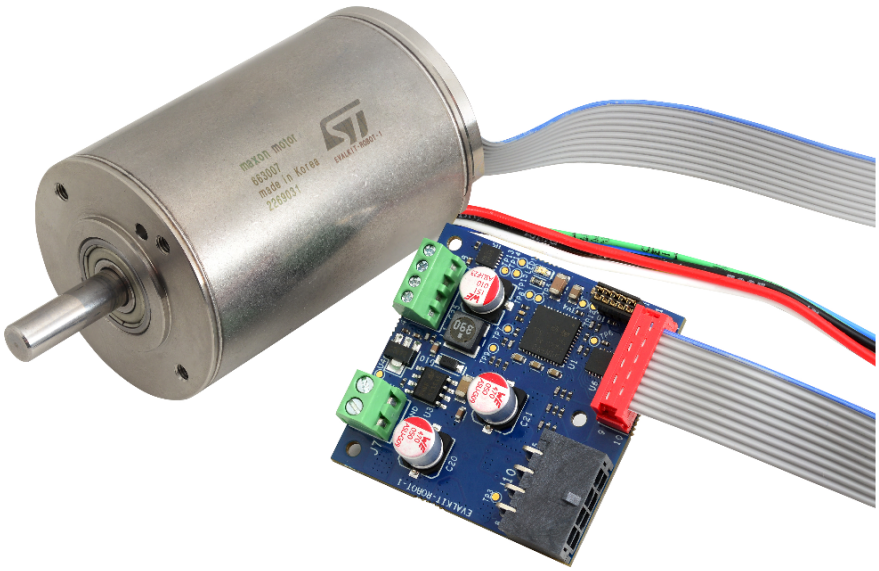
Getting started with the STSW-ROBOT-1

Introduction

The STSW-ROBOT-1 firmware example for the EVALKIT-ROBOT-1 evaluation kit offers a ready-to-use brushless servomotor solution consisting of the STSPIN32F0A control board and a maxon EC-i 40 brushless DC motor.

Figure 1. EVALKIT-ROBOT-1



1 Hardware and software requirements

The STSW-ROBOT-1 requirements are the following:

- One EVALKIT-ROBOT-1
- A 36 V / 120 W DC power supply(1)
- An RS485 2-wires serial port
- A communication software package based on MODBUS protocol
- One of the supported development toolchains:
  - IAR Embedded Workbench for ARM,
  - Keil microcontroller development kit
  - STM32CubeIDE.
- An SWD programmer/debugger supporting the STM32F0 family, such as the STLINK-V3 from STMicroelectronics.

1. *The operating range of the control electronics is between 12 V and 45 V. However, the system provides best performance with a supply voltage of 36 V ±20 %*

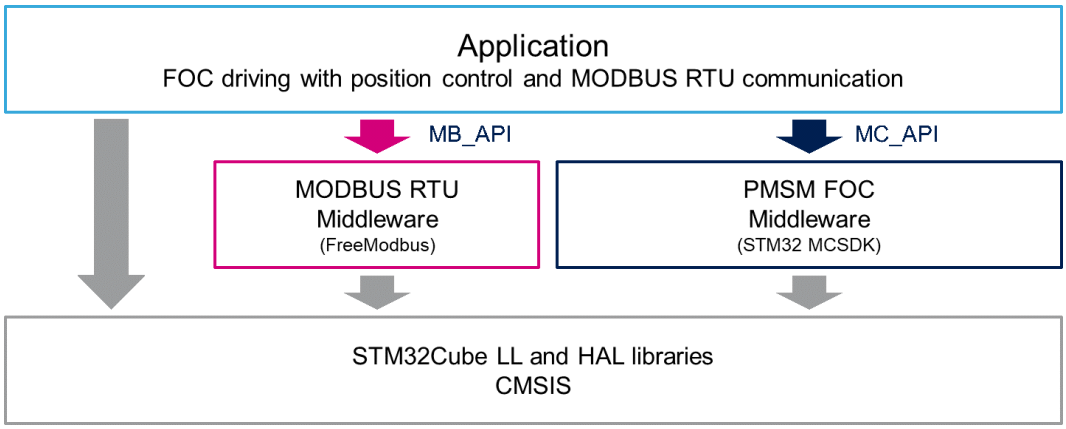
2 Getting started

This firmware example is based on the MCSDK implementing high performance PMSM FOC control with positioning features. The application firmware interfaces with the middleware via the specific API (MC_API).

The MODBUS RTU communication is based on the FreeModbus library and the application firmware interfaces with the middleware via the specific API (MB_API).

Peripheral management is performed by the STM32Cube low level divers (LL) and Hardware Abstraction Layer (HAL) based on the standard CMSIS library.

Figure 2. Firmware architecture



2.1 Folders structure

This section provides an overview of the package contents:

- MotorControl Workbench project
- **Binary** folder containing the compiled binary file
- **EVALKIT_ROBOT_1_SDK543_Positioning_LL** folder containing the source code, the CobeMX project file and the toolchains' projects

The source code sub-folder structure is the following:

- **Drivers**: STM32 HAL and LL libraries and CMSIS drivers
- **MCSDK_v5.4.3**: Motor control Field Oriented Control middleware
- **modbus**: MODBUS communication middleware
- **Inc**: Application include files
- **Src**: Application source files
- **EWARM**: IAR Embedded Workbench V8 project
- **MDK-ARM**: Keil-ARM uVision V5 project

2.2 Main application code

The main application consists of the following sections:

1. **Initialization**: Peripherals are configured and data structures are initialized
2. **Encoder alignment**: Mechanical alignment procedure is executed
3. **Main loop**: Infinite loop monitoring command request, execution and faults
4. **Motor control loop (interrupt based)**: executes the PMSM FOC algorithm (part of MCSDK)
5. **MODBUS loop (interrupt based)**: manages the MODBUS RTU communication

2.3 MC_API

The Motor Control API (Src\mc_api.c) are part of the Motor Control SDK and provides the key functions controlling the motor.

The functions listed below are used by the firmware example, for more details regarding Motor Control SDK API, refer to the relevant User Manual.

- **MC_StartMotor1**: Initiates the start-up procedure for the motor including the encoder alignment
- **MC_StopMotor1**: Initiates the stop procedure for the motor
- **MC_ProgramPositionCommandMotor1**: Programs the execution of a position command indicating the target mechanical position (fTargetPosition) in radian and the expected execution time (fDuration) in seconds
- **MC_GetControlPositionStatusMotor1**: Returns the status of the positioning control state machine
- **MC_GetAlignmentStatusMotor1**: Returns the status of the encoder alignment (start-up sequence)
- **MC_GetOccurredFaultsMotor1**: Returns a bitfield showing the new faults detected by the motor control algorithm
- **MC_AcknowledgeFaultMotor1**: Acknowledge a Motor Control fault occurred during motor control enabling a new command execution

2.4 MB_API

The MODBUS RTU API (modbus\mb_API.c) manages the direct inputs, coils and holding registers of the application.

- **setMovementCompleted**: Sets the DONE direct input high or low
- **setZeroReached**: Sets the ALIGN direct input high or low
- **getTargetPos_Deg**: Returns the target position in degrees stored in holding registers
- **getMovementDuration_s**: Returns the target movement duration (seconds) according to the value in the holding registers (milliseconds)
- **getNewCommand**: Returns true if MOVE coil is high
- **clearCommand**: Forces MOVE coil low

3 Customization

The firmware default configuration can be modified using the MotorControl Workbench and the STM32CubeMX software.

3.1 MODBUS section customization

The MODBUS section (i.e., slave address, coils, direct inputs, holding registers, etc.) can be customized by modifying the following code files:

- mb_API.c/.h: API interfacing the MODBUS with application level
- mbtask.c/.h: Implementation of the callbacks managing:
  - Holding registers read/write
  - Input registers read
  - Coils read/write
  - Discrete inputs read

The following table provides the parameters defining the MODBUS data model and slave address:

Table 1. MODBUS configuration parameters

Parameter	Location	Description
MB_SLAVE_ADDRESS	mbtask.h	Set the slave address of the device
REG_INPUT_START	mbtask.h	Input registers starting counter
REG_INPUT_NREGS	mbtask.h	Number of input registers (bytes)
REG_HOLDING_START	mbtask.h	Holding registers starting counter
REG_HOLDING_NREGS	mbtask.h	Number of holding registers (bytes)
REG_COILS_START	mbtask.h	Coils starting counter
REG_COILS_SIZE	mbtask.h	Number of coils (bits)
REG_DISC_START	mbtask.h	Discrete inputs starting counter
REG_DISC_SIZE	mbtask.h	Number of discrete inputs (bits)
MB_FUNC_OTHER_REP_SLAVEID_ENABLED	mbconfig.h	Enable/Disable Report Slave ID function
MB_FUNC_READ_INPUT_ENABLED	mbconfig.h	Enable/Disable read input register function
MB_FUNC_READ_HOLDING_ENABLED	mbconfig.h	Enable/Disable read holding register function
MB_FUNC_WRITE_HOLDING_ENABLED	mbconfig.h	Enable/Disable write holding register function
MB_FUNC_WRITE_MULTIPLE_HOLDING_ENABLED	mbconfig.h	Enable/Disable write multiple holding register function
MB_FUNC_READ_COILS_ENABLED	mbconfig.h	Enable/Disable read coils function
MB_FUNC_WRITE_COIL_ENABLED	mbconfig.h	Enable/Disable write coils function
MB_FUNC_WRITE_MULTIPLE_COILS_ENABLED	mbconfig.h	Enable/Disable write multiple coils function
MB_FUNC_READ_DISCRETE_INPUTS_ENABLED	mbconfig.h	Enable/Disable read discrete inputs function
MB_FUNC_READWRITE_HOLDING_ENABLED	mbconfig.h	Enable/Disable read/write holding register function

3.2 Starting from MotorControl Workbench project

The package contains the MotorControl Workbench project used for code generation. Using this project, it is possible to modify all the configurations related to the control algorithm and related peripherals.

Note: The communication protocol included in the MCSDK does not support RS485 implemented in the EVALKIT-ROBOT-1 board.

After the configuration changes, follow the steps in the procedure below:

Step 1. Click on “Generate Code” button.

Step 2. Select your target toolchain and check the LL library option.

Step 3. According to the changes, you have two options:

- No changes to the peripheral setup: use the “UPDATE” function. This option does not overwrite the STM32CubeMX project. You can immediately open the output project for the target toolchain and jump to step 4 in Section 3.3 .
- Some changes on the peripheral setup or a complete clean-up of the project is needed: use the “GENERATE” function and follow the steps below.

Step 4. After the code generation is completed, click on “Run STM32CubeMX”.

Step 5. The automatically generated STM32CubeMX project miss some peripheral configurations. The following changes must be done:

Step 5a. Reset configuration for PB0 (Reset_State option in drop-down list)

Step 5b. Configure PF1 as GPIO_EXTI1 and label it “M1_ENCODER_Z”

Step 5c. Configure PA15 as GPIO_Output and label it “RS485_DE”

Step 5d. Configure PF0 as GPIO_Output and label it “LED”

Step 5e. Configure the USART1 for asynchronous operation and map TX/RX to PB6/PB7 GPIOs. No hardware flow control is required.

Details about configuration of the UART are shown in the following figures.

Figure 3. UART configuration panel (basic)

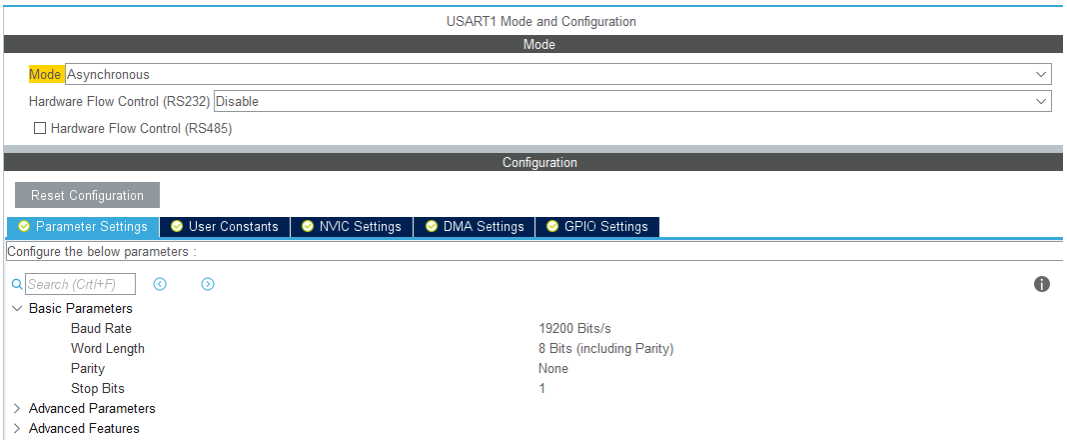
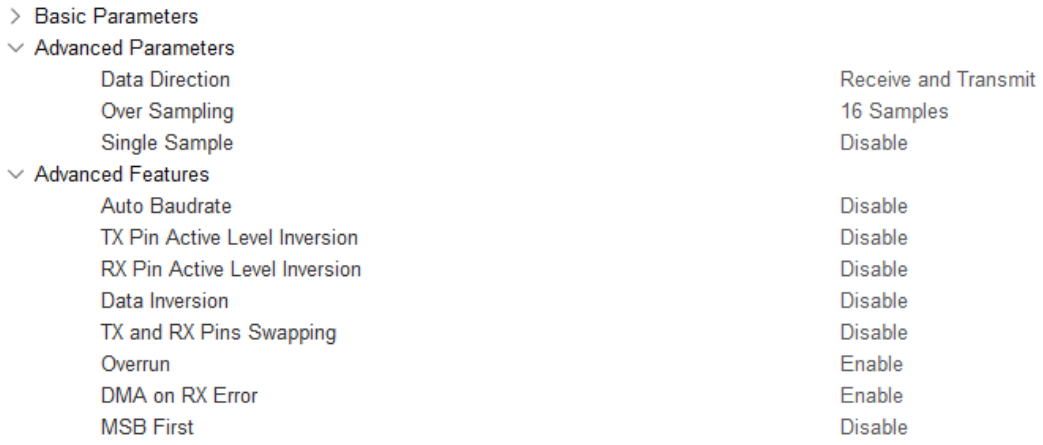


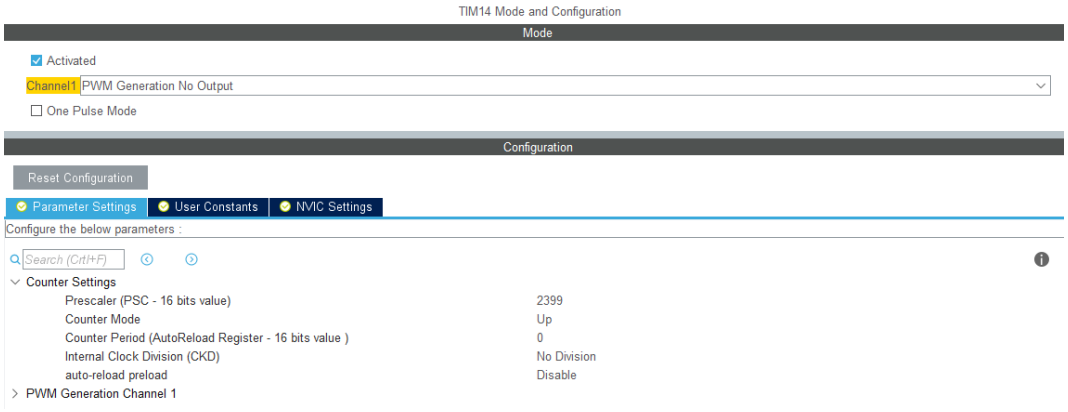
Figure 4. UART configuration panel (advanced)



Step 5f. Configure one of the available timers as PWM no output for the MODBUS RTU timeout management (by default TIM14).

It must be set in order to increment at a 50 µs rate (PSC = 2399 for 48 MHz clock).

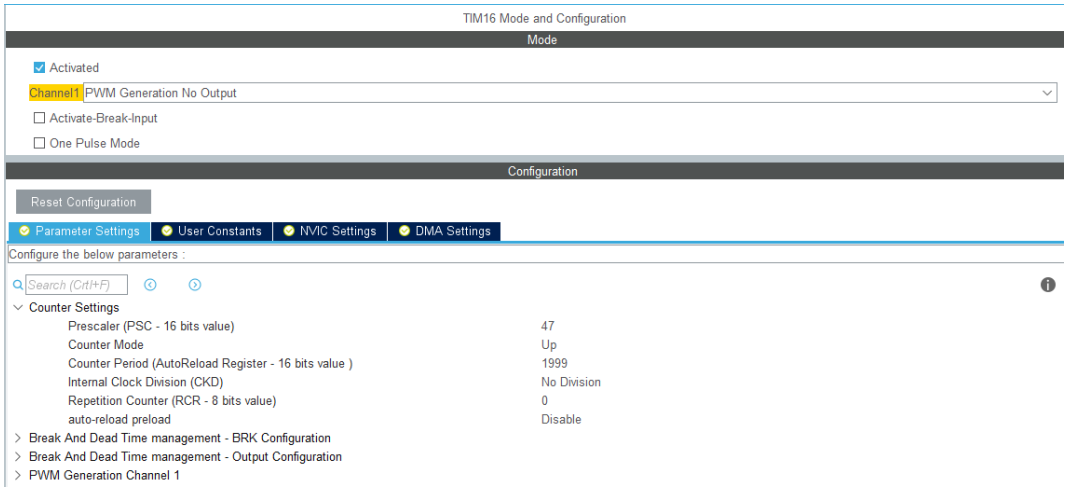
Figure 5. TIM14 configuration panel



Step 5g. Configure one of the available timers as PWM no output for the MODBUS execution scheduler (by default TIM16).

It must be set in order to increment at 1 µs rate (PSC = 47 for 48 MHz clock) and with a period of 2 ms (ARR = 1999).

Figure 6. TIM16 configuration panel



Step 5h. Enable interrupts for USART (priority 3) and timers (priority 2).

From the code generation point of view, select sequence ordering, IRQ handler generation and HAL_Handler call for the timers. For the USART only, select sequence ordering and IRQ handler generation.

Figure 7. NVIC configuration panel

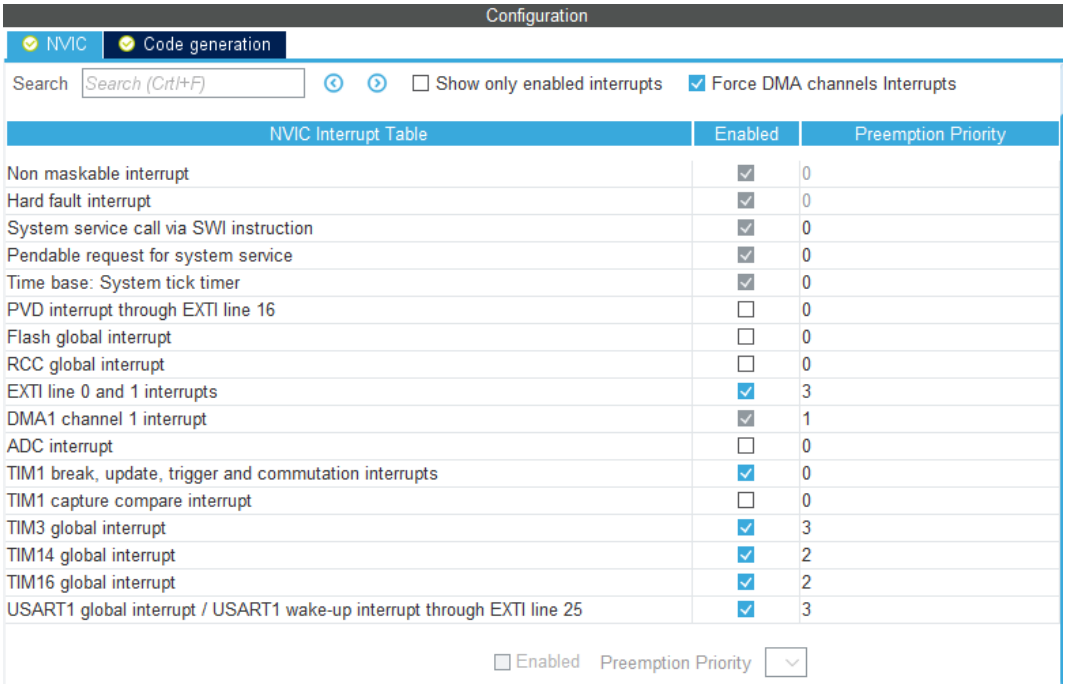
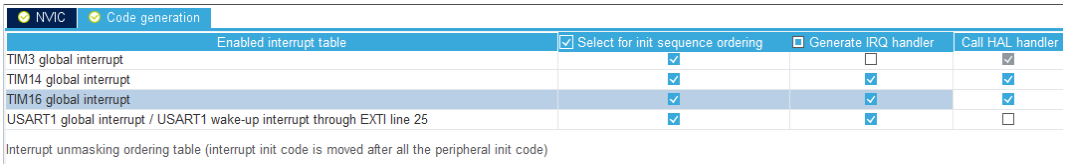


Figure 8. NVIC configuration panel (code generation)



Step 6. Select your target toolchain and check other options in the “Project Manager” tab.

Step 7. Click on “Generate Code” button.

Step 8. After the code generation is completed, click on “open project”.

Step 9. Open the stm32f0xx_it.c file and modify the empty interrupt handlers.

Step 10. Modify the IRQ_Handler of the MODBUS RTU Timeout timer as shown below:

Figure 9. MODBUS RTU timeout IRQ handler

```
105  void TIM14_IRQHandler(void)
106  {
107    /* USER CODE BEGIN TIM14_IRQn 0 */
108    if(LL_TIM_IsActiveFlag_UPDATE(TIM14) & LL_TIM_IsEnabledIT_UPDATE(TIM14))
109    {
110      LL_TIM_ClearFlag_UPDATE(TIM14);
111      pxMBPortCBTimerExpired();
112    }
113    /* USER CODE END TIM14_IRQn 0 */
114    /* USER CODE BEGIN TIM14_IRQn 1 */
115
116    /* USER CODE END TIM14_IRQn 1 */
117  }
```

Step 11. Modify the IRQ_Handler of the MODBUS scheduler timer as shown below:

Figure 10. MODBUS scheduler timer IRQ handler

```
122  void TIM16_IRQHandler(void)
123  {
124    /* USER CODE BEGIN TIM16_IRQn 0 */
125    LL_TIM_ClearFlag_UPDATE(TIM16);
126    eMBPoll();
127    /* USER CODE END TIM16_IRQn 0 */
128    /* USER CODE BEGIN TIM16_IRQn 1 */
129
130    /* USER CODE END TIM16_IRQn 1 */
131  }
```

Step 12. Modify the IRQ_Handler of the UART interface as shown below:

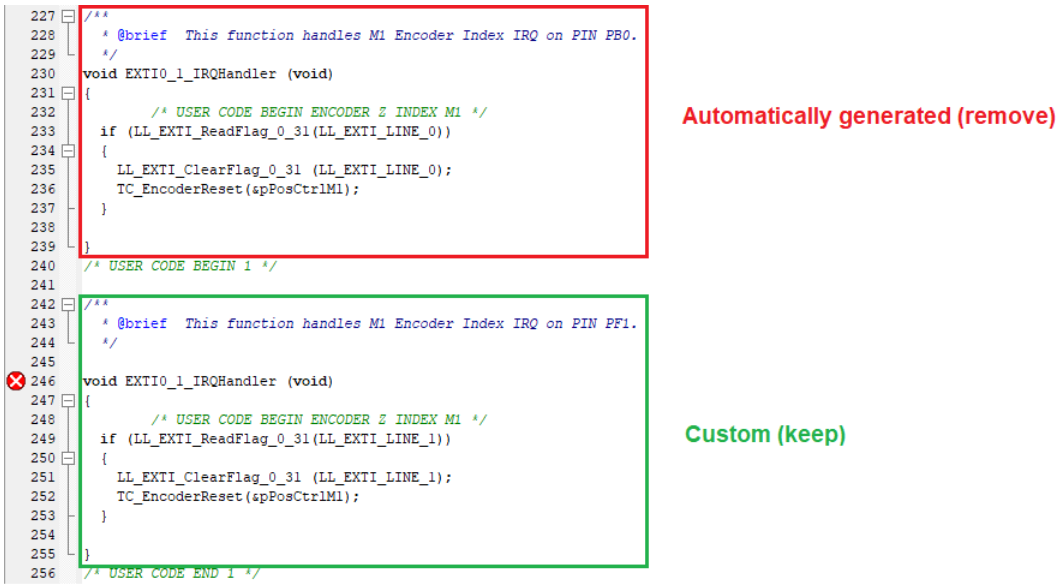
Figure 11. UART IRQ handler

```
136  void USART1_IRQHandler(void)
137  {
138    /* USER CODE BEGIN USART1_IRQn 0 */
139
140    if(LL_USART_IsActiveFlag_RXNE(USART1) & LL_USART_IsEnabledIT_RXNE(USART1))
141    {
142      pxMBFrameCBByteReceived();
143      return;
144    }
145
146    if(LL_USART_IsActiveFlag_TXE(USART1) & LL_USART_IsEnabledIT_TXE(USART1))
147    {
148      pxMBFrameCBTransmitterEmpty();
149      return;
150    }
151
152    if(LL_USART_IsActiveFlag_TC(USART1) & LL_USART_IsEnabledIT_TC(USART1))
153    {
154      LL_GPIO_ResetOutputPin(RS485_DE_GPIO_Port, RS485_DE_Pin);
155      LL_USART_ClearFlag_TC(USART1);
156      return;
157    }
158    /* USER CODE END USART1_IRQn 0 */
159    /* USER CODE BEGIN USART1_IRQn 1 */
160
161    /* USER CODE END USART1_IRQn 1 */
162  }
```

Step 13. Open stm32f0xx_mc_it.c file and replace the automatically generated EXTI0_1_IRQHandler function with the custom on available in the USER CODE 1 section.

You can find it between the /* USER CODE BEGIN 1 */ and USER CODE END 1 */ comments at the end of the file as shown below.

Figure 12. Custom EXTI0_1_IRQHandler handler



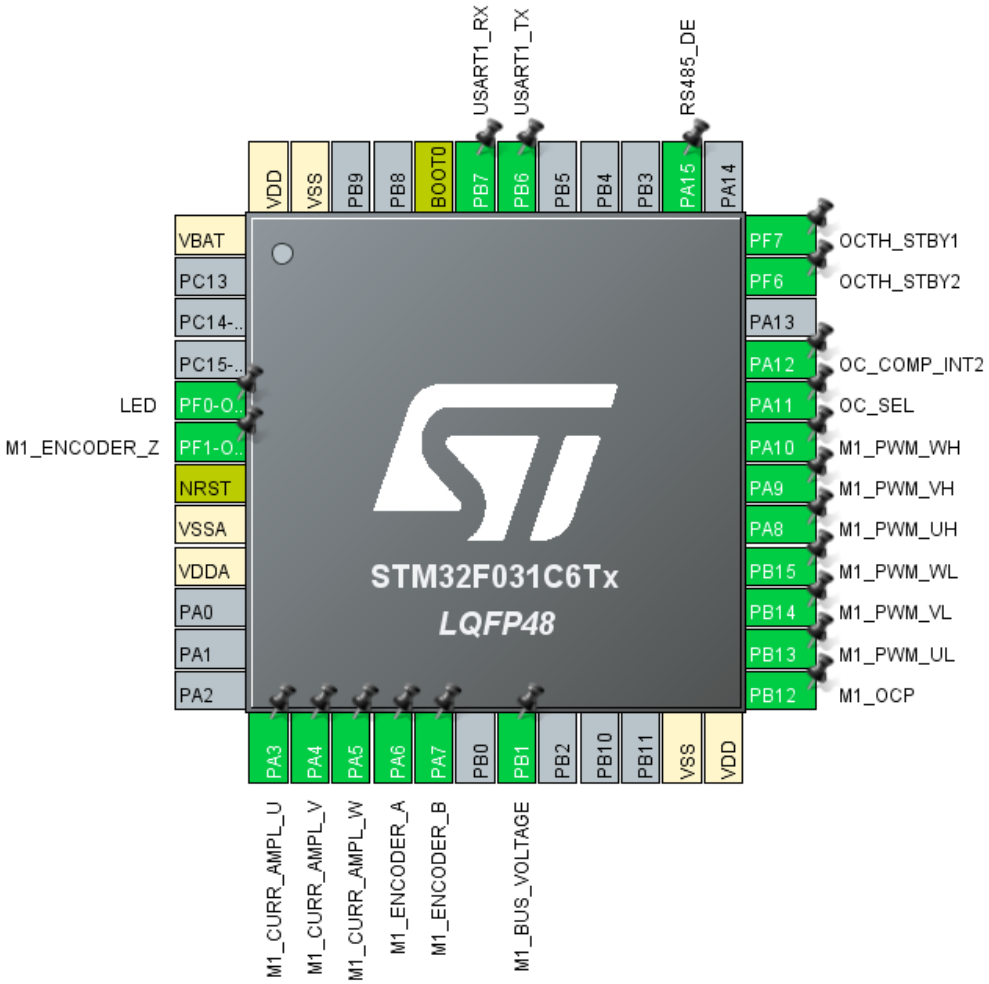
Step 14. Compile the project.

3.3 Starting from STM32CubeMX project

The package contains the STM32CubeMX project used for the code generation. Using this project, it is possible to modify the configuration of all the hardware NOT directly related to the motor control algorithm.

Note: The configuration of the peripherals related to the motor control algorithm should be modified though the MotorControl Workbench only.

Figure 13. **STM32CubeMX project file pinout configuration**



The following list of peripherals that can be re-configured using STM32CubeMX:

- Standard GPIOs
  - **LED**: status LED driving output
  - **RS485_DE**: RS485 DE signal
  - **OCTH_STBY1/OCTH_STBY2**: internal STSPIN32F0A signal setting the overcurrent threshold
  - **OC_COMP_INT2**: internal STSPIN32F0A OC comparator secondary output
  - **OC_SEL**: internal STSPIN32F0A signal setting overcurrent protection mode
  - **PA0, PA1, PA2**: unused GPIOs connected to the digital Hall effect sensors of the motor
  - **PA13, PA14**: SWD interface pins (using this pin for other functions disables the debug interface)
- UART interface
- Not motor control timers:
  - **TIM14**: MODBUS timeout counter
  - **TIM16**: MODBUS polling timer
  - **TIM2**: unused timer also connected to the digital Hall effect sensors of the motor (PA0, PA1, PA2)
  - **TIM17**: unused timer

- Other unused peripherals:
  - RTC
  - CRC
  - Watchdog

After the configuration changes, following the step-by-step procedure below:

Step 1. Select your target toolchain and check other options in the “Project Manager” tab.

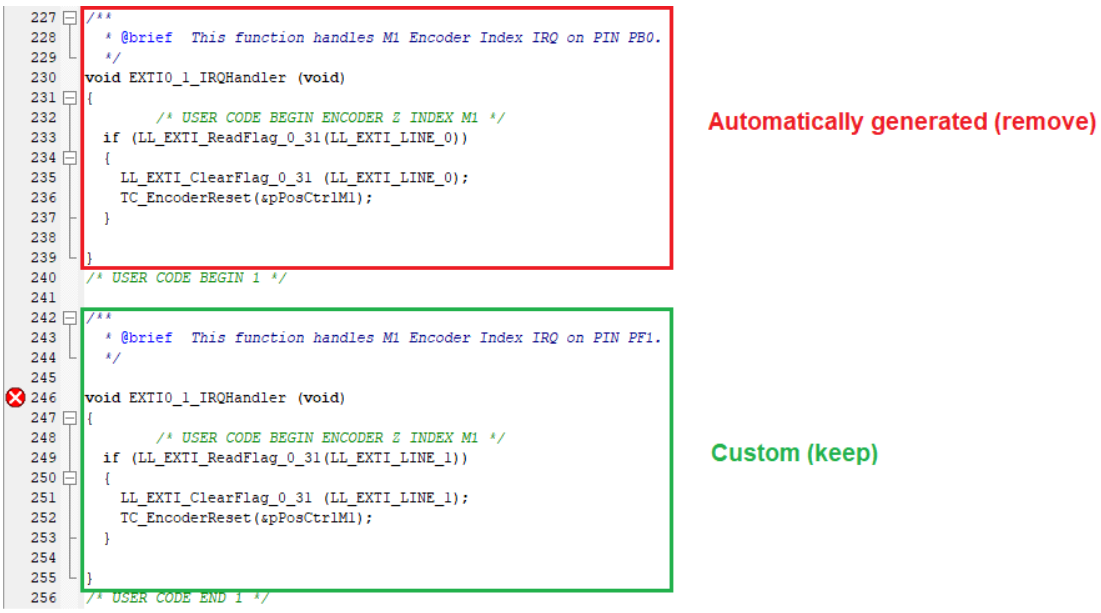
Step 2. Click on “Generate Code” button.

Step 3. After the code generation has completed, click on “open project”.

Step 4. Open stm32f0xx_mc_it.c file and replace the automatically generated EXTI0_1_IRQHandler function with the custom on available in the USER CODE 1 section.

You can find it between the /* USER CODE BEGIN 1 */ and USER CODE END 1 */ comments at the end of the file as shown below.

Figure 14. Custom EXTI0_1_IRQHandler handler



Step 5. Compile the project.

Revision history

Table 2. **Document revision history**

Date	Version	Changes
07-Apr-2020	1	Initial release.

Contents

List of figures

List of tables