)
        return ERROR_FAIL;

    return STATUS_OK;
}
```

Step 28. Implementing the SPI Transfer Completion

The completion of the SPI transfer is signaled by DMA interrupts. These run into callback functions that must be defined here. The names and details are specific to the target platform.

Here, there are two different callbacks according to the initiated transfer (transmit only and transmit/receive). However, in the latter case, two DMA interrupts are actually received, but only the receive interrupt triggers the callback. Because the callback into the AFBR-S50 library can trigger the next transfer within the interrupt, you must ensure that both interrupts are actually handled, or setting up the next transfer could fail. To achieve this, the real callback is only triggered if the transmit interrupt was already handled. Otherwise, the transmit callback is set up to trigger the final callback.

The real callback is triggered using a common helper function that also features a status. In addition, it resets the SPI CS signal (high) to indicate the end of the transfer.

Listing 18: File "API/s2pi.c" – Triggering the provided callback function

```
/*!*****************************************************************************
 * @brief    Triggers the callback function with the provided status.
 * @details It first checks if a callback function is present,
 *          otherwise it returns immediately.
 *          The callback function is reset to 0, and must be set up again
 *          for the next transfer, if required.
 * @param    status The status to be provided to the callback funcition.
 * @return     Returns the status received from the callback function
 ******************************************************************************/
static inline status_t S2PI_CompleteTransfer(status_t status)
{
    s2pi_.Status = STATUS_IDLE;

    /* Deactivate CS (set high), as we use GPIO pin */
    HAL_GPIO_WritePin(s2pi_.GPIOs[S2PI_CS].Port, s2pi_.GPIOs[S2PI_CS].Pin, GPIO_PIN_SET);

    /* Invoke callback if there is one */
    if (s2pi_.Callback != 0)
    {
        s2pi_callback_t callback = s2pi_.Callback;
        s2pi_.Callback = 0;
        status = callback(status, s2pi_.CallbackData);
    }
    return status;
}
```

Based on this, the callbacks from the interrupts can be implemented.

Listing 19: File "API/s2pi.c" – Implementation of the SPI completion callbacks

```
  /**
    * @brief  Tx Transfer completed callback.
    * @param  hspi pointer to a SPI_HandleTypeDef structure that contains
    *         the configuration information for SPI module.
    * @retval None
    */

  void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi)
  {
      S2PI_CompleteTransfer(STATUS_OK);
  }

  /**
    * @brief  DMA SPI transmit receive process complete callback for delayed transfer.
    * @param  hdma pointer to a DMA_HandleTypeDef structure that contains
    *               the configuration information for the specified DMA module.
    * @retval None
    */
  void SPI_DMATransmitReceiveCpltDelayed(DMA_HandleTypeDef *hdma)
  {
      SPI_HandleTypeDef *hspi = (SPI_HandleTypeDef *)(((DMA_HandleTypeDef *)hdma)->Parent); /*
  Derogation MISRAC2012-Rule-11.5 */
      HAL_SPI_TxCpltCallback(hspi);
  }

  /**
    * @brief  Tx Transfer completed callback.
    * @param  hspi pointer to a SPI_HandleTypeDef structure that contains
    *               the configuration information for SPI module.
    * @retval None
    */
  void HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi)
  {
     /* Note: This interrupt callback is always invoked by the RX interrupt from the HAL. However, the
  * order of RX and TX is not specified on the device. Occasionally, the RX interrupt occurs before
  * the TX interrupt which means the SPI transfer is not yet completely finished upon the occurrence
  * of the RX interrupt. Thus, the start of a new SPI transfer may fail, since the AFBR-S50 API
  * starts it right from the interrupt callback function.
  * In order to overcome the feature, the invocation of the API callback is scheduled to whatever IRQ
  * comes last: */
      if ( hspi->hdmatx->Lock == HAL_UNLOCKED ) /* TX Interrupt already received */
          HAL_SPI_TxCpltCallback(hspi);
      else /* There is still the TX DMA Interrupt we have to wait for */
          hspi->hdmatx->XferCpltCallback = SPI_DMATransmitReceiveCpltDelayed;
  }
```

Step 29. Implementing the SPI Transfer Abort

The SPI transfer must also be able to be aborted before it is done. If no transfer is in progress, this is not an error, but nothing needs to be aborted.

Listing 20: File "API/s2pi.c" – Aborting the SPI transfer

```
/*!*****************************************************************************
 * @brief   Terminates a currently ongoing asynchronous SPI transfer.
 * @details When a callback is set for the current ongoing activity, it is
 *          invoked with the #ERROR_ABORTED error byte.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t S2PI_Abort(void)
{
    status_t status = s2pi_.Status;

    /* Check if something is ongoing. */
    if(status == STATUS_IDLE)
    {
        return STATUS_OK;
    }

    /* Abort SPI transfer. */
    if(status == STATUS_BUSY)
    {
        HAL_SPI_Abort(&hspi1);
    }

    return STATUS_OK;
}
```

The callback function is triggered from the SPI abort callback.

Listing 21: File "API/s2pi.c" – Triggering the callback function on abort

```
/**
  * @brief  SPI Abort Complete callback.
  * @param  hspi SPI handle.
  * @retval None
  */
void HAL_SPI_AbortCpltCallback(SPI_HandleTypeDef *hspi)
{
    S2PI_CompleteTransfer(ERROR_ABORTED);
}
```

Step 30. Implementing the SPI Transfer Error Handling

In case of an error, the callback function must be notified also. This is done from the SPI error callback.

Listing 22: File "API/s2pi.c" – Triggering the callback function on error

```
/**
  * @brief  SPI error callback.
  * @param  hspi pointer to a SPI_HandleTypeDef structure that contains
  *               the configuration information for SPI module.
  * @retval None
  */
void HAL_SPI_ErrorCallback(SPI_HandleTypeDef *hspi)
{
    S2PI_CompleteTransfer(ERROR_FAIL);
}
```

Step 31. Implementing the External Interrupt Handling

Finally, the external interrupt used by the AFBR-S50 device to indicate data must be implemented.

First, the callback function in case of this interrupt needs to be able to be set up.

Listing 23: File "API/s2pi.c" – Preparing the external interrupt callback

```
/*!*****************************************************************************
 * @brief   Set a callback for the GPIO IRQ for a specified S2PI slave.
 *
 * @param   slave The specified S2PI slave.
 * @param   callback A callback function to be invoked when the specified
 *                       S2PI slave IRQ occurs. Pass a null pointer to disable
 *                       the callback.
 * @param   callbackData A pointer to a state that will be passed to the
 *                       callback. Pass a null pointer if not used.
 *
 * @return  Returns the \link #status_t status\endlink:
 *           - #STATUS_OK: Successfully installation of the callback.
 *           - #ERROR_S2PI_INVALID_SLAVE: A wrong slave identifier is provided.
 *******************************************************************************/
status_t S2PI_SetIrqCallback(s2pi_slave_t slave,
                             s2pi_irq_callback_t callback,
                             void * callbackData)
{
    s2pi_.IrqCallback = callback;
    s2pi_.IrqCallbackData = callbackData;

    return STATUS_OK;
}
```

Then, the possibility of reading the next interrupt line must be implemented.

Listing 24: File "API/s2pi.c" – Readout of the external interrupt status

```
/*!*****************************************************************************
 * @brief   Reads the current status of the IRQ pin.
 * @details In order to keep a low priority for GPIO IRQs, the state of the
 *          IRQ pin must be read in order to reliable check for chip timeouts.
 *
 *          The execution of the interrupt service routine for the data-ready
 *          interrupt from the corresponding GPIO pin might be delayed due to
 *          priority issues. The delayed execution might disable the timeout
 *          for the eye-safety checker too late causing false error messages.
 *          In order to overcome the issue, the state of the IRQ GPIO input
 *          pin is read before raising a timeout error in order to check if
 *          the device has already finished but the IRQ is still pending to be
 *          executed!

 * @param   slave The specified S2PI slave.
 * @return  Returns 1U if the IRQ pin is high (IRQ not pending) and 0U if the
 *          devices pulls the pin to low state (IRQ pending).
 ******************************************************************************/
uint32_t S2PI_ReadIrqPin(s2pi_slave_t slave)
{
    return HAL_GPIO_ReadPin(s2pi_.GPIOs[S2PI_IRQ].Port, s2pi_.GPIOs[S2PI_IRQ].Pin);
}
```

Finally, the callback function must be set up.

Listing 25: File "API/s2pi.c" – Implementation of the external interrupt callback

```
/**
  * @brief  EXTI line detection callbacks.
  * @param  GPIO_Pin Specifies the pins connected EXTI line
  * @retval None
  */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == s2pi_.GPIOs[S2PI_IRQ].Pin && s2pi_.IrqCallback)
    {
        s2pi_.IrqCallback(s2pi_.IrqCallbackData);
    }
}
```

With this setup, the S2PI module is complete.

4.3 Timer API

Now, the API interface must be implemented.

Step 32. Creation of the Timer File

The timer API is implemented in a new file, `timer.c`, in the `API` folder.

Step 33. Adding the Timer Includes

First, the headers declaring the API functions to be implemented are included. This means the API header and the header for the generated implementation.

Listing 26: File "API/timer.c" – Include statements

```
#include "tim.h"
#include "driver/timer.h"
```

Step 34. Implementing the Timer Initialization

Next, the timer initialization routine is implemented. The prototype is found in `driver/timer.h`.

It calls the automatic hardware initialization for each timer.

Listing 27: File "API/timer.c" – Timer initialization

```
/*!*****************************************************************************
 * @brief   Initializes the timer hardware.
 * @return   -
 *****************************************************************************/
void Timer_Init(void)
{
    /* Initialize the timers, see generated main.c */
    MX_TIM2_Init();
    MX_TIM4_Init();
    MX_TIM5_Init();

    /* Start the timers relevant for the LTC */
    HAL_TIM_Base_Start(&htim2);
    HAL_TIM_Base_Start(&htim5);
}
```

Step 35. Implementing the LTC Readout

The readout of the LTC timer is implemented. The prototype is found in `platform/AFBR-S50_timer.h`, included from `driver/timer.h`.

It reads both chained timers and returns the value, possibly looping if a counter wraparound might have occurred.

Listing 28: File "API/timer.c" – Lifetime counter readout

```
/*!*****************************************************************************
 * @brief   Obtains the lifetime counter value from the timers.
 *
 * @details The function is required to get the current time relative to any
 *          point in time, e.g. the startup time. The returned values \p hct and
 *          \p lct are given in seconds and microseconds respectively. The current
 *          elapsed time since the reference time is then calculated from:
 *
 *          t_now [µsec] = hct * 1000000 µsec + lct * 1 µsec
 *
 * @param   hct A pointer to the high counter value bits representing current
 *              time in seconds.
 * @param   lct A pointer to the low counter value bits representing current
 *              time in microseconds. Range: 0, .., 999999 µsec
 * @return  -
 ******************************************************************************/
void Timer_GetCounterValue(uint32_t * hct, uint32_t * lct)
{
    /* The loop makes sure that there are no glitches
       when the counter wraps between htim2 and htm2 reads. */
    do {
        *lct = __HAL_TIM_GET_COUNTER(&htim2);
        *hct = __HAL_TIM_GET_COUNTER(&htim5);
    }
    while (*lct > __HAL_TIM_GET_COUNTER(&htim2));
}
```

Step 36. Implementing the PIT Start/Stop

The PIT timer can be started and stopped by the appropriate API functions. The callback parameter and the period are stored. If a running timer is enabled with the same period, nothing should happen.

If the timer interval does not fit into the 16-bit timer with microsecond granularity, the prescaler is used to reduce the granularity and the period is reduced.

Listing 29: File "API/timer.c" – Starting and stopping the PIT

```
/*! Storage for the callback parameter */
static void * callback_param_;

/*! Timer interval in microseconds */
static uint32_t period_us_;

/*!***************************************************************************
 * @brief   Starts the timer for a specified callback parameter.
 * @details Sets the callback interval for the specified parameter and starts
 *          the timer with a new interval. If there is already an interval with
 *          the given parameter, the timer is restarted with the given interval.
 *          Passing a interval of 0 disables the timer.
 * @param   dt_microseconds The callback interval in microseconds.
 * @param   param An abstract parameter to be passed to the callback. This is
 *                  also the identifier of the given interval.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 *****************************************************************************/
status_t Timer_Start(uint32_t period, void * param)
{
    callback_param_ = param;

    if (period == period_us_)
        return STATUS_OK;

    period_us_ = period;
    uint32_t prescaler = SystemCoreClock / 1000000U;

    while (period > 0xFFFF)
    {
        period >>= 1U;
        prescaler <<= 1U;
    }

    assert(prescaler <= 0x10000U);

    /* Set prescaler and period values */
    __HAL_TIM_SET_PRESCALER(&htim4, prescaler - 1);
    __HAL_TIM_SET_AUTORELOAD(&htim4, period - 1);

    /* Enable interrupt and timer */
    __HAL_TIM_ENABLE_IT(&htim4, TIM_IT_UPDATE);
    __HAL_TIM_ENABLE(&htim4);

    return STATUS_OK;
}
```

```
/*!*****************************************************************************
 * @brief    Stops the timer for a specified callback parameter.
 * @details    Stops a callback interval for the specified parameter.
 * @param    param An abstract parameter that identifies the interval to be stopped.
 * @return     Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t Timer_Stop(void * param)
{
    period_us_ = 0;
    callback_param_ = 0;

    /* Disable interrupt and timer */
    __HAL_TIM_DISABLE_IT(&htim4, TIM_IT_UPDATE);
    __HAL_TIM_ENABLE(&htim4);;

    return STATUS_OK;
}

/*!*****************************************************************************
 * @brief   Sets the timer interval for a specified callback parameter.
 * @details Sets the callback interval for the specified parameter and starts
 *          the timer with a new interval. If there is already an interval with
 *          the given parameter, the timer is restarted with the given interval.
 *          If the same time interval as already set is passed, nothing happens.
 *          Passing a interval of 0 disables the timer.
 * @param   dt_microseconds The callback interval in microseconds.
 * @param   param An abstract parameter to be passed to the callback. This is
 *                  also the identifier of the given interval.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t Timer_SetInterval(uint32_t dt_microseconds, void * param)
{
    return dt_microseconds ? Timer_Start(dt_microseconds, param) : Timer_Stop(param);
}
```

Step 37. Implementing the PIT Interrupt Handling

Finally, the interrupt caused by the expired PIT timer is handled, and the callback function is triggered, if defined.

Listing 30: File "API/timer.c" – PIT interrupt handling

```
/*! Callback function for PIT timer */
static timer_cb_t timer_callback_;

/*!*****************************************************************************
 * @brief   Installs an periodic timer callback function.
 * @details Installs an periodic timer callback function that is invoked whenever
 *          an interval elapses. The callback is the same for any interval,
 *          however, the single intervals can be identified by the passed
 *          parameter.
 *          Passing a zero-pointer removes and disables the callback.
 * @param   f The timer callback function.
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 *******************************************************************************/
status_t Timer_SetCallback(timer_cb_t f)
{
    timer_callback_ = f;
    return STATUS_OK;
}

/**
  * @brief  Period elapsed callback in non-blocking mode
  * @param  htim TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* Trigger callback if the interrupt belongs to TIM4 and there is a callback */
    if (htim==&htim4 && timer_callback_)
    {
        timer_callback_(callback_param_);
    }
}
```

4.4 Optional: UART API

Optionally, the UART interface can be implemented now.

Step 38. Creation of the UART File

The timer API is implemented in a new file, `uart.c`, in the `API` folder.

Step 39. Adding the UART Includes

First, the headers declaring the API functions to be implemented are included. This means the API header and the header for the generated implementation.

Listing 31: File "API/uart.c" – Include statements

```
#include "dma.h"
#include "usart.h"
#include "driver/uart.h"

#include <stdio.h>
#include <string.h>
#include <stdarg.h>
```

Step 40. Defining the UART Variables

Now there are several variables to be defined:

- An indication if a transfer is ongoing
- A buffer for formatting the output message
- A callback function with a parameter

All of these can be static variables.

Listing 32: File "API/uart.c" – UART variable definitions

```
/*! The busy indication for the uart */
static volatile bool isTxBusy_ = false;

/*! The buffer for the uart print */
static uint8_t buffer_[1024];

/*! The callback for the uart */
static uart_tx_callback_t txCallback_ = 0;

/*! The callback state for the uart */
static void * txCallbackState_ = 0;
```

Step 41. Implementing the UART Initialization

The UART initialization is completely generated in the `Core/Src/usart.c` file and only needs to be called; however, as the UART uses DMA, it also must be initialized first.

Listing 33: File "API/uart.c" – UART initialization

```
/*!**************************************************************************
 * @brief   Initialize the Universal Asynchronous Receiver/Transmitter
 *          (UART or LPSCI) bus and DMA module
 * @param   -
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ****************************************************************************/
status_t UART_Init(void)
{
    MX_DMA_Init();
    MX_USART2_UART_Init();

    return STATUS_OK;
}
```

Step 42. Implementing the UART Send Operation

Now you can implement the transmission of the data using DMA. If the line is still busy, you can skip the transfer.

Listing 34: File "API/uart.c" – UART send operation

```
/*!*****************************************************************************
 * @brief   Writes several bytes to the UART connection.
 * @param   txBuff Data array to write to the uart connection
 * @param   txSize The size of the data array
 * @param   f Callback function after tx is done, set 0 if not needed;
 * @param   state Optional user state that will be passed to callback
 *                  function; set 0 if not needed.
 * @return  Returns the \link #status_t status\endlink:
 *           - #STATUS_OK (0) on success.
 *           - #STATUS_BUSY on Tx line busy
 *           - #ERROR_NOT_INITIALIZED
 *           - #ERROR_INVALID_ARGUMENT
 ******************************************************************************/
status_t UART_SendBuffer(uint8_t const * txBuff, size_t txSize, uart_tx_callback_t f, void * state)
{
    /* Verify arguments. */
    if( !txBuff || txSize == 0 )
        return ERROR_INVALID_ARGUMENT;

    if (isTxBusy_)
        return STATUS_BUSY;

    /* Set Tx Busy Status. */
    isTxBusy_ = true;
    txCallback_ = f;
    txCallbackState_ = state;

    HAL_UART_Transmit_DMA(&huart2, (uint8_t *) txBuff, txSize);

    return STATUS_OK;
}
```

Step 43. Implementing the UART Send Completion

In the callback after the transmission, the status is set to idle again, and the requested callback is called, if there is one.

Listing 35: File "API/uart.c" – UART send completion

```
/**
  * @brief  Tx Transfer completed callbacks.
  * @param  huart  Pointer to a UART_HandleTypeDef structure that contains
  *                the configuration information for the specified UART module.
  * @retval None
  */
void HAL_UART_TxCpltCallback( UART_HandleTypeDef *huart )
{
    isTxBusy_ = false;

    status_t status = huart->gState == HAL_UART_STATE_ERROR ? ERROR_FAIL : STATUS_OK;

    if (txCallback_)
    {
        txCallback_(status, txCallbackState_);
    }
}
```

Step 44. Implementing the Formatted Output Using print()

To be able to send data from the example application, the `print()` function is implemented to send the data over the UART interface.

Listing 36: File "API/uart.c" – UART formatted output

```
/*!****************************************************************************
 * @brief   printf-like function to send print messages via UART.
 *
 * @details Defined in "driver/uart.c" source file.
 *
 *          Open an UART connection with 115200 bps, 8N1, no handshake to
 *          receive the data on a computer.
 *
 * @param   fmt_s The usual printf parameters.
 *
 * @return  Returns the \link #status_t status\endlink (#STATUS_OK on success).
 ******************************************************************************/
status_t print(const char *fmt_s, ...)
{
    va_list ap;
    va_start(ap, fmt_s);
    int len = vsnprintf((char *) buffer_, sizeof(buffer_), fmt_s, ap);
    va_end(ap);
    if (len < 0)
        return ERROR_FAIL;

    UART_SendBuffer(buffer_, len, 0, 0);

    return STATUS_OK;
}

#undef UART_Print
```

4.5 Board API

The Board API implements the remaining board initialization that is not related to the previous devices.

Step 45. Creation of the Board File

The Board API is implemented in a new file, `board.c`, in the `API` folder. This file should be created as an empty source file.

Step 46. Implementing the Board File

The implementation is simple: The clock initialization is mapped to the automatically generated function, and the COP implementation (watchdog) is omitted.

Listing 37: File "API/board.c" – Clock Initialization

```
#include "argus.h"
#include "board/clock_config.h"
#include "driver/cop.h"

extern void SystemClock_Config(void);

/* Initialize the board with clocks. */
void BOARD_ClockInit(void)
{
    SystemClock_Config();
    return STATUS_OK;
}

/* No watchdog installed */
void COP_Disable(void) {}
```

Chapter 5: Running the Example Application

5.1 Creating the Example Application

The example application contains a periodic readout of the AFBR-S50 with evaluation of the data in a single file and is a starting point for individual development.

If the optional UART interface is implemented, it can output the calculation result over the serial line emulation, and a terminal with appropriate settings can be used to receive the data.

Running the example application is not difficult after the previous preparation:

Step 47. Copying the Example Application

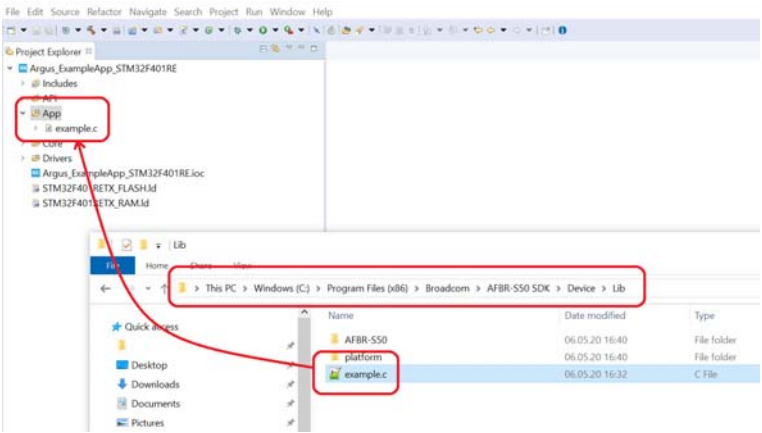
First, the example application code is copied from the source folder of the SDK into the `App/example.c` file. The file is found at the following location:

```
C:\Program Files (x86)\Broadcom\AFBR-S50 SDK\Device\Lib\example.c
```

NOTE: The actual path may change depending on the actual installation directory.

The IDE then automatically detects the new file; otherwise, restart the IDE to force detection.

Figure 31: Copied Application File in the IDE

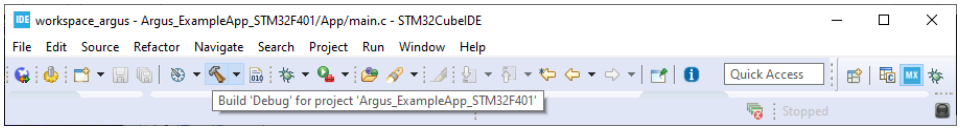


Step 48. Compiling and Running the Example Application

With all the preparations performed in the previous steps, the example application is ready to compile and run with no modification.

To compile it, click the build icon.

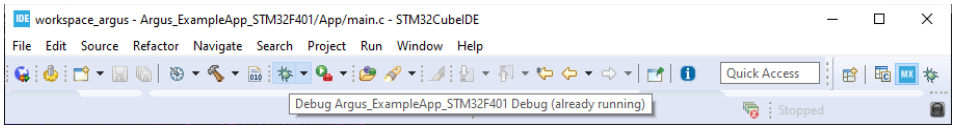
Figure 32: Building the Example Application in the IDE



The compilation should be successful with no errors.

To run the application, select debugging, which automatically transfers the build.

Figure 33: Debugging the Example Application in the IDE

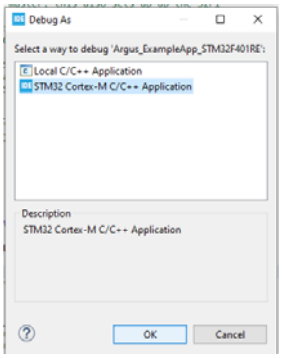


NOTE:

- The device must be attached to a USB port.
- The project must be properly selected in the Project Explorer to start the debugging.

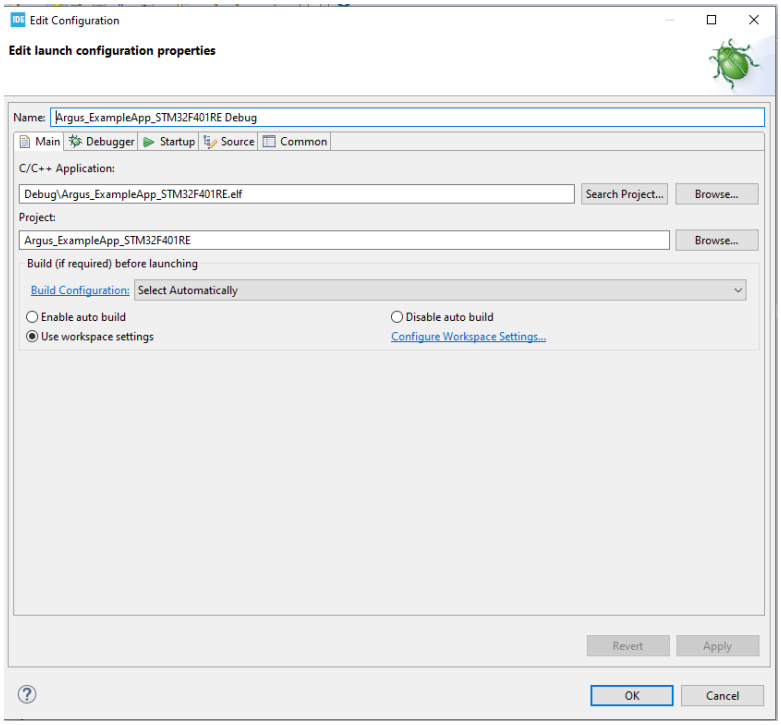
The debugger interface must be selected the first time.

Figure 34: Select the Debugger Interface



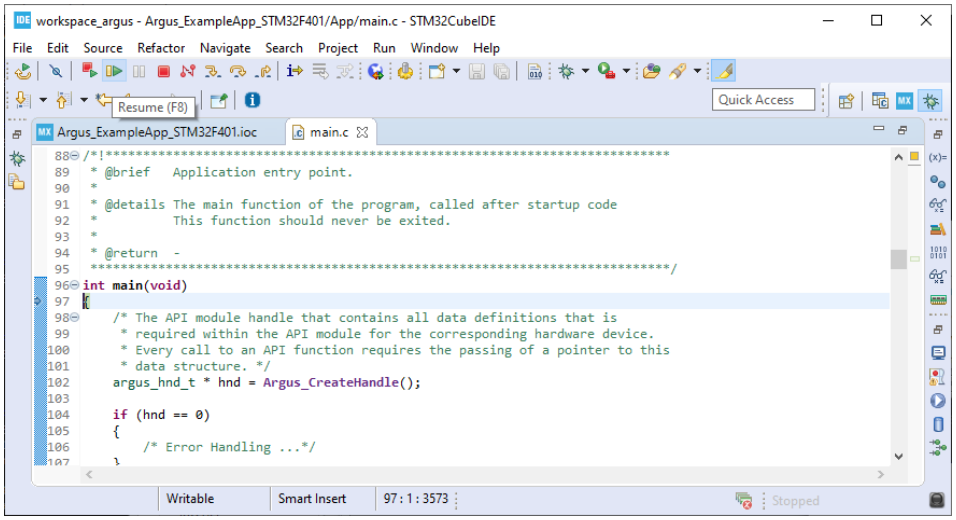
Use the default debug configuration by clicking **OK**.

Figure 35: Debug Configuration



After the debugger is started, the application is suspended at the beginning of the main function. Unless you want to step through the code, run it by clicking the resume symbol.

Figure 36: Running the Example Application in the IDE



NOTE: When attaching the device to the USB port, a virtual serial interface is automatically created for the UART interface. You can start a terminal emulation on the machine (with the connection parameters set previously) to see the device measurement results.

Figure 37: Porting Guide Serial Port Settings

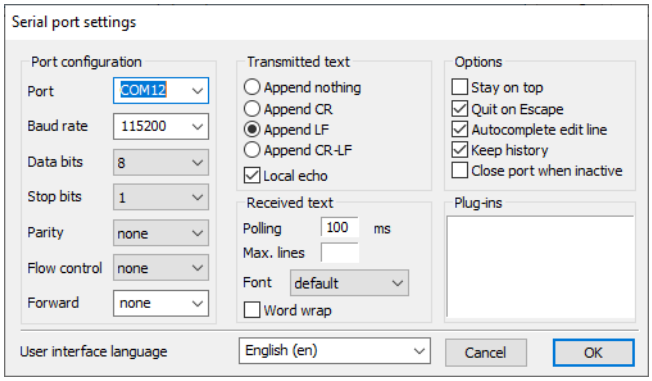
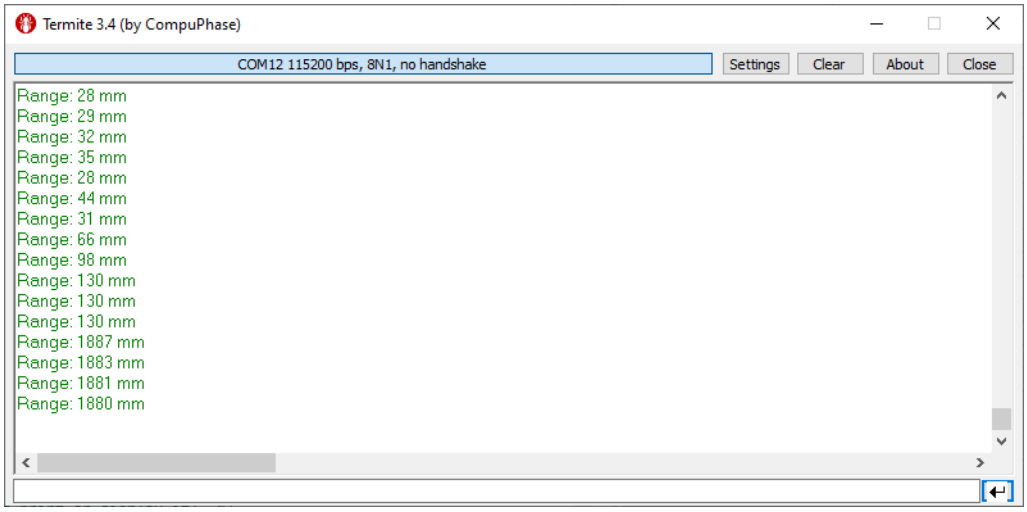


Figure 38: Porting Guide Terminal Stream



To understandably display the streamed range values from the serial port in the terminal, the sensor's frame rate was set to 10 Hz. Appendix A, Modifying the Example Application, describes how to set a target frame rate with the API.

Appendix A: Modifying the Example Application

Now you are ready to create a full application in the App folder according to your needs.

A starting point can be to increase the SPI speed to the capability of the board.

Increasing the SPI speed (line #)

```
/* #define SPI_BAUD_RATE 6000000 */
#define SPI_BAUD_RATE 21000000
```

You can also increase the frame rate by setting a smaller frame time.

Adapting the frame time (line #)

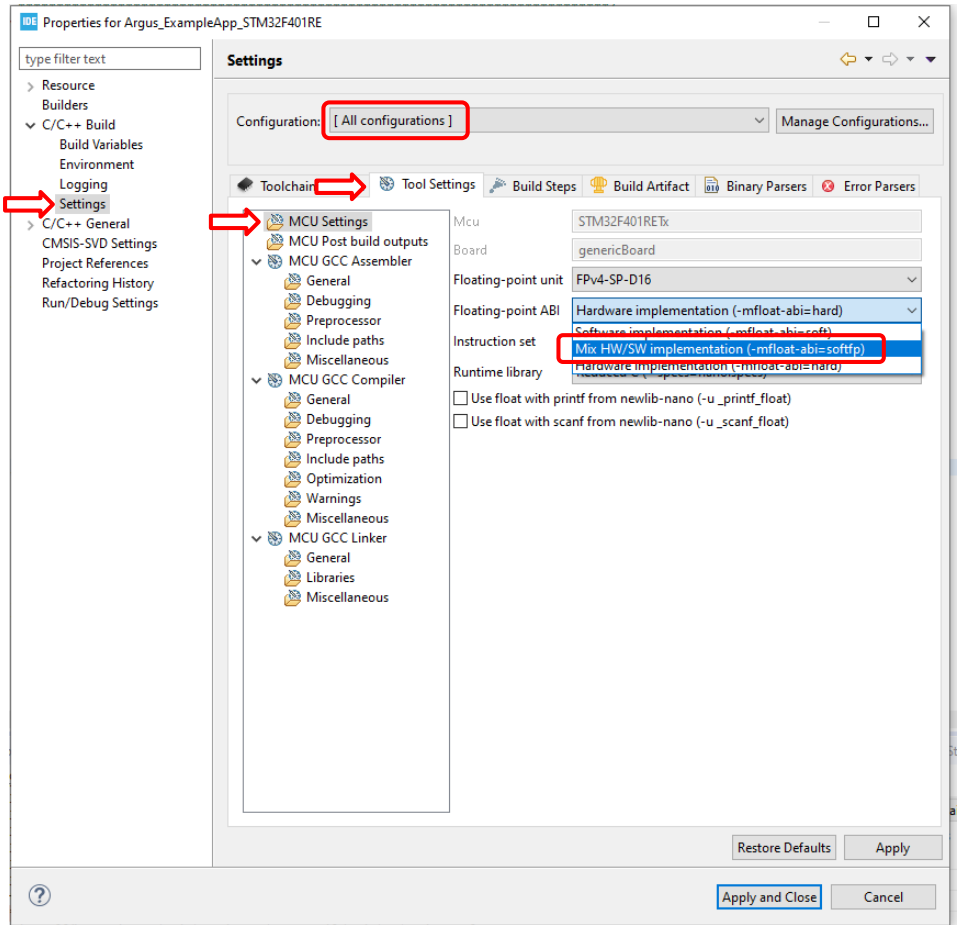
```
/*  AFBR-S50_SetConfigurationFrameTime(hnd, 100000); // 0.1 second = 10 Hz */
    AFBR-S50_SetConfigurationFrameTime(hnd, 1000);   // 0.001 second = 1000 Hz
```

If you use the terminal emulation, you may need to increase the UART baud rate as well, to deliver all measurement results in time.

A.1 Setting Up Floating-Point ABI for Soft Floating Point Usage

Currently, the current API version comes without floating-point support. To be able to link successfully, the same floating point implementation style should be set, so that the floating-point ABI has to be set to **-mfloat-abi=softfp** in the project settings.

Figure 39: Setting Floating-Point ABI



Revision History

Version 1.0, June 22, 2020

- Initial document release.


BROADCOM®