elektor books

Advanced Programming with STM32 Microcontrollers

Master the software tools behind the STM32 microcontroller

Majid Pakdel

elektor
design > share > sell

Advanced Programming with STM32 Microcontrollers

Majid Pakdel

elektor
design > share > sell

an Elektor Publication

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street
London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344

First published in the United Kingdom 2020

● British Library Cataloguing in Publication Data
Catalogue record for this book is available from the British Library

● ISBN: 978-3-89576-410-3

● EISBN: 978-3-89576-411-0

● EPUB: 978-3-89576-412-7

Prepress production: DMC ¦ daverid.com
Printed in the Netherlands by Wilco

Table of Contents

• **Preamble** 8

Chapter 1 • Introduction to STM32 ARM Cortex-M Microcontrollers 11

1.1 • Introduction 11
1.2 • Software Tools 11
1.3 • STM32 Hardware Design and Serial Programmer 13
1.4 • Proposed STM32F103RET6 Header Board 15
1.5 • Summary 17

Chapter 2 • Beginning First Projects with STM32 Microcontroller 19

2.1 • Introduction 19
2.2 • Beginning the First Project 20
2.3 • The Second Project 36
2.4 • The Third Project 38
2.5 • The Fourth Project 41
2.6 • Summary 45

Chapter 3 • Adding LCD and Delay Libraries to a Project 46

3.1 • Introduction 46
3.2 • STM32CubeMX Settings and Adding LCD Library to the Project 46
3.3 • Adding LCD and Delay Libraries from Book files Download Section 51
3.4 • Summary 51

Chapter 4 • External Interrupts in STM32 Microcontrollers 52

4.1 • Introduction 52
4.2 • External Interrupt Project Settings 52
4.3 • Summary 61

Chapter 5 • Analogue to Digital Converters (ADCs) in STM32 Microcontrollers 62

5.1 • Introduction 62
5.2 • STM32CubeMX settings for ADC Project 62
5.3 • Adding Codes and Libraries to the ADC Project 66
5.4 • Summary 68

Chapter 6 • Digital to Analogue Converters (DACs) in STM32 Microcontrollers 69

6.1 • Introduction 69
6.2 • STM32CubeMX settings for DAC Project 69
6.3 • Adding Codes to main.c 73
6.4 • Summary 75

Chapter 7 • Timers and Counters in STM32 Microcontrollers 76

7.1 • Introduction 76
7.2 • Timer Project Settings 76
7.3 • Counter Project Settings 86

7.4 • Summary . . . 90

Chapter 8 • Pulse Width Modulation (PWM) in STM32 Microcontrollers . . . 91

8.1 • Introduction . . . 91
8.2 • PWM Project Settings . . . 91
8.3 • Summary . . . 96

Chapter 9 • Real-Time Clock (RTC) in STM32 Microcontrollers . . . 97

9.1 • Introduction . . . 97
9.2 • RTC Project Settings . . . 97
9.3 • Summary . . . 105

Chapter 10 • Serial Communication in STM32 Microcontrollers . . . 106

10.1 • Introduction . . . 106
10.2 • Sending Data Project Settings . . . 107
10.3 • Receiving Data Project Settings . . . 112
10.4 • Summary . . . 115

Chapter 11 • Synchronous Peripheral Interface (SPI) in STM32 Microcontrollers . . . 116

11.1 • Introduction . . . 116
11.2 • SPI Settings for Master Microcontroller . . . 118
11.3 • SPI Settings for Slave Microcontroller . . . 118
11.4 • Header Code of main.c for Master Microcontroller . . . 120
11.5 • Header Code of main.c for the Slave Microcontroller . . . 121
11.6 • The main.c File Settings for Master and Slave Microcontrollers . . . 123
11.7 • Summary . . . 125

Chapter 12 • Watchdog Timer in STM32 Microcontrollers . . . 126

12.1 • Introduction . . . 126
12.2 • WWDG Timer Project Settings . . . 127
12.3 • IWDG Timer Project Settings . . . 130
12.4 • Summary . . . 132

Chapter 13 • Inter-Integrated Circuit (I^2C) in STM32 Microcontrollers . . . 133

13.1 • Introduction . . . 133
13.2 • I^2C Settings for Slave Microcontroller . . . 134
13.3 • I^2C Settings for Master Microcontroller . . . 137
13.4 • Summary . . . 140

Chapter 14 • Direct Memory Access (DMA) in STM32 Microcontrollers . . . 141

14.1 • Introduction . . . 141
14.2 • DMA Settings for ADC . . . 141
14.3 • Timer, ADC, and DMA Project Settings . . . 144
14.4 • DMA and USART Project Settings . . . 148
14.5 • Summary . . . 152

Chapter 15 • Real-Time Operating System (RTOS) in STM32 Microcontrollers 153

15.1 • Introduction 153
15.2 • FreeRTOS Project Settings 154
15.3 • Summary 158

Chapter 16 • STM32 Microcontrollers Programming using STM32duino 159

16.1 • Introduction 159
16.2 • Installing STM32duino in Arduino IDE 159
16.3 • Automation Control System Programming 166
16.4 • Multitasking in PLC Programming 170
16.5 • Summary 175

Chapter 17 • Finite State Machine (FSM) Programming in STM32 Microcontrollers . . 176

17.1 • Introduction 176
17.2 • FSM programming using mikroC PRO for ARM 178
17.3 • FSM Programming using Mbed Online Compiler 184
17.4 • FSM Modular Programming Using Functions 188
17.5 • Summary 191

Chapter 18 • STM32 Microcontrollers Programming using MATLAB/Simulink 192

18.1 • Introduction 192
18.2 • Pulse Generation Programming using Simulink 192
18.3 • SPWM Generation Programming for Three Phase Inverter using Simulink 205
18.4 • Summary 209

• **Index** 212

• Preamble

This book covers topics related to various programming methods, hardware, and software tools for the STM32. Various software tools have been used in the book, including Keil MDK ARM (uVison5 IDE), IAR Embedded Workbench for ARM, SW4STM32, Mbed online compiler, mikroC PRO for ARM, Arduino IDE (STM32duino), and MATLAB/Simulink embedded coder. Most of these are freely available over the internet. Proteus has been used for simulation of some programs and in PCB design of the proposed STM32F103RET6 based development board mentioned in chapter 1. Readers can use the STM32F103RET6 based header board, or lower cost, cheaper header boards including blue pill (STM32F103C8T6), NUCLEO-F103RB, STM32F030F4P6 Mini Development Board, Discovery kit with STM32F407VG MCU, etc. for hardware implementation of the programs contained in this book.

Chapter Overview:

Chapter 1
Introduction to STM32 ARM Cortex-M 32-bit Microcontrollers: An Introduction to various software tools used in STM32 microcontroller programming. Serial programming in boot mode is explained in detail.

Chapter 2
Beginner programming with STM32: Simple projects with SMT32CubeMX, IAR Embedded Workbench for ARM, and simulation using Proteus.

Chapter 3
Adding LCD and Delay Libraries to a Project: Learning how to add a library including the start-up functions of different external hardware and various functions to a project where they are not in HAL libraries.

Chapter 4
External Interrupts in STM32 Microcontrollers: External interrupts and their settings are discussed in detail.

Chapter 5
Analogue to Digital Converters (ADCs) in STM32 Microcontrollers: Analogue to digital converters (ADCs) are very important tools for designing hardware that produces analogue signals and their connection to STM32 microcontrollers. ADC settings are explained in detail.

Chapter 6
Digital to Analogue Converters (DACs) in STM32 Microcontrollers: Digital to analogue converters (DACs) operate inversely to analogue to digital converters and convert digital values to analogue. DAC settings are discussed in detail.

Chapter 7
Timers and Counters in STM32 Microcontrollers: All timing, scheduling, and delays

without stopping the program are managed by a timer operating in parallel with the processor. A timer is a counter that can count the regular pulses of an oscillator and generate regular times. This counter can count the external pulses applied to a microcontroller. Timer and counter settings are explained in detail.

Chapter 8
Pulse Width Modulation (PWM) in STM32 Microcontrollers: PWM wave is generated in Timer units which can produce PWM signals. PWM settings are discussed in detail.

Chapter 9
Real-Time Clock (RTC) in STM32 Microcontrollers: The Real-Time Clock (RTC) is one of the intelligent system design requirements for accessing real-time and date. RTC settings are discussed in detail.

Chapter 10
Serial Communication in STM32 Microcontrollers: Serial communication is one of the important sections in communication between a microcontroller and other devices. The serial communication protocol settings are explained in detail.

Chapter 11
Synchronous Peripheral Interface (SPI) in STM32 Microcontrollers: The SPI protocol is a synchronous bidirectional serial communication interface. This interface is one of the important and useful communication protocols in microcontrollers since it operates at a very high speed. SPI protocol settings are explained in detail.

Chapter 12
Watchdog Timer in STM32 Microcontrollers: The watchdog timer is one of the important fundamentals in microcontrollers. It is used to reset the CPU when it hangs. STM32 microcontrollers have two watchdog timer units, called the Independent Watchdog (IWDG) and Window Watchdog (WWDG). Watchdog timer settings are discussed in detail.

Chapter 13
Inter-Integrated Circuit (I^2C) in STM32 Microcontrollers: The Inter-Integrated Circuit (I^2C) is a serial bus protocol consisting of two signal lines known as SCL and SDL. These are used for communication. I^2C protocol settings are explained in detail.

Chapter 14
Direct Memory Access (DMA) in STM32 Microcontrollers: To avoid holding the CPU processing time, most advanced microcontrollers have a Direct Memory Access (DMA) controller. Data transfers using DMA are implemented between memory locations without the need for CPU. DMA settings for different projects are discussed in detail.

Chapter 15
Real-Time Operating System (RTOS) in STM32 Microcontrollers: This chapter focuses on how to use a real-time OS (RTOS) to create a real-time application on an STM32 microcontroller.

Chapter 16
STM32 Microcontrollers Programming Using STM32duino: STM32 microcontrollers are particularly popular because they are programmable using the Arduino IDE (STM32duino). In this way, everybody can easily get started and build projects with STM32. In this chapter, we use the Arduino IDE to program the STM32

Chapter 17
Finite State Machine (FSM) Programming in STM32 Microcontrollers: A common design technique used in industrial automation control systems is the venerable finite state machine (FSM). Industrial automation control designers use this programming method to break complex problems into manageable states and transitions. The FSM programming method is explained using mikroC PRO for ARM and Mbed online compilers in detail.

Chapter 18
STM32 Microcontrollers Programming Using MATLAB/Simulink: The user is rapidly able to produce a model using Simulink that would otherwise require hours to build in the laboratory environment. Simulink supports linear and non-linear systems, modelled in continuous time, sampled time, or a hybrid of the two. In this chapter, we show how to generate code for STM32 microcontrollers using Simulink.

Chapter 1 ● Introduction to STM32 ARM Cortex-M Microcontrollers

1.1 ● Introduction

The STM32 series is a popular, cheap, high-performance microcontroller variant. They also have lots of support from various microcontroller development software suites. STM32 microcontrollers offer a large number of peripherals that can be interfaced with all kinds of electronic components including sensors, displays, electric motors, etc. The range of performance available with the STM32 is very extensive. Some of the most basic STM32 microcontrollers (STM32F0 and STM32F1 series) start with a 24 MHz clock frequency and are available in packages with as few as 16 pins while the STM32H7 operates at up to 400 MHz, and is available in packages with 240 pins. Moreover, the STM32L series has been designed for low-power applications running from a small battery. Development software is required to develop code, program the microcontroller, and debug. Development tools must include a compiler, debugger, and an in-circuit serial programmer (ICSP) as shown in Figure 1.1.

Figure 1.1: ICSP programmer diagram

1.2 ● Software Tools

There are several software tools available for programming STM32 microcontrollers. Most are available as Integrated Development Environments (IDE).

Common programming software tools include:

- Keil MDK ARM (uVison5 IDE) – The MDK ARM IDE is a very stable development environment allowing the development of programming code.
- IAR Embedded Workbench for ARM – IAR Embedded Workbench and the included IAR C/C++ compiler generates the fastest performing, most compact code in the industry for ARM-based applications.
- SW4STM32 – The System Workbench toolchain, called SW4STM32, is a free multi-OS software development environment based on Eclipse, which supports the full range of STM32 microcontrollers and associated boards. The SW4STM32 toolchain may be obtained from the website www.openstm32.org, which includes forums, blogs, and training for technical support. The System Workbench toolchain and its collaborative website have been built by AC6, a service company providing training and consultancy on embedded systems.

- Mbed online compiler – Jump directly into application development with Mbed OS without installing anything. Create a Mbed account to get started.
- CoIDE – A free toolchain based on the Eclipse IDE integrated along with an embedded ARM version of the free GCC compiler.
- mikroC PRO for ARM – C compiler for writing fast multimedia applications for ARM Cortex M3 and M4 devices using mikroC programming environment.
- Arduino IDE – STM32duino (https://github.com/stm32duino/wiki/wiki/Getting-Started).
- Matrix-Flowcode 8 – ARM MCUs including the popular STM32 ARM family are also supported in Flowcode.
- MATLAB/Simulink embedded coder or supported hardware packages including STM32 development boards.

ARM Cortex-M microcontrollers support two programming protocols: JTAG (named by the electronics industry association the Joint Test Action Group) and Serial Wire Debug (SWD). The block diagram of peripheral devices connected to a microcontroller with some communication protocols (which is called a system-level block diagram) is shown in Figure 1.2.

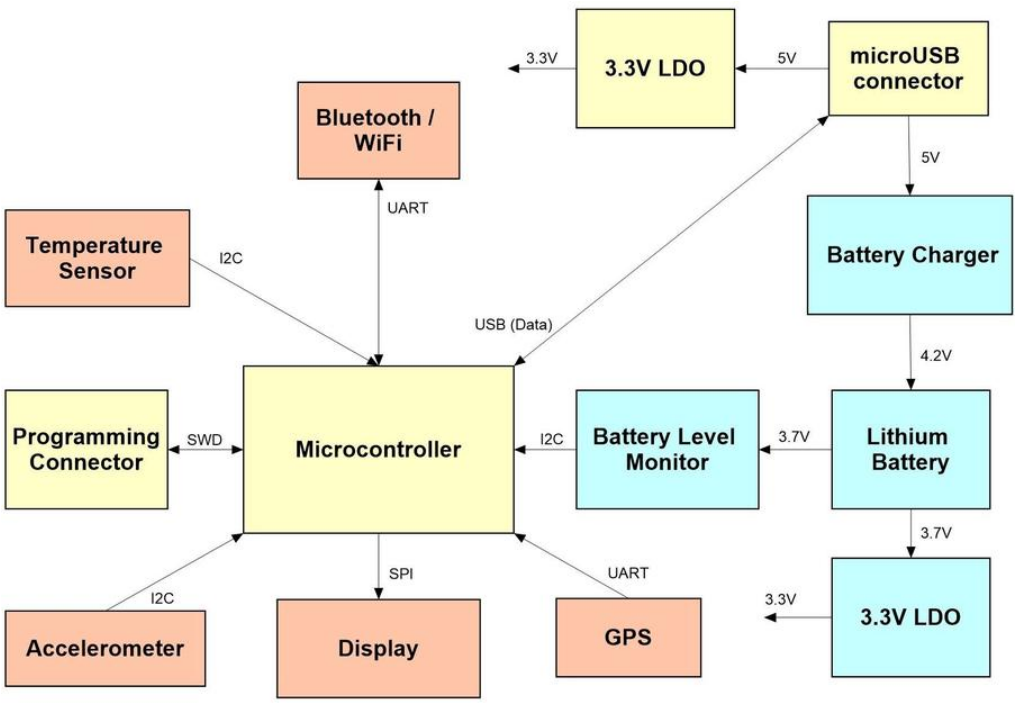

Figure 1.2: System-level block diagram

ARM Cortex-M is a 32-bit microcontroller which is generally the best choice for computationally intensive tasks compared to what is available from older 8-bit microcontrollers such as

the 8051, PIC, and AVR microcontrollers. ARM microcontrollers come in multiple types including the Cortex-M0, M1, M3, M4, and M7. Some versions are available with a Floating Point Unit (FPU) and are designated with an F in the Cortex-M4F microcontrollers. One of the benefits of Arm Cortex-M processors is their low price and high performance. Even if an 8-bit microcontroller is sufficient for your application, it might be better to consider a 32-bit Cortex-M microcontroller. There are Cortex-M microcontrollers available with very comparable pricing to some of the older 8-bit chips. A 32-bit microcontroller gives you more flexibility to extend your program and add additional features when required. STM32 microcontrollers can be divided into several subseries as shown in Figure 1.3.

STM32 Series	Cortex-Mx	Max clock (MHz)	Performance (DMIPS)
F0	M0	48	38
F1	M3	72	61
F3	M4	72	90
F2	M3	120	150
F4	M4	180	225
F7	M7	216	462
H7	M7	400	856
L0	M0	32	26
L1	M3	32	33
L4	M4	80	100
L4+	M4	120	150

Figure 1.3: Comparison of various STM32 microcontrollers

1.3 ● STM32 Hardware Design and Serial Programmer

A typical STM32F103RET6 based development board is shown in Figure 1.4. A USB to serial converter IC (FT232RL) is used to produce a virtual serial port when connecting to the PC via USB port. The TXD and RXD pins with FT232RL are connected to the RXD1 pin (PA10) and TXD1 pin (PA9) of the microcontroller. Since the FT232RL IC is generally expensive, we can use the PL2303 IC instead, as shown in Figure 1.5, or the PL2303-USB to TTL adapter module as shown in Figure 1.6. The programming of the microcontroller can be done by this virtual serial port. For programming the microcontroller, it should be set to BOOT mode. If the BOOT0 pin of the microcontroller is connected to VCC3.3 and its BOOT1 pin is connected to GND and the microcontroller is reset once using the RESET button, the microcontroller enters BOOT mode. In this mode, the microcontroller can be programmed through the virtual serial port. After programming the microcontroller for exiting from BOOT mode, BOOT0 should be connected to GND. In this case, BOOT1 can be used freely by the user. In normal operation, the BOOT0 pin of the microcontroller should be connected to GND. The STM32 Flash Loader Demonstrator software as illustrated in Figure 1.7 is used for serial port programming. For programming using this software, the microcontroller should be in BOOT mode. After programming, the microcontroller should exit from BOOT mode.

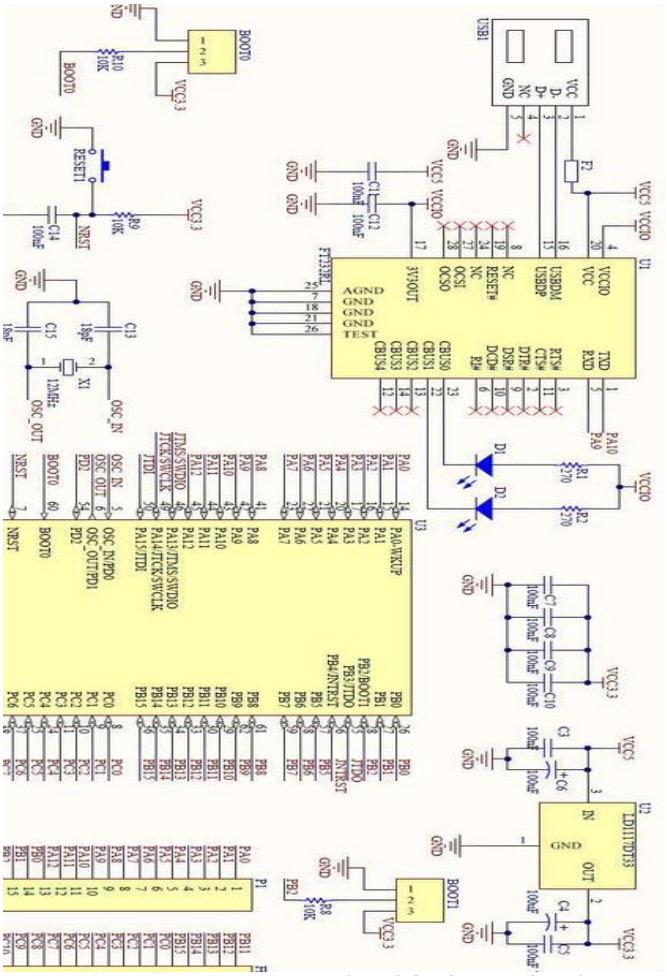

Figure 1.4: STM32F103RET6 based development board

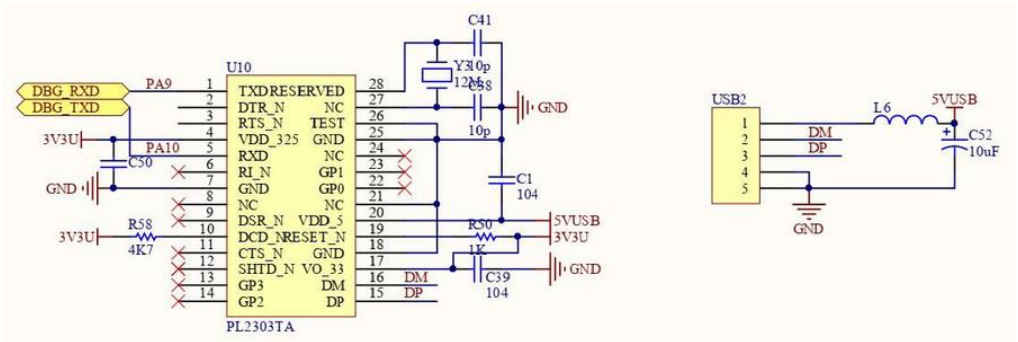

Figure 1.5: STM32F103RET6 serial programming using PL2303

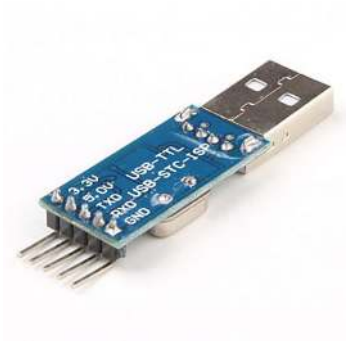

Figure 1.6: PL2303-USB to TTL adapter module

Figure 1.7: STM32 flash loader demonstrator software

1.4 ● Proposed STM32F103RET6 Header Board

The proposed minimum design schematic for the STM32F103RET6 microcontroller, the PCB layout using Proteus software, and the hardware implementation of the header board designed for the STM32F103RET6 microcontroller are shown in Figures 1.8, 1.9, and 1.10 respectively.

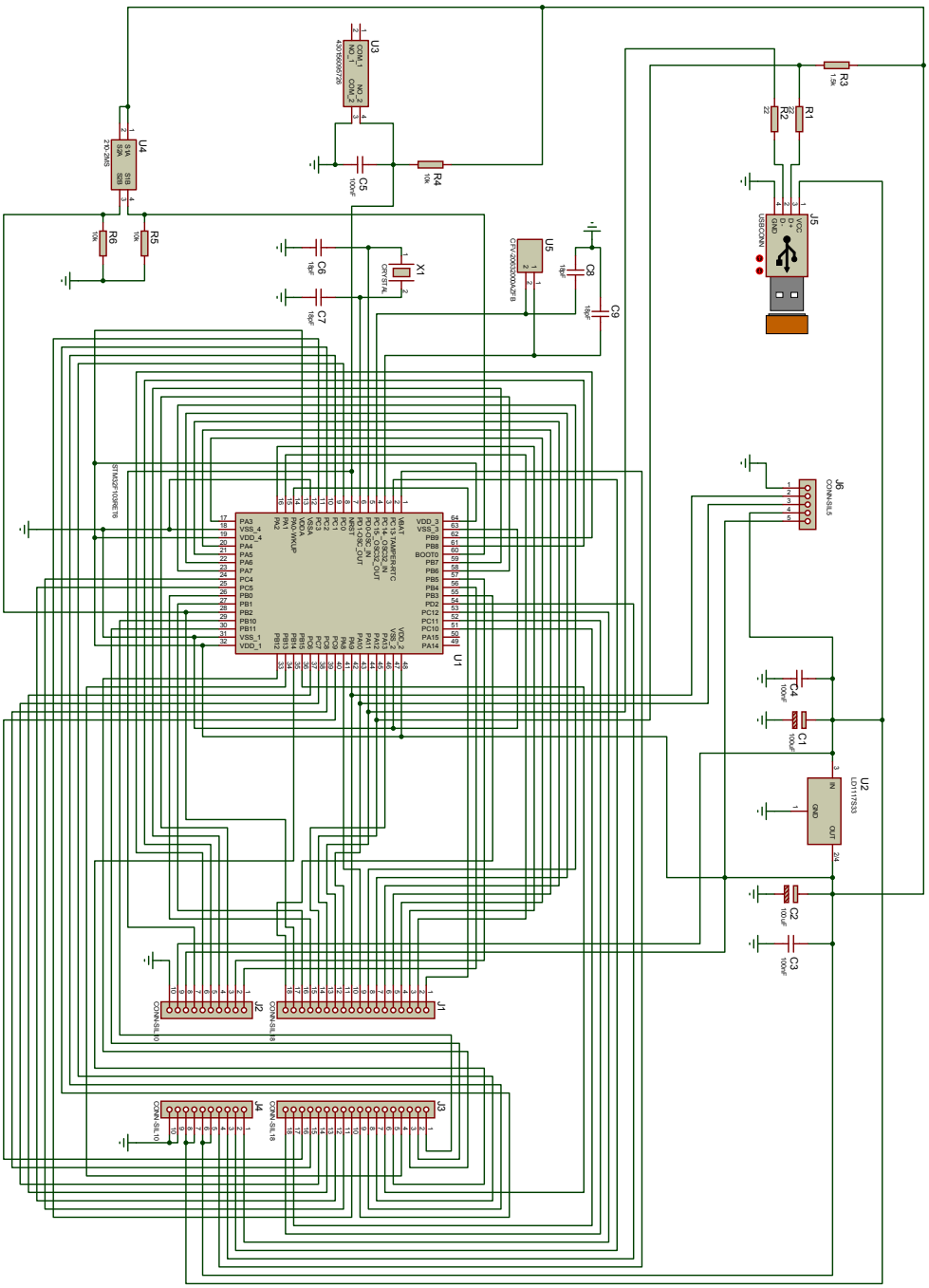

Figure 1.8: The proposed minimum designed schematic for the STM32F103RET6 microcontroller

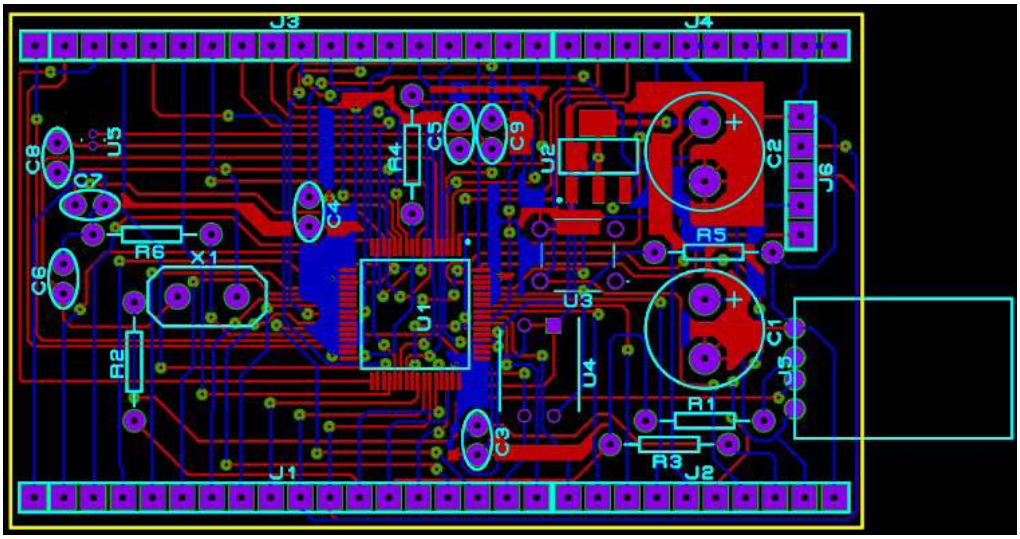

Figure 1.9: The PCB layout for the proposed STM32F103RET6 microcontroller header board

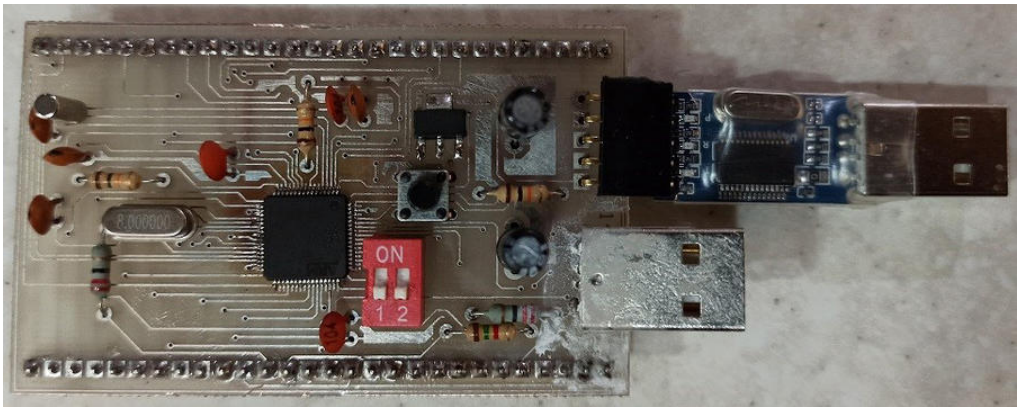

Figure 1.10: Hardware implementation of the designed header board using the STM32F103RET6 microcontroller

1.5 ● Summary

The STM32 microcontroller series is well-liked and used in a wide variety of applications. They are inexpensive and have high-performance capabilities. They also support various microcontroller development software suites. The ARM Cortex-M is a 32-bit microcontroller which is the best choice for more computationally intensive tasks compared to what is available from older 8-bit microcontrollers such as the 8051, PIC, and AVR microcontrollers. An Introduction to various software tools for STM32 microcontrollers programming and serial programming in boot mode was explained in detail. Readers can use the proposed STM32F103RET6-based header board, or lower cost, cheaper alternatives such as blue pill (STM32F103C8T6), NUCLEO-F103RB, STM32F030F4P6 Mini Development Board, Discovery

kit with STM32F407VG MCU, etc. Hardware implementations of programs are presented in the next chapters.

Chapter 2 ● Beginning First Projects with STM32 Microcontroller

2.1 ● Introduction

Firstly, STM32CubeMX software should be installed and executed. For creating a new project, the libraries of the specifically used microcontroller family should be installed. From the Help menu, install NEW Libraries, as shown in Figure 2.1.

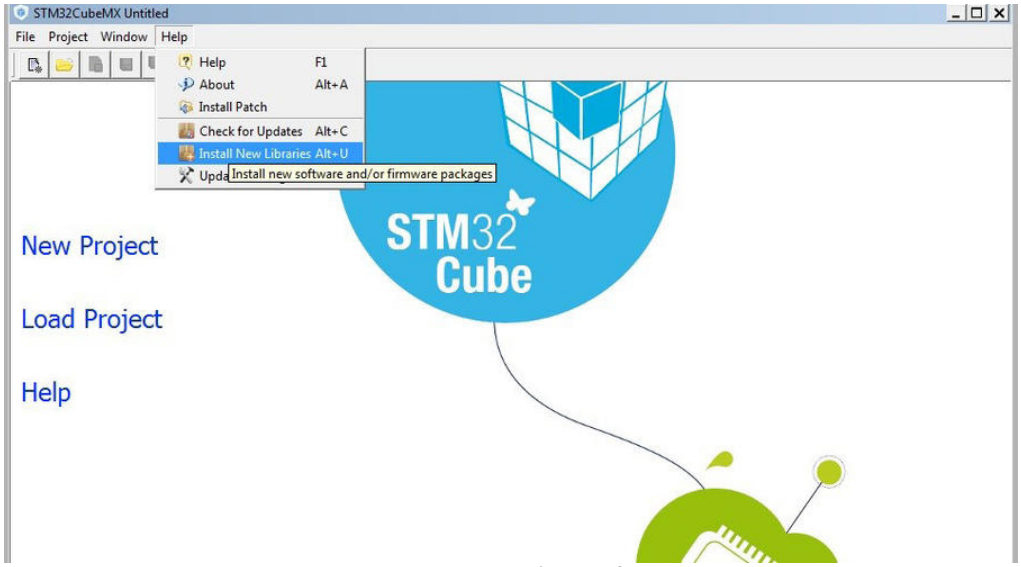

Figure 2.1: STM32CubeMX software

In the pop-up window, choose the Firmware Package for Family STM32F1 and click the 'Install Now' button as shown in Figure 2.2. After installation, a green square is shown beside the selected icon.

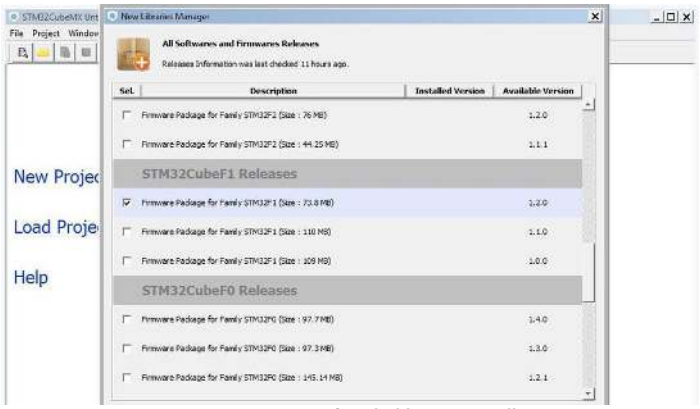

Figure 2.2: STM32F1 family library installation

2.2 • Beginning the First Project

After installation of the STM32F1 family library, come back to the software main window and select the 'New Project' icon to open the New Project window. From the MCU Selector tab, the STM32F103RETx is selected as illustrated in Figure 2.3.

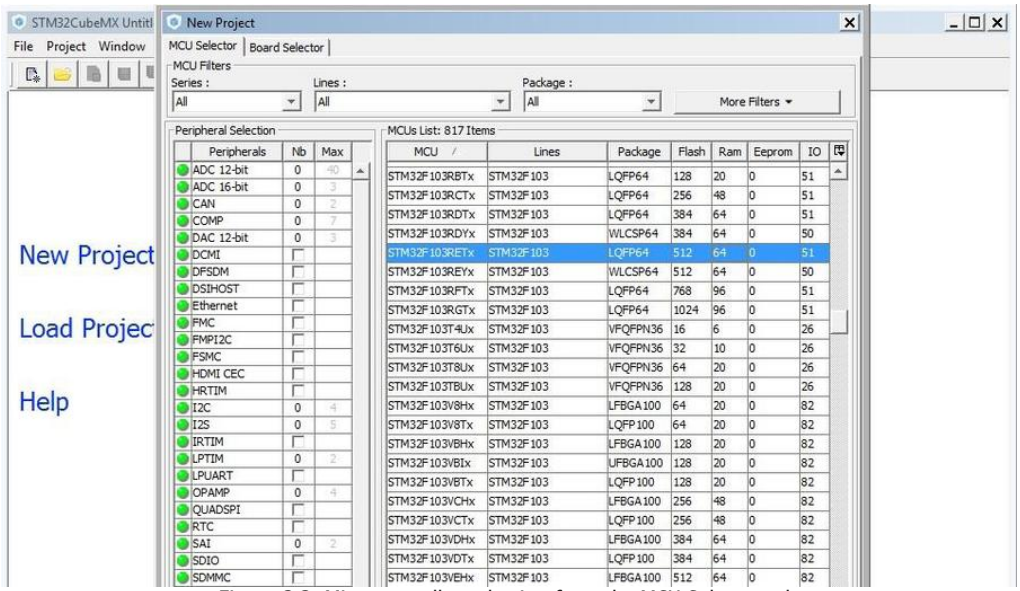

Figure 2.3: Microcontroller selecting from the MCU Selector tab

After clicking the OK button, a window pops up as shown in Figure 2.4.

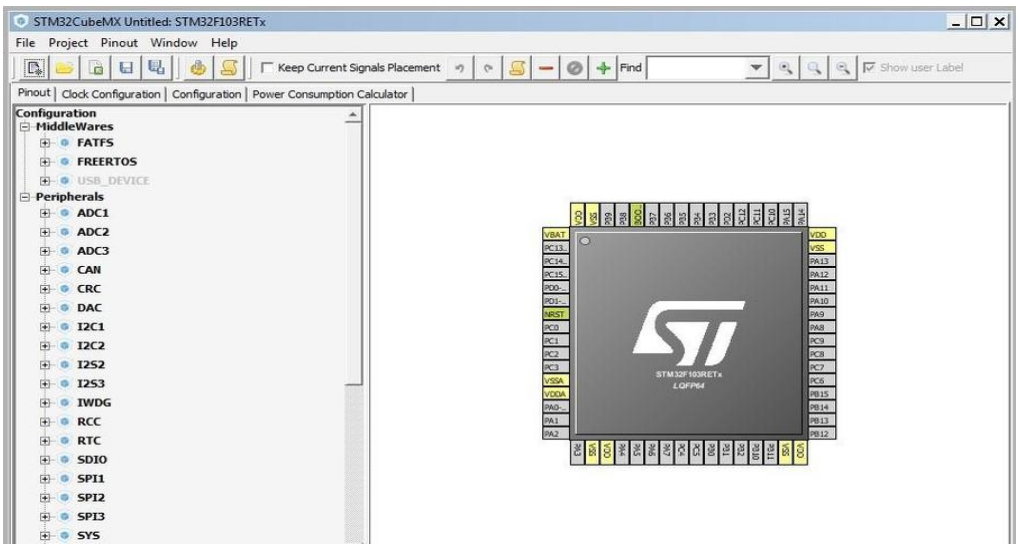

Figure 2.4: The STM32F103RETx microcontroller

We click on the PA2 pin and select the 'GPIO_Output' icon as shown in Figure 2.5.

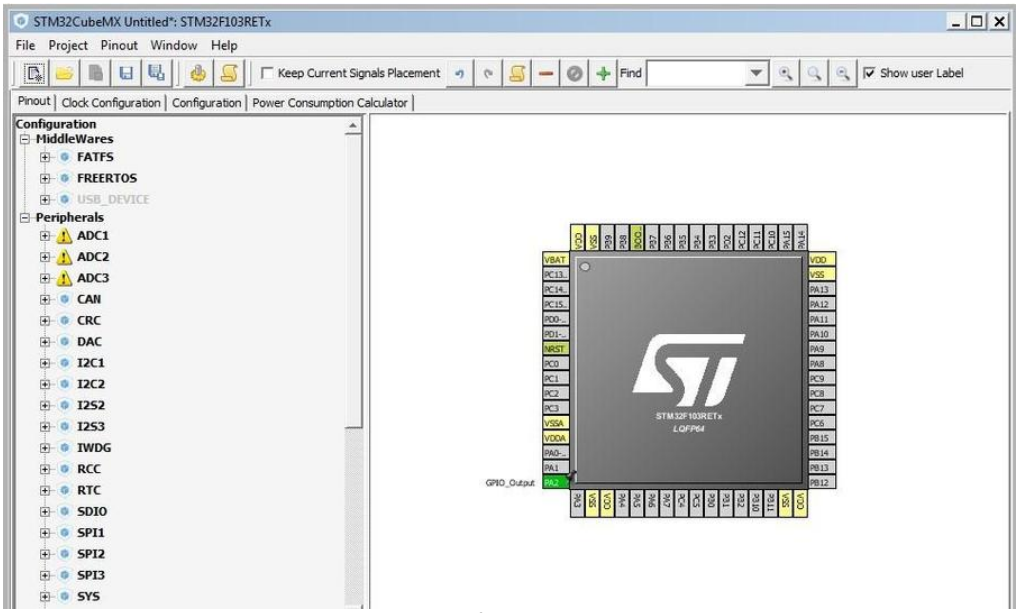

Figure 2.5: Setting the PA2 pin as GPIO_Output

The Clock Configuration tab is used for setting the frequency of the microcontroller clock. This microcontroller has two internal oscillators (8 MHz and 40 kHz). In Figure 2.6, HSE and LSE correspond to High-Speed and Low-Speed External which are related to the supplied clock with an external resonator. In the same manner, HSI and LSI represent High-Speed Internal and Low-Speed Internal respectively.

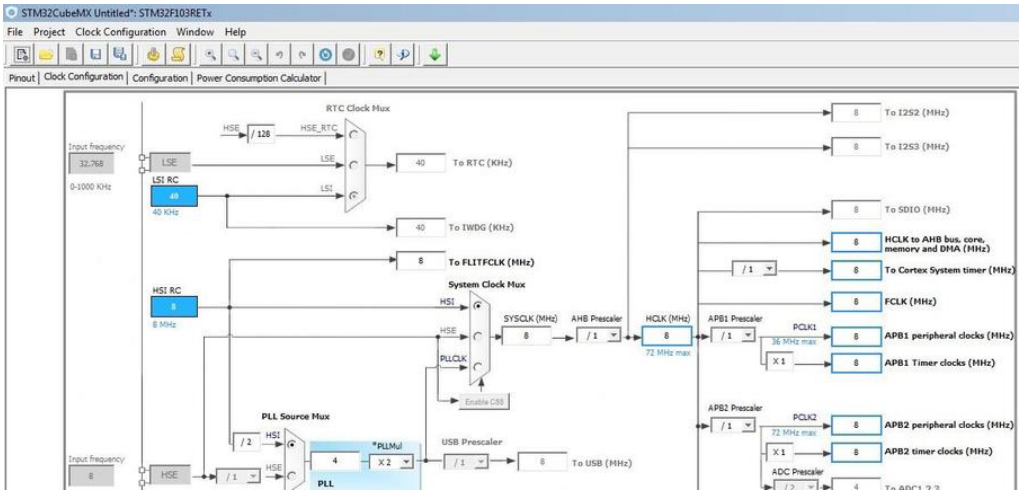

Figure 2.6: The Clock Configuration tab

The Configuration tab is used for setting the hardware enabled in the Pinout section. The GPIO button is in the system section as illustrated in Figure 2.7. When clicking on the GPIO button, the Pin Configuration window pops up, as shown in Figure 2.8.

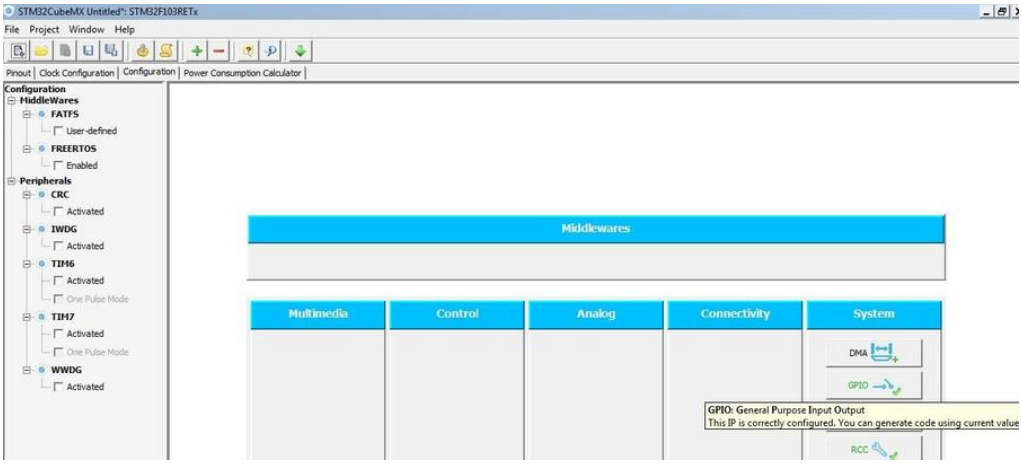

Figure 2.7: The Configuration section

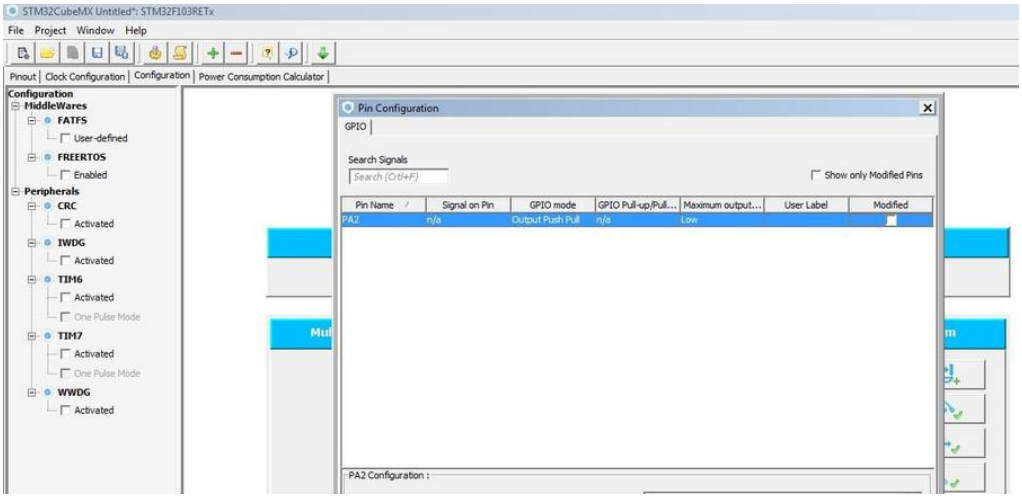

Figure 2.8: GPIO in the Pin Configuration section

In the Power Consumption Calculator section, the user can calculate the power consumed according to the hardware enabled by using a steady supply or battery and generating a report. If the power is supplied from a battery, the battery used should be selected by clicking on the 'Select Battery' button as shown in Figure 2.9.

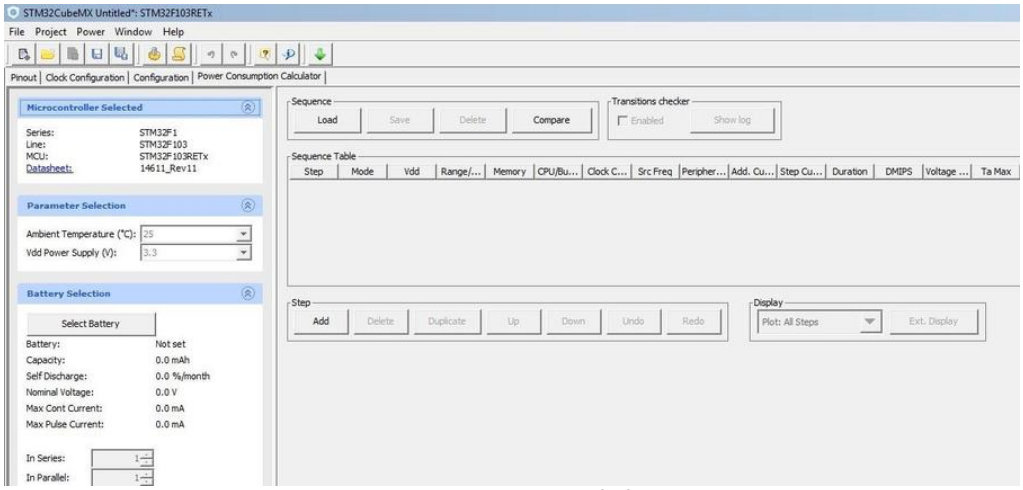

Figure 2.9: Power Consumption Calculator section

In the 'Available batteries' window, as shown in Figure 2.10, there is a list of batteries that the user can select. If the battery used is not available on the list, the user can add it by clicking on the 'Add Battery' button. The user can then specify the characteristics of the battery used in the pop window as shown in Figure 2.11.

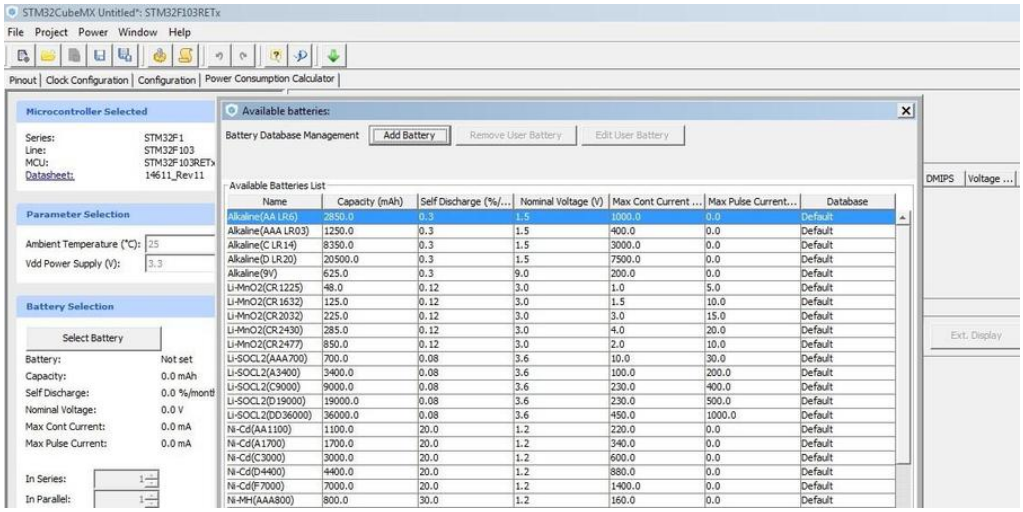

Figure 2.10: Used battery selection from the available battery list

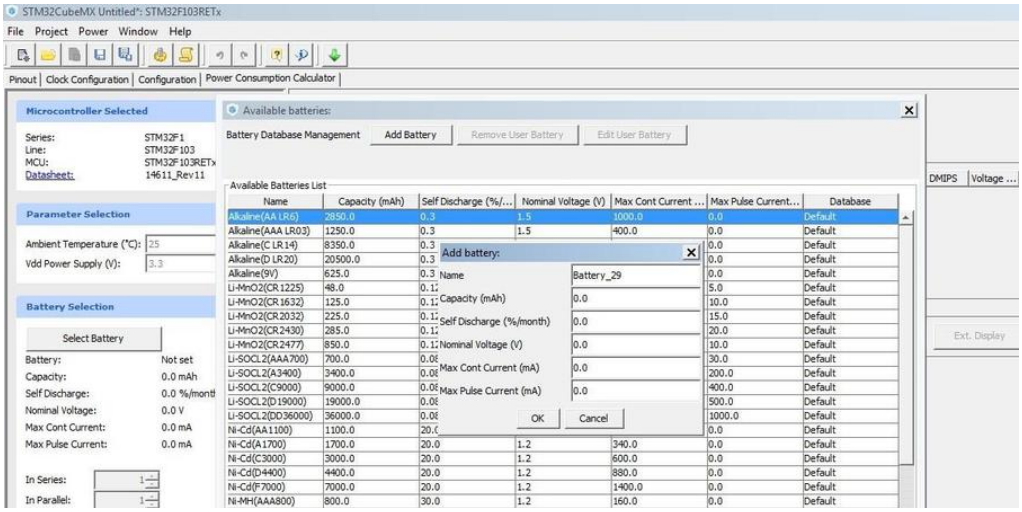

Figure 2.11: Adding a new battery

After selecting a battery, by clicking on the Add button in the Power Consumption Calculator section, the New step window appears as shown in Figure 2.12. In this window, the microcontroller operation condition, clock configuration, and selected hardware in the Pinout section are determined to calculate the consumed power. After determining the characteristics, click the 'Add' button to generate the final results. These results have been depicted in Figures 2.13 and 2.14.

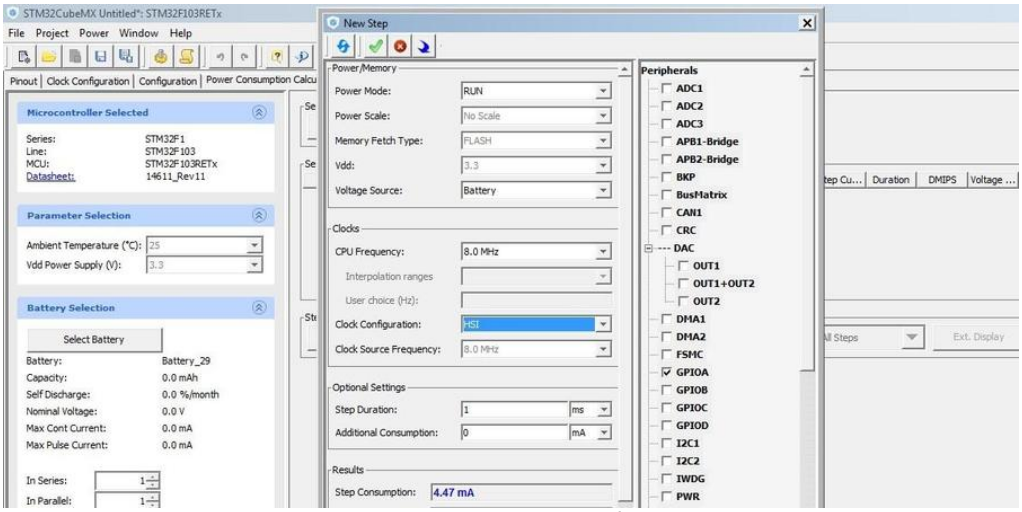

Figure 2.12: New step window

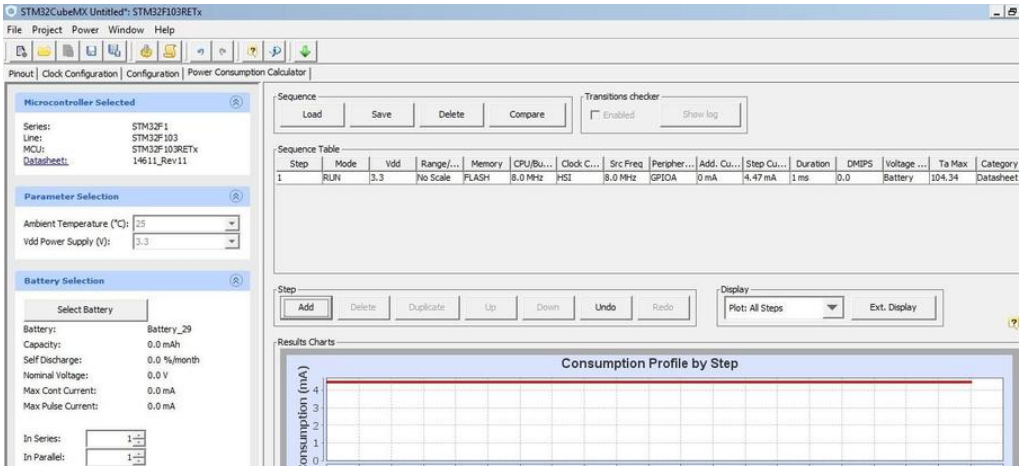

Figure 2.13: Power calculation result

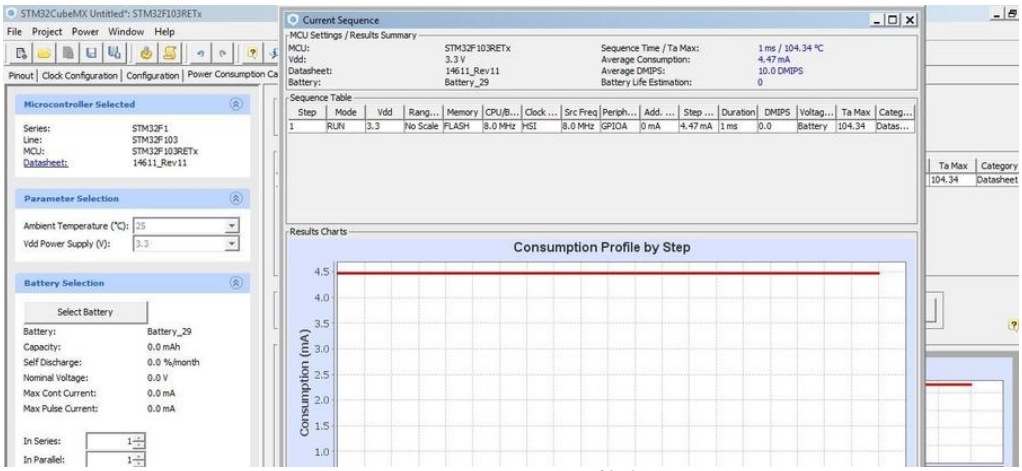

Figure 2.14: Consumption profile by step

After inputting the required settings, the project is ready to be generated in Keil, IAR, or SW4STM32. For this purpose, from the Project menu, select the 'Settings' icon to open the Project Settings window as shown in Figure 2.15. The Project Settings window constitutes Project and Code Generator tabs. The project name, location, and programming software are selected in the Project tab. In the Code Generator tab, libraries and project generation conditions can be adjusted, as shown in Figure 2.16.

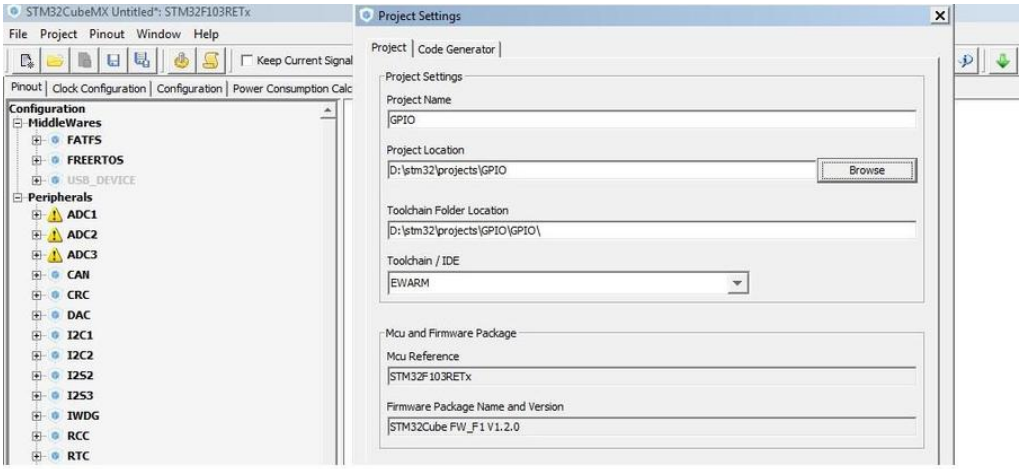

Figure 2.15: Project Settings window

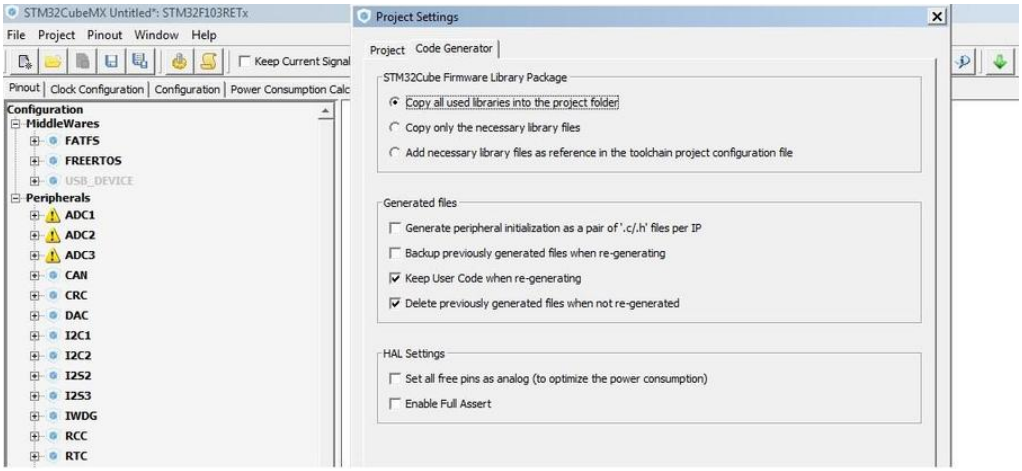

Figure 2.16: Code Generator window

After configuration, the project can be generated. Thus, in accordance with Figure 2.17 from the project menu, select the ‘Generate Code’ icon. Generate Report gives a complete report of the settings and conditions of the project in a PDF file format.

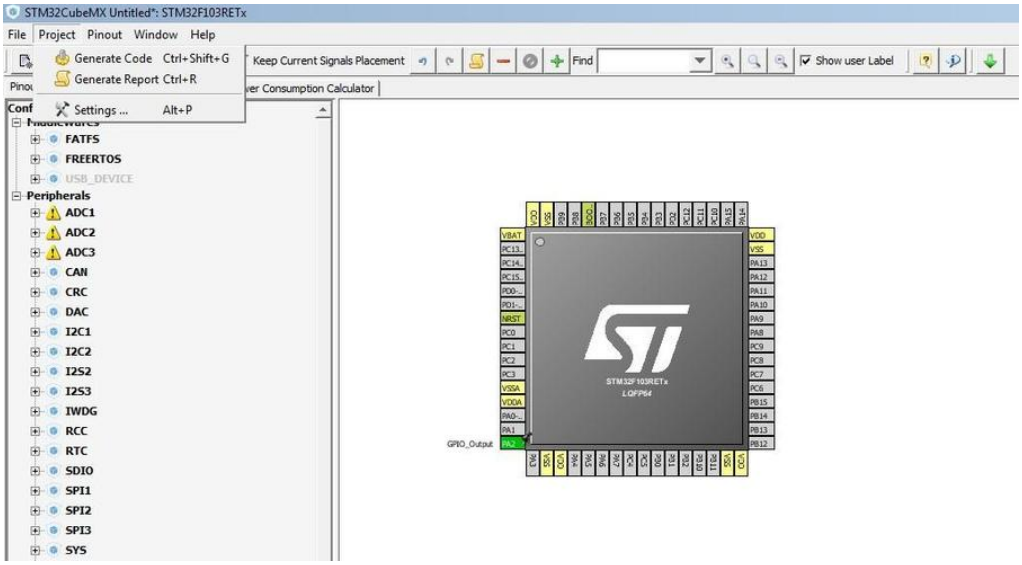

Figure 2.17: Code generating window

After generating the project, the Code Generation message appears as shown in Figure 2.18. When clicking the open project button, the project executes in the IAR software environment as shown in Figure 2.19.

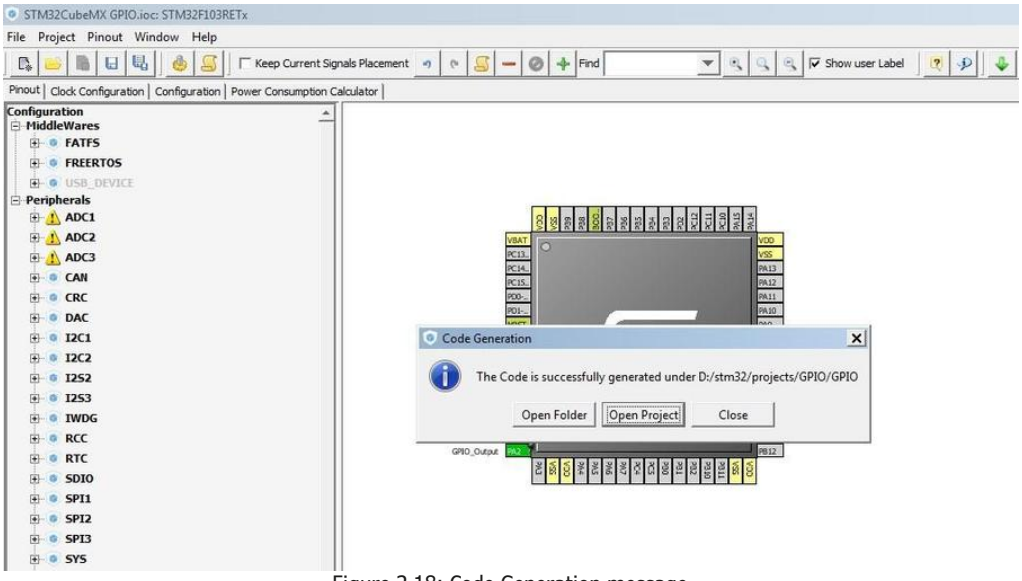

Figure 2.18: Code Generation message

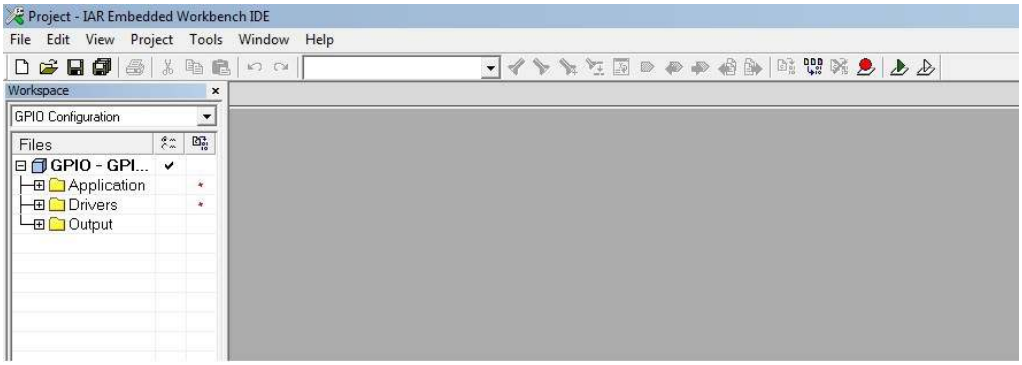

Figure 2.19: Project in IAR Embedded Workbench IDE

To set the project structure, right-click on the project name and select the 'Options' icon as shown in Figure 2.20.

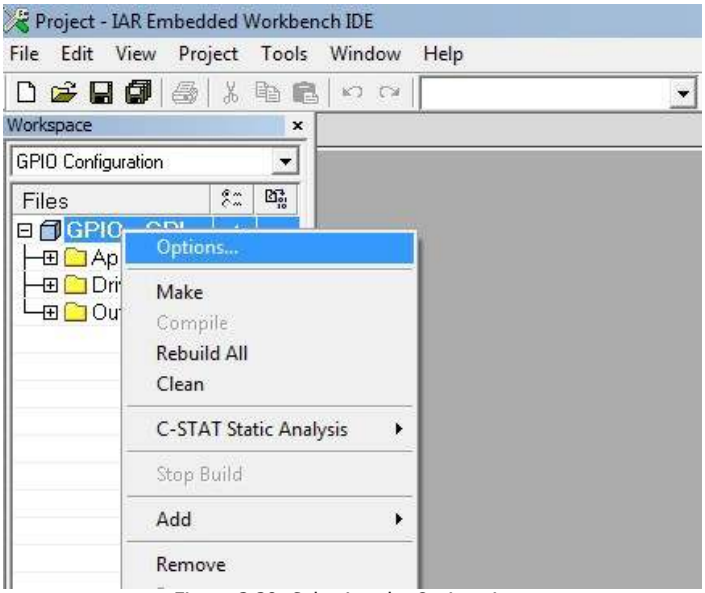

Figure 2.20: Selecting the Options icon

From the General Options select the 'Target' tab which relates to selecting the microcontroller device as shown in Figure 2.21.

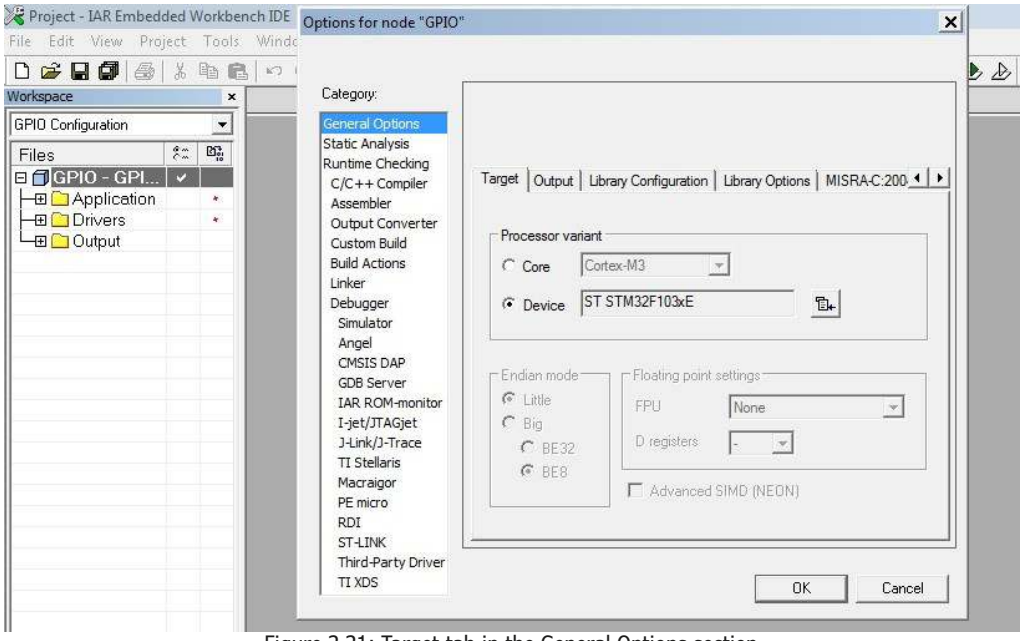

Figure 2.21: Target tab in the General Options section

The Library Configuration tab in the General Options section is related to the use of C/C++ libraries as shown in Figure 2.22.

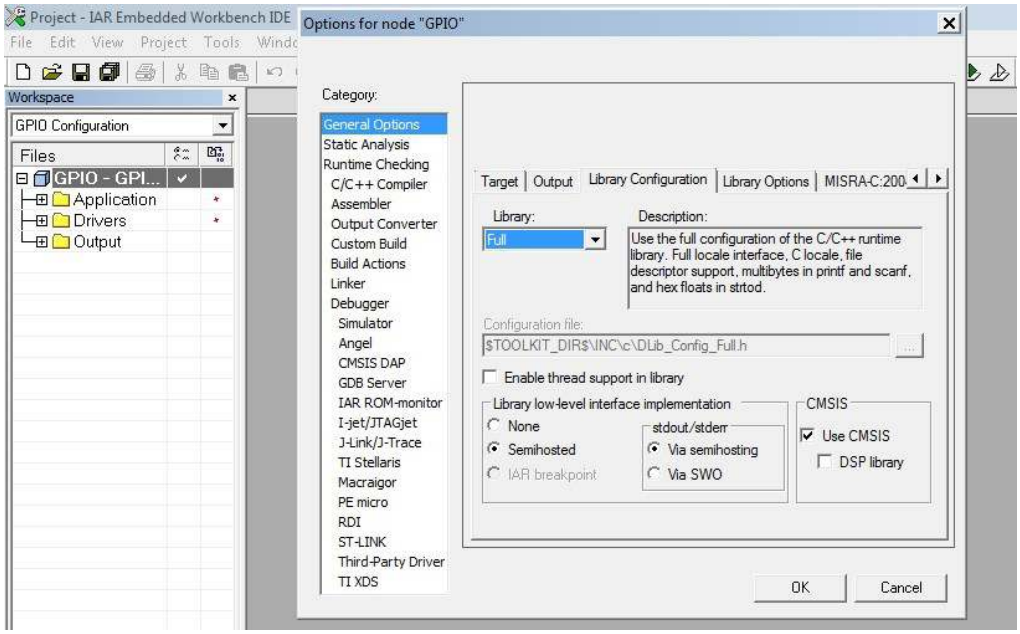

Figure 2.22: Setting the use of C/C++ libraries in the Library Configuration tab.

The Output tab in the Output Converter section is related to generating a hex file for programming the microcontroller. It is necessary to generate a hex file for programming the microcontroller using different programming software. Edit the Output tab in the Output Converter section as shown in Figure 2.23.

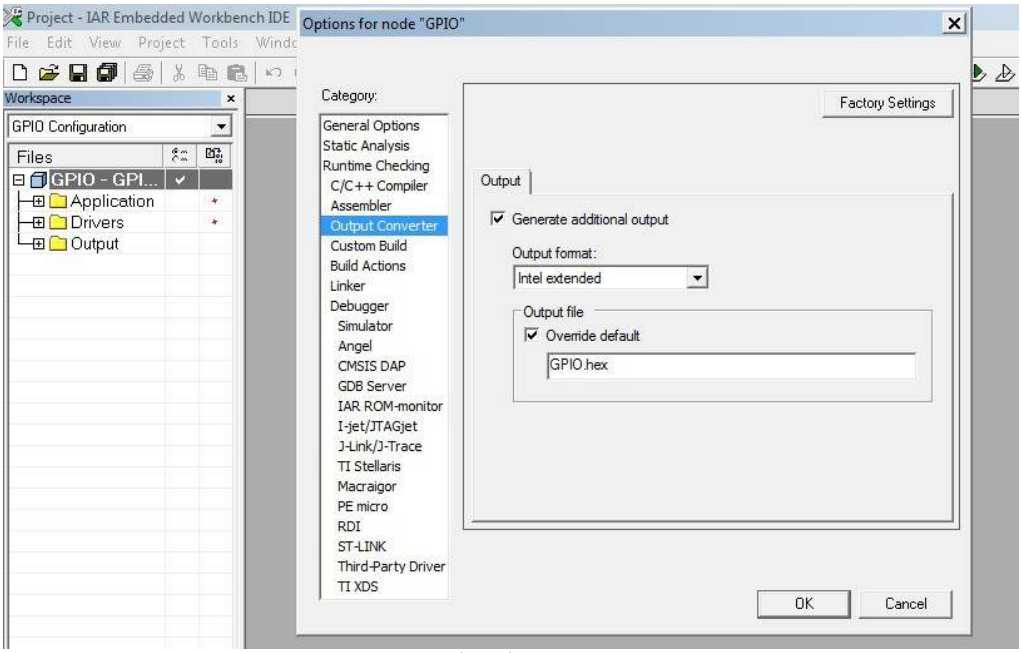

Figure 2.23: Output tab in the Output Converter section

In Figure 2.24, the tree structure of the project generated in the IAR software environment is shown. The STM32F1xx_HAL_Driver folder includes the main libraries of the hardware enabled on the microcontroller. The User folder includes the main.c file which is used for writing C code. If you want to add a new library to the project, use this folder.

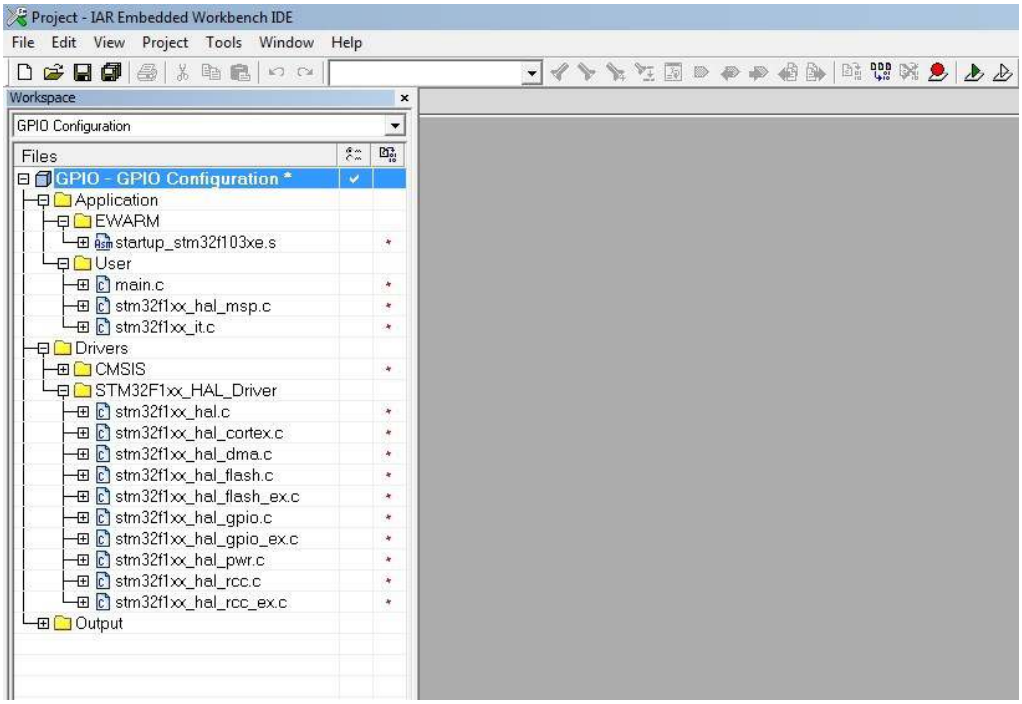

Figure 2.24: Project structure in IAR software

When clicking on the main.c file, the file contents are shown in the main software window as in Figure 2.25.

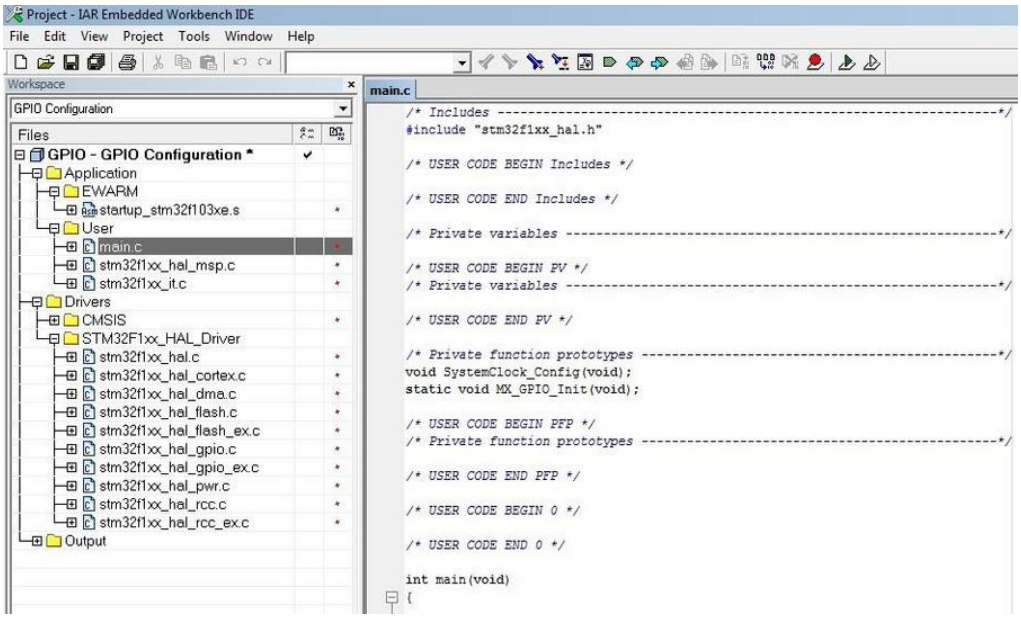

Figure 2.25: Code structure in the main.c file

On the top right-hand side of the window, as per Figure 2.26, there is a tool to show the functions available in the opened file. When selecting each function, the mouse cursor is transferred to the contents of that function.

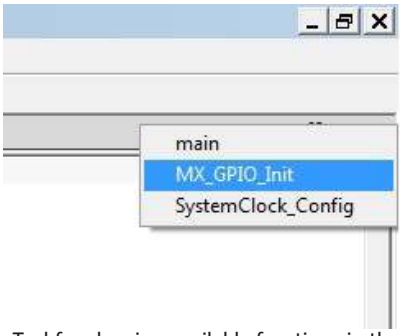

Figure 2.26: Tool for showing available functions in the current file

```
void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __GPIOA_CLK_ENABLE();

  /*Configure GPIO pin : PA2 */
  GPIO_InitStruct.Pin = GPIO_PIN_2;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}
```

Figure 2.27: Body of the MX_GPIO_Init() function

From the tree structure of the files in the STM32F1xx_HAL_Driver folder, double-click on the stm32f1xx_hal_gpio.c file to open the relative library in the main window. Use the tool on the top right of the window to show the functions used in the stm32f1xx_hal_gpio.c file as illustrated in Figure 2.28.

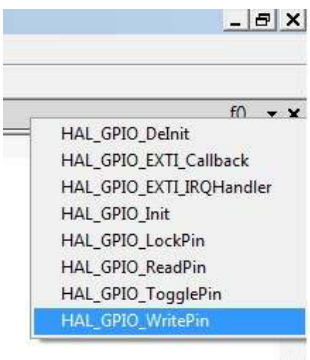

Figure 2.28: Functions available in the stm32f1xx_hal_gpio.c file

The HAL_GPIO_WritePin function is used for value allocation to the pins in output mode. When clicking this function, the mouse cursor is transferred to the location of the function as shown in Figure 2.29. In the HAL libraries on the top of each function, there are complete comments about the input and output of each function which helps the user in applying the function correctly.

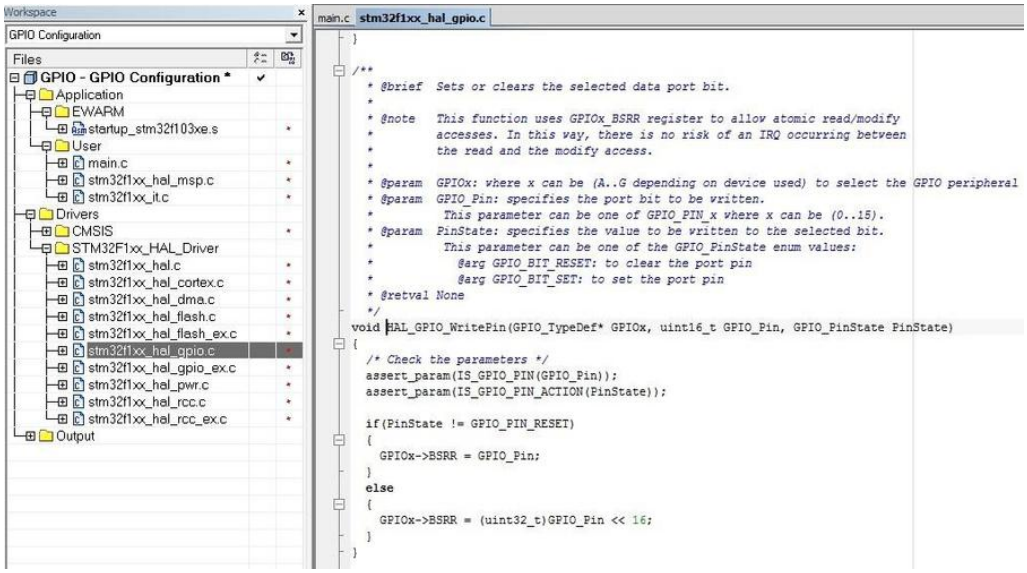

Figure 2.29: The body of the HAL_GPIO_WritePin function

In the pin setting section, the PA2 pin is set as output. We want to toggle an LED connected to the PA2 pin. To do this, use the following code in the main loop of the program, as shown in Figure 2.30.

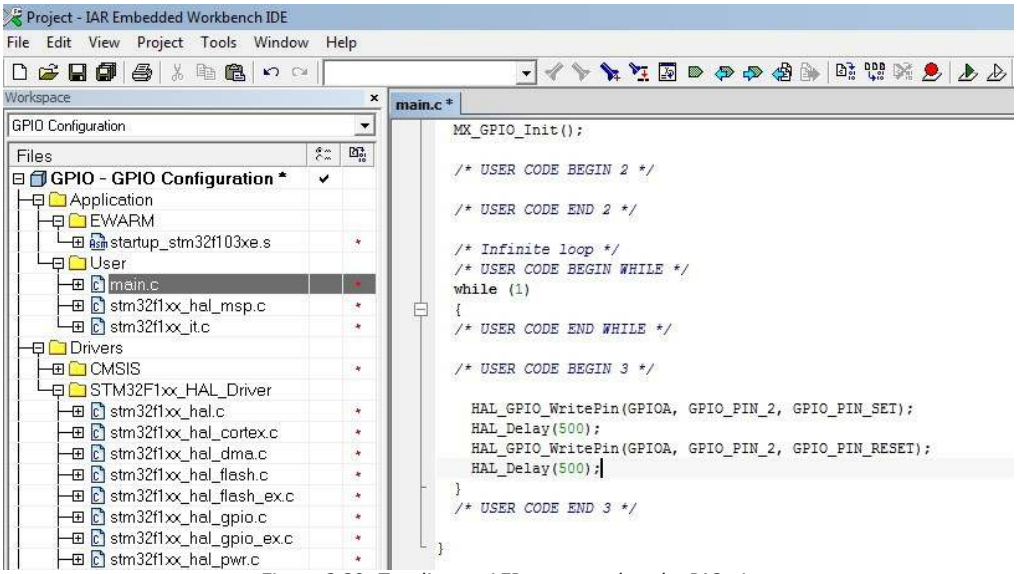

Figure 2.30: Toggling an LED connected to the PA2 pin

We can also do the same by using the HAL_GPIO_TogglePin function. It is necessary to use the following lines inside the while loop:

```
HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_2);
HAL_Delay(500);
```

After writing the code, the program is compiled and a hex file is generated. This file is transferred to the microcontroller by the programmer. For this purpose, as per Figure 2.31 from the Project menu, select the ‘Rebuild All’ icon. After the compiling procedure begins, the results are shown in the messages window, see Figure 2.32.

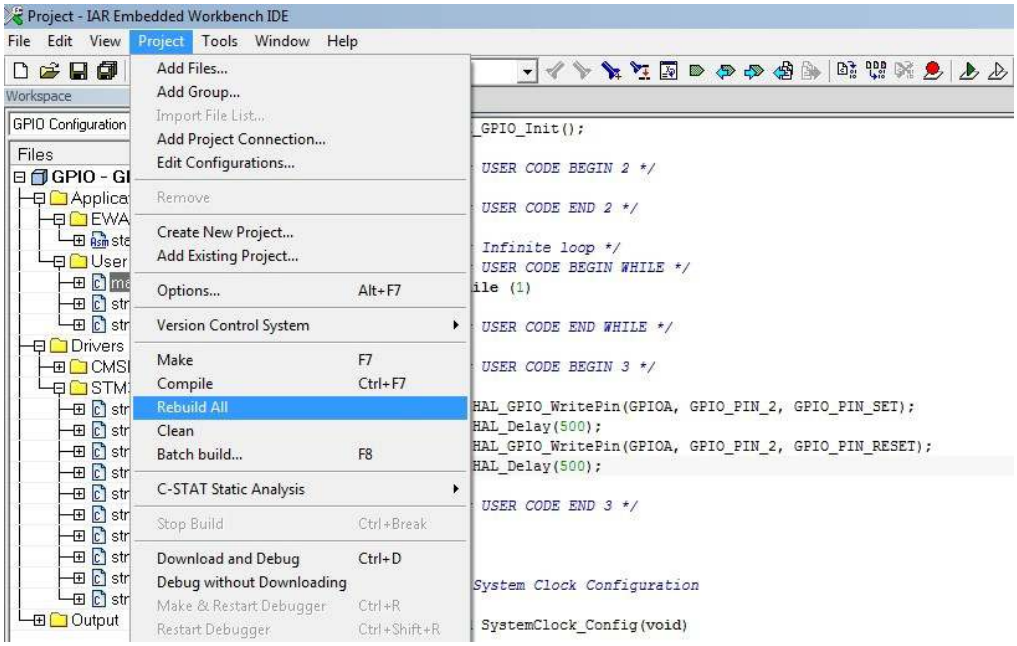

Figure 2.31: Code compiling and hex file generating

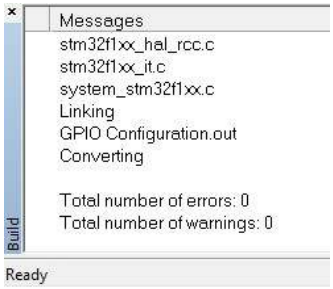

Figure 2.32: Messages window after compiling

The simulation using the generated hex file can be performed in Proteus as shown in Figure 2.33.

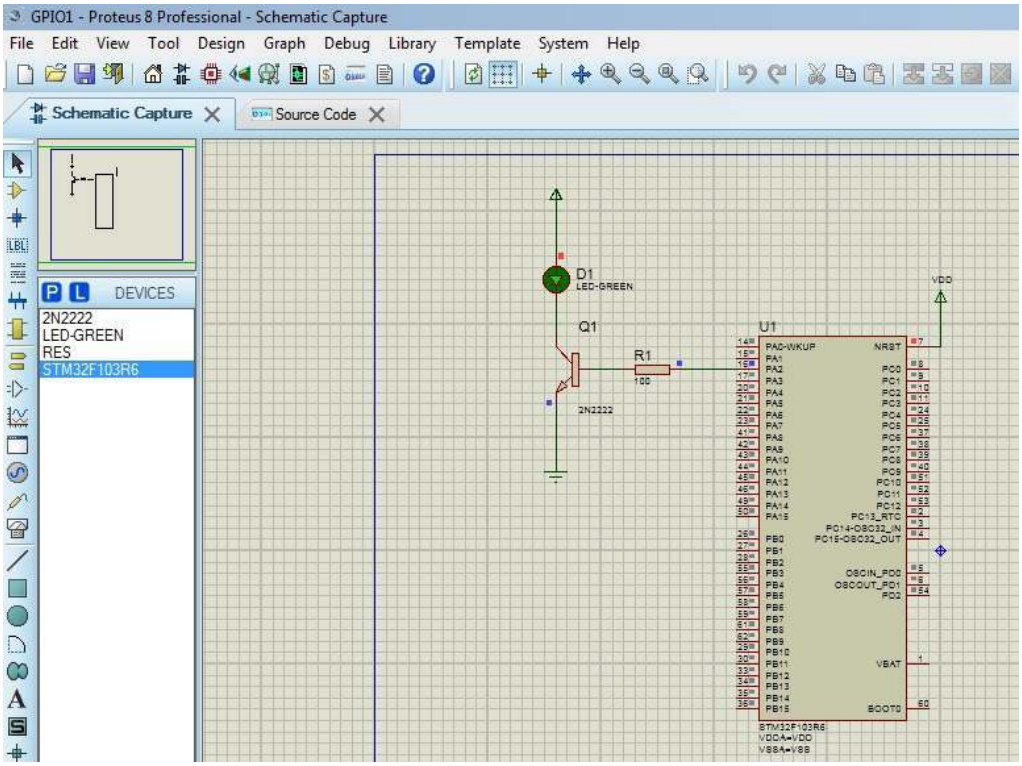

Figure 2.33: LED toggling simulation using Proteus

2.3 • The Second Project

In the next example, we want to give values to a group of output pins. Therefore, as per Figure 2.34, we set a group of port A pins as GPIO_Output and generate a new project.

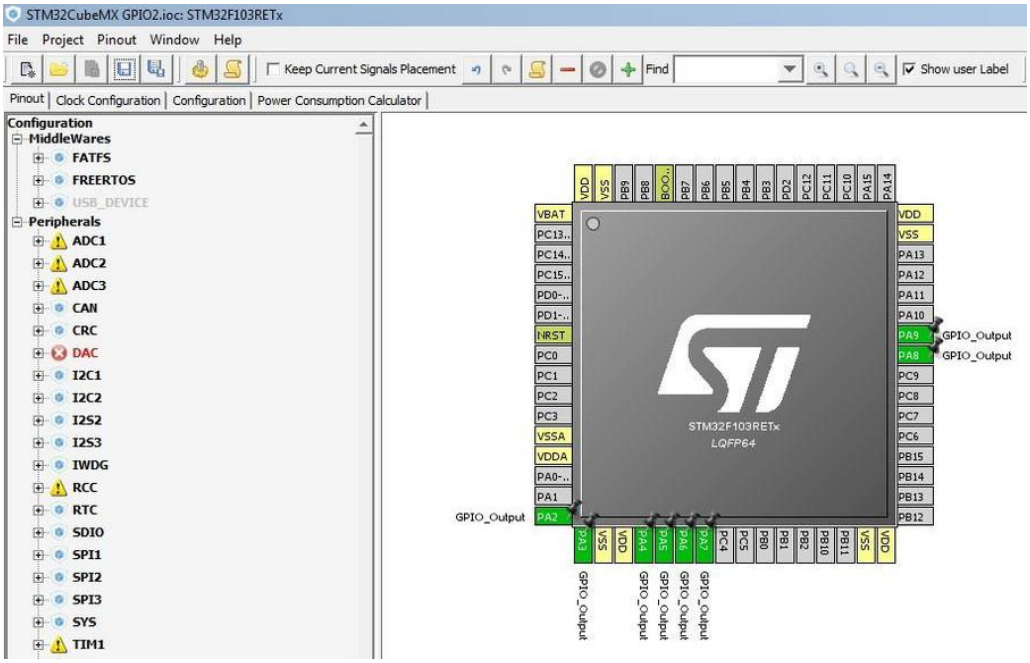

Figure 2.34: setting a group of port A pins as GPIO_Output

After generating the code, in main.c, the following code in while loop is added as shown in Figure 2.35.

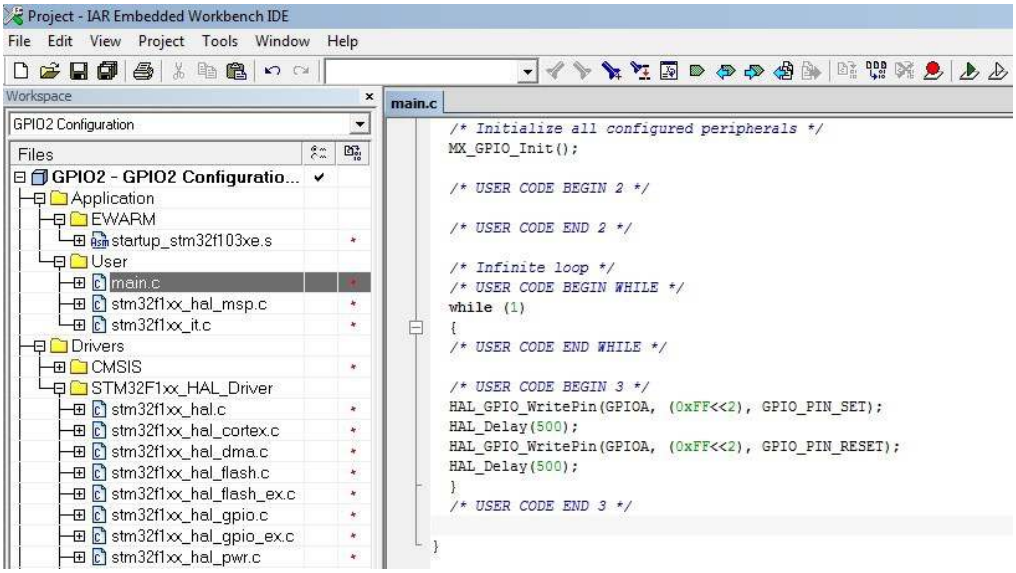

Figure 2.35: Giving values to a group of output pins

Remember, when you left shift a value x by y bites (x<<y), the leftmost y bits in x are lost. In Figure 2.35 we have (0xFF<<2) which means the hex value 0xFF is left-shifted by 2 hex values (hex values 0x in hex value 0xFF are lost and the result is hex value FF). We can then compile the project generate the hex file and simulate it using Proteus as shown in Figure 2.36.

2.4 • The Third Project

As another example, we want to drive a seven-segment. Therefore, seven pins of port A are set as GPIO_Output. We want to show counting from zero to nine on a seven-segment with a one second time interval. So, as per Figures 2.37 and 2.38, we can produce a constant array of values that will generate numbers from zero to nine. This constant array is added in the main.c file as shown in Figure 2.39.

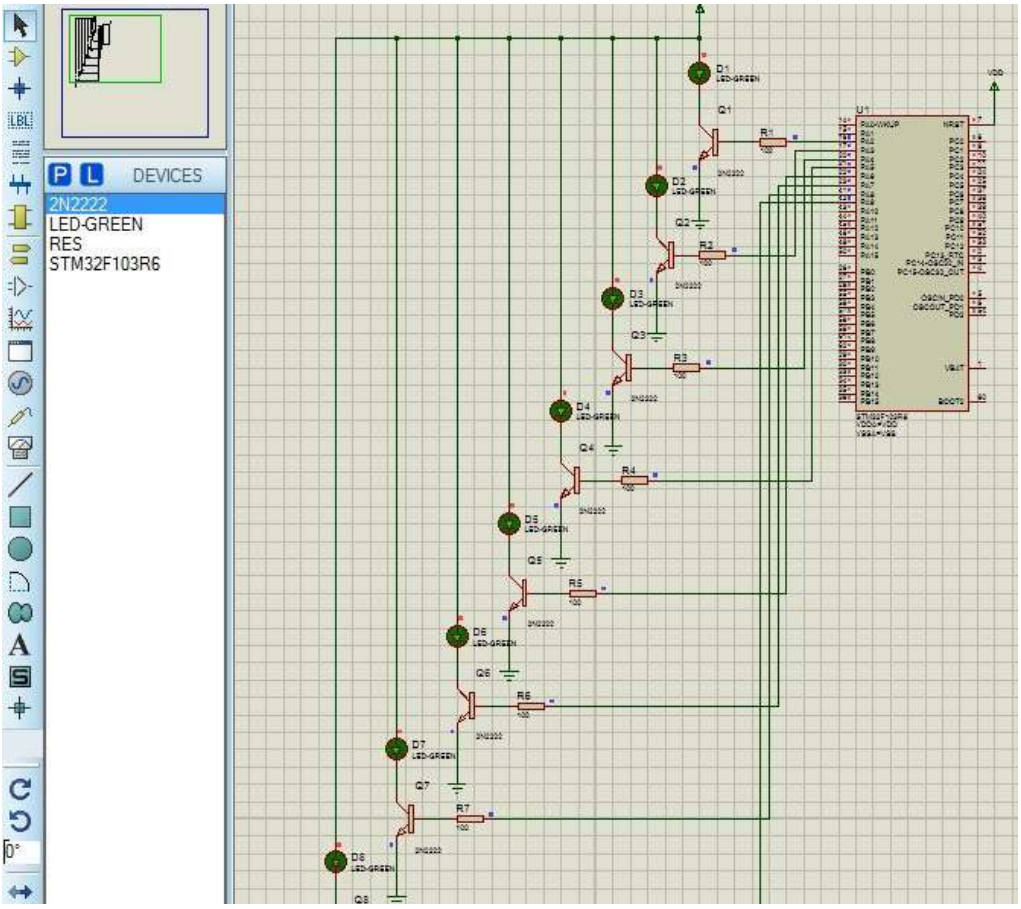

Figure 2.36: Turning on/off a group of output pins together

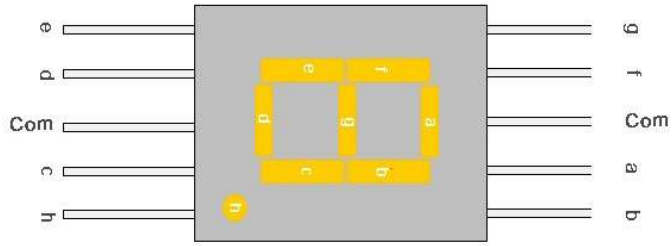

Figure 2.37: Seven-segment pins

Number	g f e d c b a	Hexadecimal
0	0 1 1 1 1 1 1	3F
1	0 0 0 0 1 1 0	06
2	1 0 1 1 0 1 1	5B
3	1 0 0 1 1 1 1	4F
4	1 1 0 0 1 1 0	66
5	1 1 0 1 1 0 1	6D
6	1 1 1 1 1 0 1	7D
7	0 0 0 0 1 1 1	07
8	1 1 1 1 1 1 1	7F
9	1 1 0 1 1 1 1	6F

Figure 2.38: Hex codes for displaying numbers on seven-segment

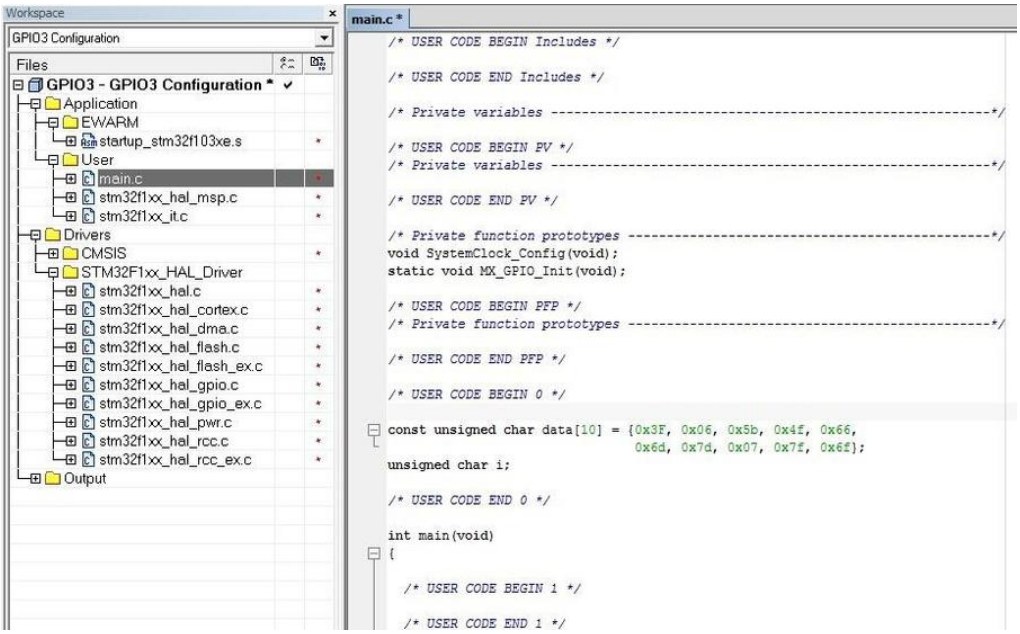

Figure 2.39: Constant array of numbers from 0 to 9 on 7-segment

The following code as illustrated in Figure 2.40 should be added in the while loop of the main.c file to show the counting values from zero to nine with a one-second time interval. We can then can compile the project, generate the hex file, and simulate it using Proteus as shown in Figure 2.41.

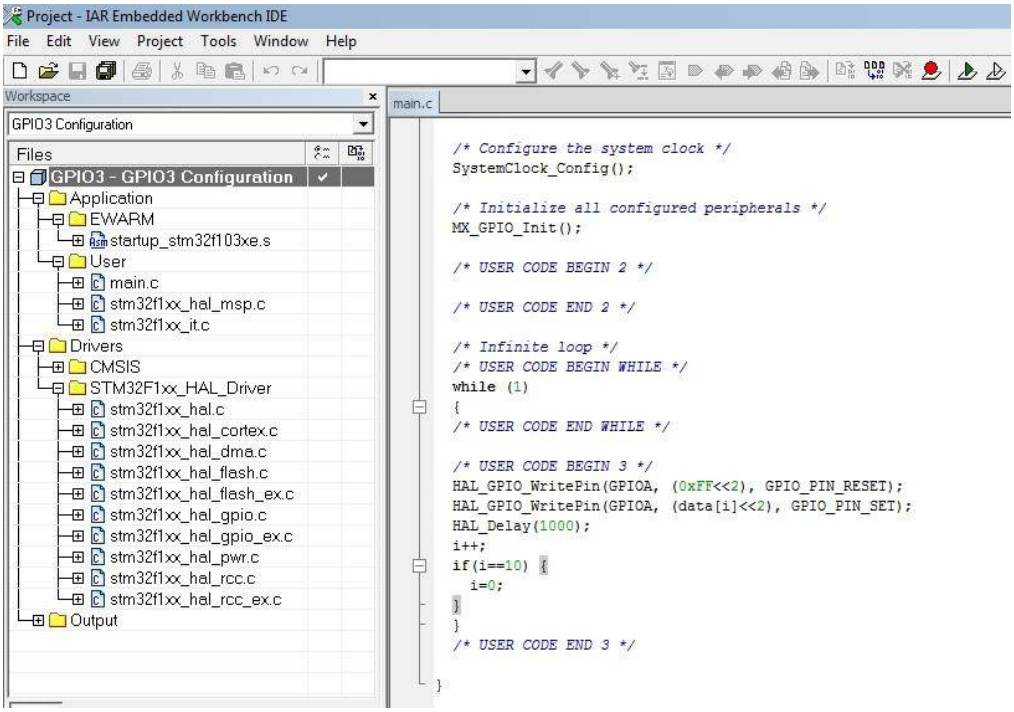

Figure 2.40: Implementation of counting values from 0 to 9 with 1s time interval

The 74LS541 is a buffer used for driving the seven-segment. The circuit uses a 74LS541 octal TTL integrated circuit that has eight high-speed buffers and drivers with a Schmitt trigger on the inputs. Similar CMOS integrated circuits like 74HCT541, 74HC541, and TTL integrated circuits like 74S541 or 74F541 can also be used. The 74LS541 buffer converts the input signal with a low slew rate into a CMOS or TTL compatible signal with a high slew rate.

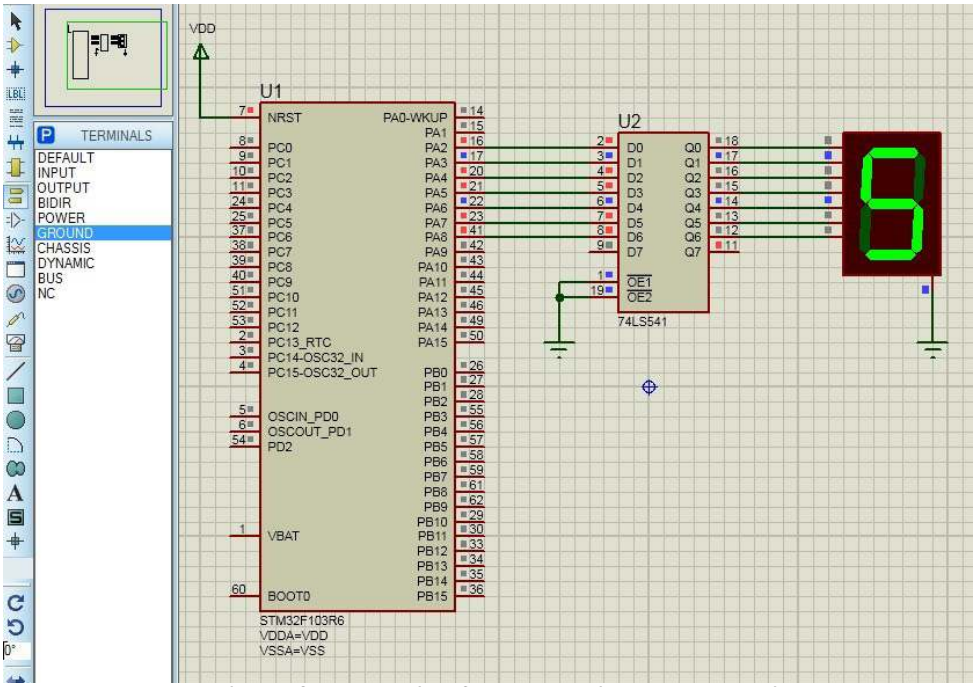

Figure 2.41: Simulation of counting values from 0 to 9 with a 1s time interval using Proteus

2.5 ● The Fourth Project

In previous examples, we considered pins as outputs. In the following example, we want to evaluate the operation of the pins in input mode. Therefore, we are going to increase the number value shown on seven-segment by a pushbutton connected to the PB10 pin. Thus, we set the PB10 pin as GPIO_Input as shown in Figure 2.42. Also, by clicking on the GPIO button in the Configuration tab, we select the PB10 pin and change its mode to Pull_up as shown in Figure 2.43.

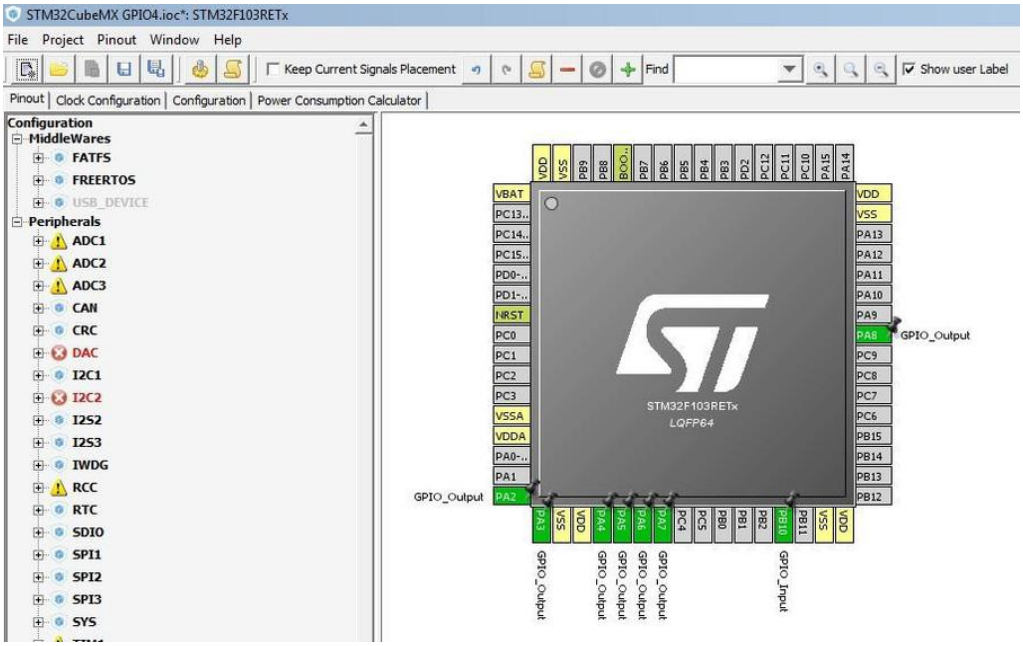

Figure 2.42: Defining the PB10 pin as GPIO_Input

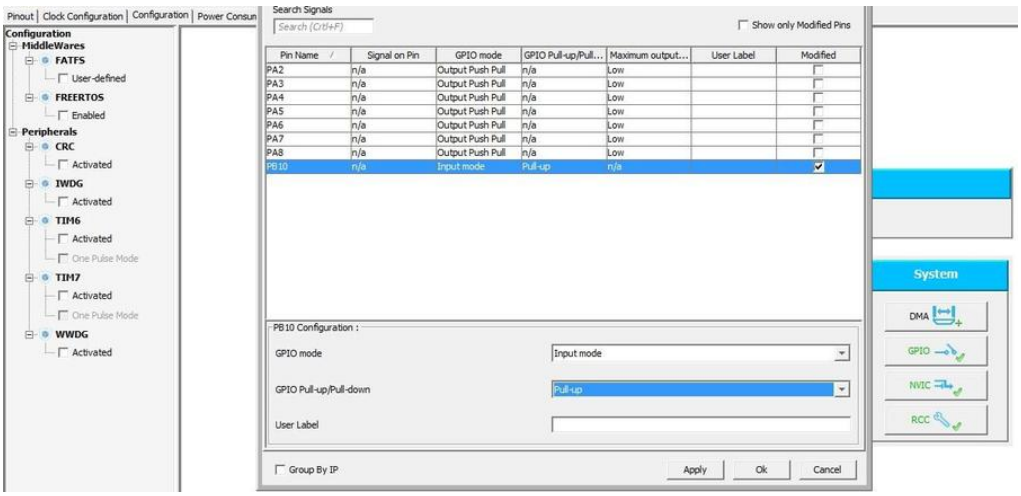

Figure 2.43: Selecting Pull_up for PB10 pin

We then generate the project. The functions for GPIO are in stm32f1xx_hal_gpio.c file as shown in Figure 2.44.

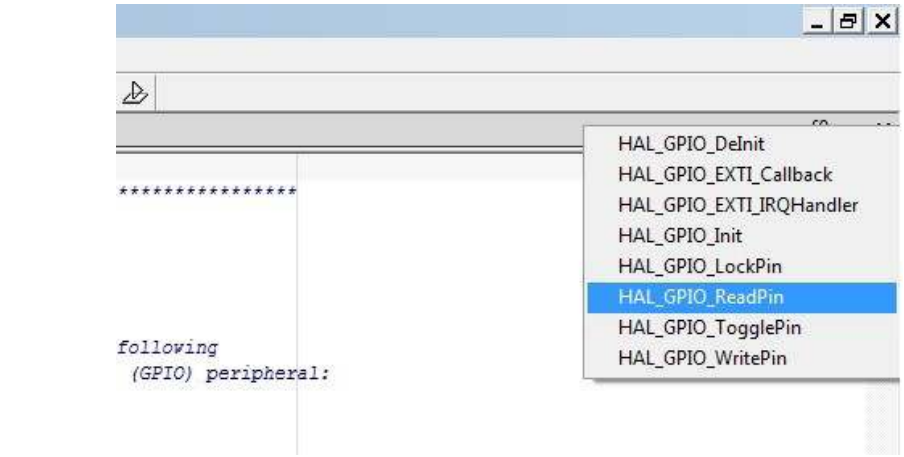

Figure 2.44: Functions available in stm32f1xx_hal_gpio.c file

The HAL_GPIO_ReadPin is used for reading the value of the input pin. The body of the HAL_GPIO_ReadPin function is illustrated in Figure 2.45.

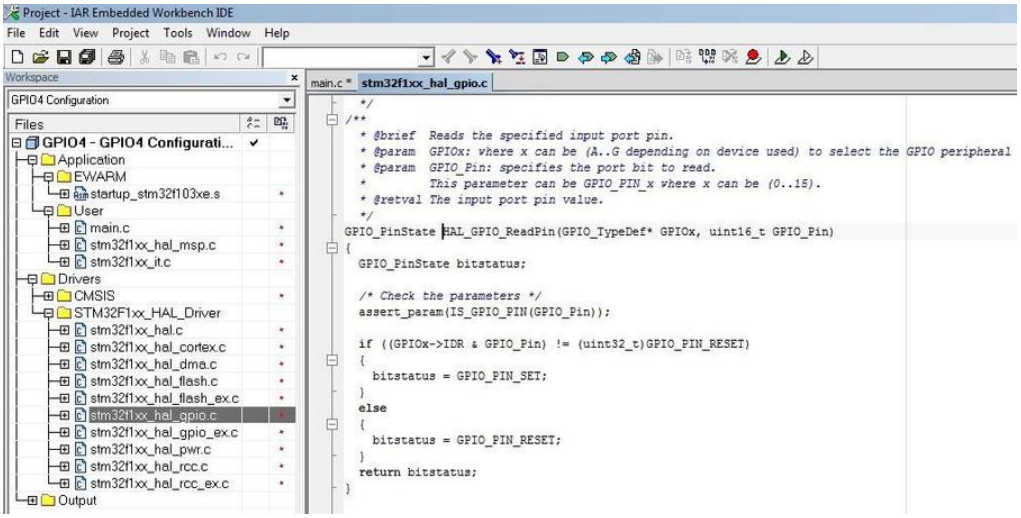

Figure 2.45: The body of the HAL_GPIO_ReadPin function

Since the PB10 pin is defined as Pull_up, its value is 1 without excitation. So, for excitation of PB10 pin, it should be connected to 0 or GND. The code shown in Figure 2.46 and 2.47 should be added in the while loop. There is a 200 ms delay to compensate for pushbutton contact vibrations and de-bouncing.

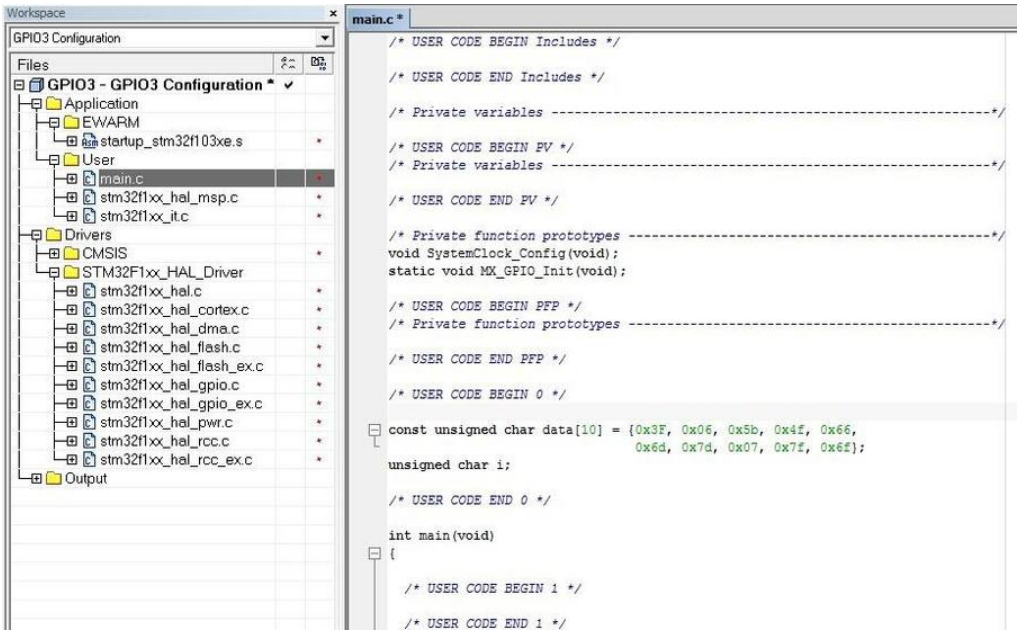

Figure 2.46: Constant array of numbers from 0 to 9 on 7-segment

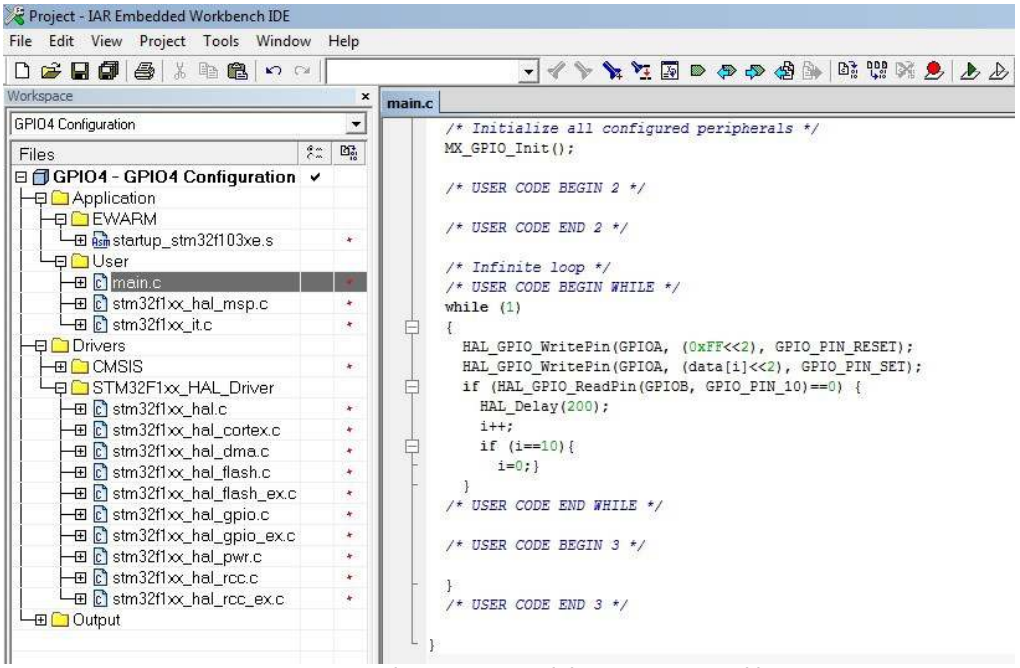

Figure 2.47: Reading input pin and the increasing variable i

We can then compile the project, generate the hex file, and simulate it using Proteus as shown in Figure 2.48.

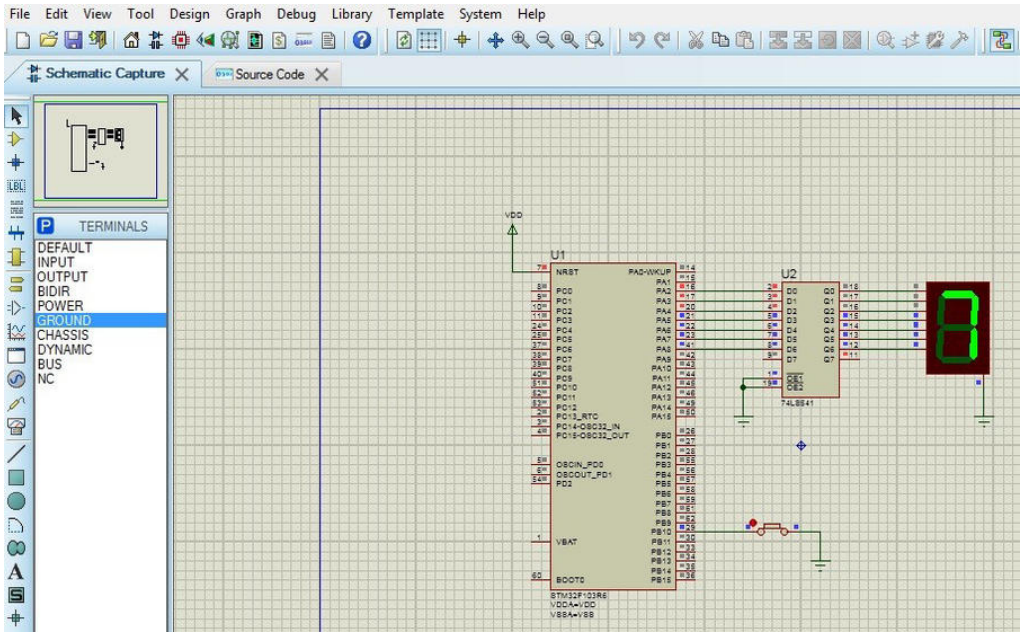

Figure 2.48: Simulation of counting values from 0 to 9 with PB10 input pushbutton using Proteus

2.6 ● Summary

In this chapter, we used STM32CubeMX and IAR Embedded Workbench. We also used Proteus for simulating hex files generated from compiling projects in the IAR Embedded Workbench for the ARM software environment.

Chapter 3 • Adding LCD and Delay Libraries to a Project

3.1 • Introduction

Libraries in the C programming language include various functions and tools. In the last chapter, we got acquainted with the HAL libraries and functions generated by STM32CubeMX. In this chapter, we will learn how to add a library including the start-up functions of different external hardware and various functions to a project where there are no HAL libraries. A standard library in C includes two files with .h and .c suffixes, known as header and source files respectively. In the header file, there are macros, function calls, and libraries used. In the source file, there is a body of functions. For using the text LCD, we should add its library to the project. A schematic of the 16×2 LCD pins is shown in Figure 3.1.

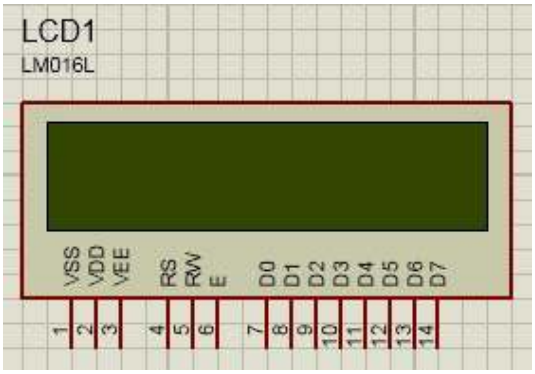

Figure 3.1: Text LCD pins

3.2 • STM32CubeMX Settings and Adding LCD Library to the Project

According to Figure 3.2, we set the appropriate pins as GPIO_Output and in push-pull mode and then generate the project.

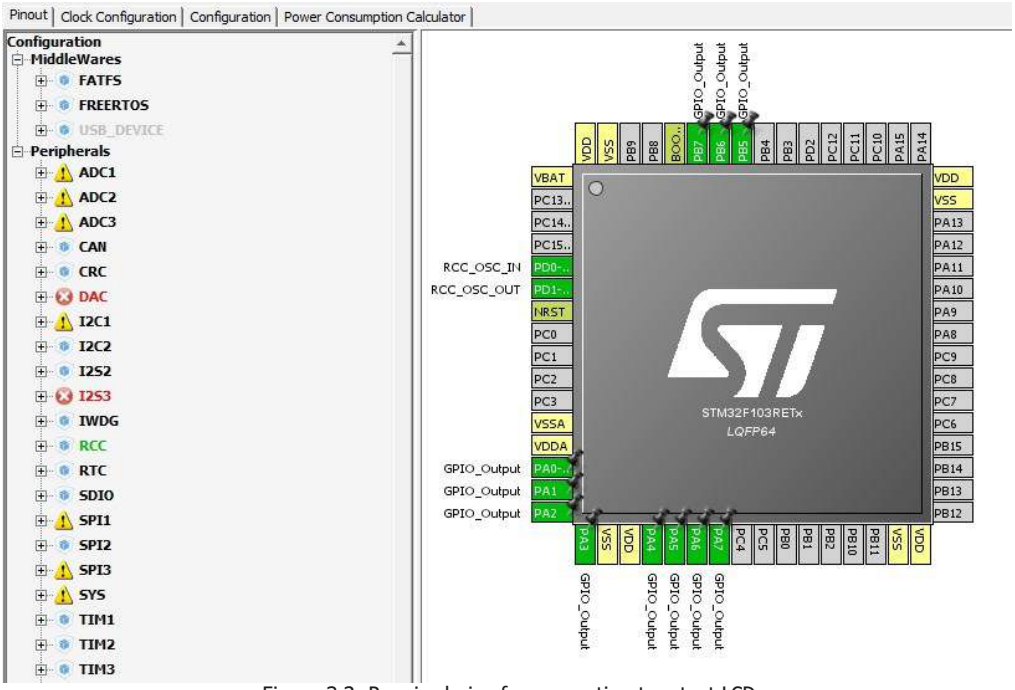

Figure 3.2: Required pins for connecting to a text LCD

After generating the project using STM32CubeMX, add an LCD library to the project. For this purpose, in the Src of the project folder, there is the main.c file. Inside this, make a folder called LCD and set the .h and .c library files as shown in Figure 3.3.

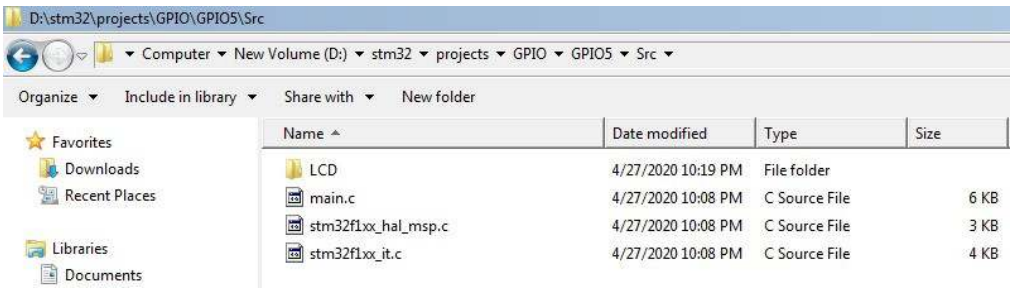

Figure 3.3: Making a folder named LCD in the Src folder

After inserting the library files into the Src folder, in accordance with Figure 3.4 in the IAR software environment and project tree structure, right-click on the Driver folder and Add -> Add Group icon to make a new group named LCD.

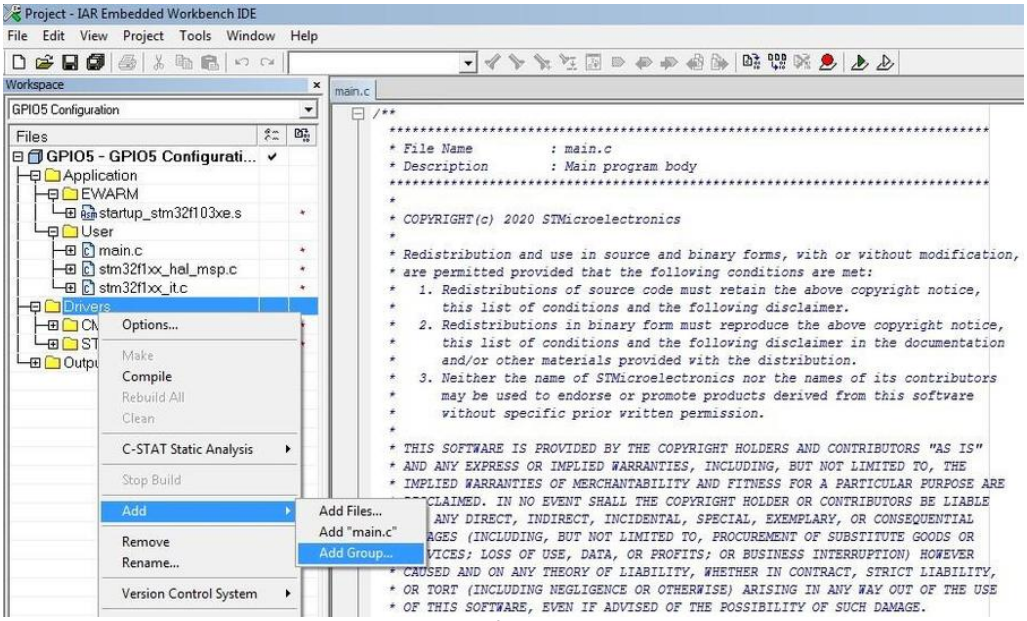

Figure 3.4: Making a new group

After making a new group, right-click it, and from Add -> Add Files add two files (.h and .c) to the project as shown in Figure 3.5. The added files are shown in Figure3.6.

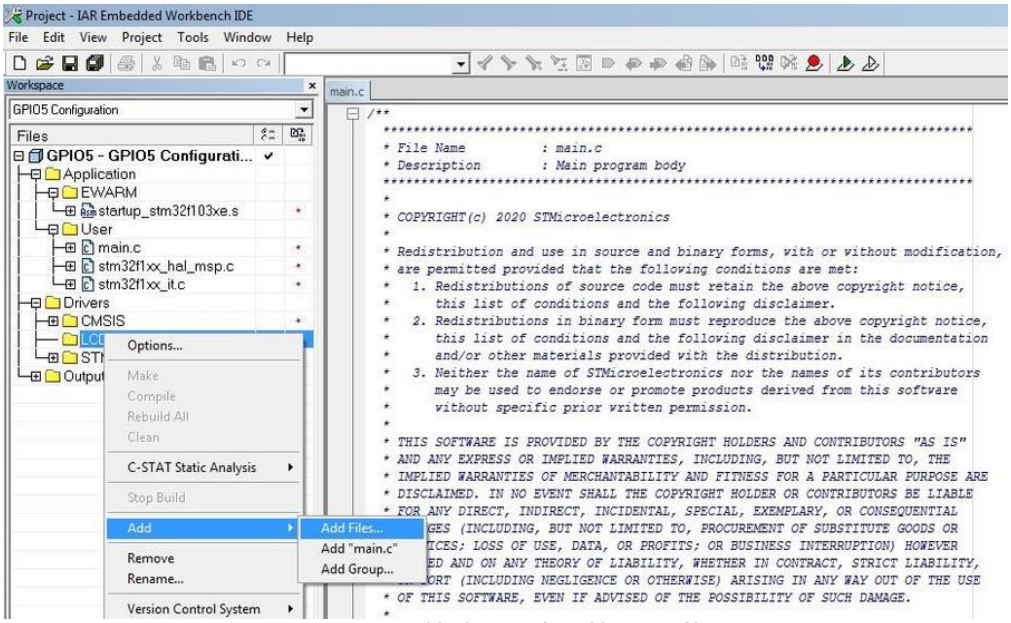

Figure 3.5: Add Files icon for adding LCD files

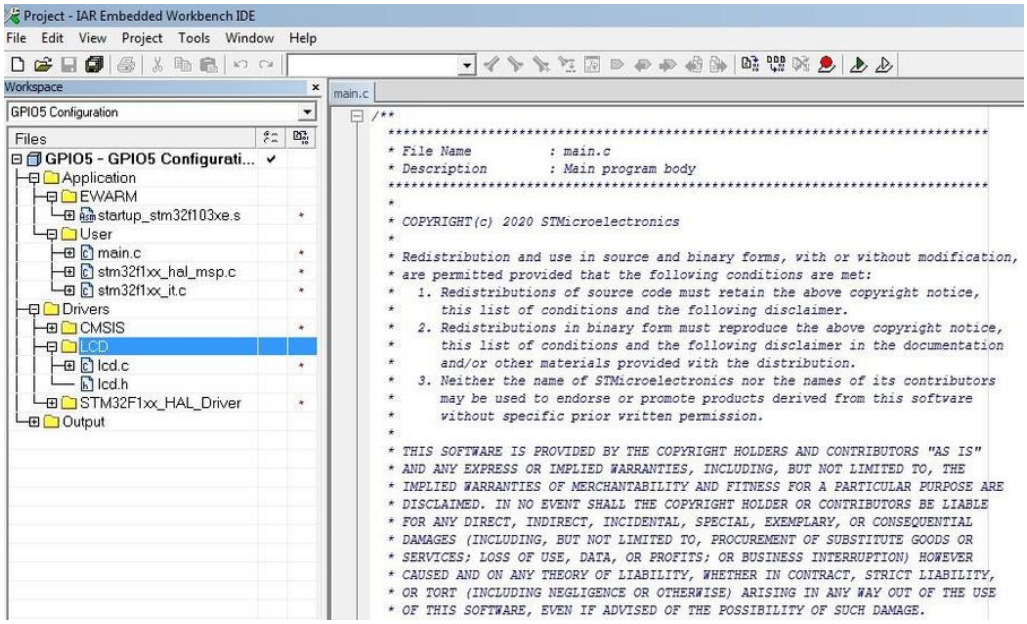

Figure 3.6: Added .h and .c files to the project structure

After adding the library files to the project structure, the library .h file should be added to the main.c file to be able to use the available functions as illustrated in Figure 3.7.

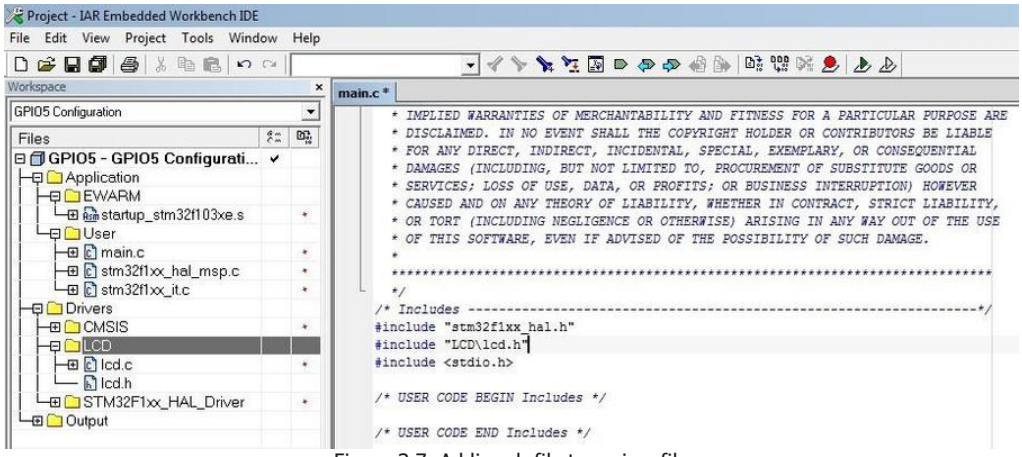

Figure 3.7: Adding .h file to main.c file

After adding the LCD library, we should adapt the library condition with the available hardware. Usually, .h files in a library are related to the settings of that library. The contents of the LCD library .h file is shown in Figure 3.8.

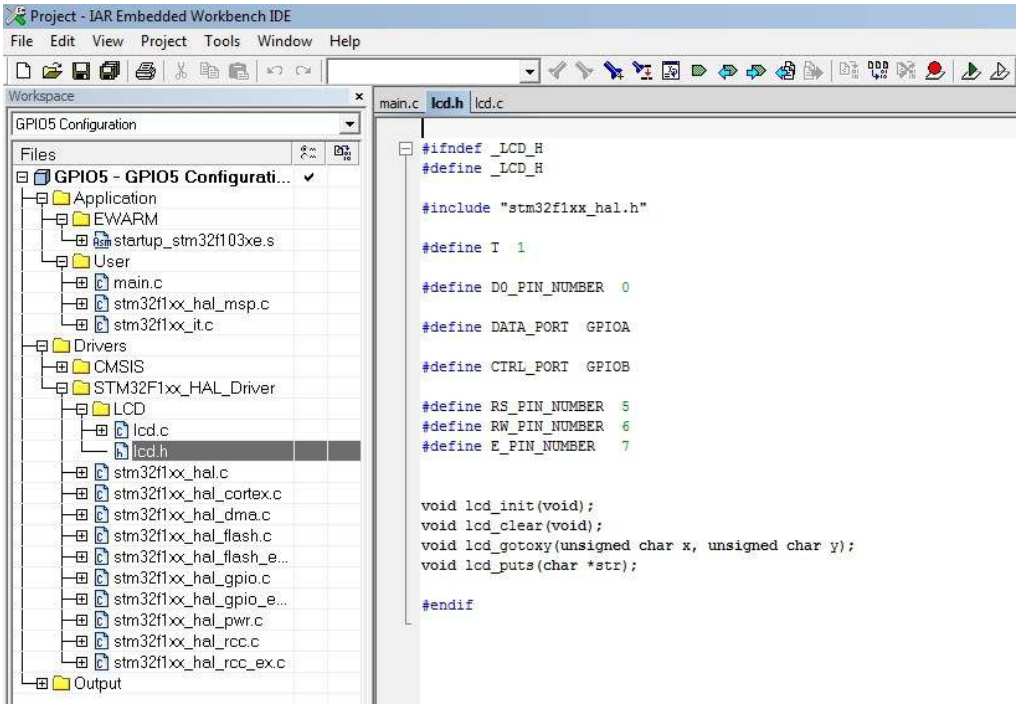

Figure 3.8: The lcd.h file contents

Let's consider an example: firstly a constant string is printed on a display then after 3 seconds, a counter begins counting. The programming code is shown in Figures 3.9 and 3.10.

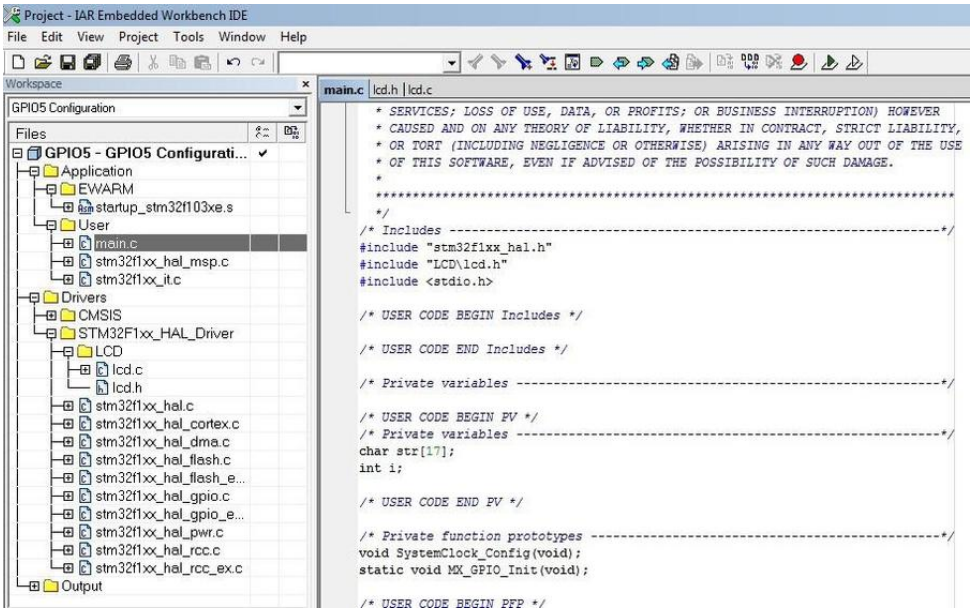

Figure 3.9: Code above the main function

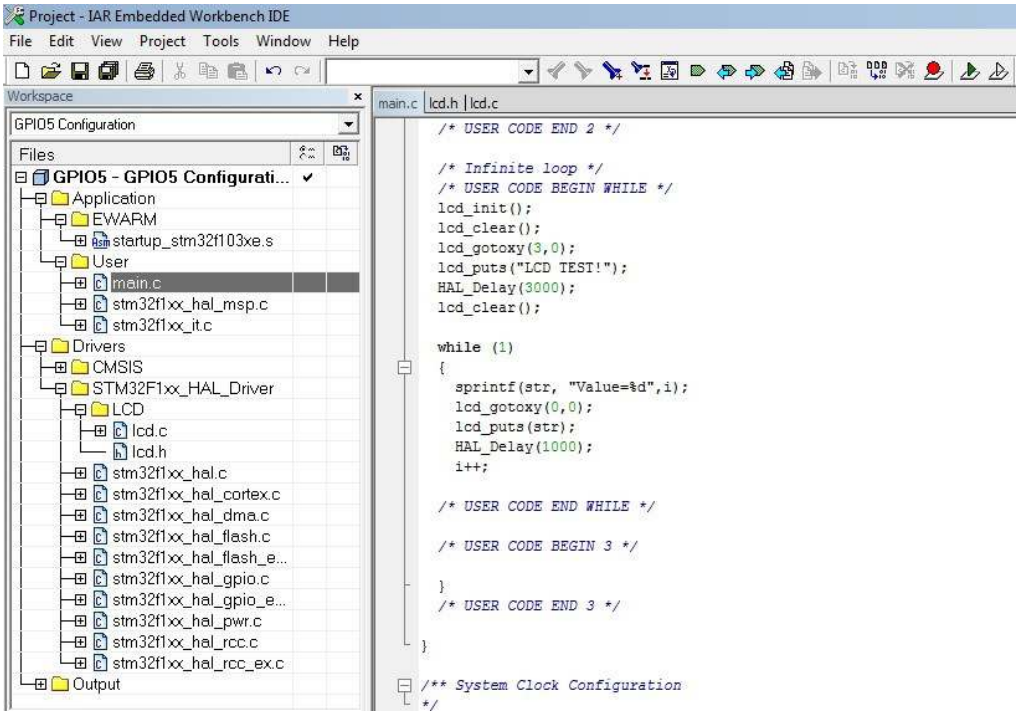

Figure 3.10: The programming code

3.3 ● Adding LCD and Delay Libraries from Book files Download Section

The lcd.c and lcd.h files for an 8-bit LCD are provided in the book's programming code files download section. For a 4-bit LCD16×2, lcd16x2.h and lcd16x2.c are also provided. An example of the main.c file using Keil uVision, including use of the delay.h and delay.c is also given.

3.4 ● Summary

In this chapter, we learned how to add LCD and Delay libraries to a project where they are not in HAL libraries. A standard library in C programming language includes two files with .h and .c suffixes called header and source files respectively. In the header file, there are macros, function calls, and libraries used. In the source file, there is a body of functions. The LCD and Delay libraries used have been included in the book programming code files download section.

Chapter 4 • External Interrupts in STM32 Microcontrollers

4.1 • Introduction

Interrupts are particularly important to microcontrollers. With interrupt occurrence in the microcontroller operation cycle, the interrupt service function is called and the code inside in it is executed. An interrupt can have different factors such as external excitation, timer interrupt, analogue to digital converter interrupt, serial communications interrupt, etc. By using an interrupt, microcontroller capability for the evaluation of available and external hardware connected to it increases. Interrupts have execution priority and always the upper priority has the execution priority compared with the others. In this chapter, we will consider the external excited interrupt. In Cortex-M microcontrollers, there is a nested vectored interrupt controller (NVIC) unit for handling interrupts. A nested interrupt means if an interrupt service function is executed and upper priority interrupt is activated, the executing interrupt service function is halted at that point and the interrupt service function with upper priority will be executed. The NVIC unit in the STM32F103RET6 microcontroller supports 16 interrupt priorities from GPIO_EXTI0 to GPIO_EXTI15. The user can select the interrupt type and priority.

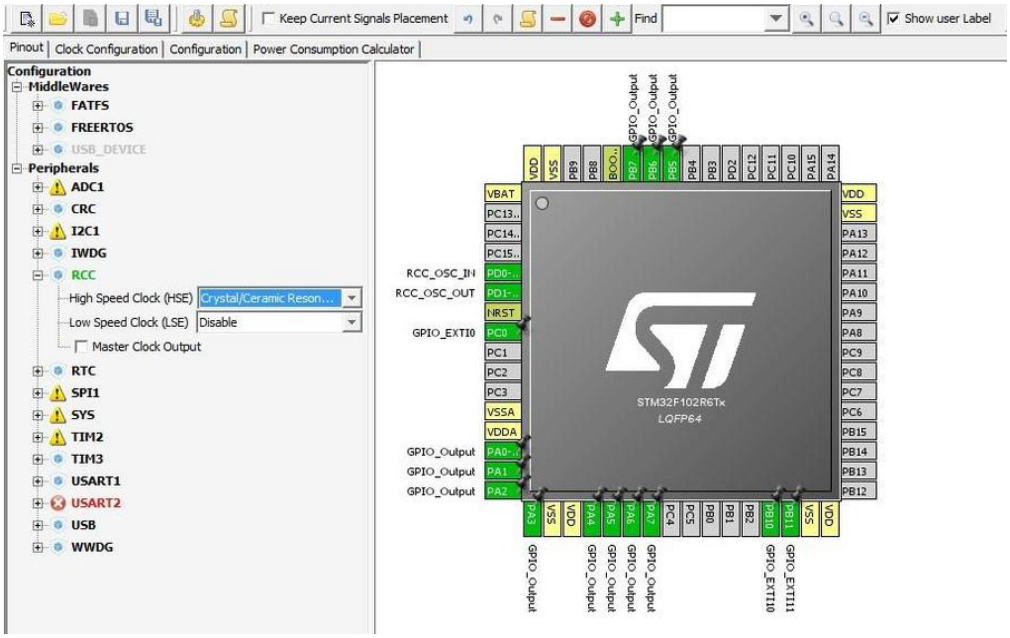

Figure 4.1: External interrupt pins selection

4.2 • External Interrupt Project Settings

In the following example, we are going to select PC0, PB10, and PB11 as external interrupt pins and by exciting each pin, add the variables related to each external interrupt and display it on an LCD. In the STM32CubeMX software and Pinout section, by clicking on

the specified pins, we activate them for external interrupts. We also determine the output pins for the LCD connection. At the end, we activate the external crystal resonator pins as shown in Figure 4.1. After setting the pins for external interrupt, we should set the NVIC unit for related interrupt. For this purpose, as shown in Figure 4.2, from the Configuration section, we select the NVIC button. The NVIC settings section is illustrated in Figure 4.3. Each factor that can produce an external interrupt is included in the list. As shown, the EXTI line0 interrupt and EXTI line [15:0] interrupts are in the list and should be activated with tick symbols in the associated squares. From Preemption priority, priority should be chosen from 0 to 15.

Figure 4.2: Configuration section and NVIC unit in System subsection

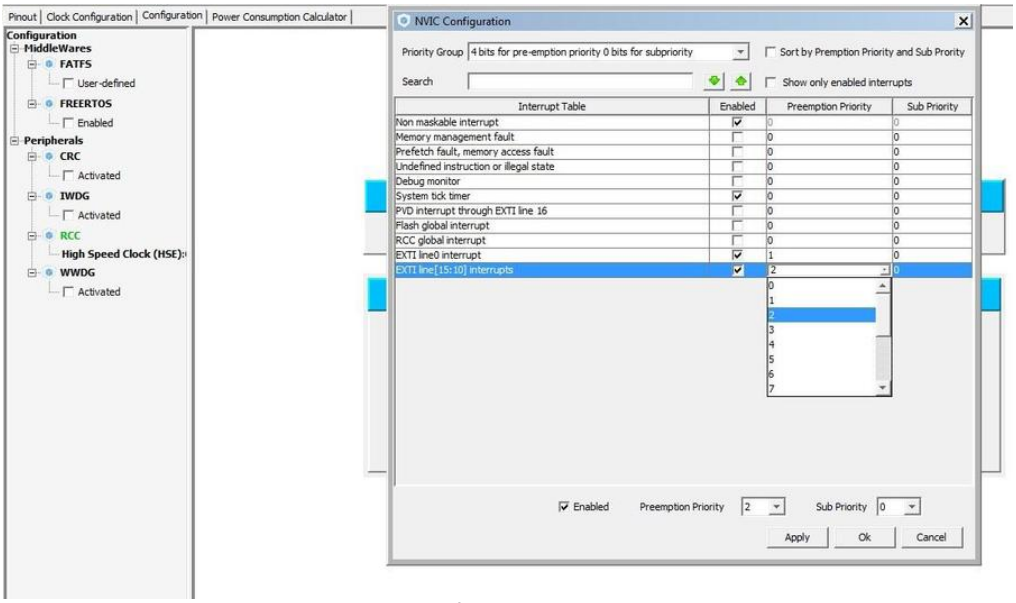

Figure 4.3: The NVIC settings section

After setting the NVIC unit, the pin mode according to excitation type and its pull up or pull down condition should be determined. For this purpose, as shown in Figure 4.4, from the configuration section, we select GPIO from the system subsection. As shown in Figure 4.5, we can set the relative pin conditions. The PC0 pin is determined by exciting with an up-going pulse and pull-down mode. The PB10 and PB11 pins are determined as exciting with a down-going pulse and pull-up mode. The project is then generated as shown in Figure 4.6.

Figure 4.4: Configuration section and GPIO in System subsection

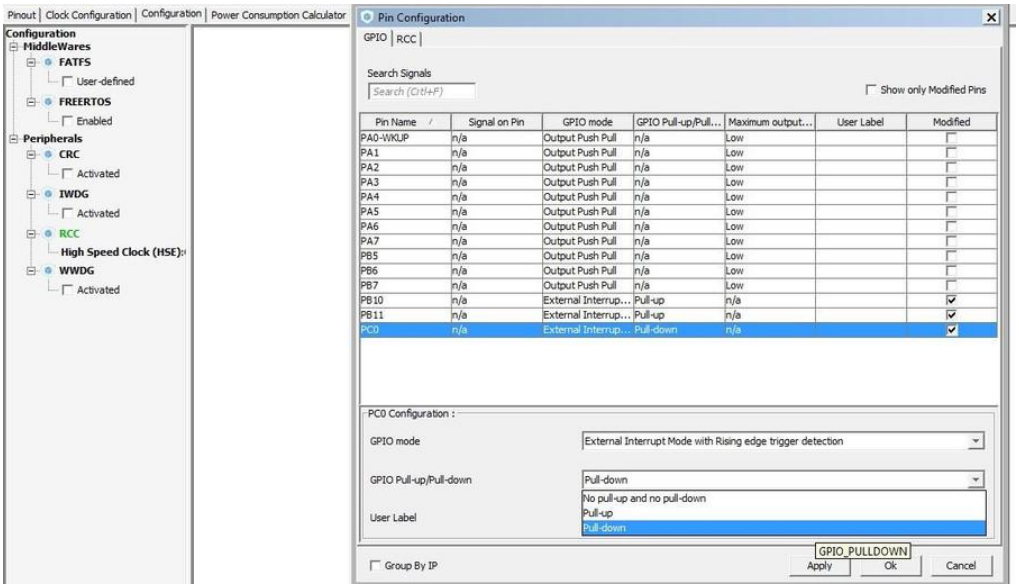

Figure 4.5: The GPIO settings section

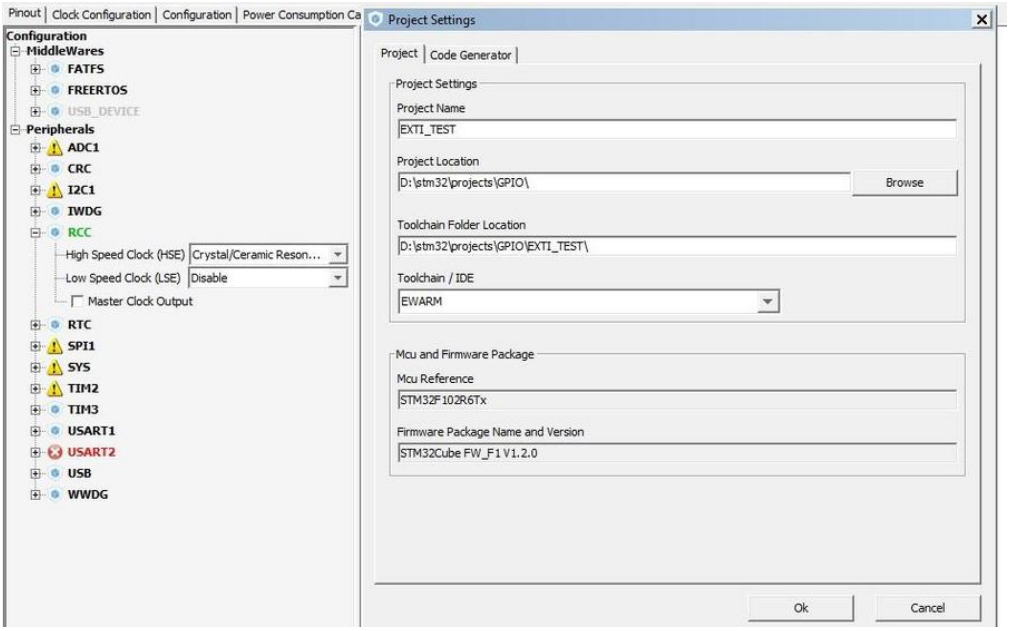

Figure 4.6: Code generation setup

The MX_GPIO_Init function body is shown in Figure 4.7. Service functions to interrupts are inside the stm32f1xx_it.c file. The functions inside this file are depicted in Figure 4.8.

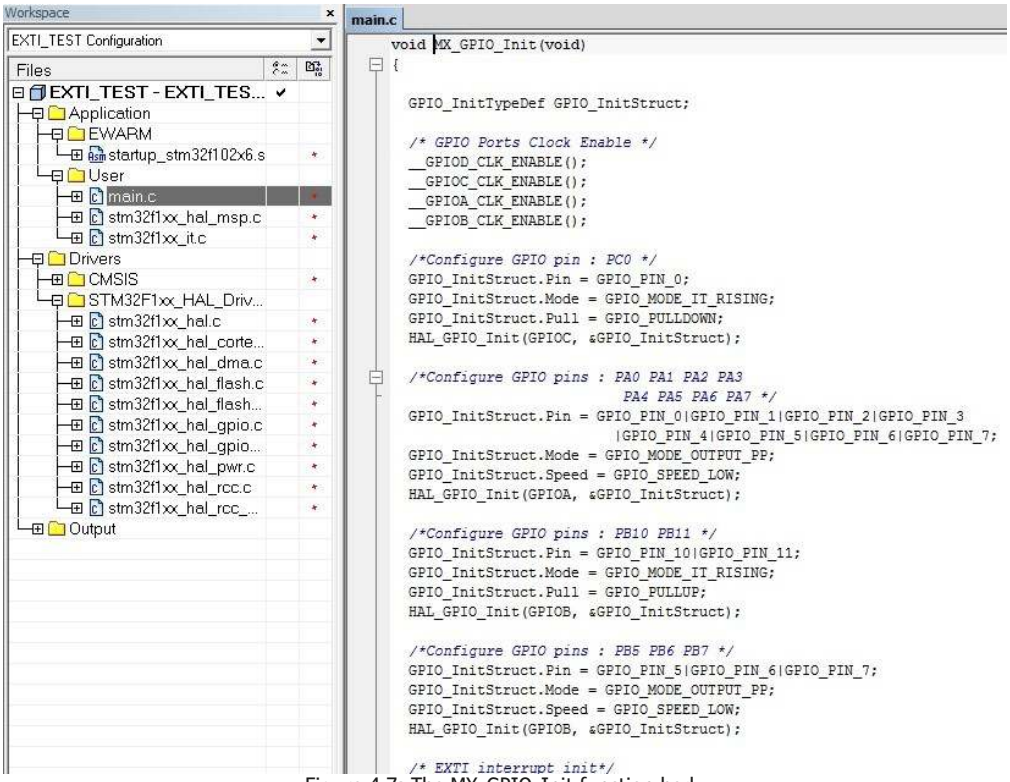

Figure 4.7: The MX_GPIO_Init function body

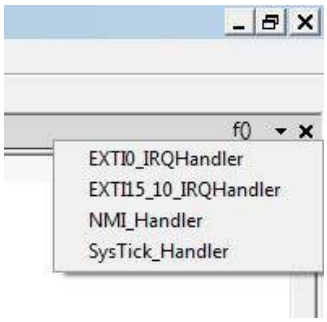

Figure 4.8: The functions inside stm32f1xx_it.c file

By clicking the EXTI0_IRQHandler and EXTI5_10_IRQHandler functions, the body of them is opened as shown in Figure 4.9.

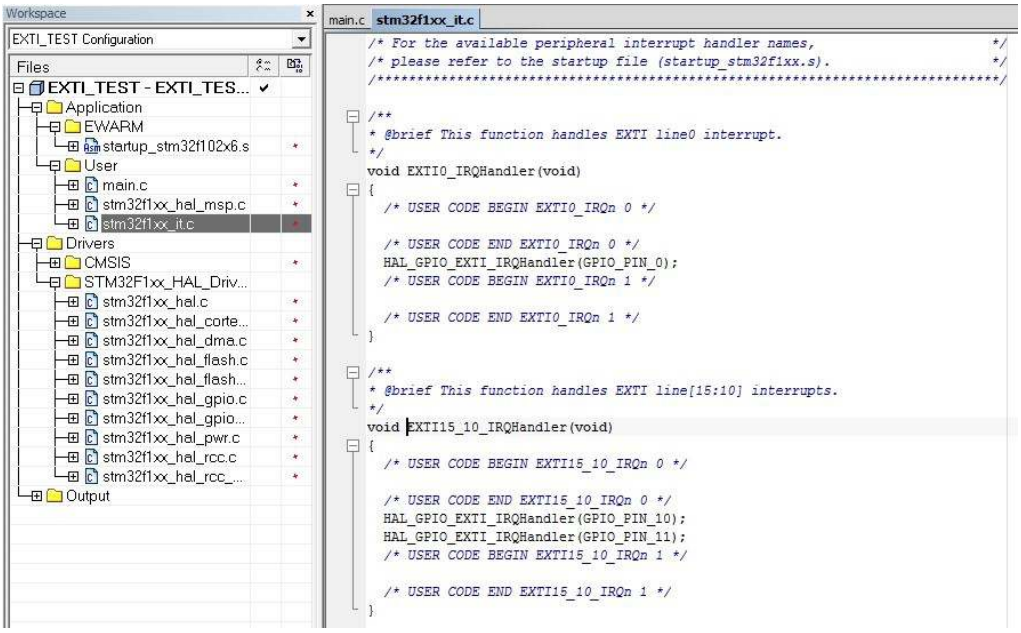

Figure 4.9: The body of the EXTI0_IRQHandler and EXTI15_10_IRQHandler functions in stm32f1xx_it.c

By right-clicking on the HAL_GPIO_EXTI_IRQHandler function and selecting Go to Definition of … we will be directed to the body of the HAL_GPIO_EXTI_IRQHandler function as shown in Figures 4.10 and 4.11. The HAL_GPIO_EXTI_Callback function inside the HAL_GPIO_EXTI_IRQHandler function is used for user code execution in external interrupt time. The body of the HAL_GPIO_EXTI_Callback function inside the stm32f1xx_hal_gpio.c file is also illustrated in Figure 4.11. The user should copy from the body of the HAL_GPIO_EXTI_Callback function and take it inside the stm32f1xx_it.c file as described in the following example:

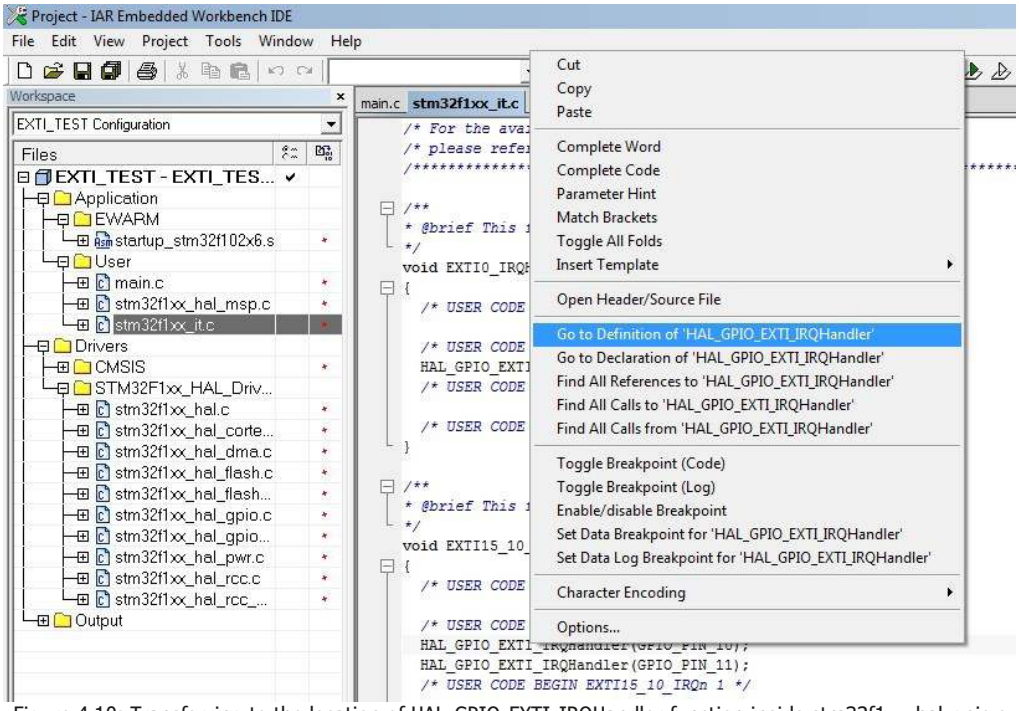

Figure 4.10: Transferring to the location of HAL_GPIO_EXTI_IRQHandler function inside stm32f1xx_hal_gpio.c

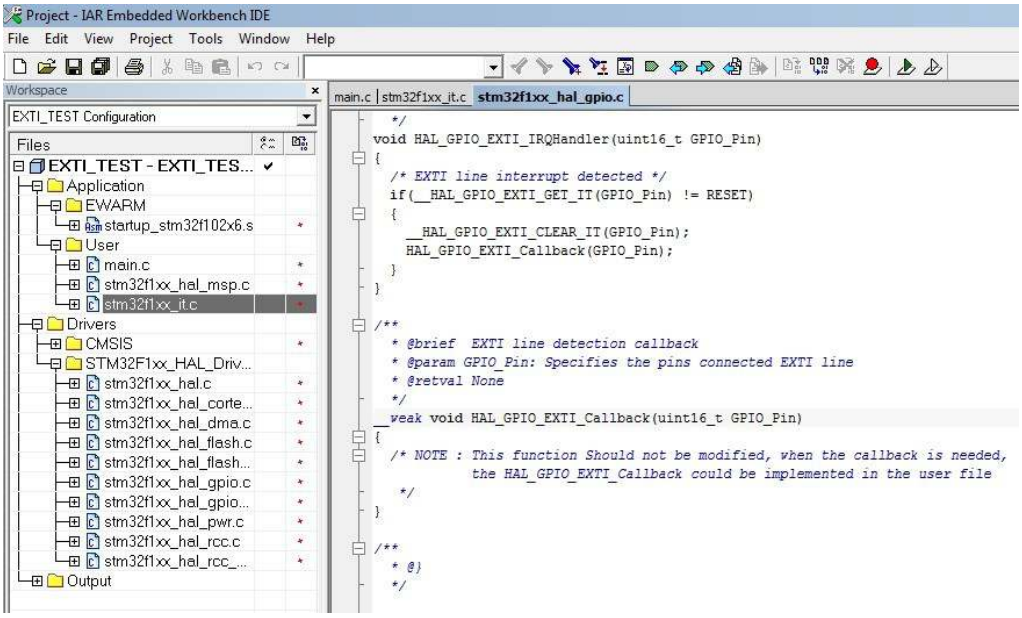

Figure 4.11: Body of the HAL_GPIO_EXTI_IRQHandler and HAL_GPIO_EXTI_Callback functions inside stm32f1xx_hal_gpio.c

In the following example, we connect three pushbuttons to the PB10, PB11, and PC0 pins and in the interrupt service function of each external interrupt, we define three counting variables corresponding to the three interrupt pushbuttons and display the values on an LCD. The code inside the main.c file is shown in Figure 4.12 and Figure 4.13. We then copy the body of the HAL_GPIO_EXTI_Callback function inside the stm32f1xx_it.c file as shown in Figure 4.14.

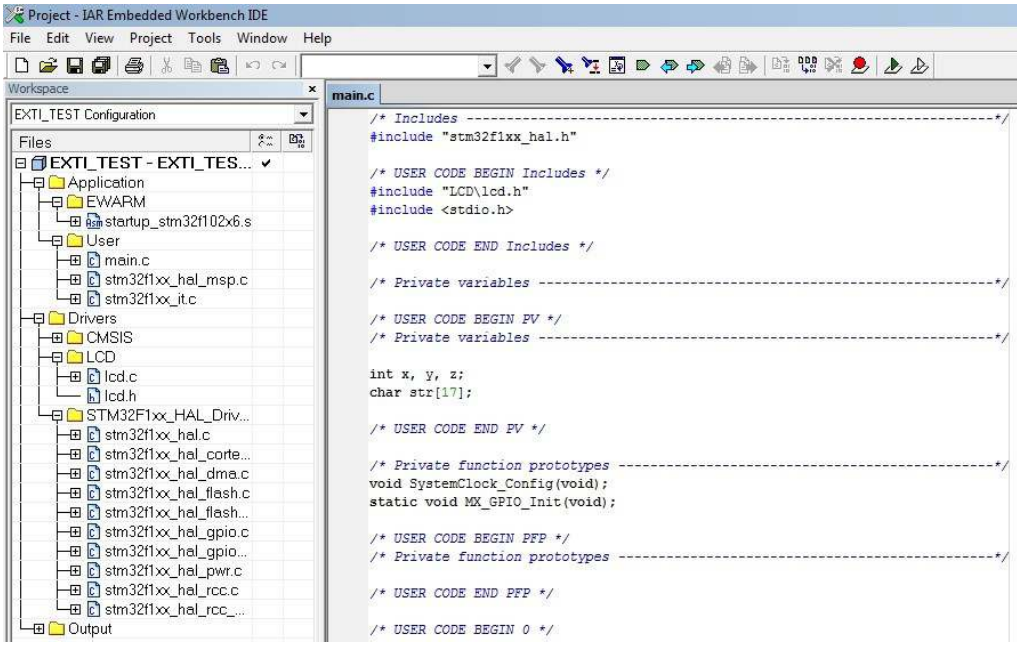

Figure 4.12: The header code of main.c

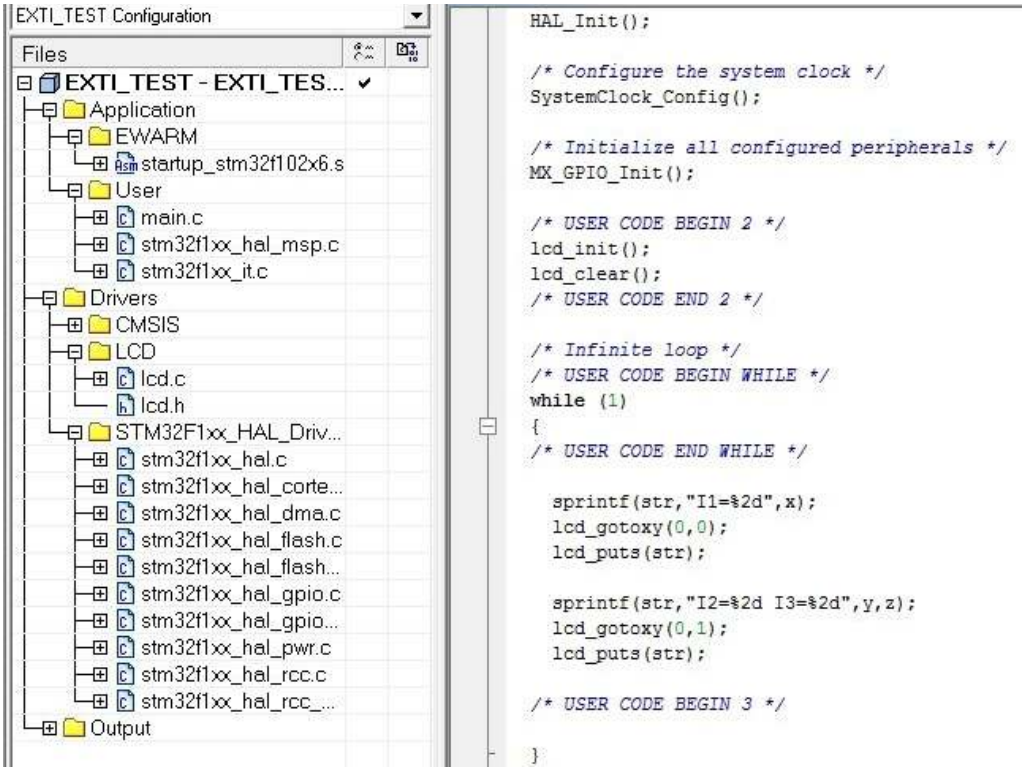

Figure 4.13: Code inside while loop in main.c

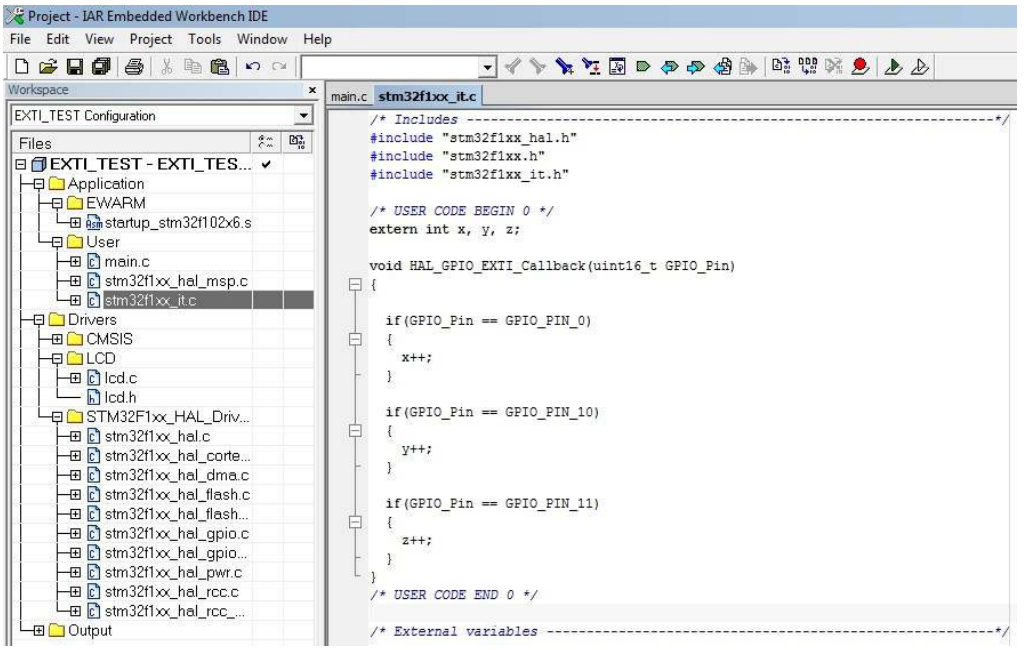

Figure 4.14: Code inside the HAL_GPIO_EXTI_Callback function in stm32f1xx_it.c

4.3 ● Summary

Interrupts in microcontrollers are important and useful. When an interrupt occurs in the microcontroller operation cycle, the interrupt service routine is performed and the code inside it is executed. An interrupt can have different factors such as external excitation, timer interrupts, analogue to digital converter interrupts, serial communications interrupts, etc. By using an interrupt, the microcontroller capacity for the evaluation of available and external hardware connected to it is increased. Interrupts have execution priority and upper priority always has execution priority. In this chapter, we discussed an external exited interrupt project. In Cortex-M microcontrollers, there is a nested vectored interrupt controller (NVIC) unit for handling interrupts.

Chapter 5 • Analogue to Digital Converters (ADCs) in STM32 Microcontrollers

5.1 • Introduction

Analogue to digital converters (ADCs) are very important tools for designing hardware that produces analogue signals. These signals can be the output of sensors, acoustic signals, medical device signals, etc. An analogue to digital converter samples from an analogue signal with a specific accuracy in determined time intervals. An analogue to digital converter has the following features: reference voltage, resolution, and sampling time. In every analogue to digital converter, the maximum level of input voltage is the supply voltage of the converter. The resolution coefficient determines the intervals of input voltage which is declared and generated with a related digital value. For example, in an analogue to digital converter with a reference voltage of 3.3 V and 12-bit resolution, the resolution coefficient is calculated as the below equation.

$$resolution\ coefficient = \frac{reference\ voltage}{2^{resolution} - 1} = \frac{3.3}{2^{12} - 1} = \frac{3.3}{4095} = 805.86080\ \mu V$$

The accuracy for reading the voltage in this analogue to digital converter is 805.86080 µV. Thus, this converter is not able to read a voltage of less than this value. The number of samples which are converted from analogue to digital in 1s duration is called the sampling rate. The minimum sampling rate of an analogue to digital converter should be twice the frequency of the input analogue signal. The STM32F103RET6 microcontroller has three analogue-to-digital units with 12-bit resolution. On this microcontroller, there are 16 specific pins for analogue input voltage.

5.2 • STM32CubeMX settings for ADC Project

In the following example, we want to enable two analogue inputs and display their values on an LCD. Therefore, in the STM32CubeMX software and Pinout section from the ADC1 subsection, we activate determined analogue input pins. Furthermore, the pins associated with LCD connection and external crystal resonator are enabled as shown in Figure 5.1. From the Clock Configuration section, we can set the ADC unit frequency in which its maximum frequency in the ADC unit is 14 MHz as depicted in Figure 5.2. From the configuration section and analogue subsection, click the ADC1 button as illustrated in Figure 5.3.

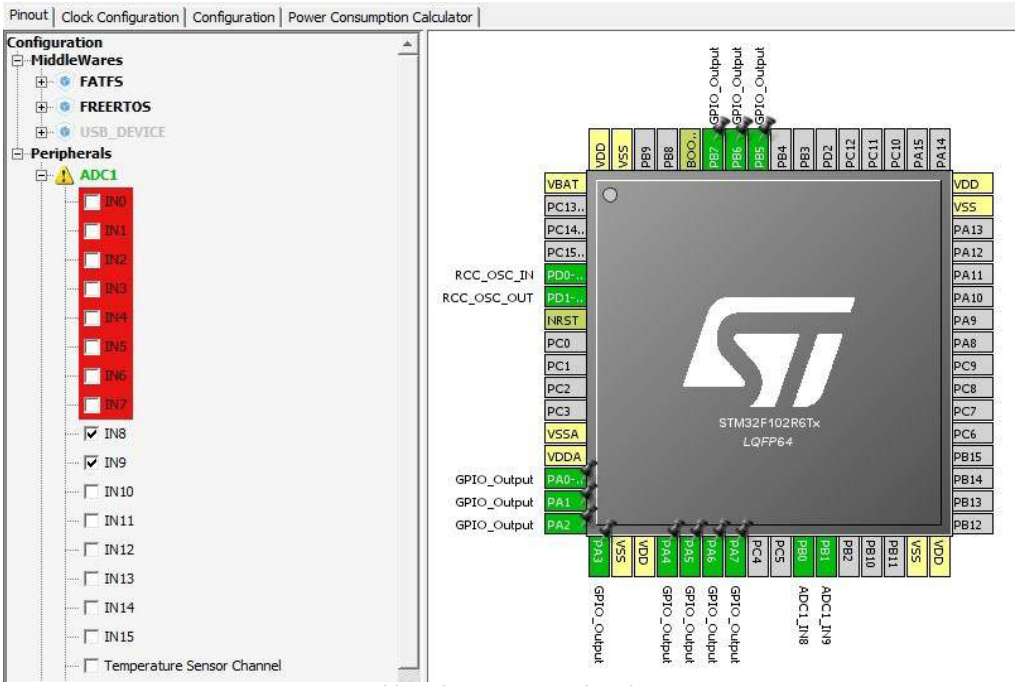

Figure 5.1: Enabling the ADC unit and analogue input pins

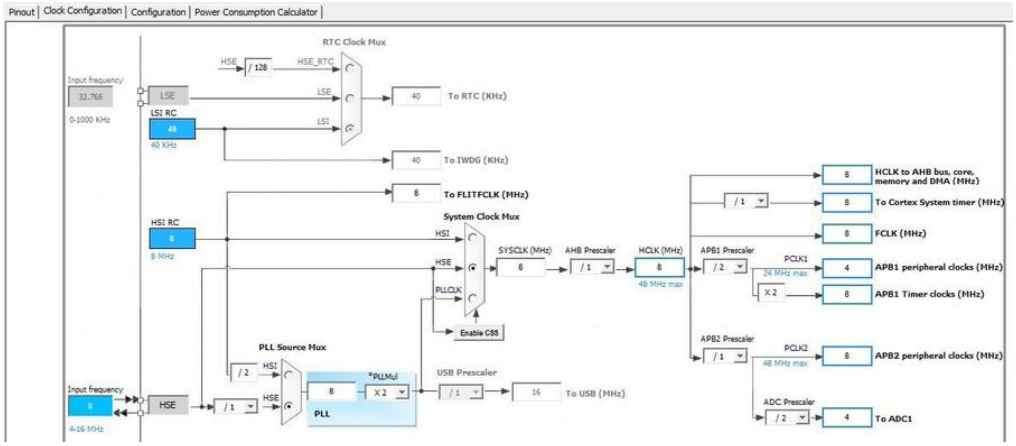

Figure 5.2: Setting the frequency of the ADC unit

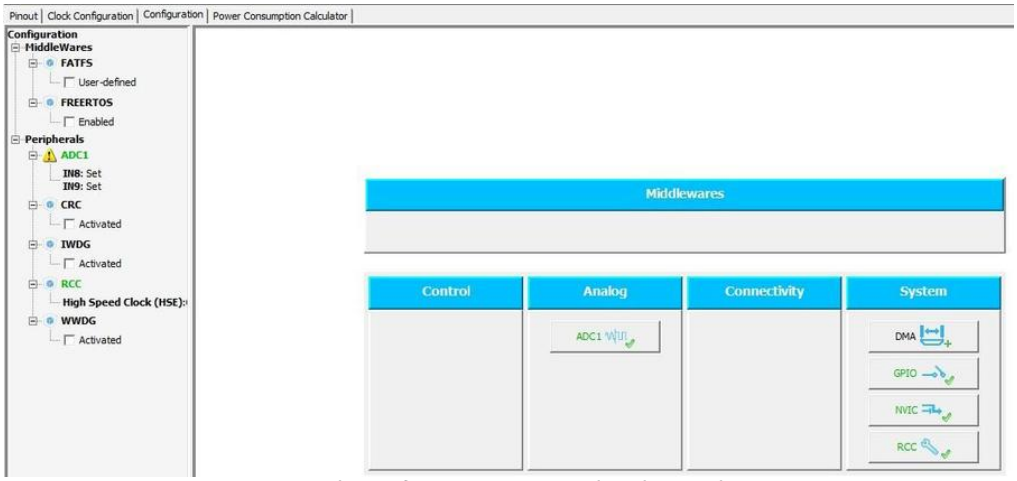

Figure 5.3: The Configuration section and Analogue subsection

From the ADC1 Configuration window, enable 'Scan Conversion Mode' and then set the number of conversion at 2 (number of inputs). Two Ranks are generated: In each Rank, we select the input channel number and sampling time for every channel as shown in Figure 5.4.

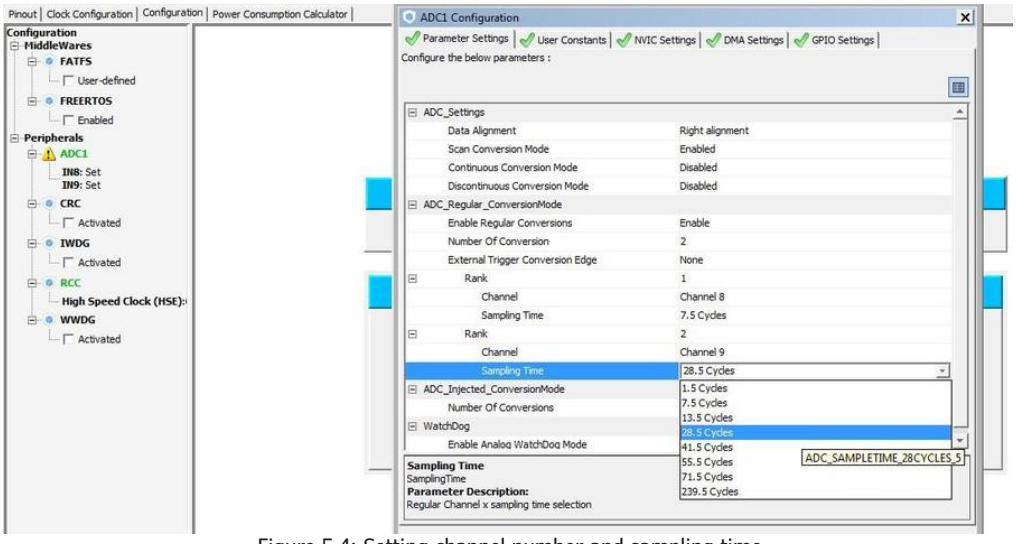

Figure 5.4: Setting channel number and sampling time

After generating the project, consider the functions of ADC. In Figure 5.5, the header of the main.c file is shown. The MX_ADC1_Init function contents are shown in Figure 5.6.

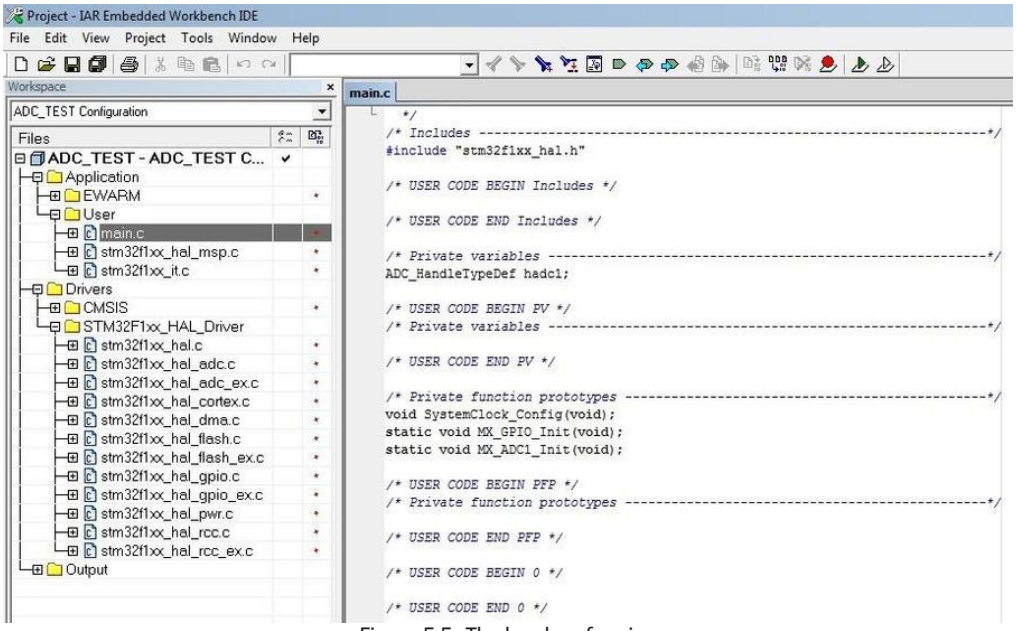

Figure 5.5: The header of main.c

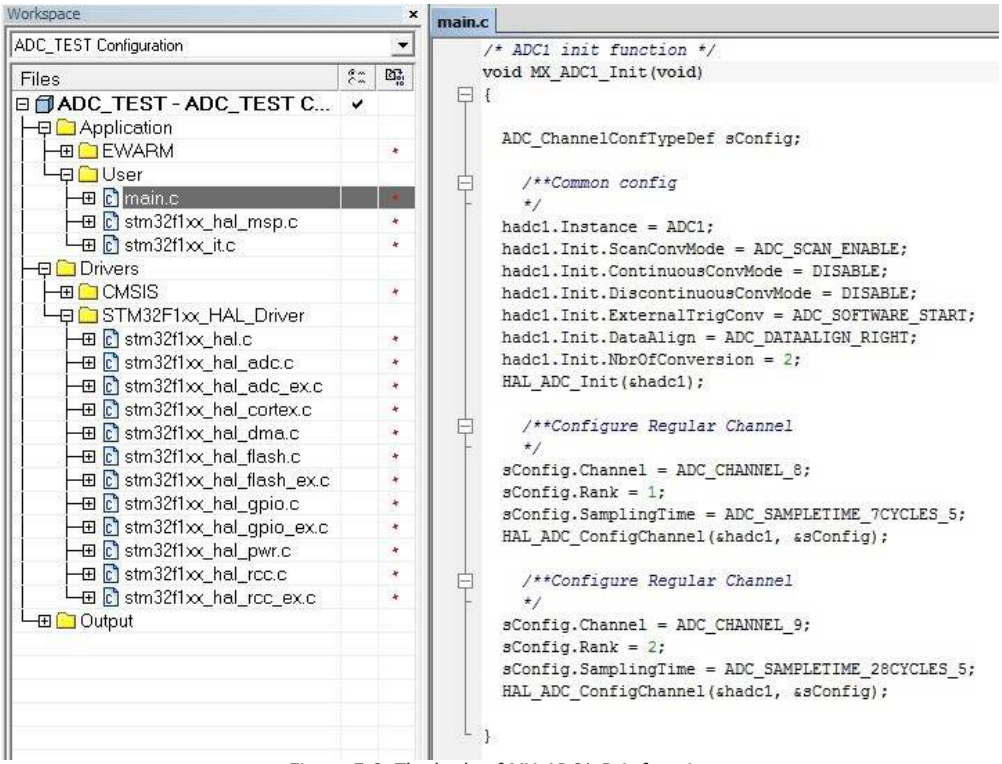

Figure 5.6: The body of MX_ADC1_Init function

The functions related to the ADC unit are in the stm32f1xx_hal_adc.c file as shown in Figure 5.7.

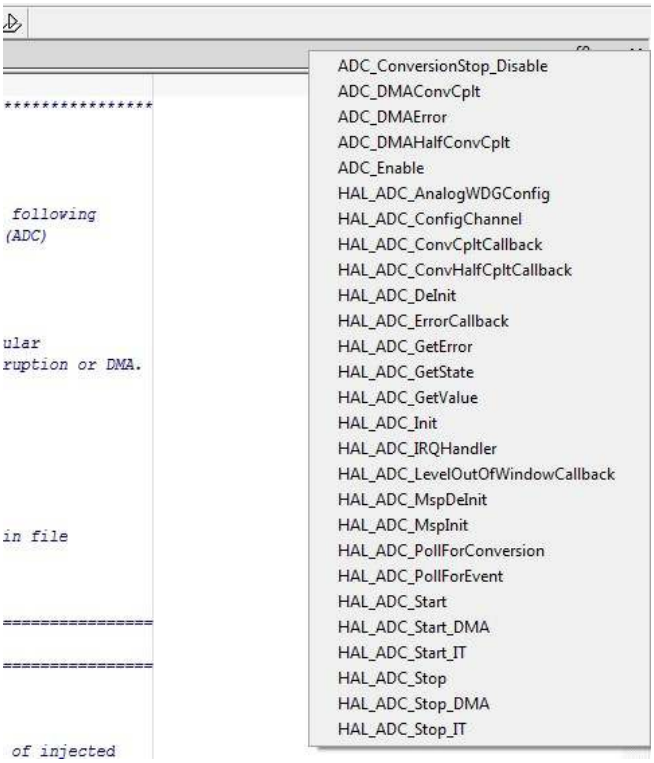

Figure5.7: The functions of ADC unit inside stm32f1xx_hal_adc.c

5.3 • Adding Codes and Libraries to the ADC Project

In the following example, we want to connect two potentiometers to the ADC1_IN8 and ADC1_IN9 pins (PB0 and PB1) and by varying them, show the input voltage values on an LCD. According to Figure 5.8, in the main.c file, we add the LCD and studio libraries and then define a float variable for voltage values and a string variable for displaying values on the LCD. The code inside the main function and before the while loop is illustrated in Figure 5.9.

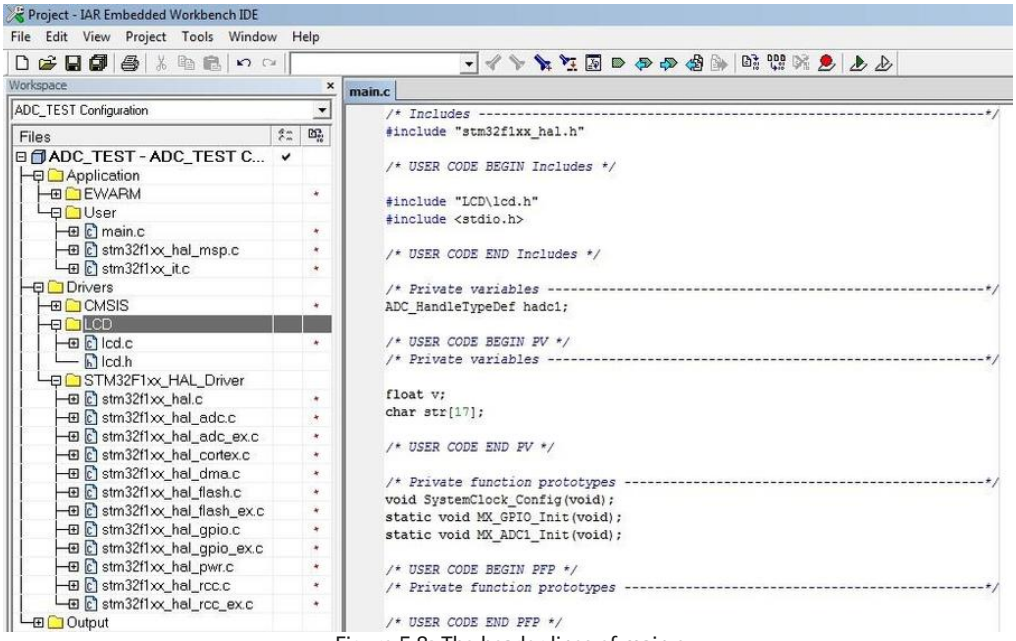

Figure 5.8: The header lines of main.c

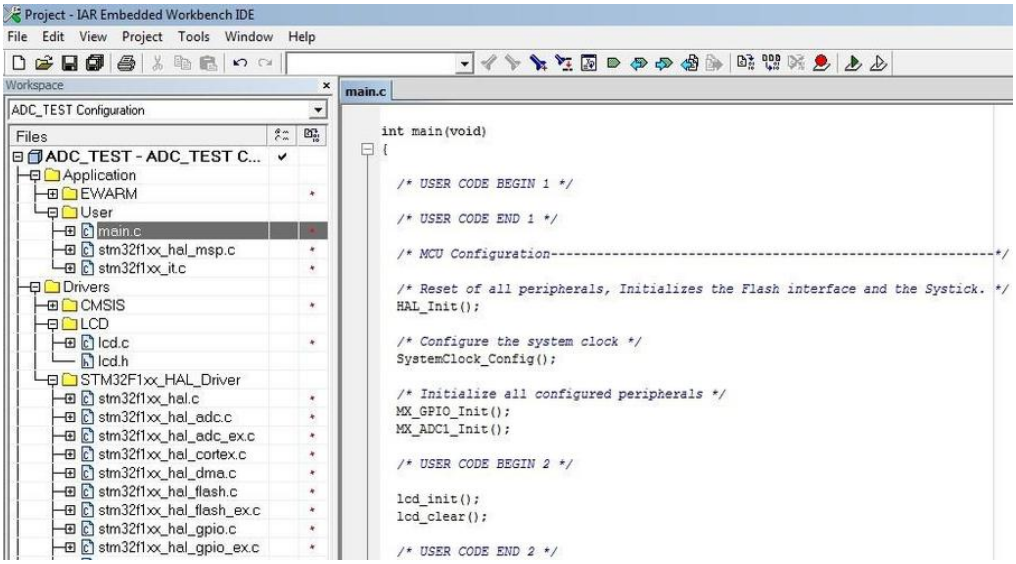

Figure 5.9: The code inside the main function

The code inside the while loop is depicted in Figure 5.10. After adding the code to the main.c file, we can compile the program and load the hex or bin file to the microcontroller using a programmer.

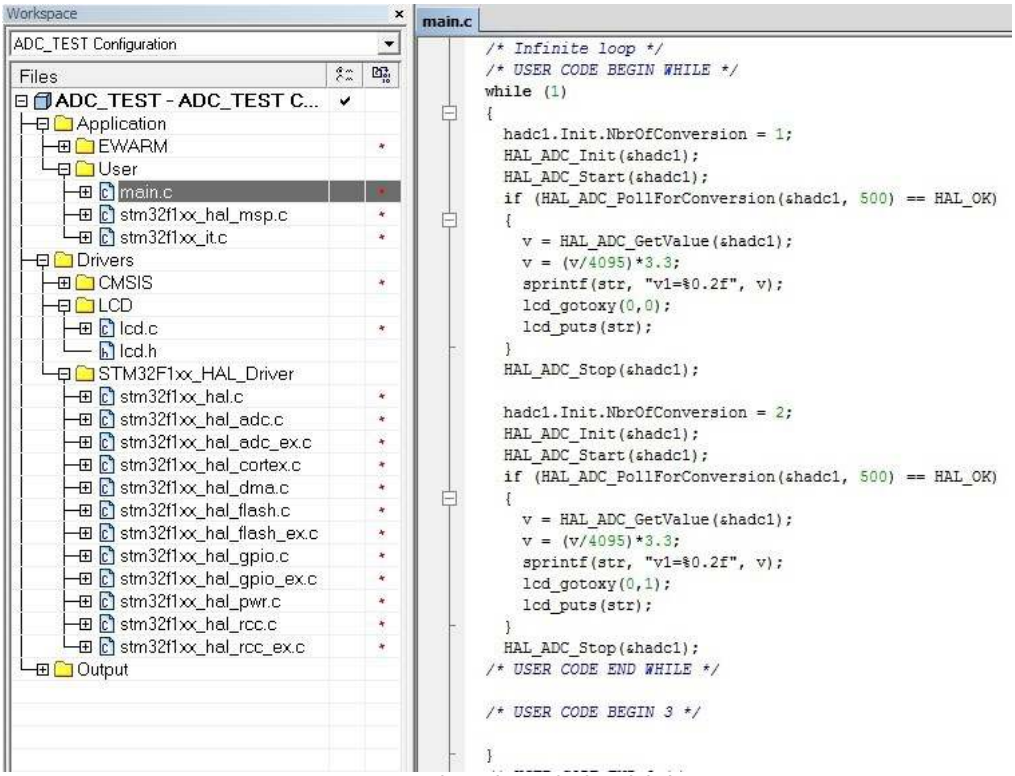

Figure 5.10: The code inside the while loop

5.4 • Summary

Analogue to digital converters (ADCs) are important tools for designing hardware that produces analogue signals. These signals can be the output of sensors, acoustic signals, and medical devices signals, etc. which are required to be converted from analogue to digital in order to be used in the digital domain. An analogue to digital converter samples from an analogue signal with specific accuracy in determined time intervals. In this chapter, we connected two potentiometers to the ADC1_IN8 and ADC1_IN9 pins (PB0 and PB1) and by varying them, displayed the input voltage values on an LCD.

Chapter 6 ● Digital to Analogue Converters (DACs) in STM32 Microcontrollers

6.1 ● Introduction

Digital to analogue converters (DACs) operate inversely to analogue to digital converters and convert a digital value to analogue. Similar to analogue to digital converters, digital to analogue converters have three main characteristics: reference voltage, resolution, and conversion time. The reference voltage and resolution conditions are completely analogous to analogue to digital converters. The only difference is that in digital to analogue converters, the digital integers are converted to voltage levels in output. A digital to analogue converter with 12-bit resolution and 3.3 V reference voltage can generate voltage values from 0 V to 3.3 V for integer values of 0 to 4095. The output calculation equation in a DAC converter is as follows:

$$generated\ voltage\ in\ DAC = \frac{reference\ voltage}{2^{resolution} - 1} \times DAC\ value$$

The STM32F103RET6 microcontroller has 2 DAC units with two separate pins for each output.

6.2 ● STM32CubeMX settings for DAC Project

In the following example, we enable one of the DAC units, generate a waveform, and display it on an oscilloscope.

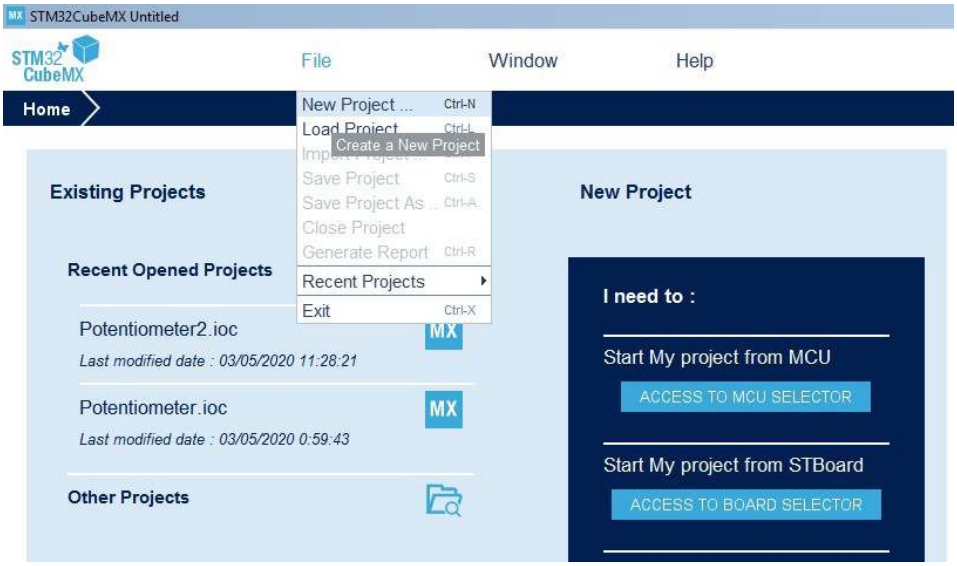

Figure 6.1: Creating a new project

In accordance with Figure 6.1, we create a new project and then select the MCU (STM32F103RET6) as shown in Figure 6.2. In the Analogue section and the DAC subsection, activate the OUT1 configuration to enable the relative pin (PA4) and enable the output buffer. We also enable the main external crystal pins as shown in Figures 6.3 and 6.4 respectively.

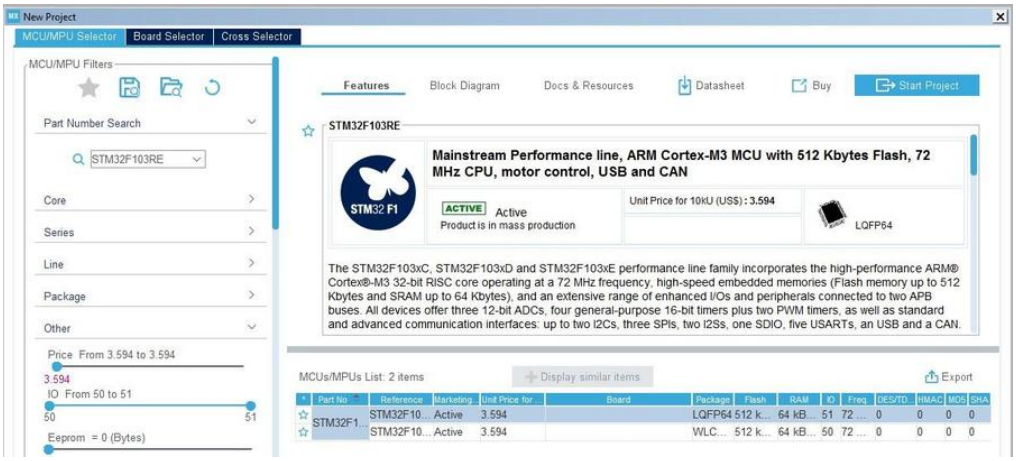

Figure 6.2: Selecting the MCU (STM32F103RET6)

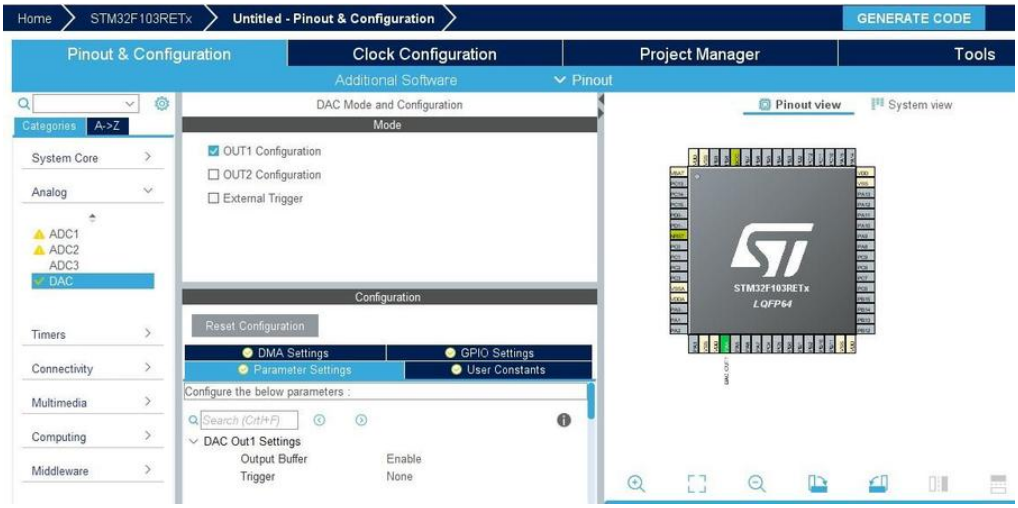

Figure 6.3: Activating OUT1 Configuration in the DAC subsection

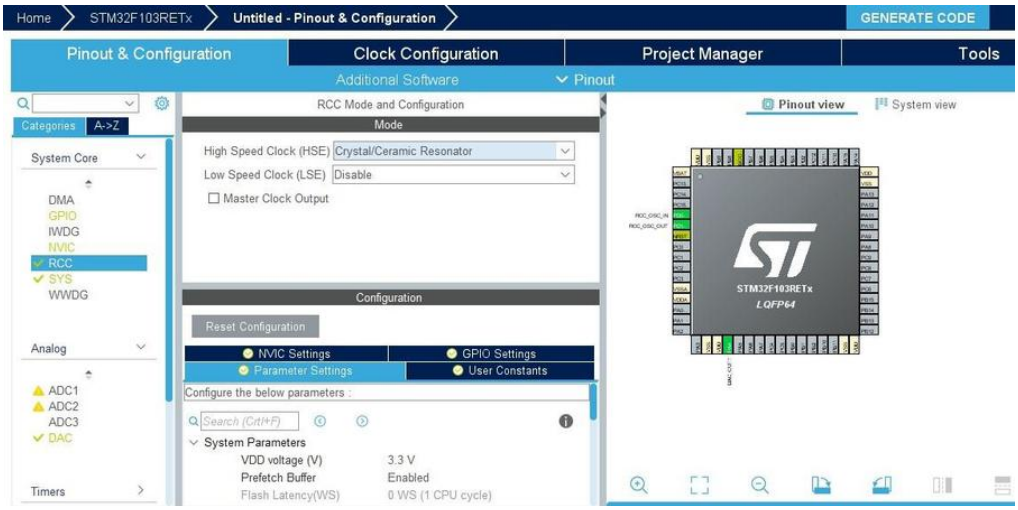

Figure 6.4: Enabling external crystal resonator

From the Project Manager section, select a project name and the compiler EWARM IDE as shown in Figure 6.5. Finally, generate the code.

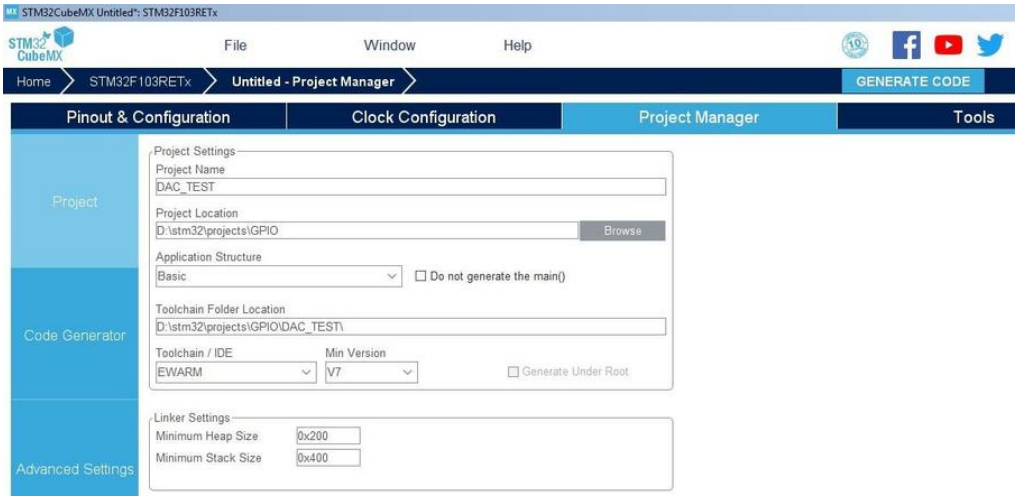

Figure 6.5: The Project Manager settings

The code before the main function is shown in Figure 6.6. The code inside the MX_DAC_Init function is illustrated in Figure 6.7.

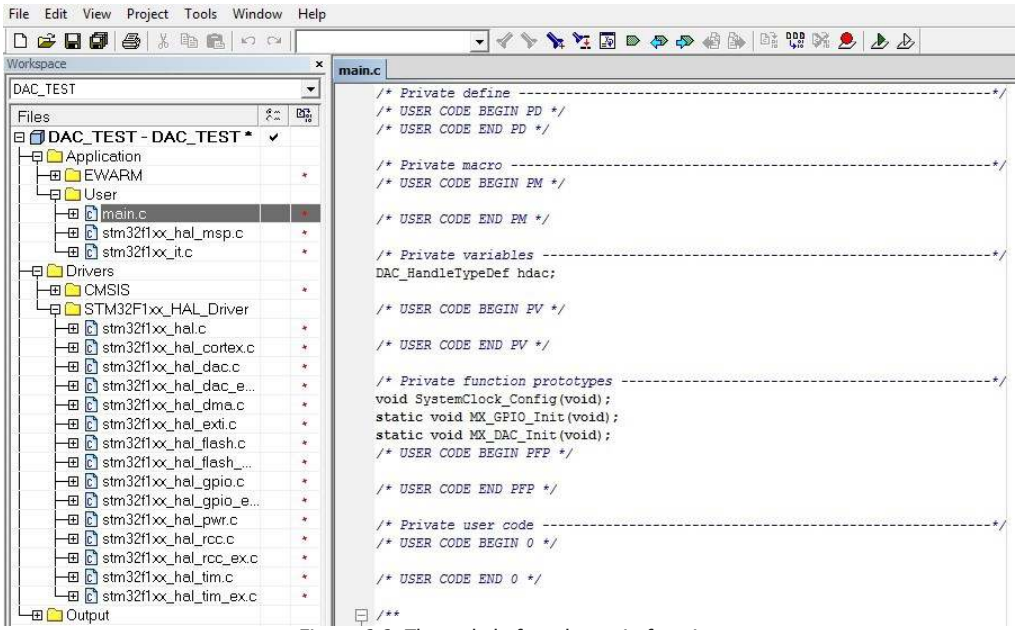

Figure 6.6: The code before the main function

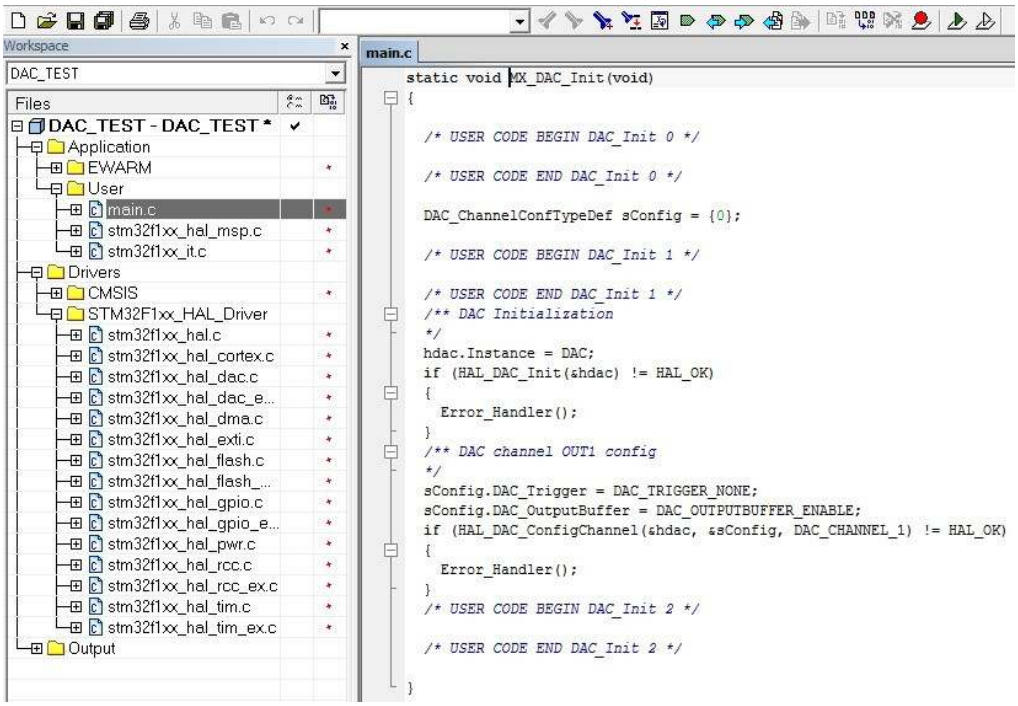

Figure 6.7: The code inside the MX_DAC_Init function

The functions related to the DAC unit are inside the stm32f1xx_hal_dac.c file and are illustrated in Figure 6.8.

Figure 6.8: The functions available in stm32f1xx_hal_dac.c

6.3 ● Adding Codes to main.c

As previously mentioned, we are going to generate an arbitrary signal with integers and display the DAC1 pin (PA4) voltage signal on an oscilloscope. In accordance with Figure 6.9, in main.c, a variable (i) as array index and the array (data) with 17 members and initial values for the DAC unit is defined. The codes inside the while loop are shown in Figure 6.10. After adding the code to the main.c file, we can compile the program and load the hex or bin file produced to the microcontroller by a programmer.

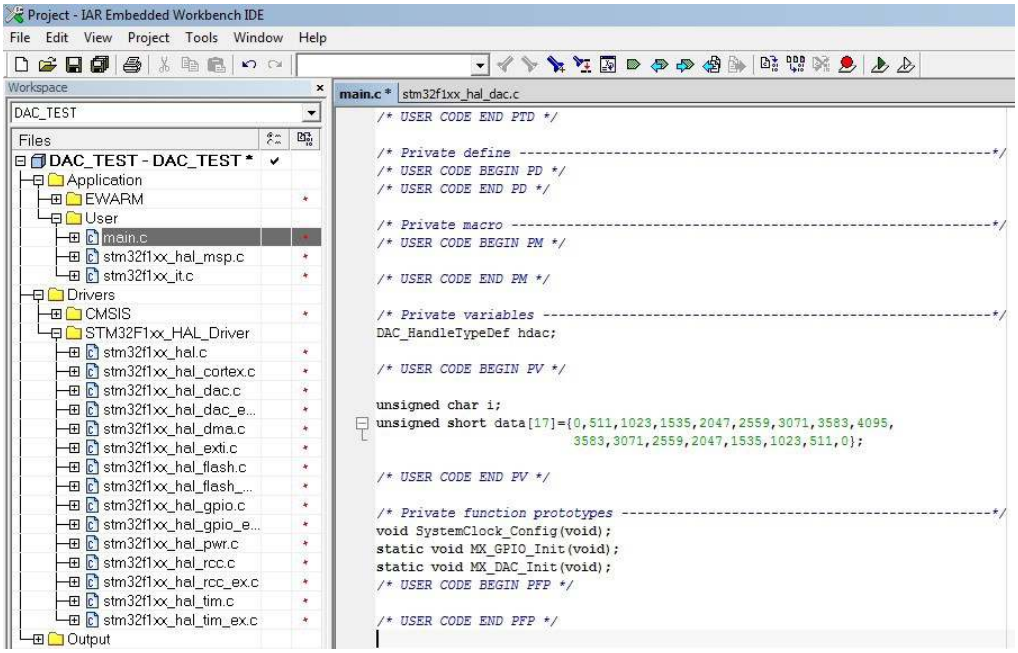

Figure 6.9: The code before the main function

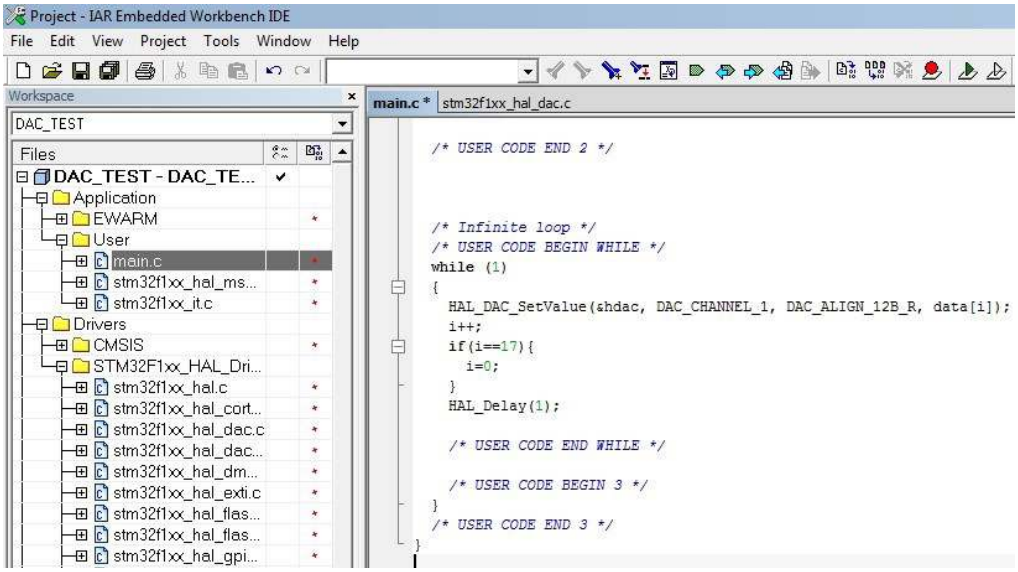

Figure 6.10: The code inside the while loop

If as in Figure 6.5, we select the Keil uVision V5 compiler, the code is generated in that compiler environment and we can add necessary code to main.c as shown in figures 6.11 and 6.12.

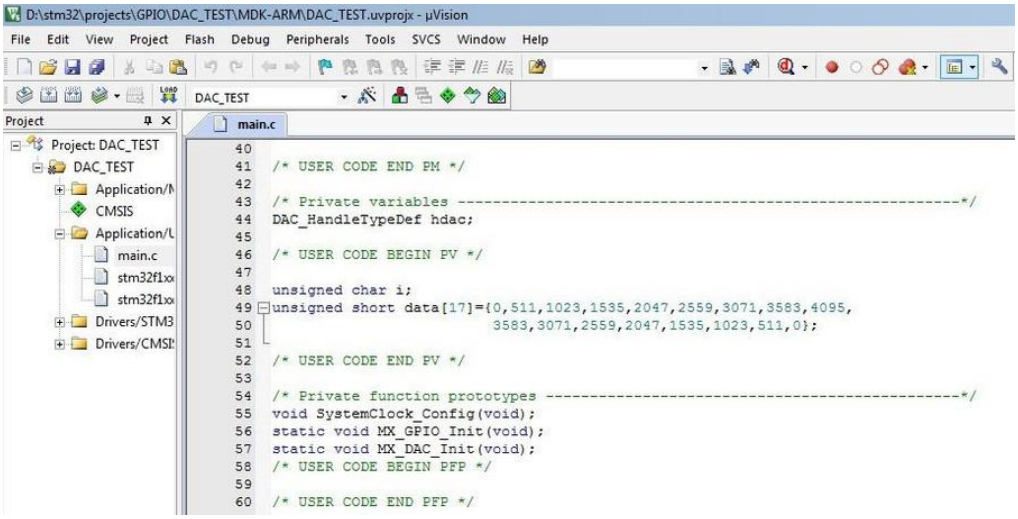

Figure 6.11: The code before the main function in Keil uVision

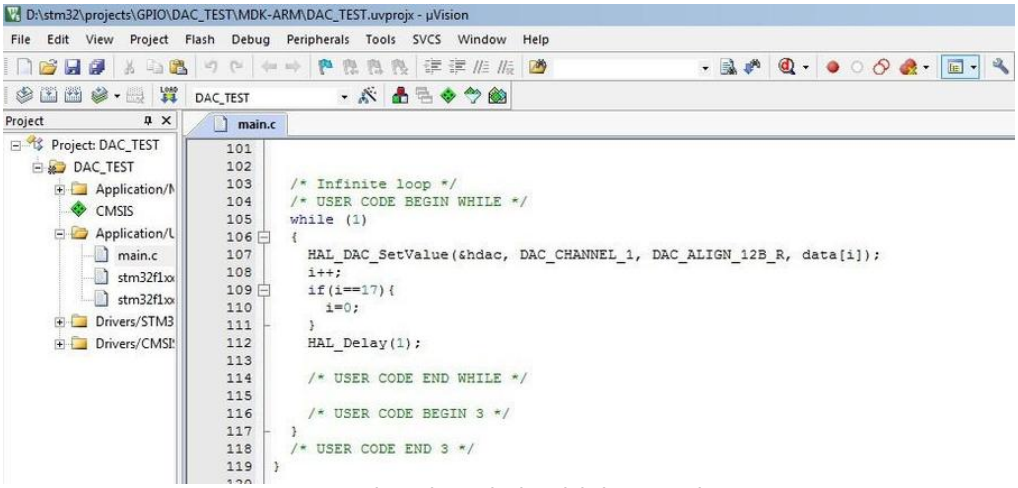

Figure 6.12: The code inside the while loop in Keil uVision

6.4 ● Summary

Digital to analogue converters (DACs) operate inversely when compared to analogue to digital converters. They convert a digital value to an analogue signal. In this chapter, we enabled a DAC unit and generated a waveform, and then displayed it on an oscilloscope.

Chapter 7 • Timers and Counters in STM32 Microcontrollers

7.1 • Introduction

Timers and counters are the most important sections of a microcontroller and have critical roles in timing and operation. Using a delay function in a program causes program execution to stop. This is not a recommendable technique in a professional program. By using timers, all delays are omitted from the program. All timing scheduling and delays (without stopping the program) are managed by a timer that operates in parallel with the processor. A timer is a counter which can count the regular pulses of an oscillator and generate regular times. This counter can count the external pulses which are applied to the microcontroller. The STM32F103RET6 microcontroller has 8 timers.

7.2 • Timer Project Settings

We are going to drive 2 timers with 250 ms and 1 s respectively and count a variable with each of them and display it on an LCD. The driving of other timers is similar. For this purpose, according to Figure 7.1 in the timers section and the TIM1 subsection, the clock source is set as the internal clock. In the parameter settings section, we set prescaler to 7999 and counter period to 249. Then from the NVIC Settings section, we activate the TIM1 update interrupt as shown in Figure 7.2.

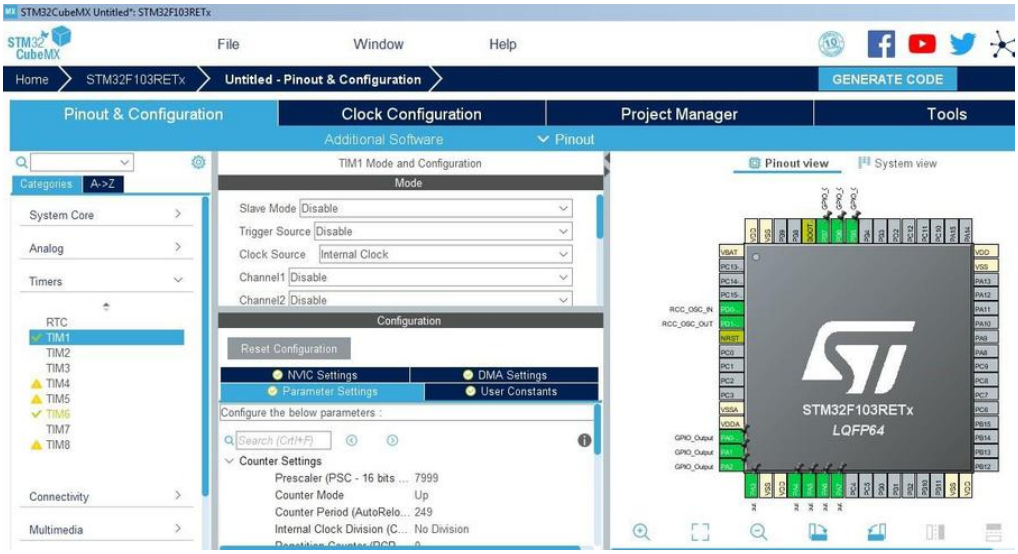

Figure 7.1: Activating TIM1 unit

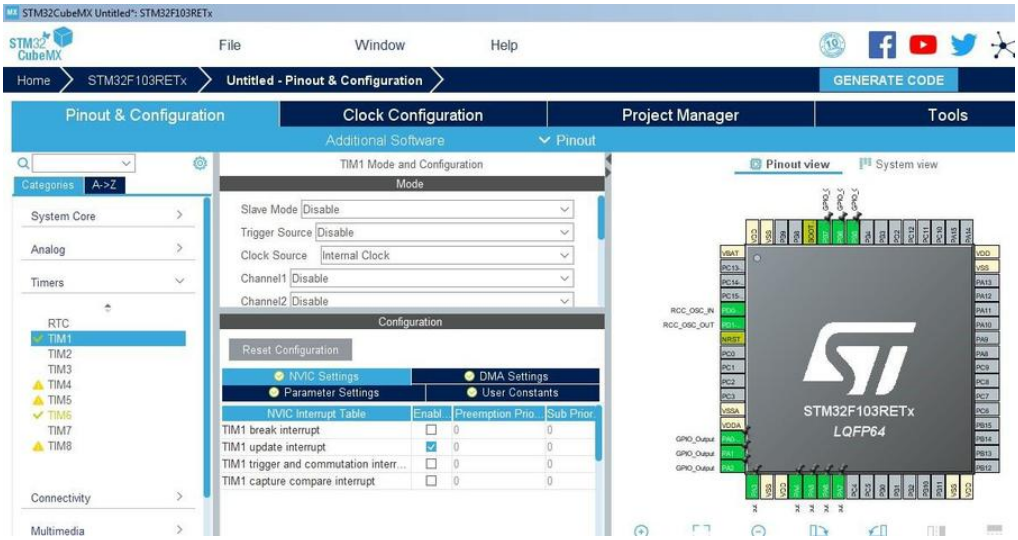

Figure 7.2: Activating TIM1 update interrupt

For activating TIM6, we select the activated icon. In the parameter settings section, we also set the prescaler to 7999 and counter period to 999 as shown in Figure 7.3. Then, from the NVIC settings section, we enable the TIM6 global interrupt as shown in Figure 7.4.

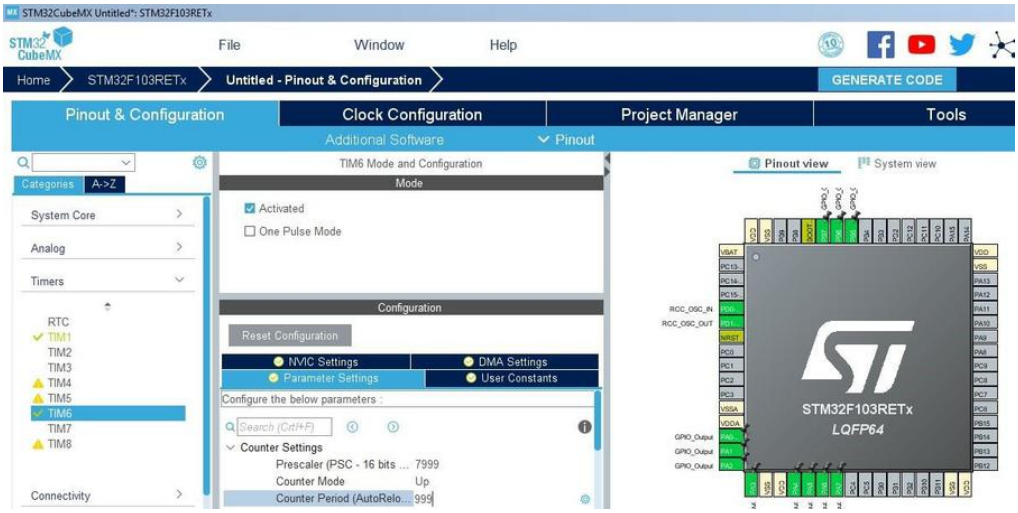

Figure 7.3: Activating TIM6 unit

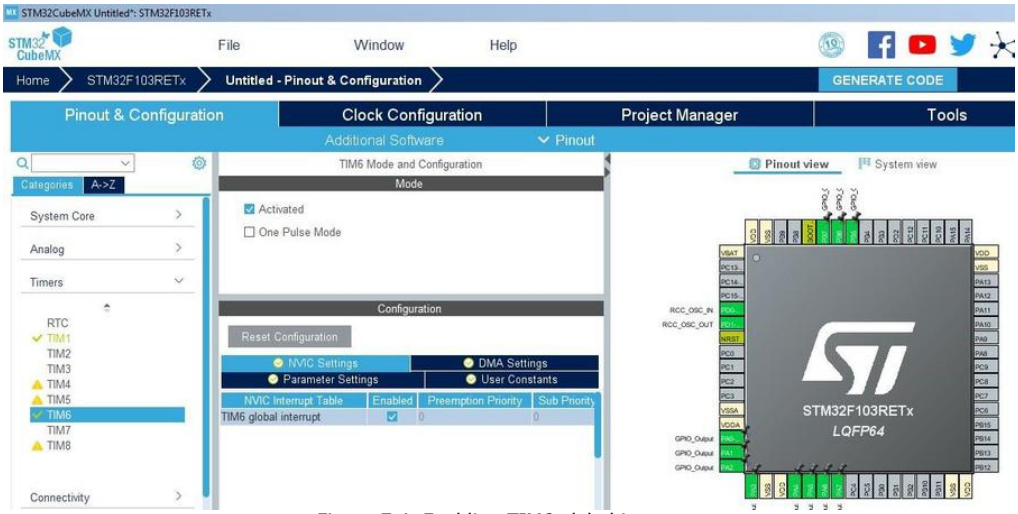

Figure 7.4: Enabling TIM6 global interrupt

We also activate the microcontroller pins related to the driving of the LCD and external crystal resonator as mentioned in previous chapters. The clock configuration section is shown in Figure 7.5. As illustrated in Figure 7.5, the APB1 (TIM6) and APB2 Timer clocks (TIM1) frequencies are selected as 8 MHz. For generating TIM1 interrupts every 250 ms, select the prescaler value equal to 7999 to divide the 8 MHz frequency of TIM1 to 8000 (from 0 to 7999) to produce a 1 kHz frequency with a 1ms period. For Counter Period, select 249 (from 0 to 249) to generate 250 ms. Similarly, for generating TIM6 interrupts every 1s, select the Prescaler value equal to 7999 to divide the 8 MHz frequency of TIM6 to 8000 (from 0 to 7999) to produce 1 kHz frequency with a 1ms period. Also, for Counter Period select 999 (from 0 to 999) to generate 1s. For determining interrupt priority from the NVIC configuration section, select 1 for TIM1 and 2 for TIM6 Preemption Priority as shown in Figure 7.6.

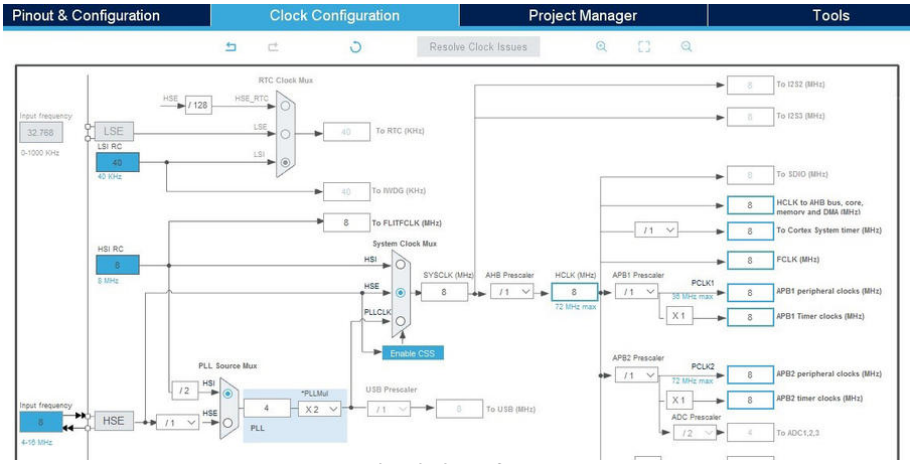

Figure 7.5: The Clock Configuration section

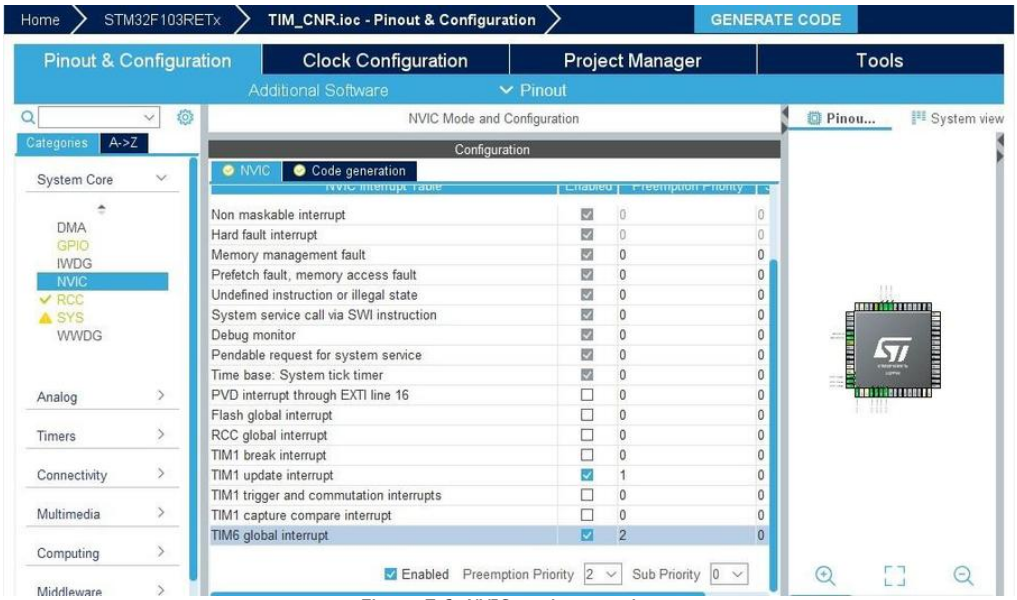

Figure 7.6: NVIC settings section

After inputting the above settings, select SW4STM32 from the Project Manager Tab and generate the code as illustrated in Figure 7.7.

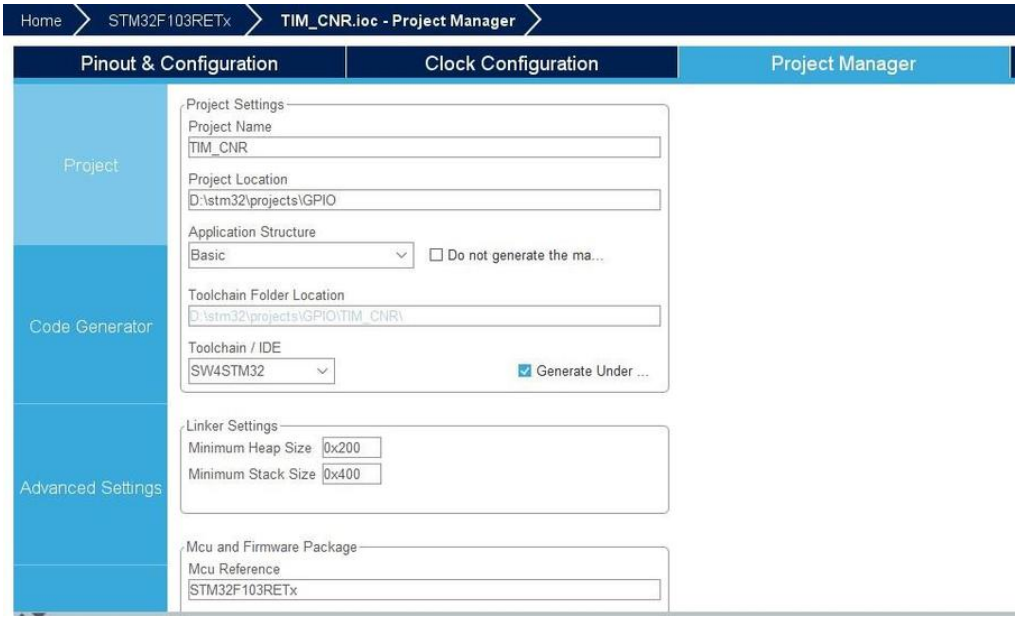

Figure 7.7: selecting SW4STM32 as the compiler

The header code of the main.c file is shown in Figure 7.8. Also, the body of functions MX_TIM1_Init and MX_TIM6_Init are depicted in Figure 7.9 and Figure 7.10 respectively.

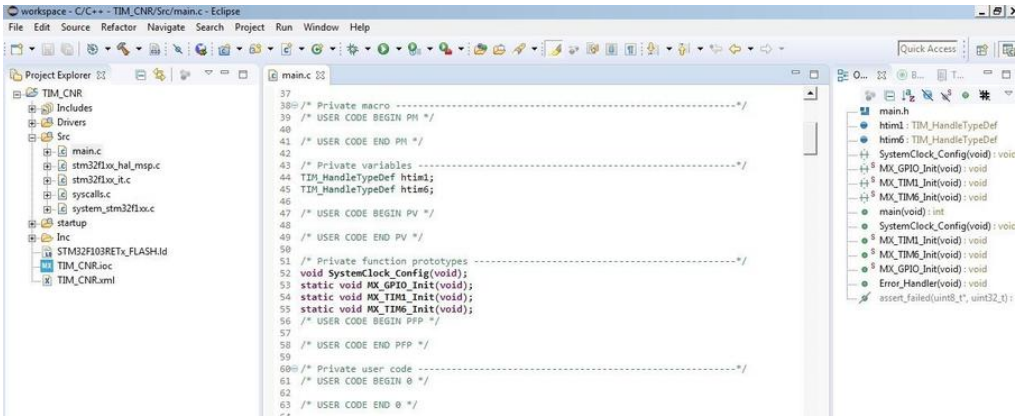

Figure 7.8: The header code of the main.c file

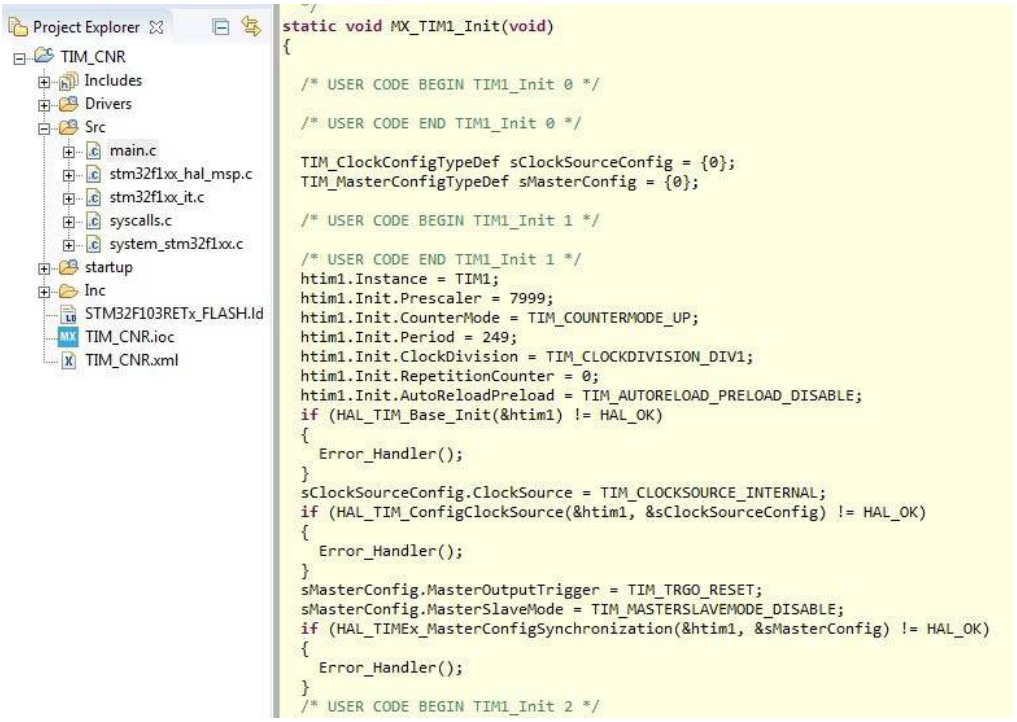

Figure 7.9: The body of function MX_TIM1_Init

```
  * @brief TIM6 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM6_Init(void)
{

  /* USER CODE BEGIN TIM6_Init 0 */

  /* USER CODE END TIM6_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM6_Init 1 */

  /* USER CODE END TIM6_Init 1 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = 7999;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = 999;
  htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM6_Init 2 */

  /* USER CODE END TIM6_Init 2 */
```

Figure 7.10: The body of function MX_TIM6_Init

The functions related to the Timer unit are inside the stm32f1xx_hal_tim.c file as shown in Figure 7.11. Interrupt service functions to the timer interrupts are inside the stm32f1xx_it.c file as shown in Figure 7.12.

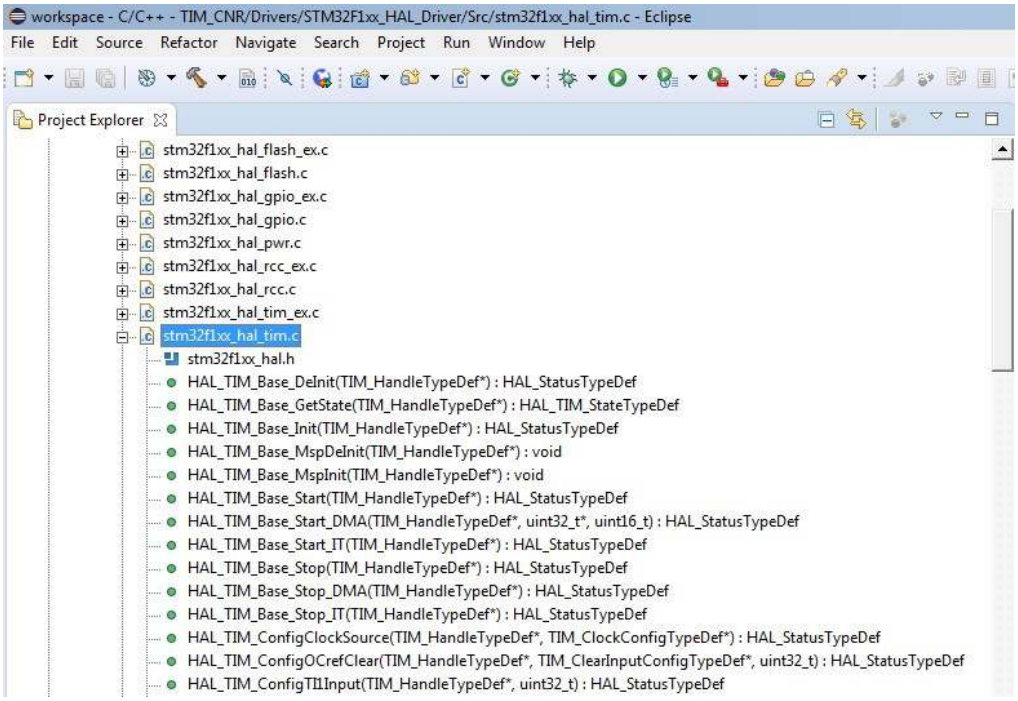

Figure 7.11: The Timer functions inside the stm32f1xx_hal_tim.c file

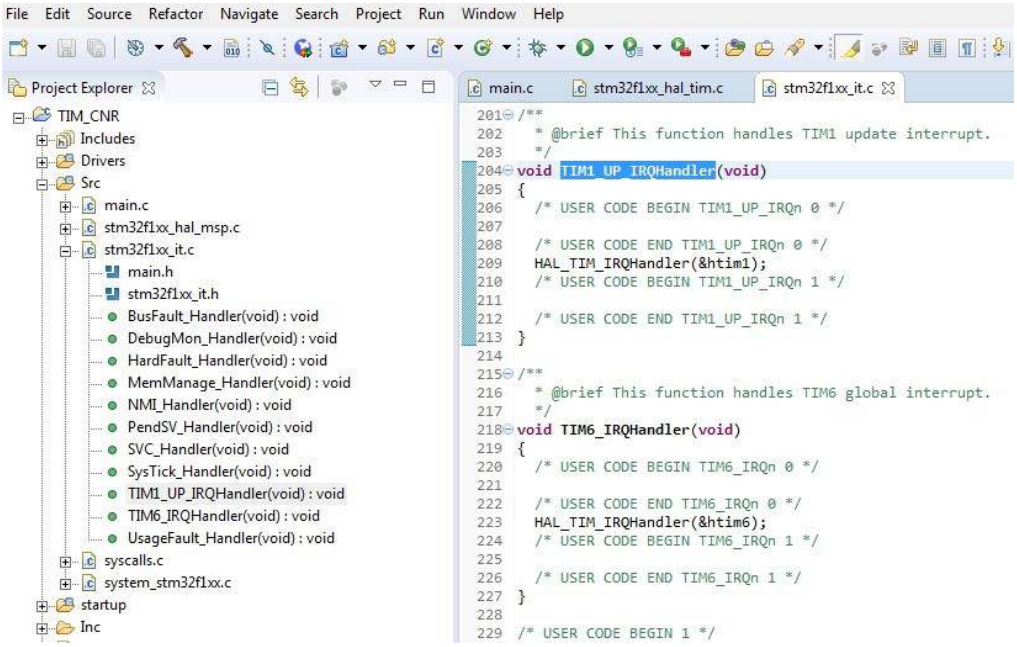

Figure 7.12: Timer interrupt functions inside the stm32f1xx_it.c file

As previously mentioned, we are going to drive two timers (TIM1 and TIM6) and generate a 250 ms interrupt with TIM1 and a 1 s interrupt with TIM6 and then count a variable with each timer and display it on an LCD. The codes before the main function are illustrated in Figure 7.13.

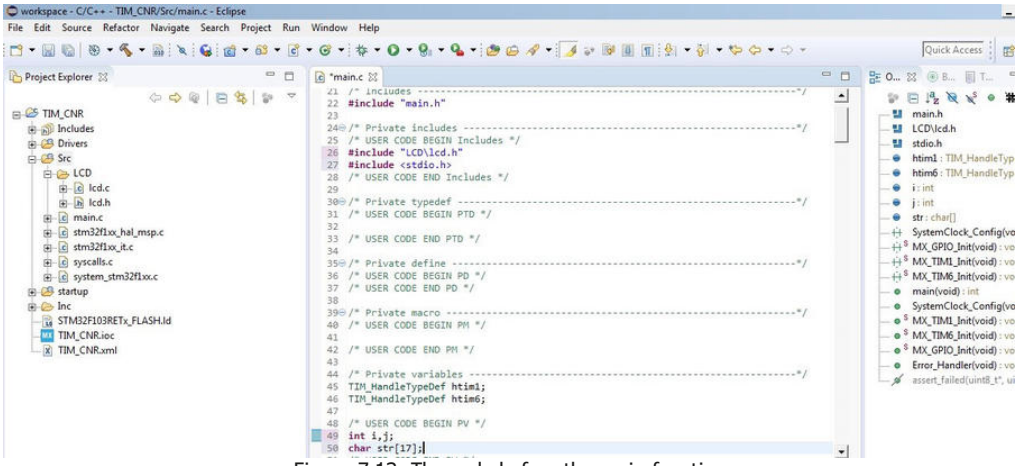

Figure 7.13: The code before the main function

The code inside the while loop is shown in Figure 7.14. The header code of the stm32f1xx_ it.c file is illustrated in Figure 7.15.

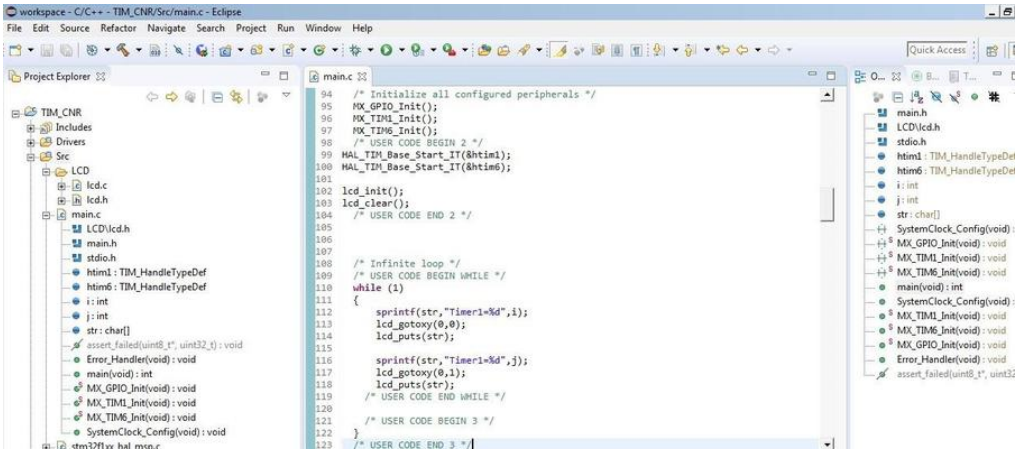

Figure 7.14: The code inside the while loop

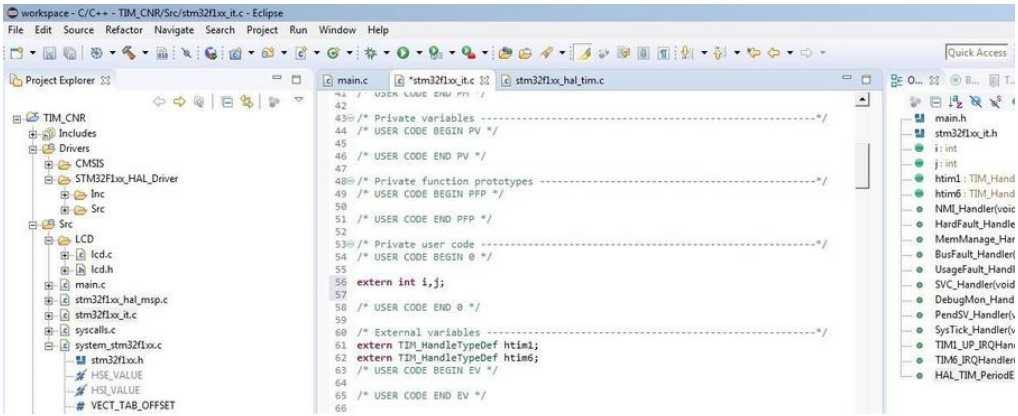

Figure 7.15: The header code of the stm32f1xx_it.c file

The executing function of user codes using HAL_TIM_PeriodElapsedCallback function (inside the stm32f1xx_hal_tim.c file) which is copied inside the stm32f1xx_it.c file is shown in Figure 7.16.

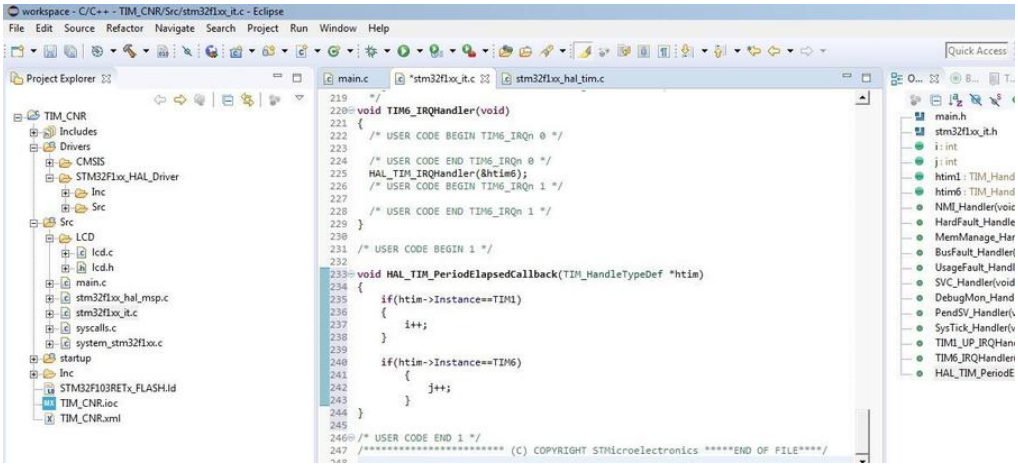

Figure 7.16: The executing function of user codes copied inside the stm32f1xx_it.c file

We can then compile the project by right-clicking the project name and selecting the ‘build project’ icon as illustrated in Figure 7.17.

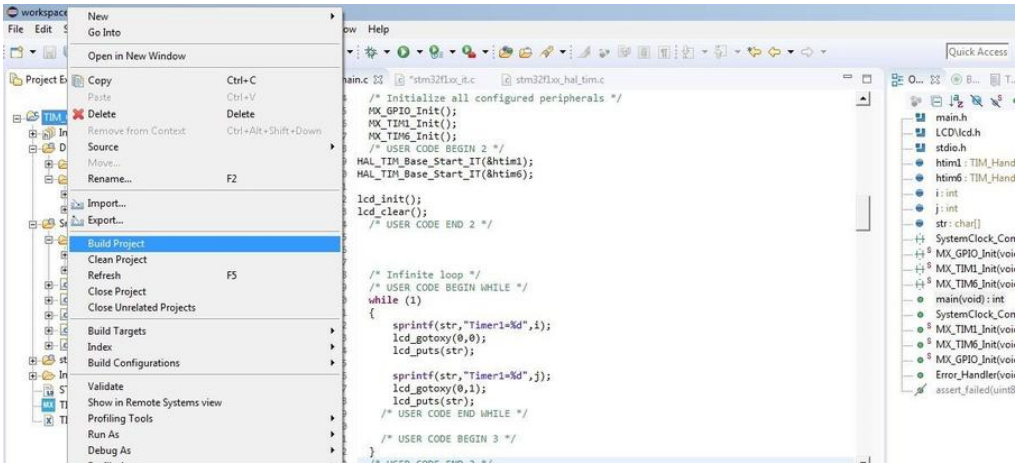

Figure 7.17: Compiling the project

After successfully compiling the project, the hex file generated for programming the microcontroller will be inside the debug folder as shown in Figure 7.18.

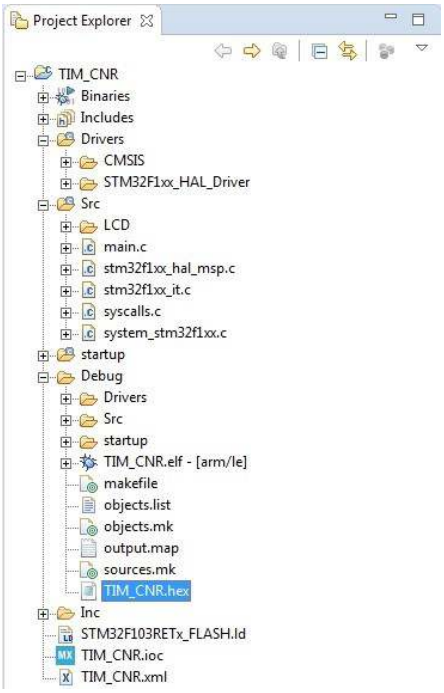

Figure 7.18: The generated hex file inside the Debug folder

7.3 • Counter Project Settings

Timers are counters that count regular pulses of an oscillator. Timers also can count pulses that are applied externally to a pin. In this case, the timer is called a counter. We want to enable the TIM1 unit as a counter of external pulses and connect it to a pin by a pushbutton and excite it with rising edge pulses, counting them with a variable associated with the counter and then display it on an LCD. For this purpose, in accordance with Figure 7.19 from the Pinout & Configuration section and TIM1 subsection, set the slave mode to External Clock Mode 1 and Trigger Source as TI1FP1 and then activate the pins related to driving the LCD as GPIO_Output. Also enable the external crystal resonator pins. From the parameter settings, settings such as Prescaler (0), Counter Mode (Up), Counter Period (65535 for 16 bits' value) Trigger Polarity (Rising Edge), and Trigger Filter (15) are enabled. In the system core section and TIM subsection, set pull-down for GPIO mode (since we chose trigger polarity as rising edge before) as shown in Figure 7.20. From the system core section and NVIC subsection, enable the TIM1 update interrupt. When the counter value reaches the counter period value, the TIM1 interrupt is activated. Finally, we can generate code in the SW4STM32 software environment as illustrated in Figure 7.21. The body of the MX_TIM1_ Init function is shown in Figure 7.22.

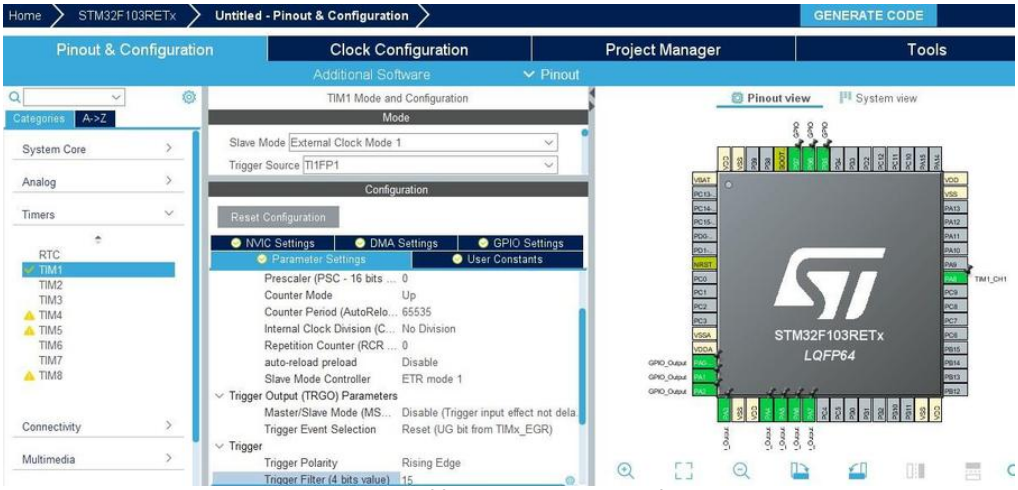

Figure 7.19: Enabling TIM1 as an external counter

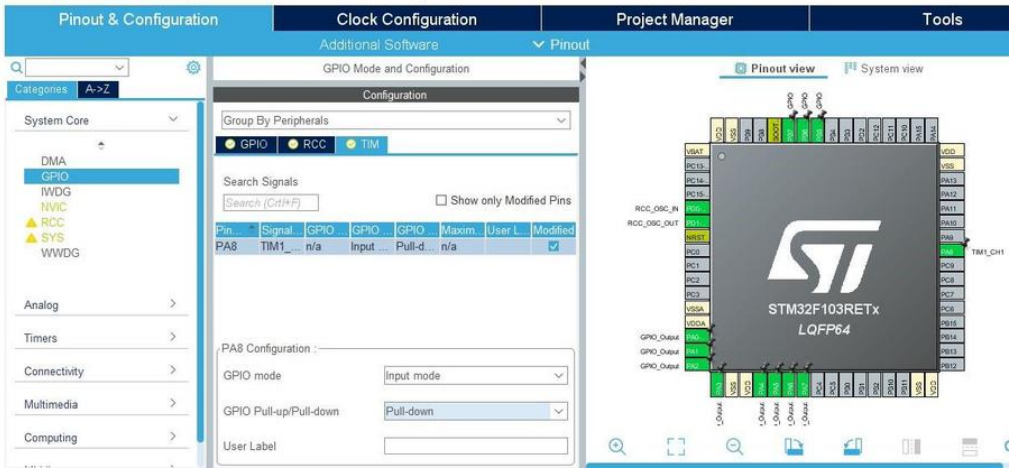

Figure 7.20: Pull-down setting for GPIO mode of TIM1

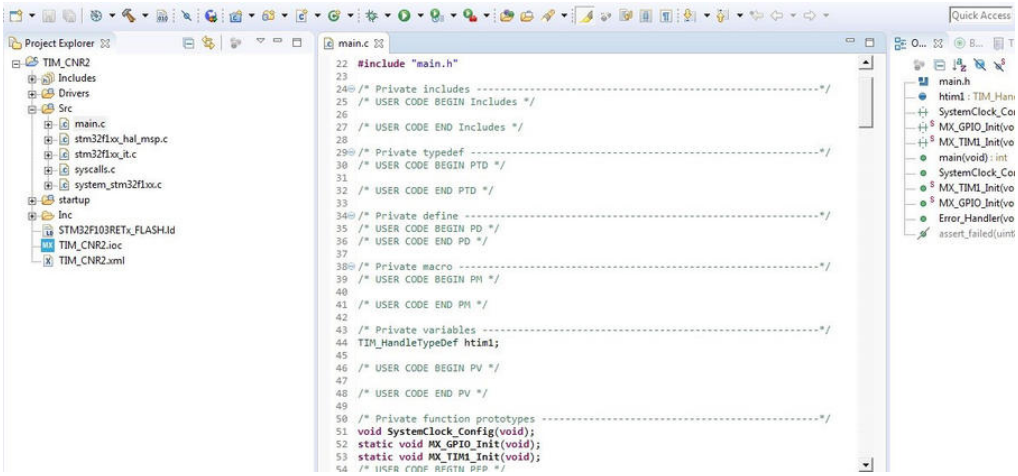

Figure 7.21: The header code of main.c

```
  * @retval None
  */
static void MX_TIM1_Init(void)
{

  /* USER CODE BEGIN TIM1_Init 0 */

  /* USER CODE END TIM1_Init 0 */

  TIM_SlaveConfigTypeDef sSlaveConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM1_Init 1 */

  /* USER CODE END TIM1_Init 1 */
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 0;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 65535;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
  sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
  sSlaveConfig.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
  sSlaveConfig.TriggerFilter = 15;
  if (HAL_TIM_SlaveConfigSynchro(&htim1, &sSlaveConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM1_Init 2 */
```

Figure 7.22: The body of function MX_TIM1_Init

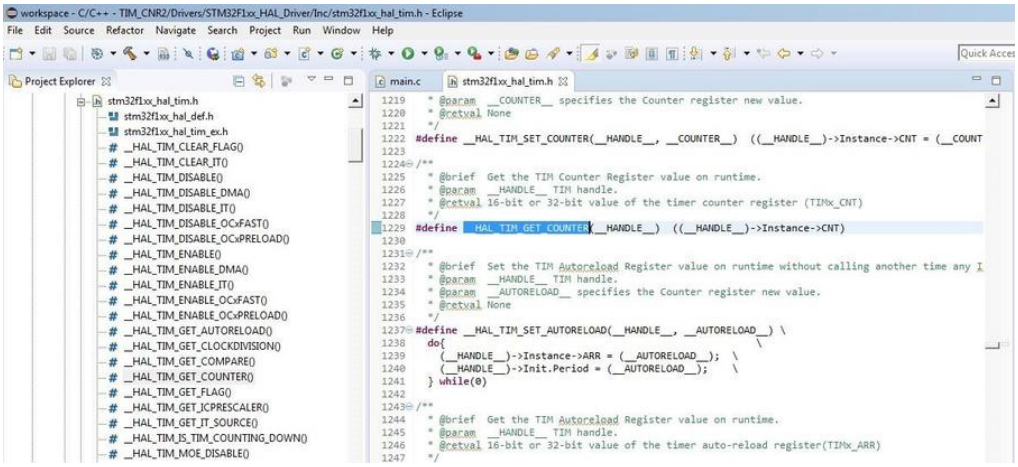

Figure 7.23: HAL_TIM_GET_COUNTER and HAL_TIM_SET_COUNTER macros inside stm32f1xx_hal_tim.h file

The HAL_TIM_GET_COUNTER and HAL_TIM_SET_COUNTER macros inside the stm32f1xx_hal_tim.h file are shown in Figure 7.23. We will use these in the following example. Since we use the LCD library in this project, add the LCD folder (including lcd.h and lcd.c files) to

the Src folder as shown in Figure 7.24. Then right-click on the Src folder and select refresh as shown in Figure 7.25. The code before the main function of the main.c file is shown in Figure 7.26.

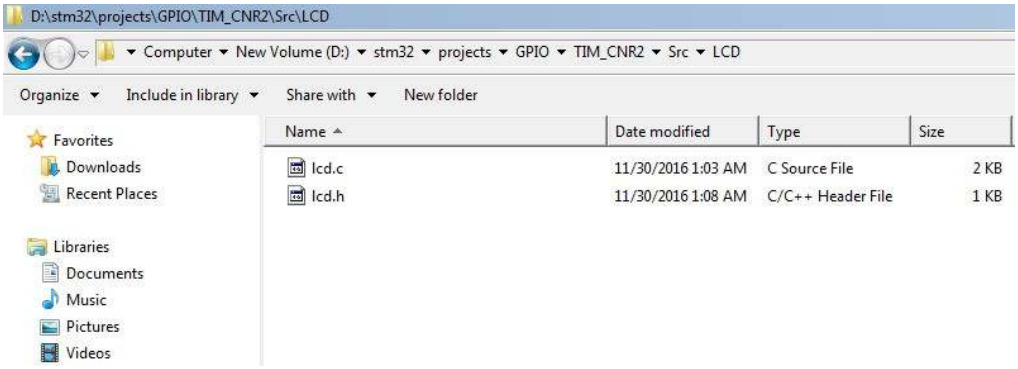

Figure 7.24: Adding an LCD folder to the Src folder of the project

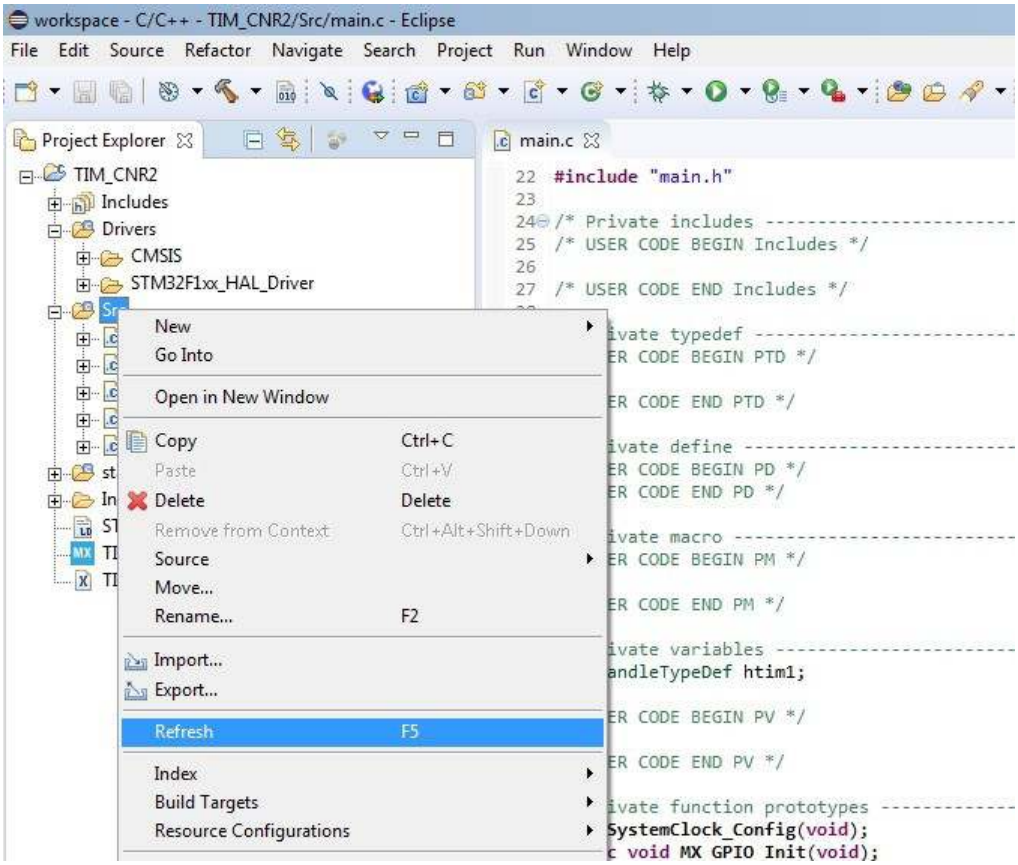

Figure 7.25: Refreshing Src folder of the project

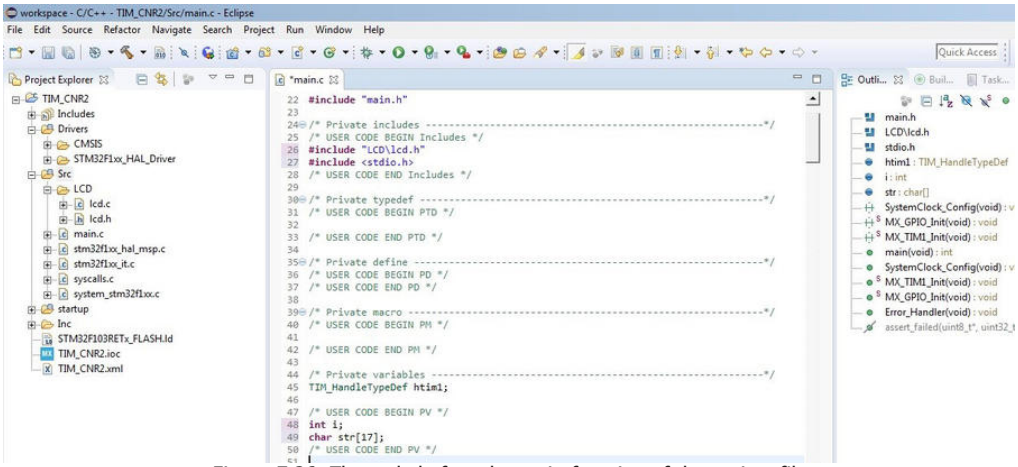

Figure 7.26: The code before the main function of the main.c file

The initial structures for the LCD and TIM1 functions and using HAL_TIM_SET_COUNTER and HAL_TIM_GET_COUNTER macros before and inside while loop is shown in Figure 7.27. We then right-click on the project and select 'build project' to compile the project and generate the hex code for programming the microcontroller.

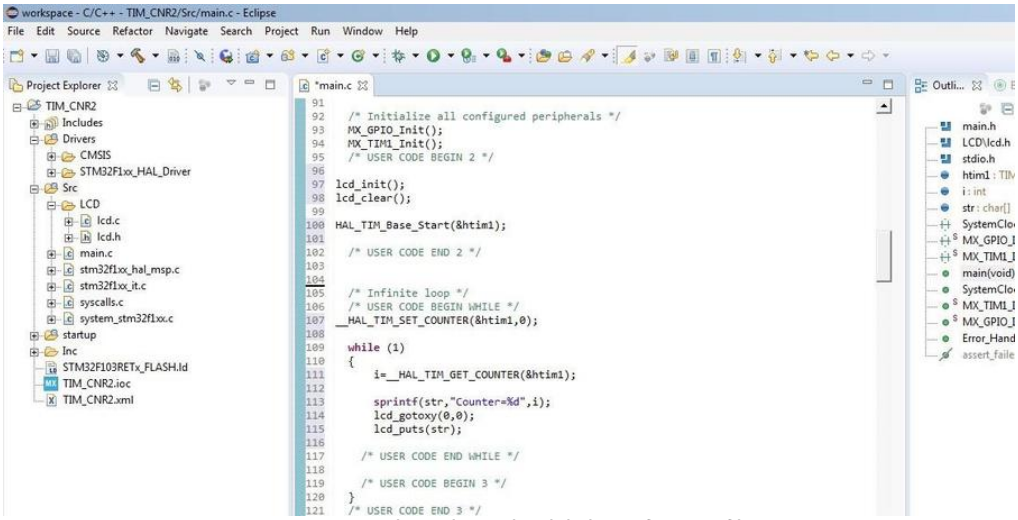

Figure 7.27: The code inside while loop of main.c file

7.4 • Summary

All timing scheduling and delays without stopping the program are managed by a timer that is operated in parallel with the processor. A timer is a counter which can count the regular pulses of an oscillator and generate regular times. This counter can count the external pulses which are applied to the microcontroller. In this chapter, the timer and counter project settings were explained in detail.

Chapter 8 ● Pulse Width Modulation (PWM) in STM32 Microcontrollers

8.1 ● Introduction

Pulse Width Modulation (PWM) is an important feature of microcontrollers and is used for data transmission. It can also be used for controlling the power of devices by changing the pulse width of the waveform. The on-time duration of a waveform is called the duty cycle which is the ratio of on-time duration to the period of the signal. It is usually determined by percentage. PWM wave is generated in timer units which can produce PWM signals. The STM32F103RET6 microcontroller has 8 timer units of which 6 units can generate PWM signals. Each Timer can produce 4 PWM signals with a 16-bit resolution.

8.2 ● PWM Project Settings

In the following example, we are going to enable two Timers (TIM1 and TIM4) and display them on an oscilloscope with different duty cycles. From the Timers section and TIM1 subsection, select the clock Source as the internal clock and channel1 as PWM Generation CH1. From parameter Settings, undertake settings such as Prescaler (0) so that its maximum for 16-Bit value is 65535, Counter Mode (Up), Counter Period (255) as shown in Figure 8.1. The frequency of the PWM signal is calculated as follows:

$$PWM\ signal\ frequency = \frac{input\ frequency\ to\ Timer\ unit}{(1 + Prescaler) \times Counter\ Period}$$

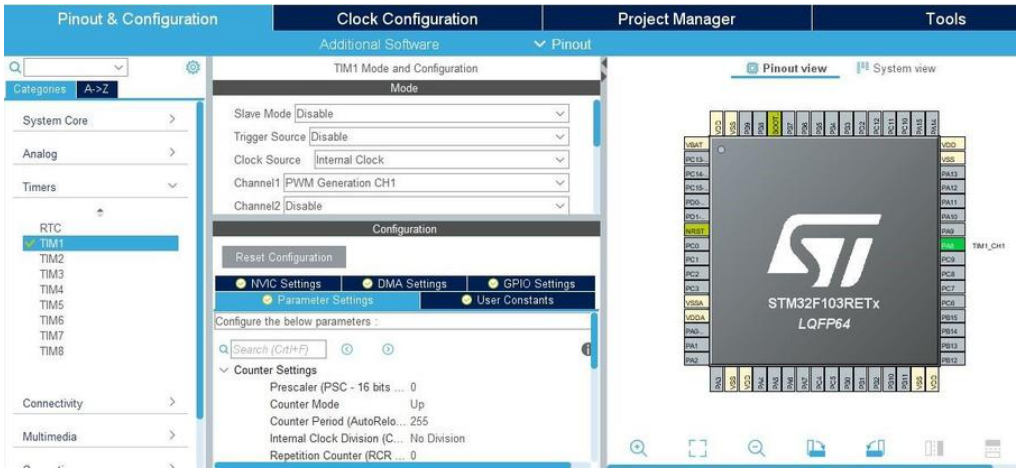

Figure 8.1: TIM1 settings for PWM signal generation

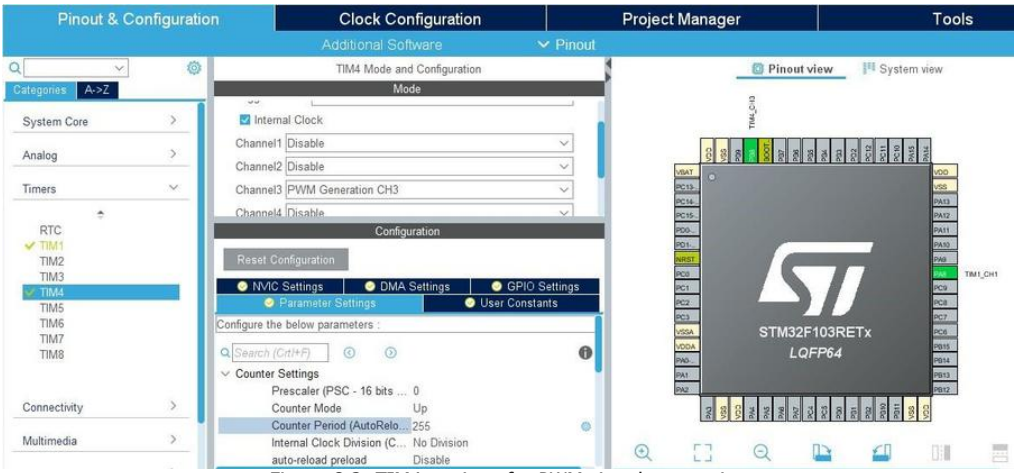

Figure 8.2: TIM4 settings for PWM signal generation

Since the APB1 and APB2 timer clock frequencies are 8 MHz, the frequency of PWM signals will be 31.25 kHz. Similarly, from the timers section and TIM4 subsection, select clock source as the internal clock and Channel3 as PWM Generation CH3. From parameter settings, undertake settings such as Prescaler (0) so that the maximum for 16-Bit value is 65535, Counter Mode (Up), Counter Period (255) as shown in Figure 8.2. Also, enable the pins related to the external crystal resonator as illustrated in Figure 8.3.

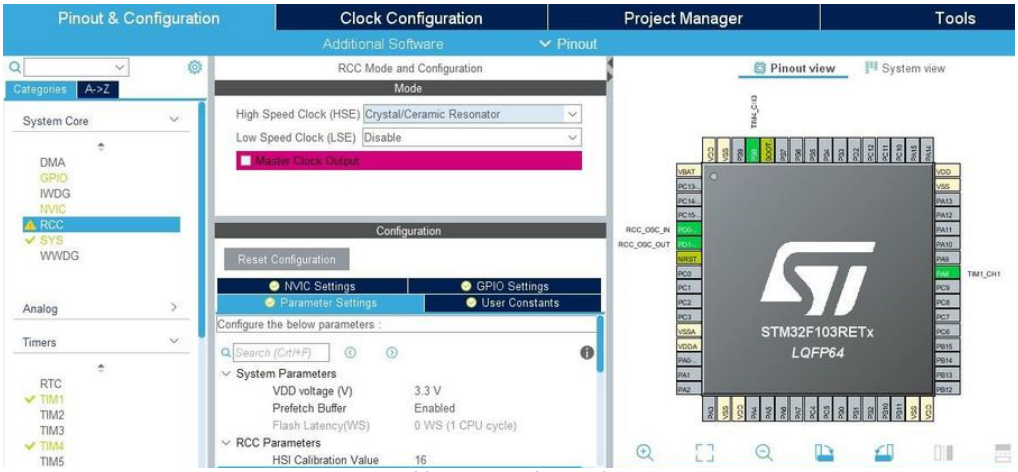

Figure 8.3: Enabling external crystal resonator pins

Finally, from the Project Manager section, we can generate code in the SW4STM32 software environment as illustrated in Figure 8.4. The body of the MX_TIM1_Init function is shown in Figure 8.5. The body of the MX_TIM4_Init function is depicted in Figure 8.6.

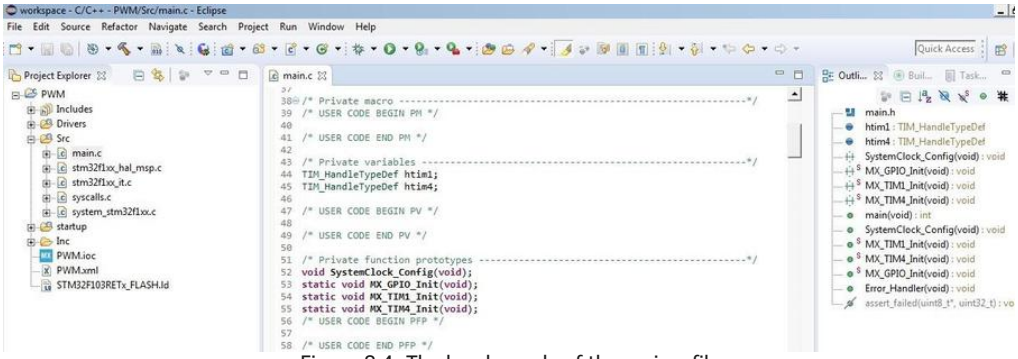

Figure 8.4: The header code of the main.c file

```
  * @param None
  * @retval None
  */
static void MX_TIM1_Init(void)
{

  /* USER CODE BEGIN TIM1_Init 0 */

  /* USER CODE END TIM1_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};

  /* USER CODE BEGIN TIM1_Init 1 */

  /* USER CODE END TIM1_Init 1 */
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 0;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 255;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
```

Figure 8.5: The body of function MX_TIM1_Init

```
  * @brief TIM4 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM4_Init(void)
{

  /* USER CODE BEGIN TIM4_Init 0 */

  /* USER CODE END TIM4_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};

  /* USER CODE BEGIN TIM4_Init 1 */

  /* USER CODE END TIM4_Init 1 */
  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 255;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
```

Figure 8.6: The body of function MX_TIM4_Init

The Timer functions are inside the stm32f1xx_hal_tim.c file as shown in Figure 8.7. The HAL_TIM_PWM_Start function is used for enabling PWM generation in timer units. The __HAL_TIM_SET_COMPARE macro inside the stm32f1xx_hal_tim.h file is used for reaching the comparison value for PWM signal generation as illustrated in Figure 8.8. We will use this macro in the following example. The duty cycle of a PWM signal is calculated according to the following equation:

$$duty\ cycle\ of\ PWM\ signal = \frac{comparison\ value}{Counter\ Period} \times 100\%$$

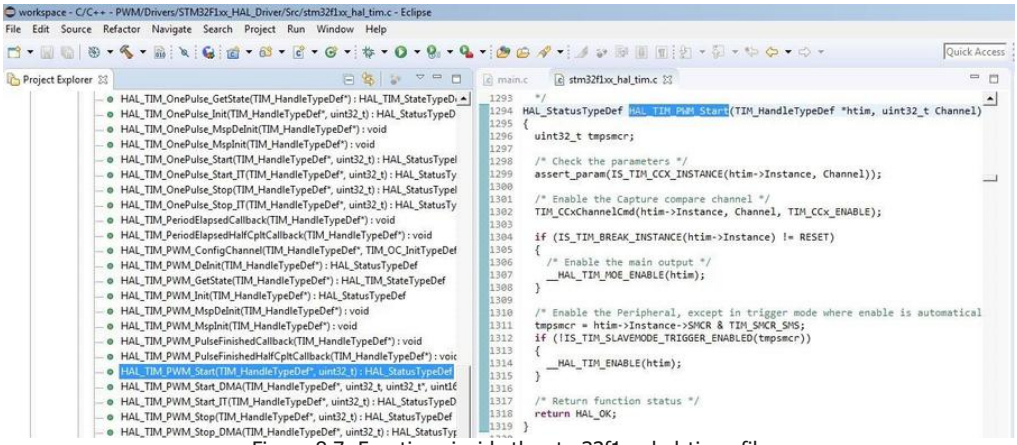

Figure 8.7: Functions inside the stm32f1xx_hal_tim.c file

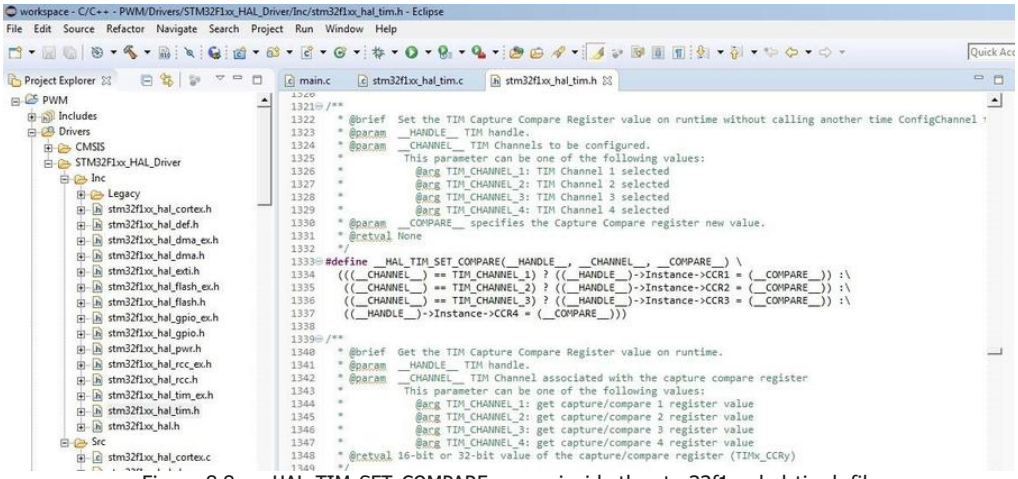

Figure 8.8: __HAL_TIM_SET_COMPARE macro inside the stm32f1xx_hal_tim.h file

As previously mentioned, we are going to enable two timer units as PWM outputs and display them on an oscilloscope with two different duty cycles. The initializing codes for TIM1 and TIM4 as well as the codes inside the while loop are illustrated in Figure 8.9.

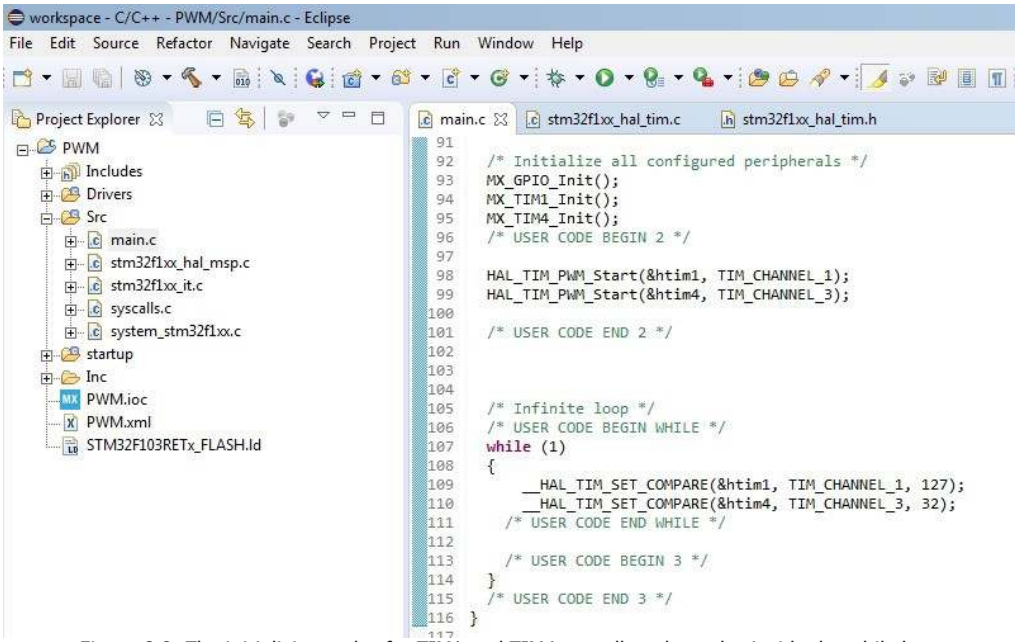

Figure 8.9: The initializing codes for TIM1 and TIM4 as well as the codes inside the while loop

Right-click on the project and select 'Build Project' to compile and generate the hex code for programming the microcontroller.

8.3 • Summary

PWM wave is generated in Timer units which can produce PWM signals. In this chapter, the PWM project settings were explained in detail.

Chapter 9 ● Real-Time Clock (RTC) in STM32 Microcontrollers

9.1 ● Introduction

The Real-Time Clock (RTC) is one of the intelligent system design requirements for accessing real-time and date. RTC chips are also available separately. However, most ARM microcontrollers include an RTC section. The clock of an RTC unit is supplied from an oscillator with a frequency of 32.768 kHz which is provided by a quartz crystal part called the 'time crystal'. RTC units usually have a backup battery that restores the time and date when the main power supply is disconnected. In an RTC unit, the frequency of 32.768 kHz (that is 2 to the power of 15) is pre-scaled to 32768 to generate the 1 Hz frequency. This frequency is then used as a counting basis in RTC units for producing the second, minute, hour, and date.

9.2 ● RTC Project Settings

In the following example, we are going to enable the RTC unit with associated pins and set the second, minute, and hour as well as the date and display them on an LCD. Moreover, we want to generate pulses with a 1 Hz frequency on the RTC output pin. For this purpose, according to Figure 9.1 from the System Core and RCC subsection, we set the Low-Speed Clock (LSE) as the Crystal/Ceramic Resonator to enable the pins related to the crystal (with the frequency of 32.768 kHz) connection to the microcontroller.

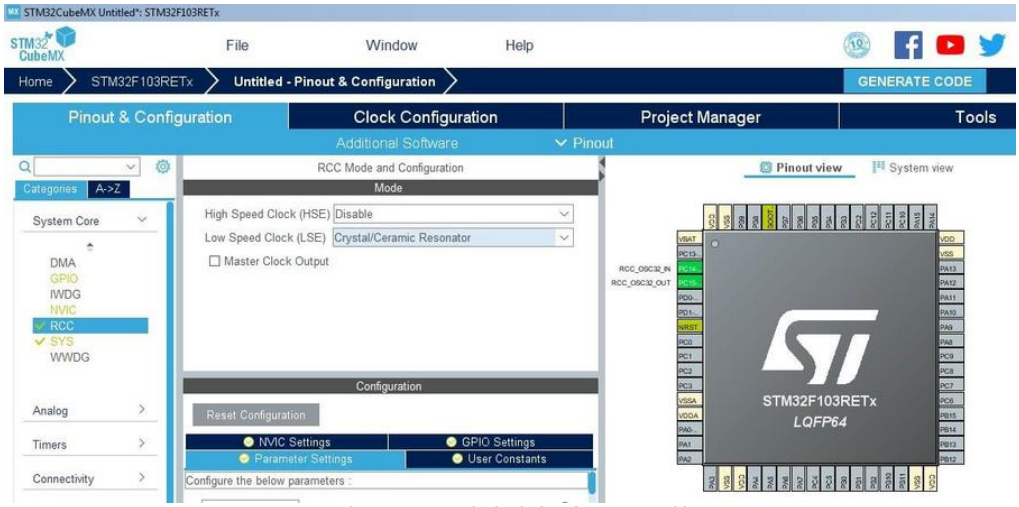

Figure 9.1: Setting the Low-Speed Clock (LSE) as Crystal/Ceramic Resonator

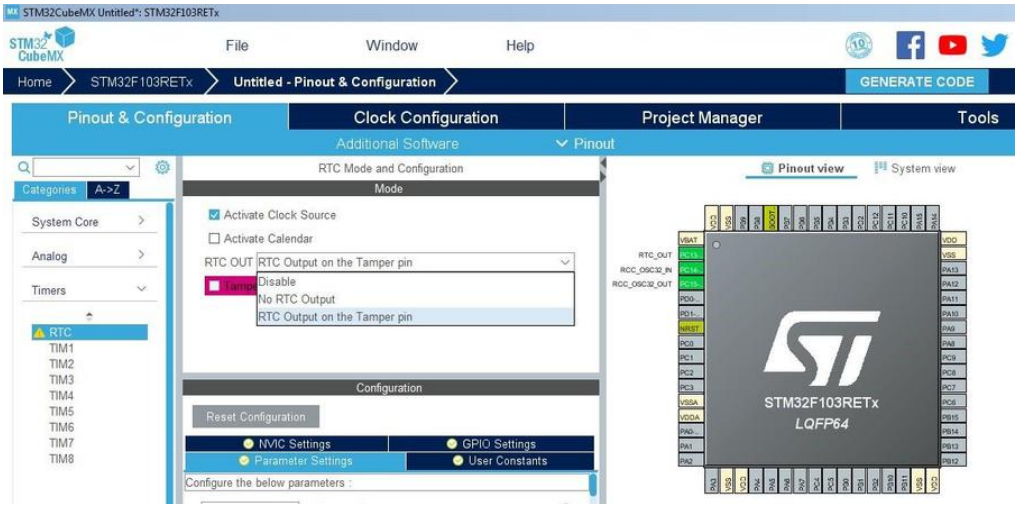

Figure 9.2: Setting the RTC OUT as RTC Output on the Tamper pin

From the timers section and RTC subsection, we then set RTC OUT as the RTC Output on the tamper pin to enable the tamper pin of the RTC unit as shown in Figure 9.2. We also activate the pins related to driving LCD as GPIO_Output and enable the external crystal resonator as depicted in Figure 9.3.

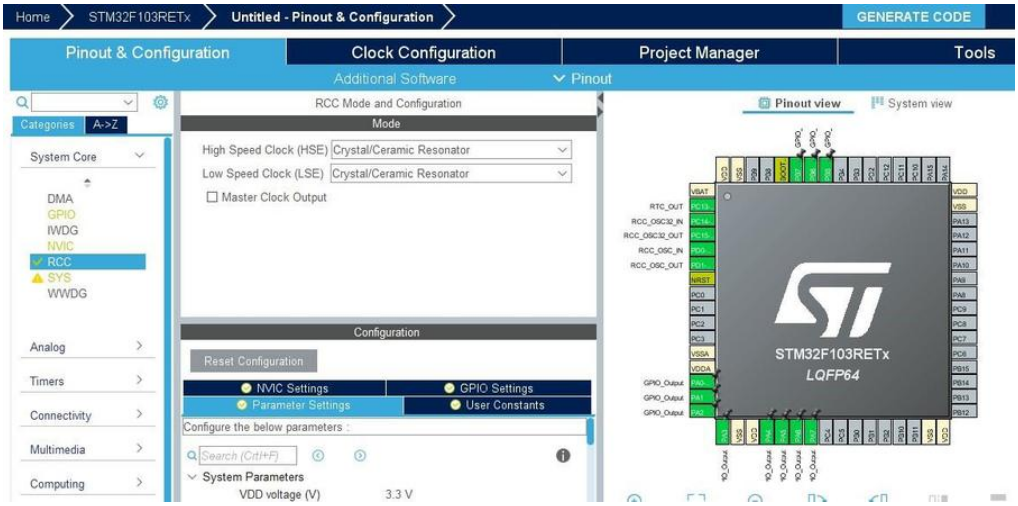

Figure 9.3: Enabling pins related to driving LCD and the external crystal resonator

In the clock Configuration tab, select LSE for setting the RTC unit clock frequency as illustrated in Figure 9.4.

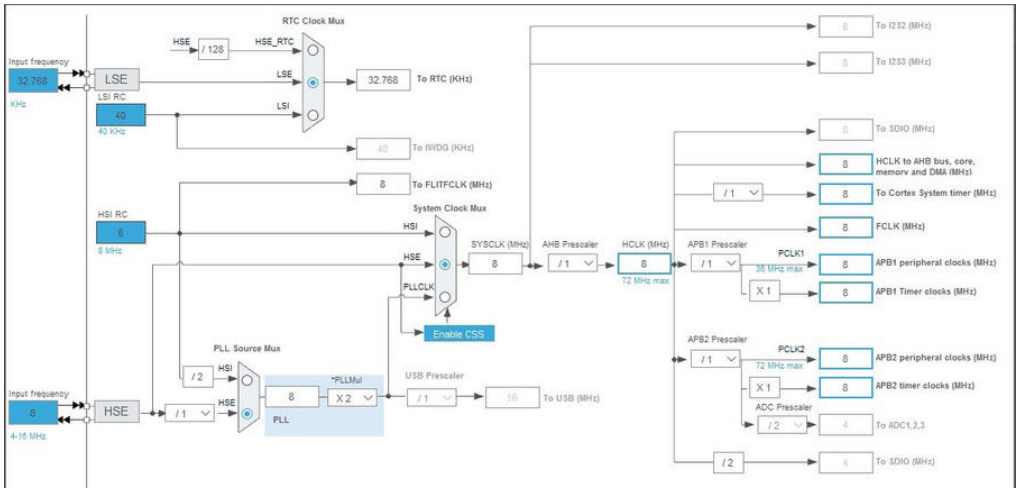

Figure 9.4: Selecting LSE for setting the RTC unit clock frequency

In the timers section and RTC subsection, open the parameter settings for the RTC unit. In the calendar time and date subsets, enter the initial time and date. In the general subset, enable auto predivider calculation. Set the asynchronous predivider value as automatic Predivider calculation Enabled and Output as the second pulse signal on the TAMPER pin as shown in Figure 9.5.

Figure 9.5: Parameter Settings for RTC unit

Also, from the timers and RTC subsection in the NVIC Settings tab, we can enable RTC global interrupt for every second (clock frequency of 1 Hz in the RTC unit) as shown in Figure 9.6.

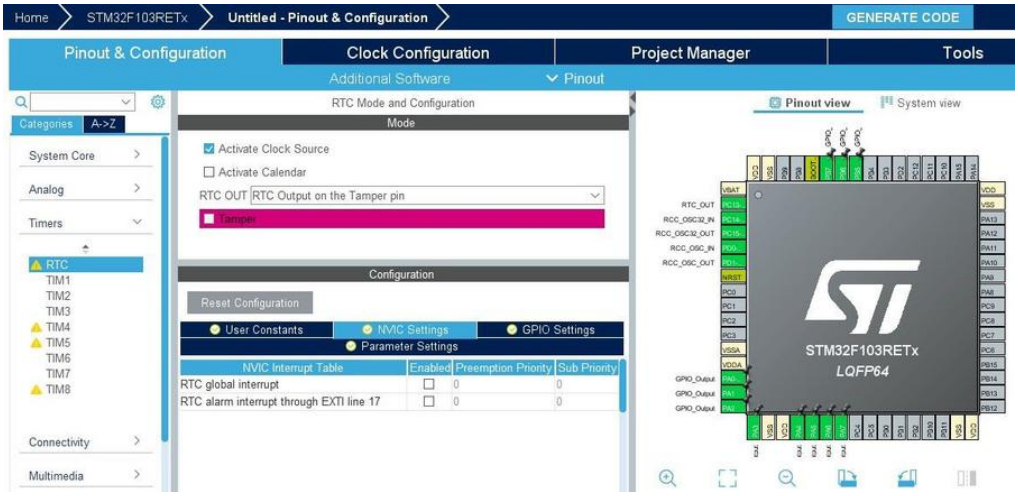

Figure 9.6: Enabling RTC global interrupt

We also select RTC from the timers section and activate calendar as illustrated in Figure 9.7. From the project manager tab, select SW4STM32 as the compiler and generate the code. The header code of the main.c file is shown in Figure 9.8

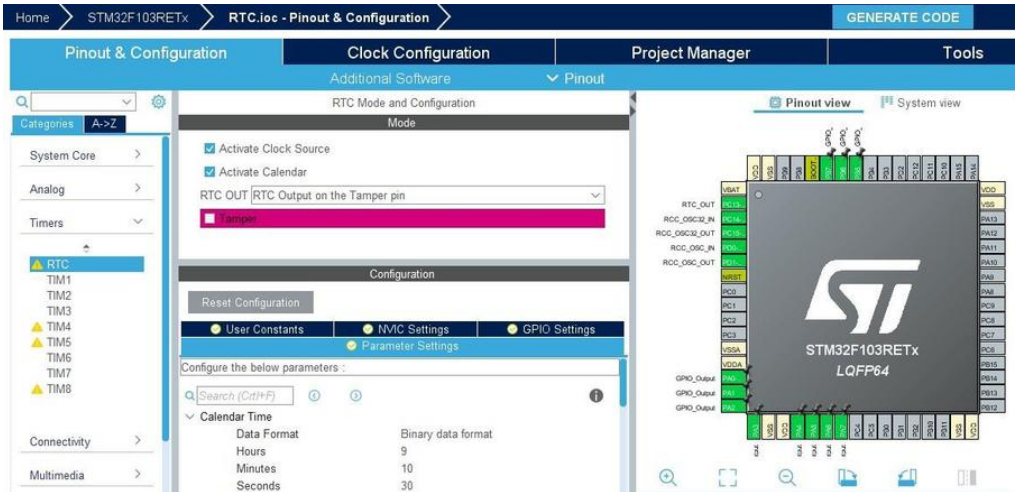

Figure 9.7: Activating Calendar in the RTC subsection

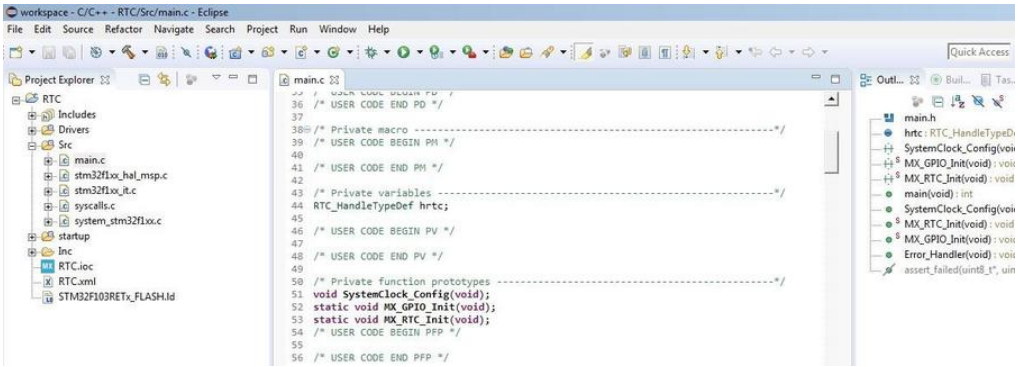

Figure 9.8: The header code of main.c

The body of the MX_RTC_Init function is illustrated in Figure 9.9.

```
static void MX_RTC_Init(void)
{

  /* USER CODE BEGIN RTC_Init 0 */

  /* USER CODE END RTC_Init 0 */

  RTC_TimeTypeDef sTime = {0};
  RTC_DateTypeDef DateToUpdate = {0};

  /* USER CODE BEGIN RTC_Init 1 */

  /* USER CODE END RTC_Init 1 */
  /** Initialize RTC Only
  */
  hrtc.Instance = RTC;
  hrtc.Init.AsynchPrediv = RTC_AUTO_1_SECOND;
  hrtc.Init.OutPut = RTC_OUTPUTSOURCE_SECOND;
  if (HAL_RTC_Init(&hrtc) != HAL_OK)
  {
    Error_Handler();
  }

  /* USER CODE BEGIN Check_RTC_BKUP */

  /* USER CODE END Check_RTC_BKUP */

  /** Initialize RTC and set the Time and Date
  */
  sTime.Hours = 9;
  sTime.Minutes = 10;
  sTime.Seconds = 30;

  if (HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BIN) != HAL_OK)
  {
    Error_Handler();
  }
  DateToUpdate.WeekDay = RTC_WEEKDAY_MONDAY;
  DateToUpdate.Month = RTC_MONTH_JANUARY;
  DateToUpdate.Date = 1;
  DateToUpdate.Year = 20;

  if (HAL_RTC_SetDate(&hrtc, &DateToUpdate, RTC_FORMAT_BIN) != HAL_OK)
```

Figure 9.9: The body of MX_RTC_Init function

Functions related to the RTC unit are inside the stm32f1xx_hal_rtc.c file as shown in Figure 9.10.

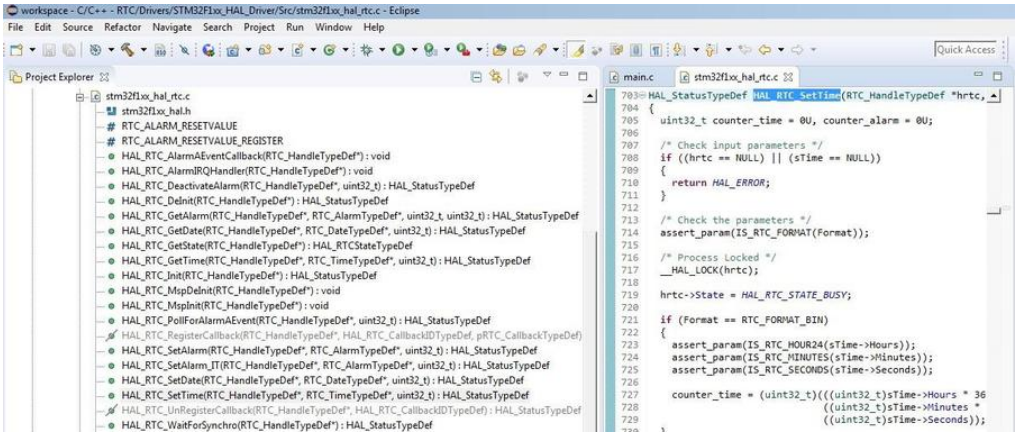

Figure 9.10: RTC unit functions in the stm32f1xx_hal_rtc.c file

As previously mentioned, we are going to set the time and date in the RTC unit and display them on an LCD. We also want to generate a pulse with a frequency of 1 Hz on the TAMPER pin. The header code of the main.c file is shown in Figure 9.11. The RTC_TimeTypeDef and RTC_DateTypeDef structure bodies are illustrated in Figure 9.12 and Figure 9.13 respectively. The code for initializing the LCD and RTC unit as well as the code inside the while loop are depicted in Figure 9.14.

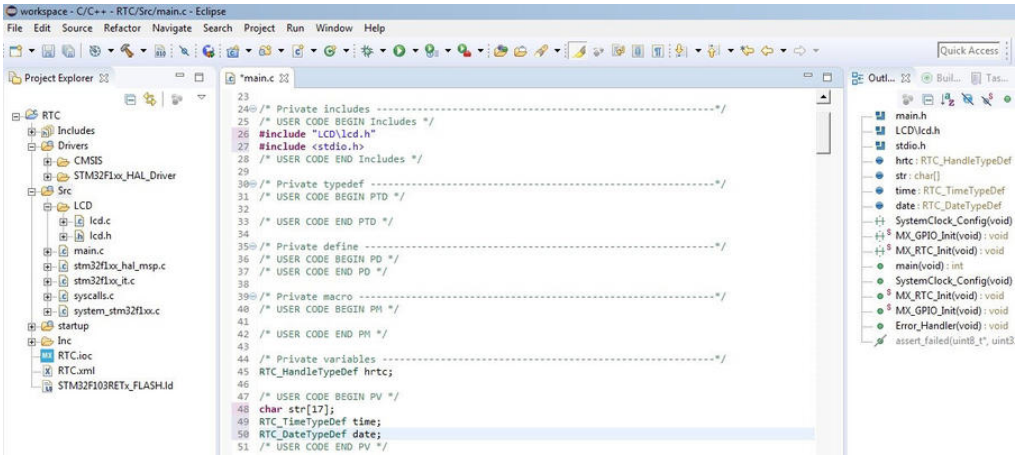

Figure 9.11: The header code of main.c

```
/**
  * @brief  RTC Time structure definition
  */
typedef struct
{
  uint8_t Hours;            /*!< Specifies the RTC Time Hour.
                                 This parameter must be a number between Min_Data = 0 and Max_Data = 23 */

  uint8_t Minutes;          /*!< Specifies the RTC Time Minutes.
                                 This parameter must be a number between Min_Data = 0 and Max_Data = 59 */

  uint8_t Seconds;          /*!< Specifies the RTC Time Seconds.
                                 This parameter must be a number between Min_Data = 0 and Max_Data = 59 */

} RTC_TimeTypeDef;
```

Figure 9.12: The RTC_TimeTypeDef structure body

```
/**
  * @brief  RTC Date structure definition
  */
typedef struct
{
  uint8_t WeekDay;  /*!< Specifies the RTC Date WeekDay (not necessary for HAL_RTC_SetDate).
                         This parameter can be a value of @ref RTC_WeekDay_Definitions */

  uint8_t Month;    /*!< Specifies the RTC Date Month (in BCD format).
                         This parameter can be a value of @ref RTC_Month_Date_Definitions */

  uint8_t Date;     /*!< Specifies the RTC Date.
                         This parameter must be a number between Min_Data = 1 and Max_Data = 31 */

  uint8_t Year;     /*!< Specifies the RTC Date Year.
                         This parameter must be a number between Min_Data = 0 and Max_Data = 99 */

} RTC_DateTypeDef;
```

Figure 9.13: The RTC_DateTypeDef structure body

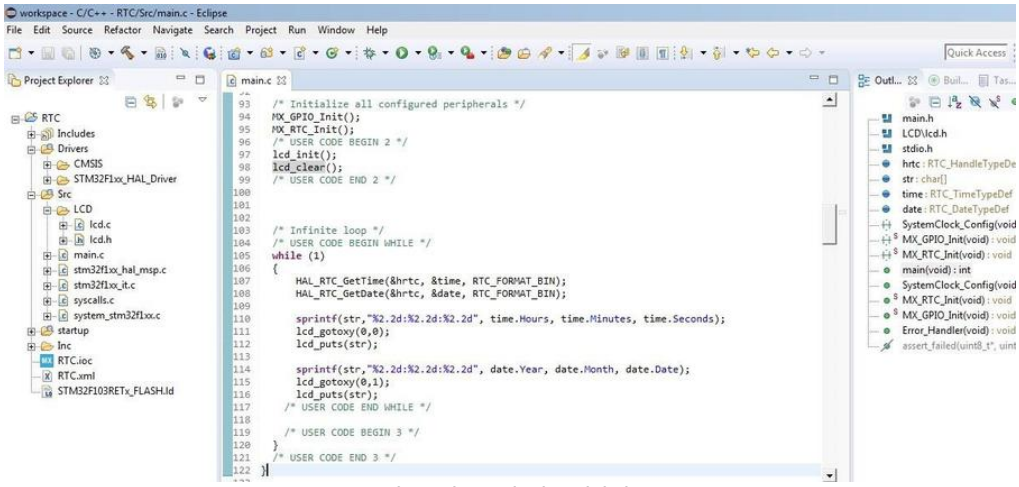

Figure 9.14: The code inside the while loop in main.c

We can then compile the project by right-clicking the project name and selecting the Build Project icon. After successfully compiling the project, the generated hex file for programming the microcontroller will be inside the Debug folder as shown in Figures 9.15 and 9.16.

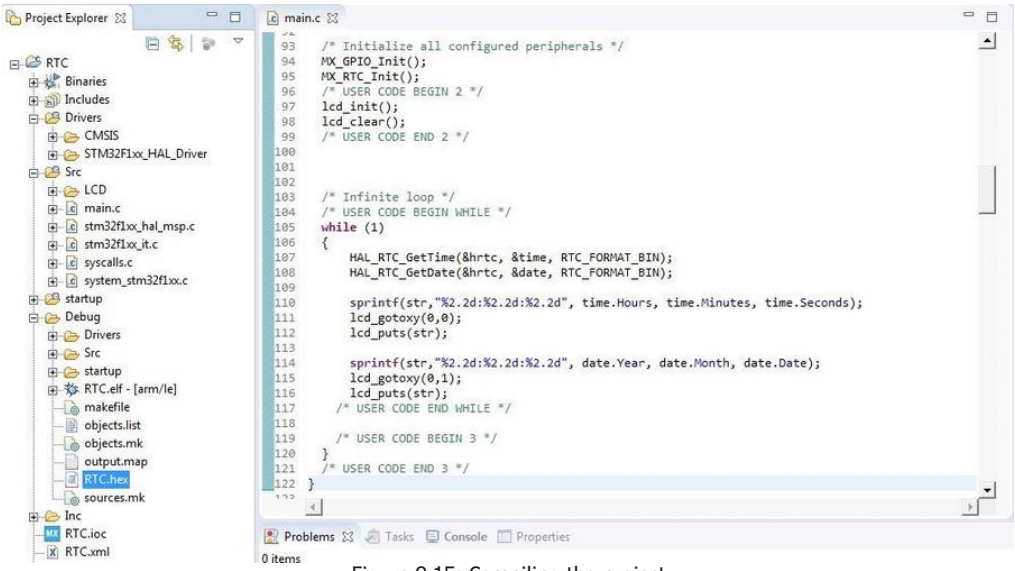

Figure 9.15: Compiling the project

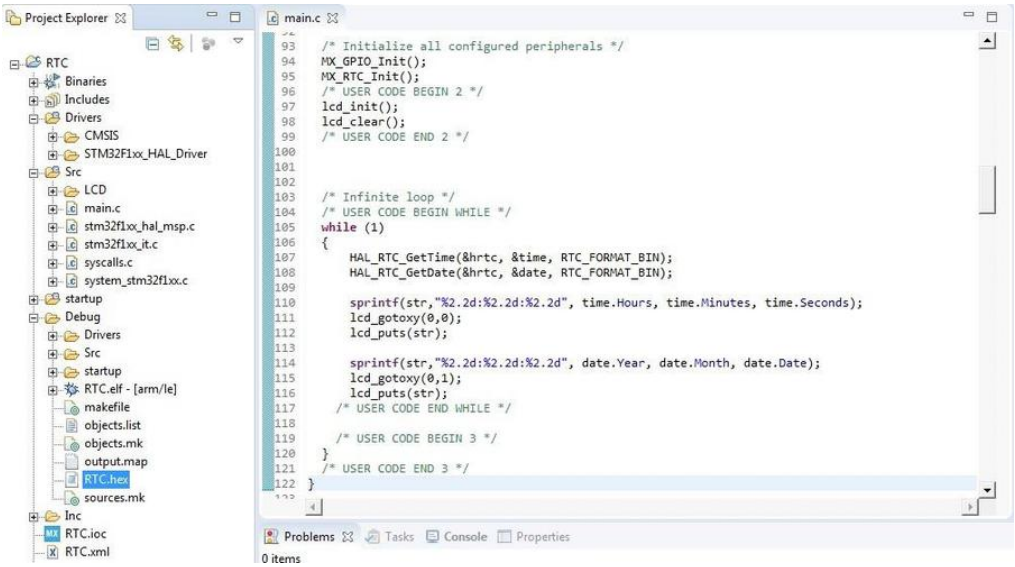

Figure 9.16: The generated hex file inside the Debug folder

9.3 ● Summary

The Real-Time Clock (RTC) is one of the intelligent system design requirements for accessing real-time and date. In this chapter, the RTC project settings were discussed in detail.

Chapter 10 • Serial Communication in STM32 Microcontrollers

10.1 • Introduction

Serial communication is one of the most important features of a microcontroller communicating with other devices. One of the main serial communication protocols is the Universal Asynchronous Receiver/Transmitter (UART) which is called asynchronous communication without a clock. This communication protocol constitutes two routes (RX and TX) which are shown in Figure 10.1.

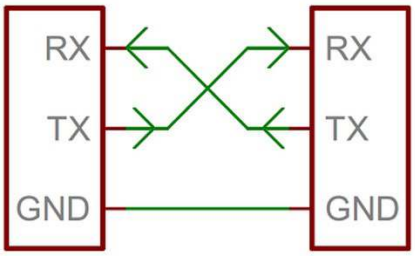

Figure 10.1: UART serial communication in the TTL level

Serial communication can also be undertaken in synchronous mode. In this case, it is called Universal Synchronous/Asynchronous Receiver/Transmitter (USART). The synchronous communication protocol includes a clock. In the following example, we are going to enable the USART1 unit in an asynchronous mode and send and receive data from a computer.

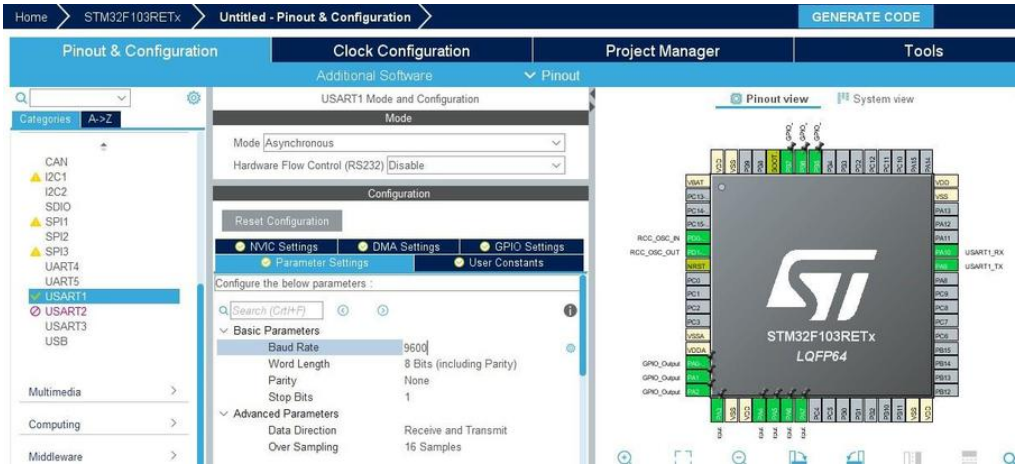

Figure 10.2: Enabling USART1 in asynchronous mode

We also want to display the characters received from a computer on an LCD.

10.2 ● Sending Data Project Settings

In accordance with Figure 10.2, in the connectivity section and USART1 subsection, set the mode to asynchronous and baud rate to 9600. Also activate the pins associated with driving the LCD and enable the external crystal resonator as depicted in Figure 10.3.

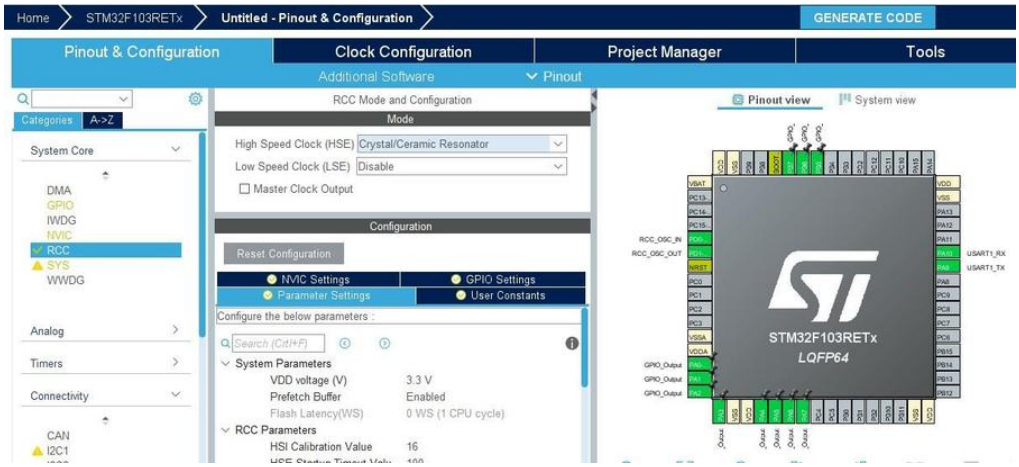

Figure 10.3: Activating pins for driving the LCD and enabling external crystal resonator

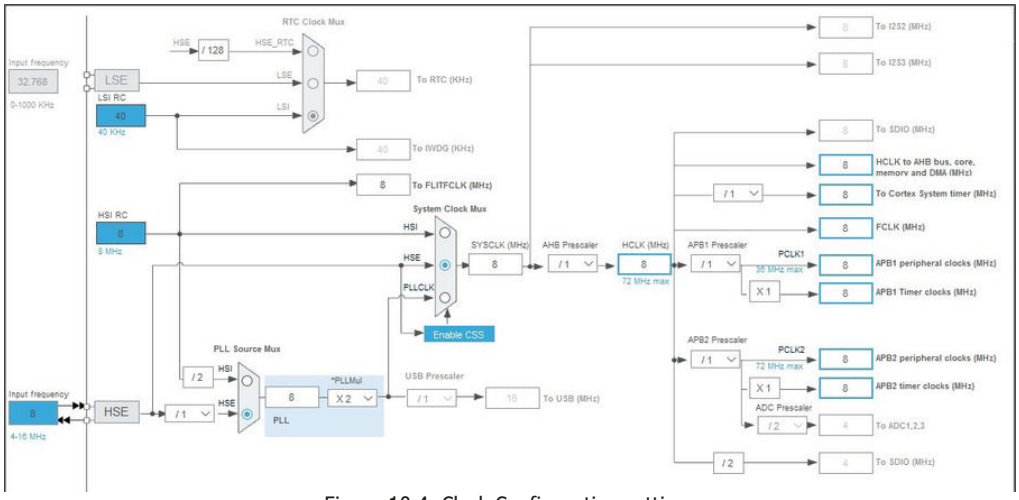

Figure 10.4: Clock Configuration settings

From the clock configuration section, set the APB2 peripheral clocks to 8 MHz with an enabling external clock resonator (HSE) as illustrated in Figure 10.4. Also, in the connectivity section, the USART1 subsection and the NVIC Settings tab, enable the USART1 global interrupt as demonstrated in Figure 10.5. The RX pin (which is input) is better set in Pull-up mode as depicted in Figure 10.6. We can then determine the USART1 global interrupt

preemption priority in the NVIC section as illustrated in Figure 10.7.

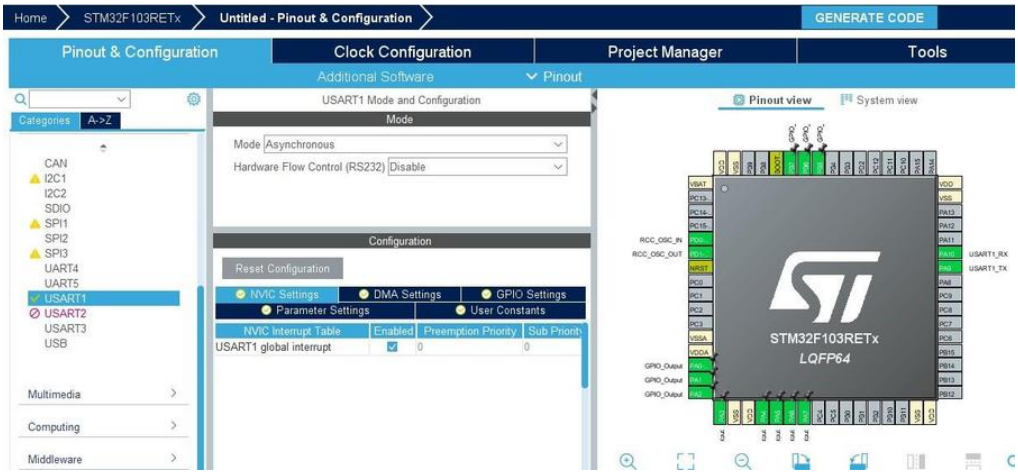

Figure 10.5: Enabling USART1 global interrupt

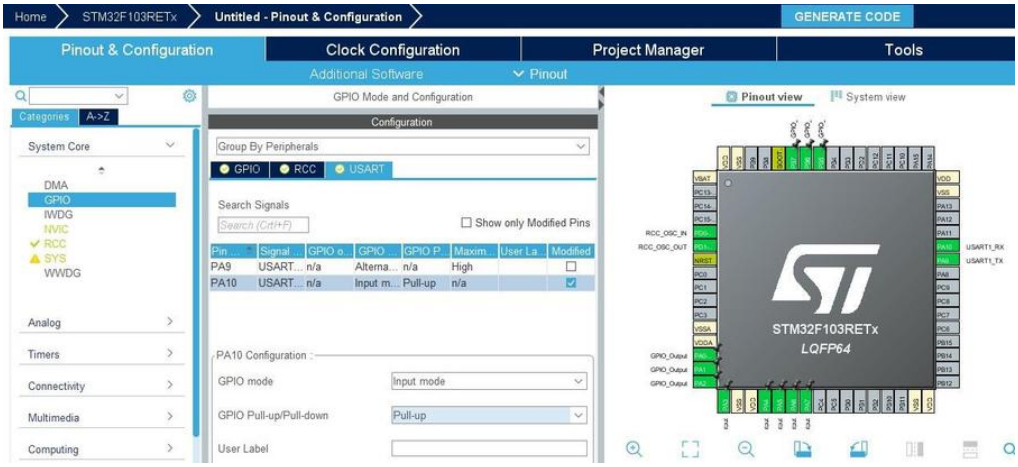

Figure 10.6: Setting the RX pin in Pull-up mode

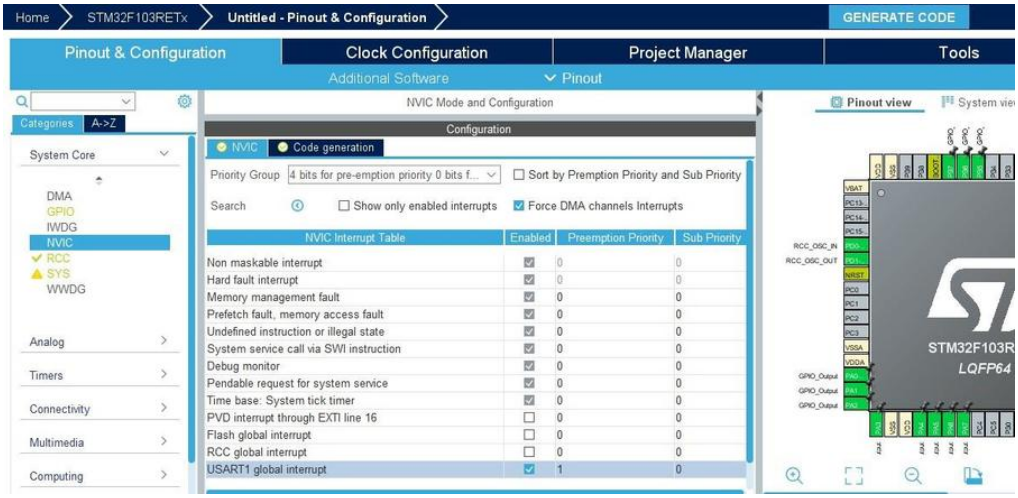

Figure 10.7: Choosing USART1 global interrupt Preemption Priority

After inputting the above settings, from the Project Manager tab, select 'SW4STM32' as the compiler and generate the code. The header code of the main.c file is depicted in Figure 10.8. The body of the MX_USART1_UART_Init function is demonstrated in Figure 10.9.

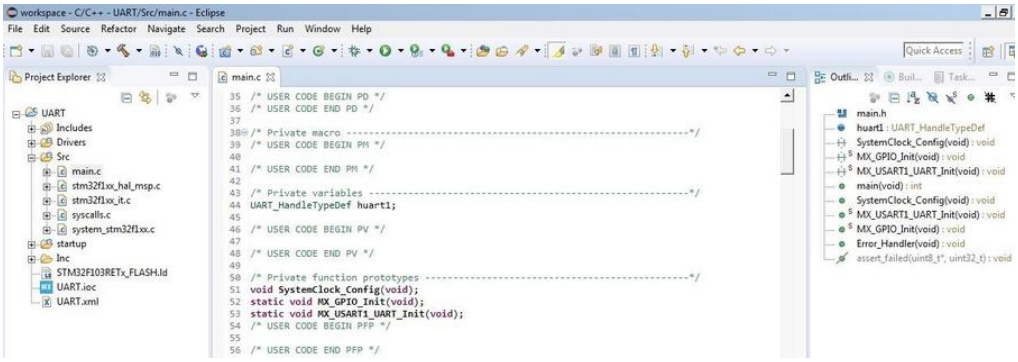

Figure 10.8: The header code of main.c

```
/**
  * @brief USART1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART1_UART_Init(void)
{

  /* USER CODE BEGIN USART1_Init 0 */

  /* USER CODE END USART1_Init 0 */

  /* USER CODE BEGIN USART1_Init 1 */

  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 9600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */

  /* USER CODE END USART1_Init 2 */

}
```

Figure 10.9: The header code of main.c

The functions relating to UART are inside the stm32f1xx_hal_uart.c file as shown in Figure 10.10.

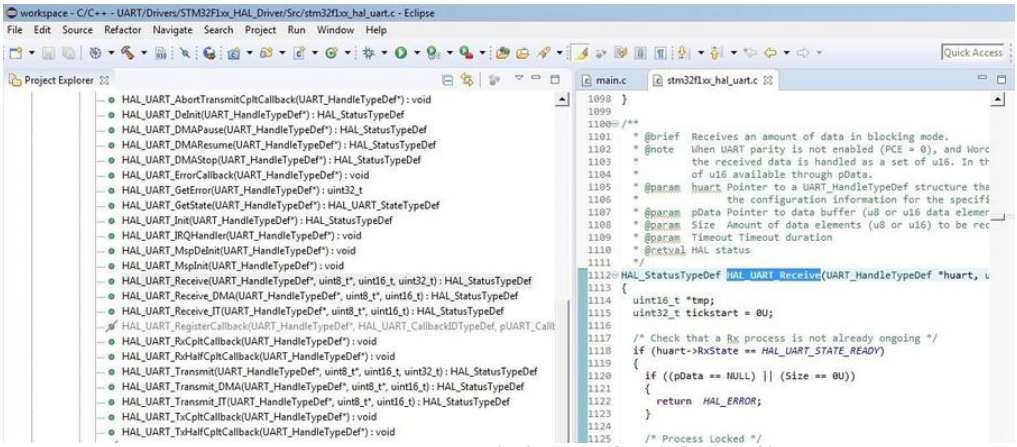

Figure 10.10: Functions inside the stm32f1xx_hal_uart.c file

For sending data from a microcontroller to a PC, we are going to add a unit to a variable every 500 ms and then make a string. The header code of the main.c file with new variables definitions is shown in Figure 10.11. The code before the main function is shown in Figure 10.12. Also, the code inside the while loop in the main function is shown in Figure 10.13.

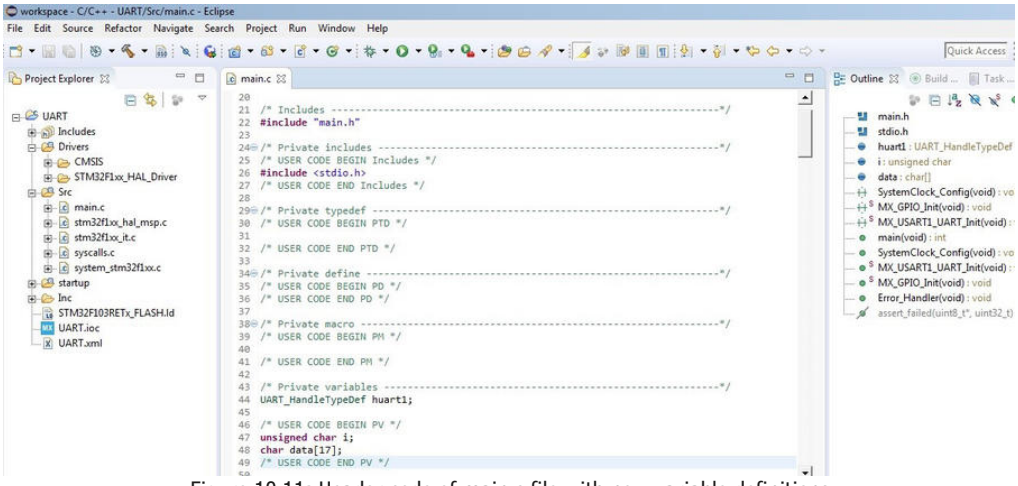

Figure 10.11: Header code of main.c file with new variable definitions

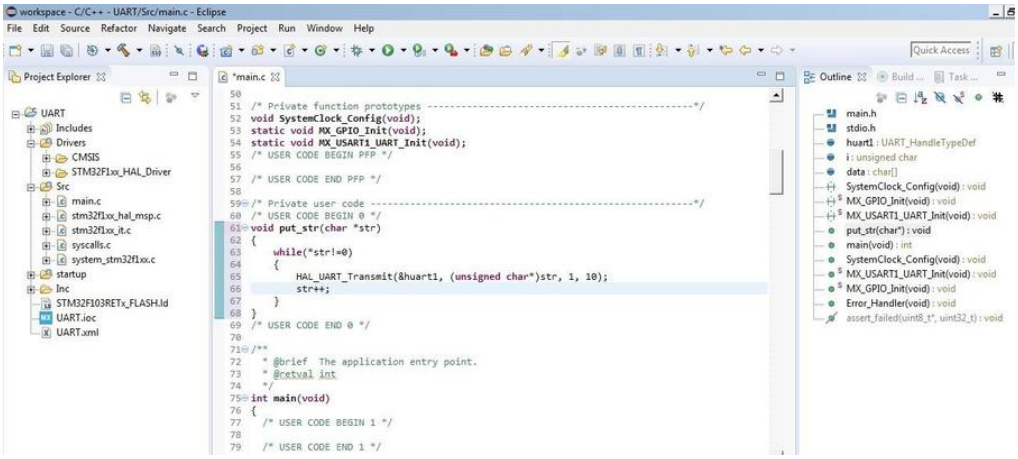

Figure 10.12: The code before the main function

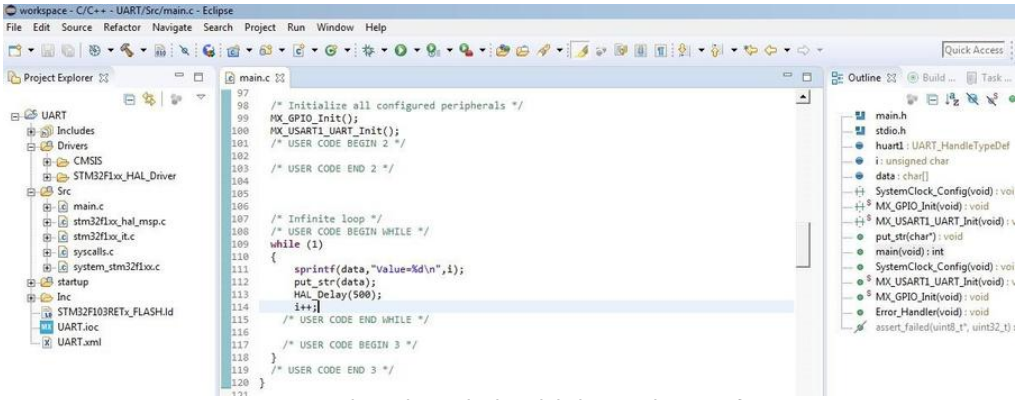

Figure 10.13: The code inside the while loop in the main function

We can then compile the project by right-clicking the project name and selecting the 'build project' icon. After successfully compiling the project, the hex file generated for programming the microcontroller will be inside the Debug folder.

10.3 • Receiving Data Project Settings

For receiving data on a microcontroller from a PC and displaying it on an LCD, consider the following example: The header code of main.c with the new variable definition is illustrated in Figure 10.14. Code inside the while loop in the main function is shown in Figure 10.15. The UART_HandleTypeDef struct definition is illustrated in Figure 10.16. The USART1 interrupt service function in stm32f1xx_it.c file is demonstrated in Figure 10.17.

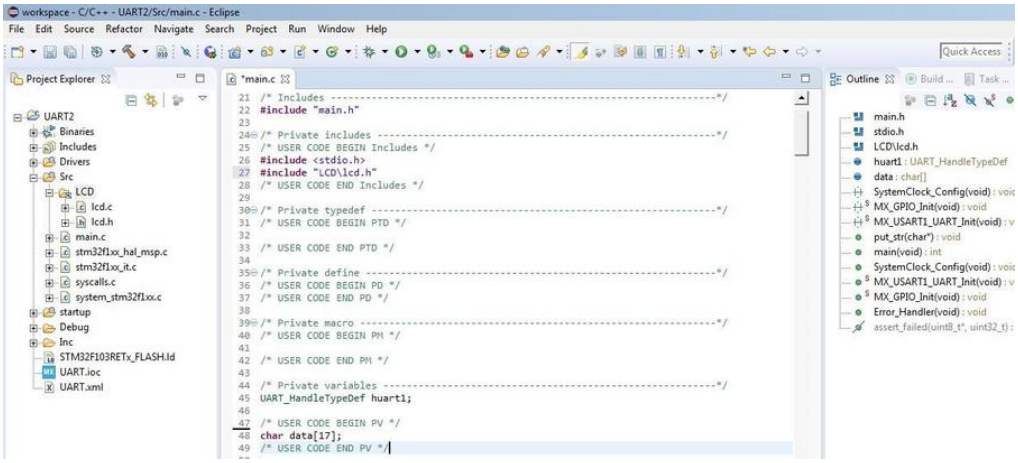

Figure 10.14: The header code of main.c file with a new variable definition

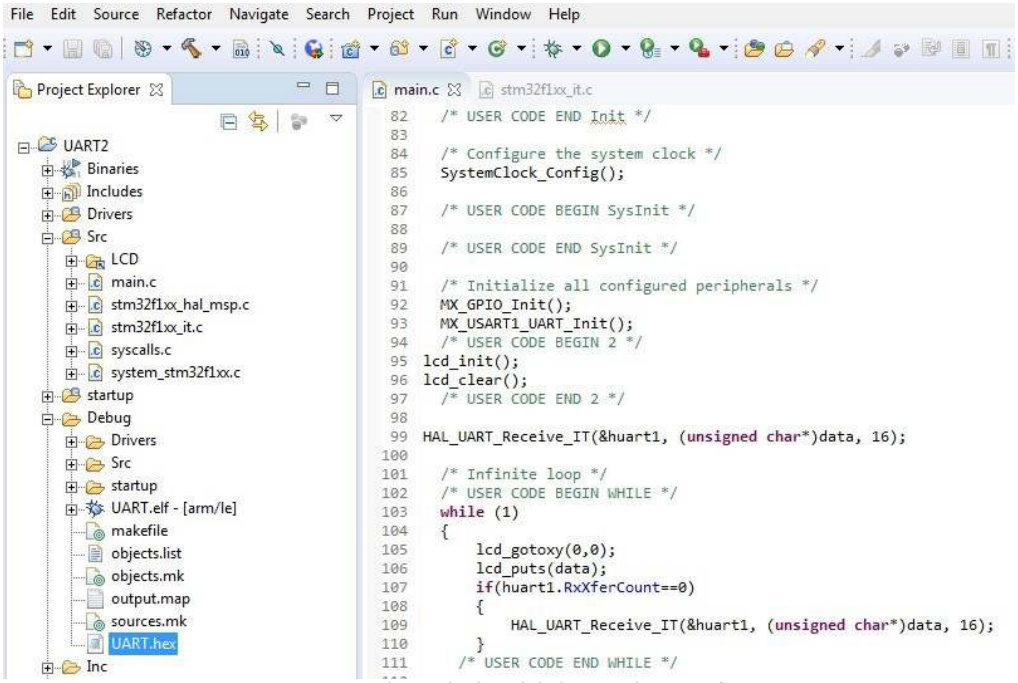

Figure 10.15: Code inside the while loop in the main function

Figure 10.16: UART_HandleTypeDef struct definition

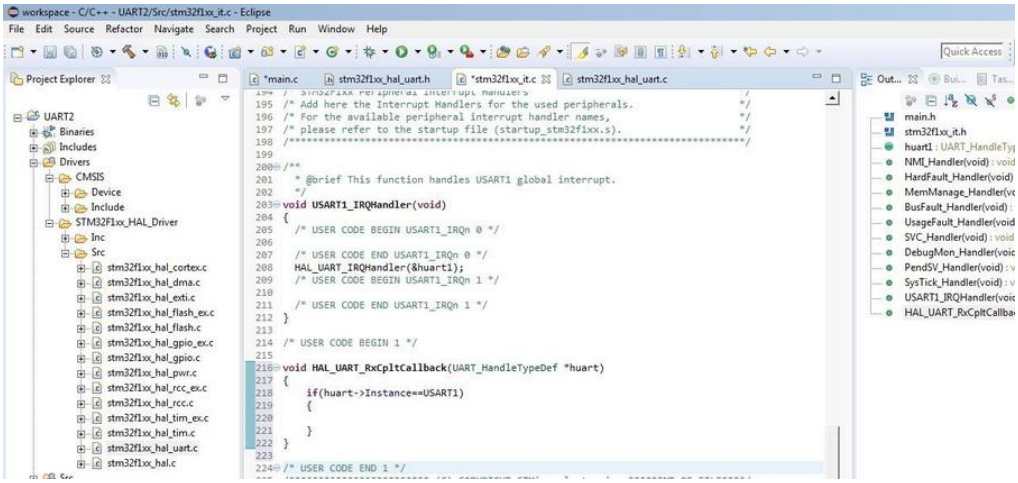

Figure 10.17: USART1 interrupt service function in stm32f1xx_it.c

The HAL_UART_RxCpltCallback function has been copied from stm32f1xx_hal_uart.c file inside stm32f1xx_it.c file as shown in Figure 10.17. Interrupt enabling is undertaken through the __HAL_UART_ENABLE_IT macro. This macro is inside the stm32f1xx_hal_uart.h file as follows:

```
#define__HAL_UART_ENABLE_IT(__HANDLE__,__INTERRUPT__)
```

Right-click the project and select 'Build Project' to compile the project and generate the hex code for programming the microcontroller.

10.4 ● Summary

Serial communication is one of the important aspects of communication between a microcontroller and other devices. In this chapter, serial communication protocol settings including sending and receiving data to and from a PC and displaying it on an LCD project were explained in detail.

Chapter 11 • Synchronous Peripheral Interface (SPI) in STM32 Microcontrollers

11.1 • Introduction

SPI protocol is a synchronous bidirectional serial communication interface. The interface is one of the most important and useful communication protocols in microcontrollers due to its fast speed. High-speed chips and devices tend to use this communication protocol. The synchronous interface includes a master unit and a slave unit or multiple slave units. The master usually generates the interface clock. Since this interface is synchronous, sending and receiving data is performed by the clock. The interface of a master and a slave using the SPI protocol is shown in Figure 11.1.

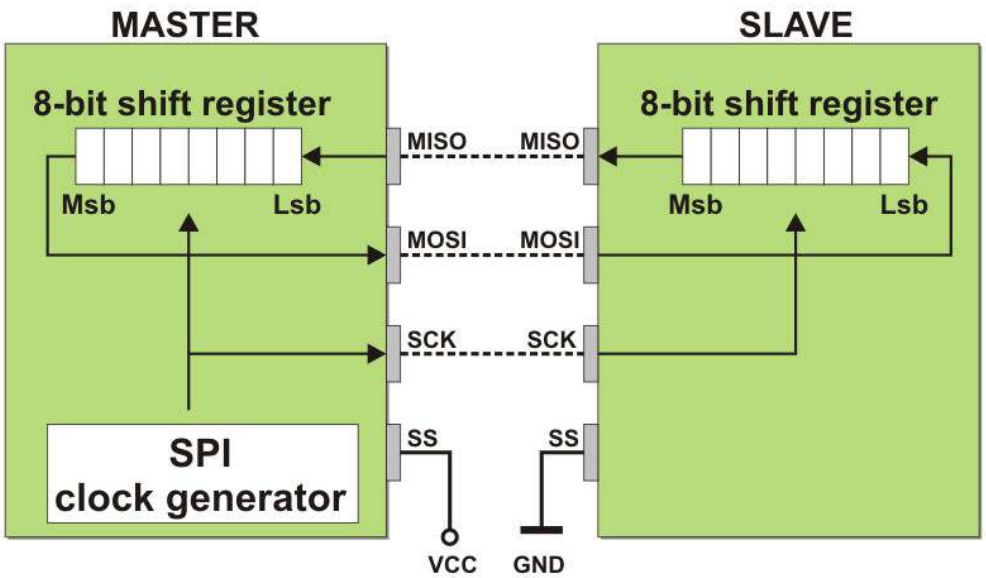

Figure 11.1: The interface of a master and a slave using the SPI protocol

The pins of an SPI protocol are as follows:

MOSI: Master Output Slave Input
MISO: Master Input Slave Output
SCK: Clock
SS: Slave Select

The block diagram of interfacing one master and multiple slaves is demonstrated in Figure 11.2.

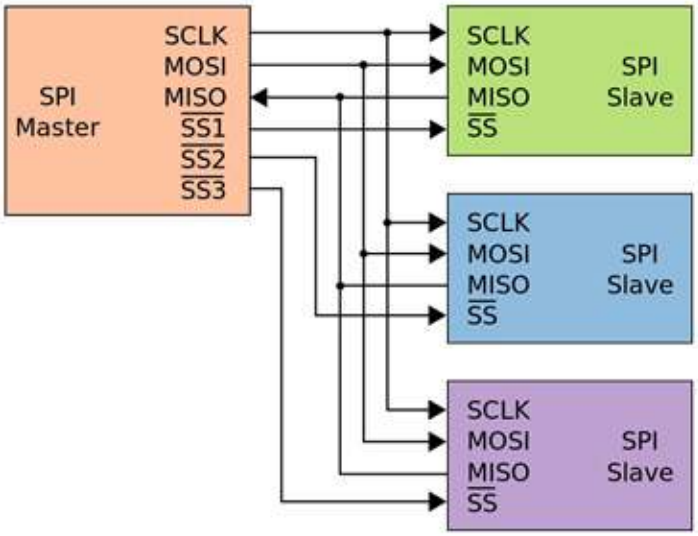

Figure 11.2: Interfacing one master and multiple slaves using SPI protocol

In the following example, we are going to use two microcontrollers as a master and slave. We will use the STM32F103RET6 microcontroller as a master and STM32F030F4P6 as a slave. The master microcontroller in the while loop of the program sends two characters: a and b, to the slave. If the character received on the slave microcontroller is A, an LED on a pin in the slave microcontroller turns on. If the character received on the slave microcontroller is B, an LED on a pin on the slave microcontroller will be off.

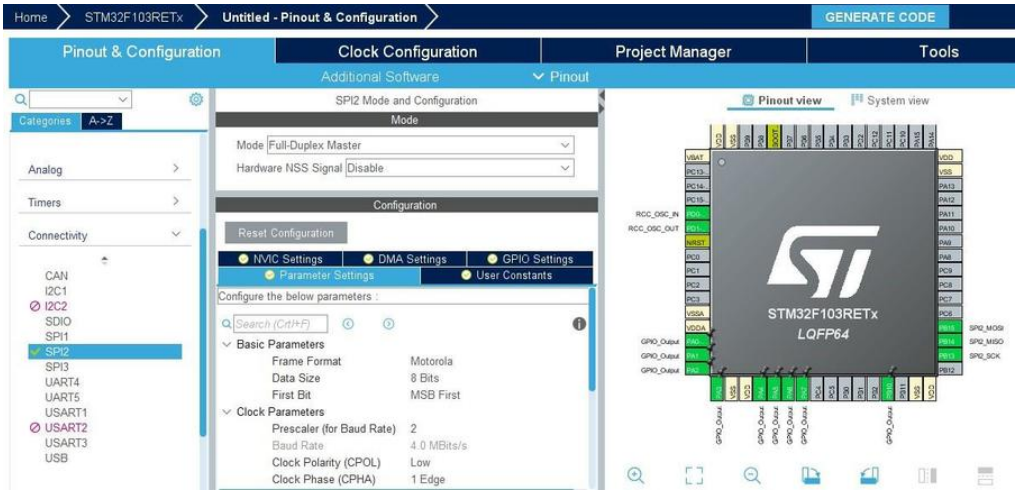

Figure 11.3: Enabling SPI2 for a master microcontroller

11.2 • SPI Settings for Master Microcontroller

Firstly, we consider the master microcontroller settings. From the connectivity section and SPI2 subsection, select the Mode as Full-Duplex Master. From the Parameter Settings tab, apply settings including data size, Prescaler, etc., as demonstrated in Figure 11.3. Also activate the driving of the LCD and external crystal resonator pins. From the clock configuration tab, set the APB1 bus clock to 8 MHz (SPI2 unit is connected to APB1 bus) with an enabling external crystal resonator (HSE) as illustrated in Figure 11.4.

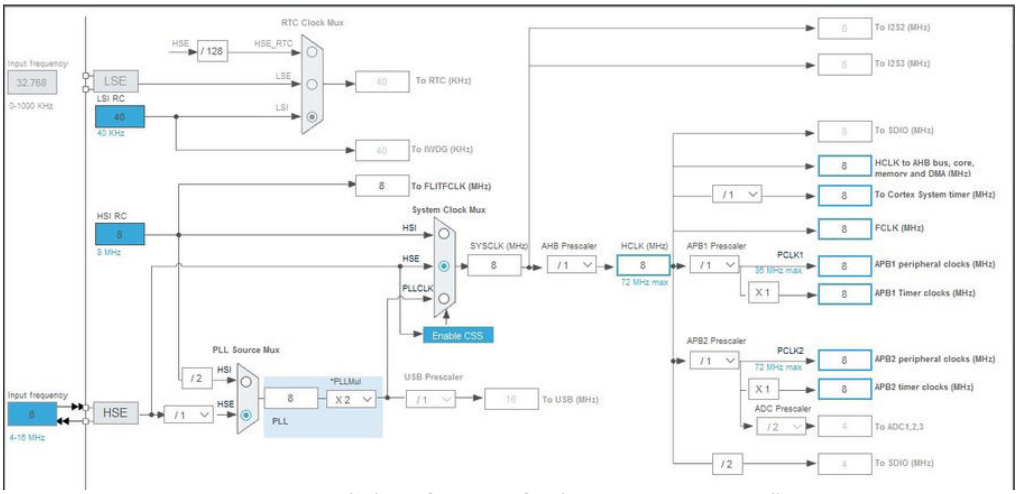

Figure 11.4: Clock Configuration for the master microcontroller

From the project manager tab, select SW4STM32 as the compiler and generate the code for the master microcontroller.

11.3 • SPI Settings for Slave Microcontroller

Settings for the slave microcontroller (STM32F030F4P6) are as follows: From the connectivity section and SPI1 subsection, select the Mode as Full-Duplex Master and Hardware NSS Signal as Hardware NSS Input Signal. From the Parameter Settings tab then apply settings such as data size, Prescaler, etc., as shown in Figure 11.5. The clock configuration tab is shown in Figure 11.6. The SPI interface clock settings are undertaken on the master unit. In the connectivity section and SPI1 subsection, from the NVIC Settings tab, we can enable the SPI1 global interrupt as illustrated in Figure 11.7. From the system core section and NVIC subsection, we can then determine preemption priority for SPI1 global interrupt as shown in Figure 11.8. After undertaking the above settings, from the project manager tab, select SW4STM32 as the compiler and generate the code for the slave microcontroller.

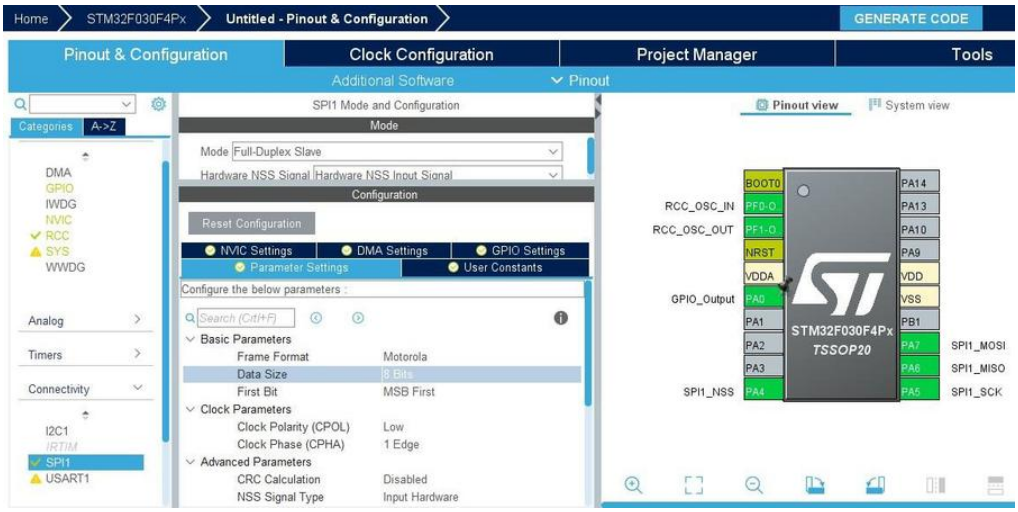

Figure 11.5: Enabling SPI1 for slave microcontroller

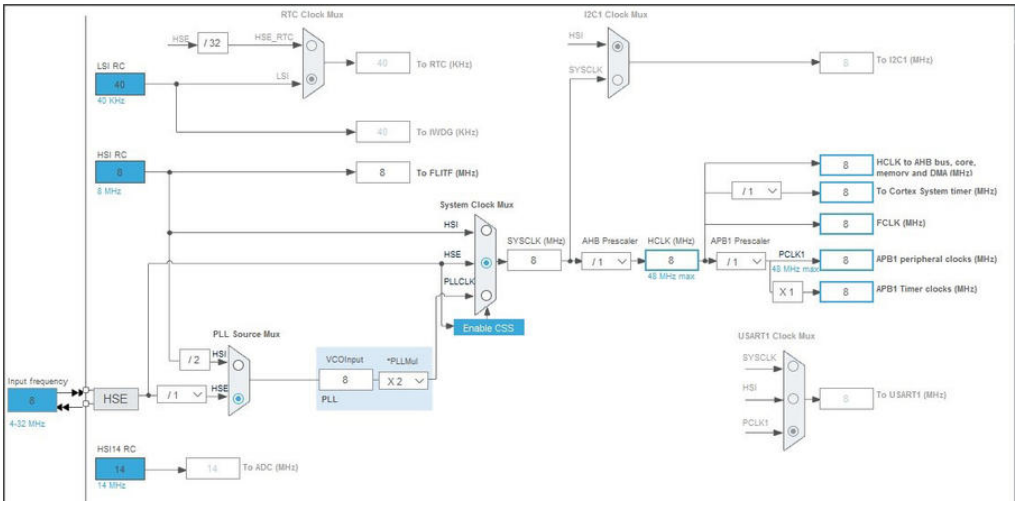

Figure 11.6: Clock Configuration for slave microcontroller

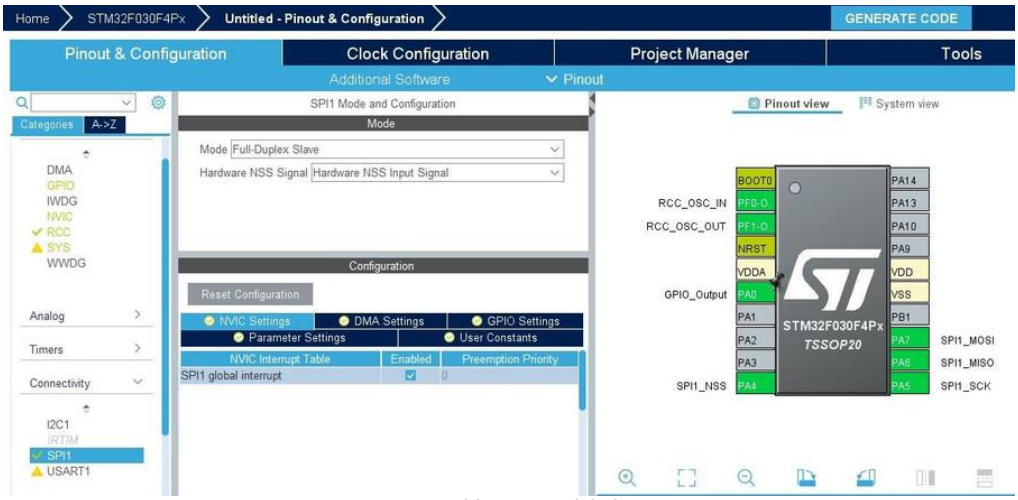

Figure 11.7: Enabling SPI1 global interrupt

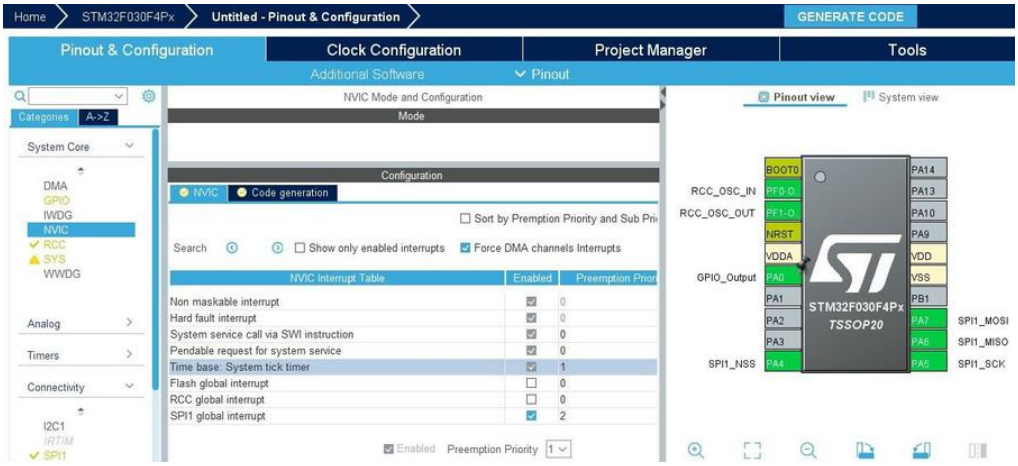

Figure 11.8: Selecting Preemption Priority for SPI1 global interrupt

11.4 • Header Code of main.c for Master Microcontroller

The header code of the main.c file for the master microcontroller is shown in Figure 11.9. The code inside the MX_SPI2_Init function is demonstrated in Figure 11.10.

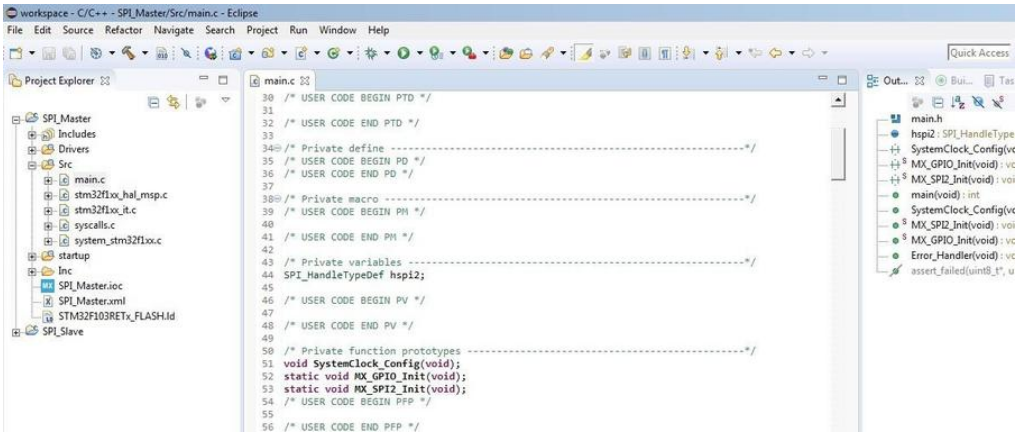

Figure 11.9: The header code of main.c (master)

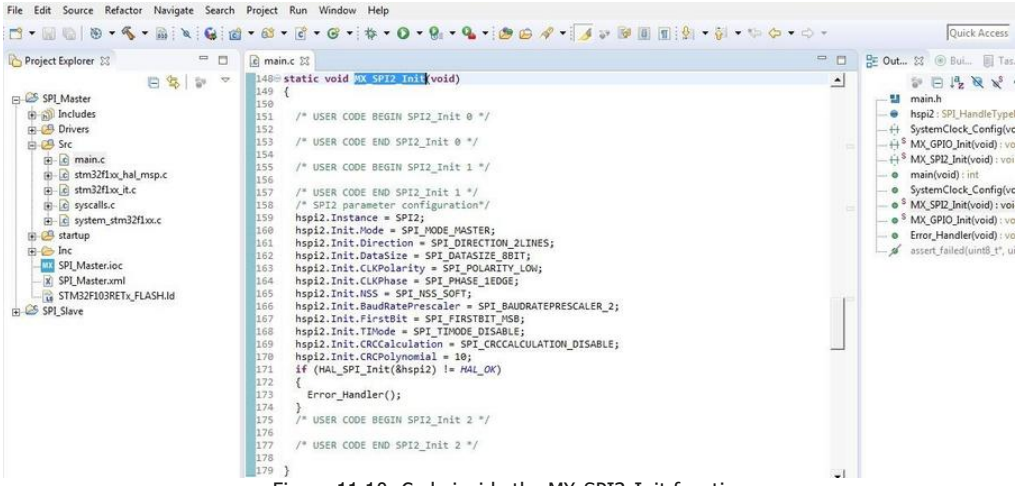

Figure 11.10: Code inside the MX_SPI2_Init function

11.5 ● Header Code of main.c for the Slave Microcontroller

The header code of the main.c file for the slave microcontroller is shown in Figure 11.11. The code inside the MX_SPI2_Init function is illustrated in Figure 11.12. Functions inside the stm32f1xx_hal_spi.c file are depicted in Figure 11.13.

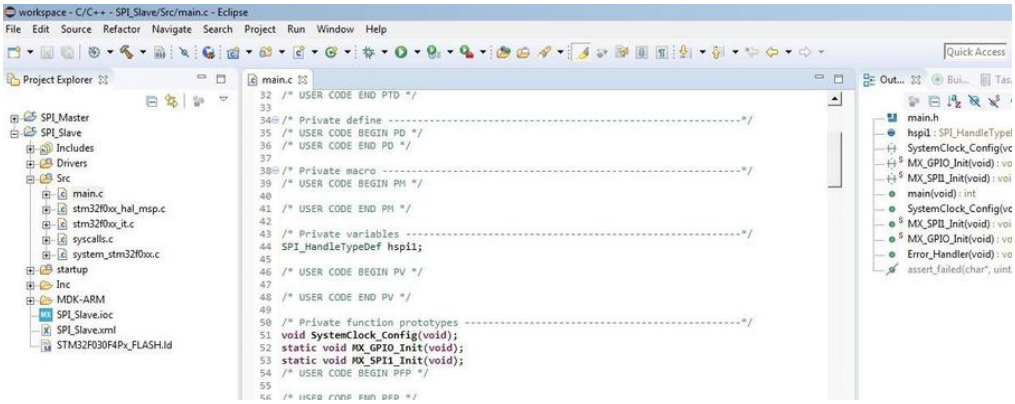

Figure 11.11: The header code of main.c slave)

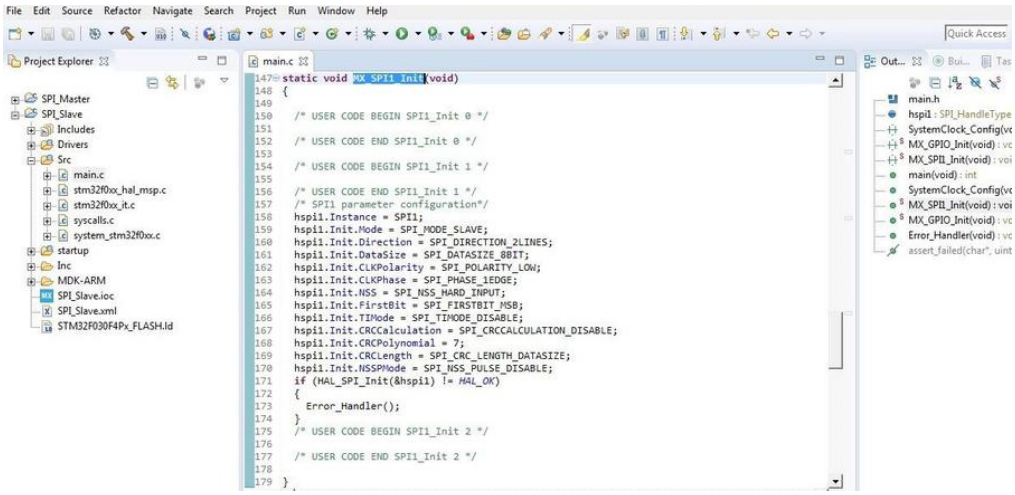

Figure 11.12: Code inside the MX_SPI1_Init function

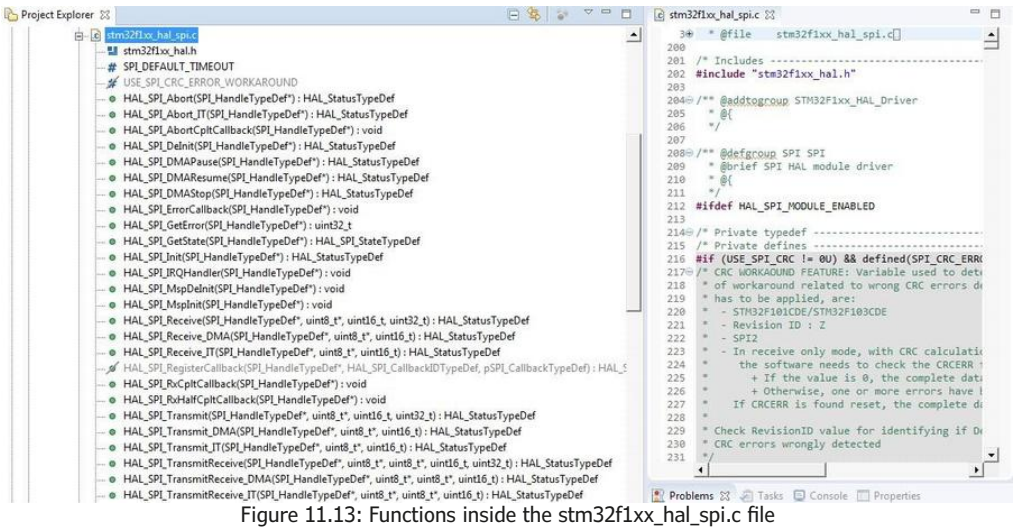

Figure 11.13: Functions inside the stm32f1xx_hal_spi.c file

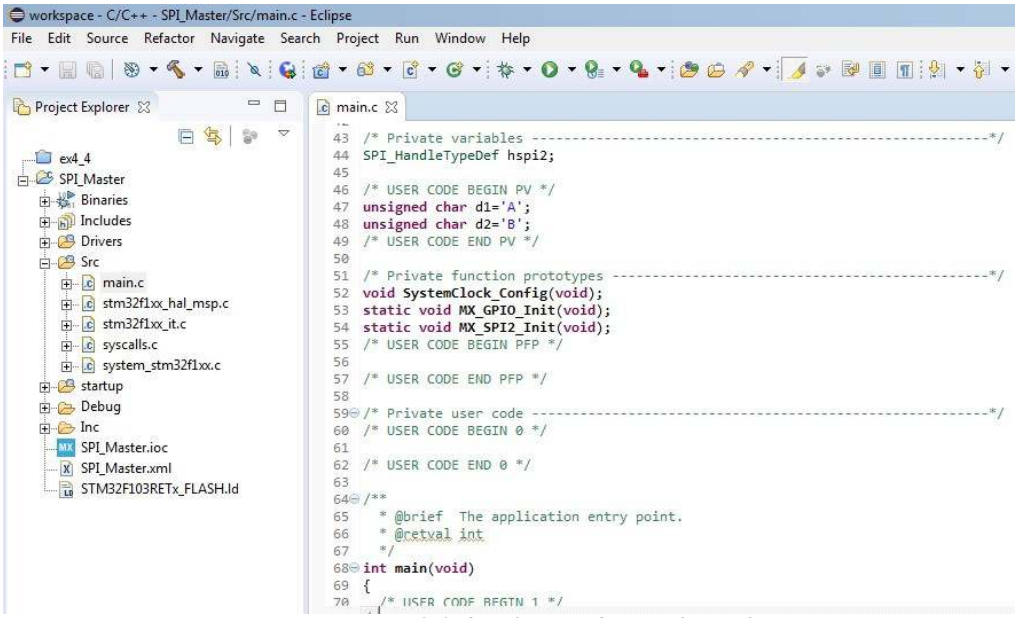

Figure 11.14: Code before the main function (master)

11.6 ● The main.c File Settings for Master and Slave Microcontrollers

The code before the main function for the master microcontroller is shown in Figure 11.14. Also, the code inside the while loop is demonstrated in Figure 11.15. The code before the main function of the slave microcontroller is shown in Figure 11.16. Also, the code inside the while loop is depicted in Figure 11.17. The HAL_SPI_RxCpltCallback function inside the stm32f0xx_hal_spi.c file (as shown in Figure 11.18) is used for the interrupt service routine and has been copied from stm32f0xx_hal_spi.c file inside stm32f0xx_it.c file as illustrated in Figure 11.19. Also, user code can be inserted inside the SPI1_IRQHandler function.

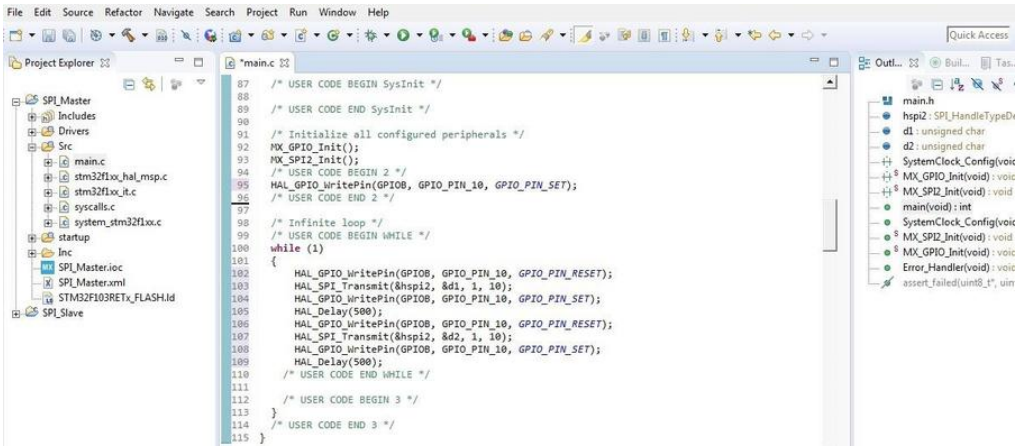

Figure 11.15: Code inside the while loop (master)

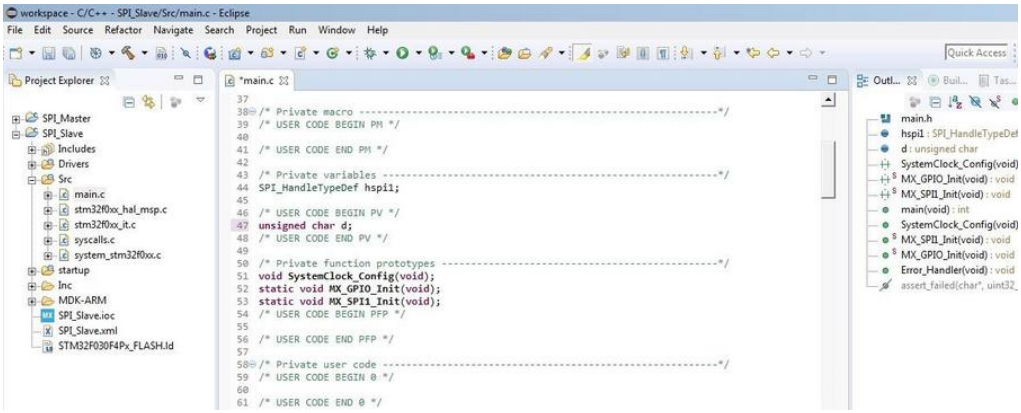

Figure 11.16: Code before the main function (slave)

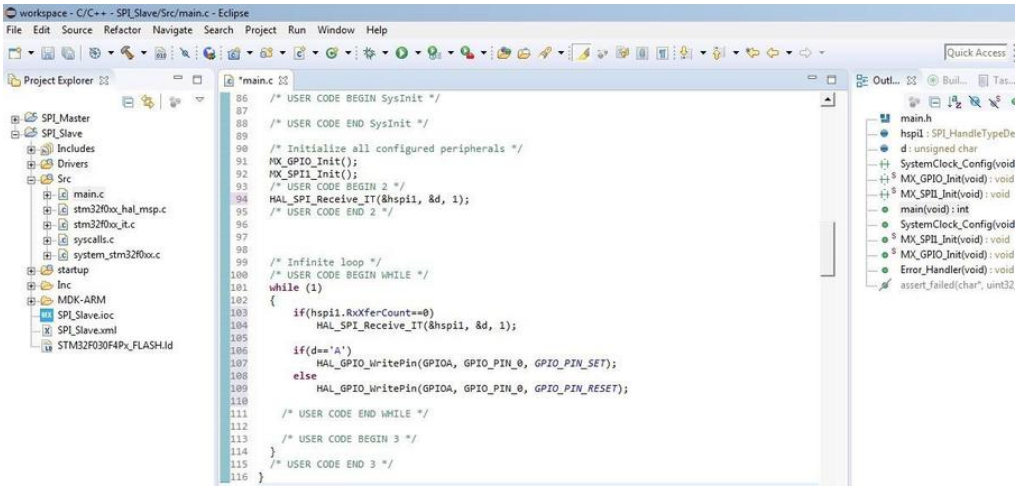

Figure 11.17: Code inside the while loop (slave)

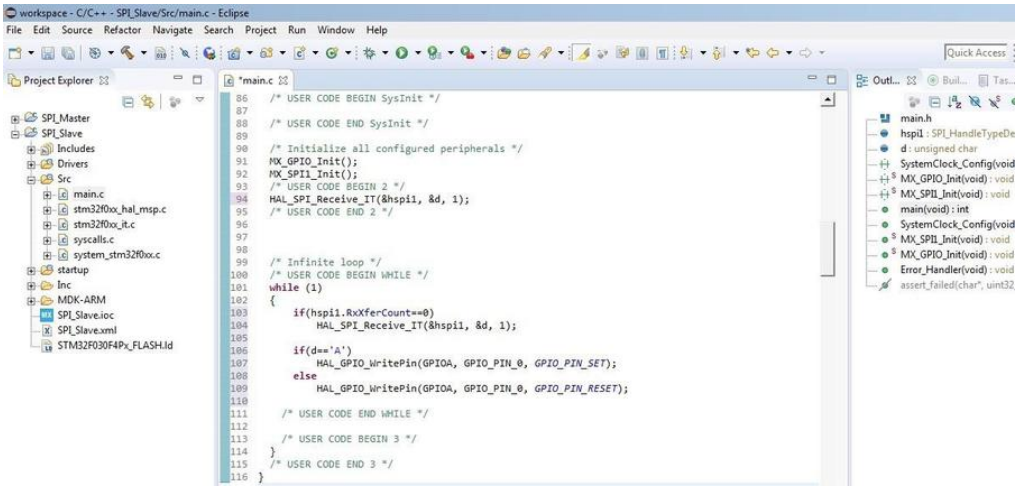

Figure 11.18: HAL_SPI_RxCpltCallback function inside the stm32f0xx_hal_spi.c file

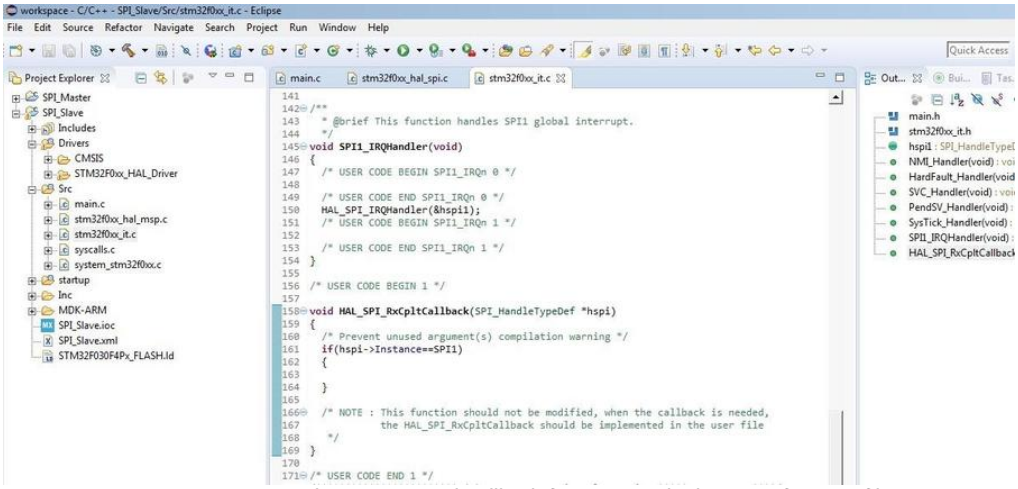

Figure 11.19: Copied HAL_SPI_RxCpltCallback function inside the stm32f0xx_it.c file

Interrupt enabling is achieved using the __HAL_SPI_ENABLE_IT macro. This macro is inside the stm32f0xx_hal_spi.h file as follows:

```
#define__HAL_SPI_ENABLE_IT(__HANDLE__,__INTERRUPT__)
```

Also, the receive interrupt enabling element is the SPI_IT_RXNE function. Right-click on the project and select 'Build Project' to compile the project and generate the hex code for programming the master and slave microcontrollers.

11.7 ● Summary

The SPI protocol is a synchronous bidirectional serial communication interface. This interface is one of the most important and useful communication protocols in microcontrollers due to its very high speed. In this chapter, SPI protocol settings for master and slave microcontrollers was explained in detail.

Chapter 12 • Watchdog Timer in STM32 Microcontrollers

12.1 • Introduction

The watchdog timer is used to reset the CPU when it hangs. It is adjustable and resets the CPU at predetermined times. However, the CPU in the program main loop resets the watchdog timer and does not allow the watchdog timer to reach the determined time and reset the CPU. The watchdog timer resets the CPU only when it hangs and does not reset itself. The watchdog timer structure is shown in Figure 12.1.

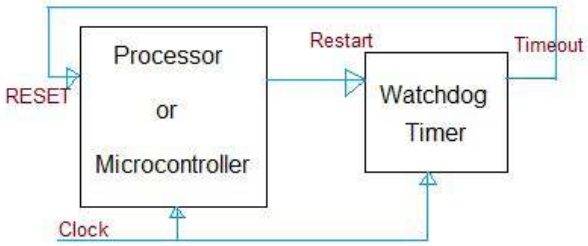

Figure 12.1: Watchdog timer structure

STM32 microcontrollers have two watchdog timer units called the Independent Watchdog (IWDG) and Window Watchdog (WWDG). An operation diagram of the WWDG timer is shown in Figure 12.2.

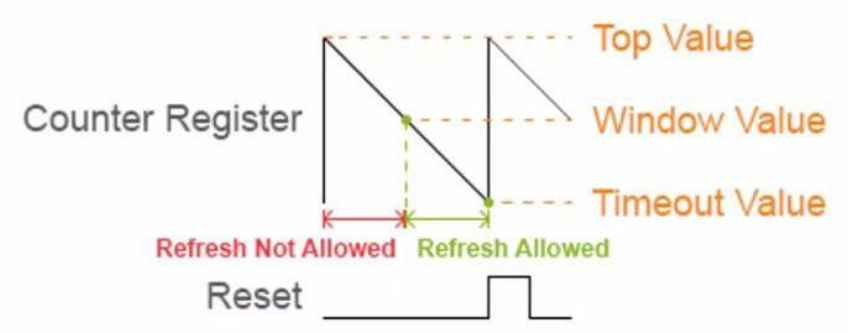

Figure 12.2: The operation diagram of WWDG timer

The IWDG timer has a clock independent of the main clock. It stays active even if the main clock fails.

12.2 ● WWDG Timer Project Settings

We consider the following example for WWDG timer operation: From the system core section and SYS subsection, choose to debug as Serial Wire as illustrated in Figure 12.3. From the system core and the RCC subsection, set the external crystal resonator as demonstrated in Figure 12.4. Also, from the Clock Configuration tab, set PCLK1 equal to 36 MHz as shown in Figure 12.5. Finally, we need to enable the WWDG timer, LED_GREEN, LED_YELLOW as outputs, and SW as input and pull-up mode as shown in Figure 12.6 and Figure 12.7.

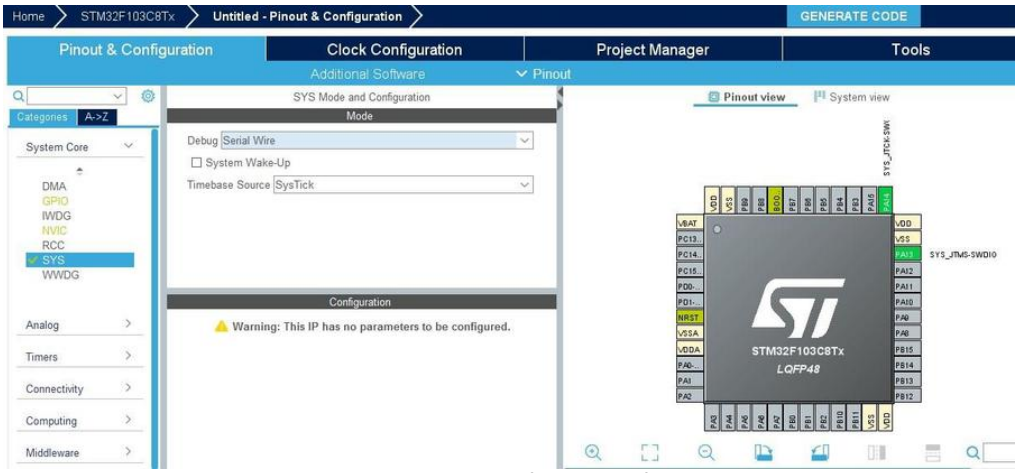

Figure 12.3: Setting Debug as Serial Wire.

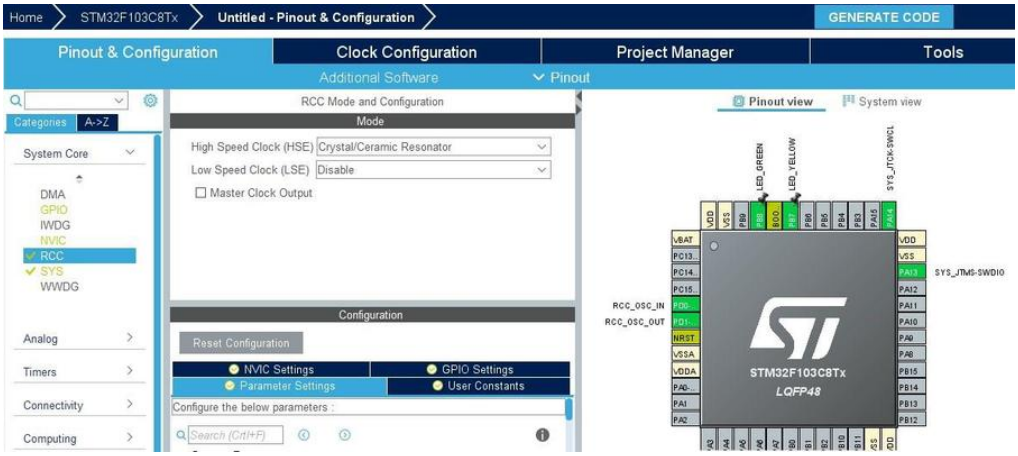

Figure 12.4: Setting external crystal resonator

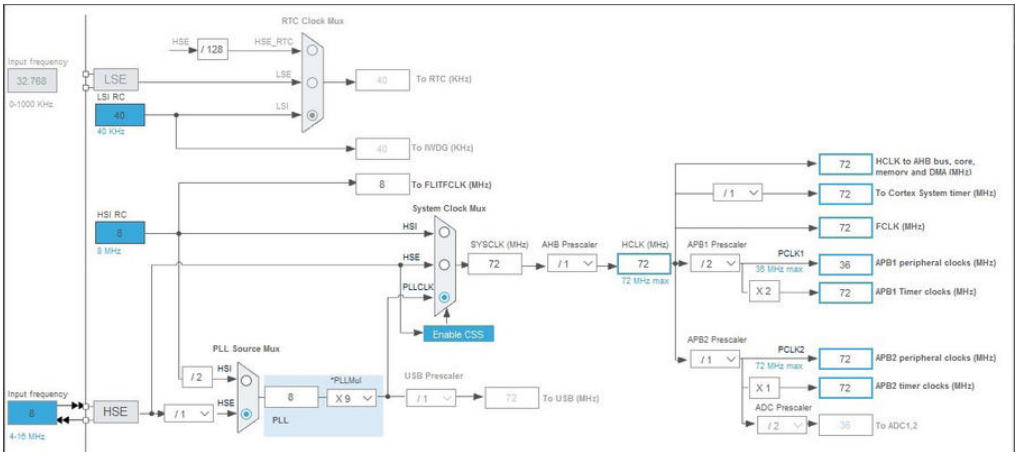

Figure 12.5: Setting PCLK1 at 36 MHz

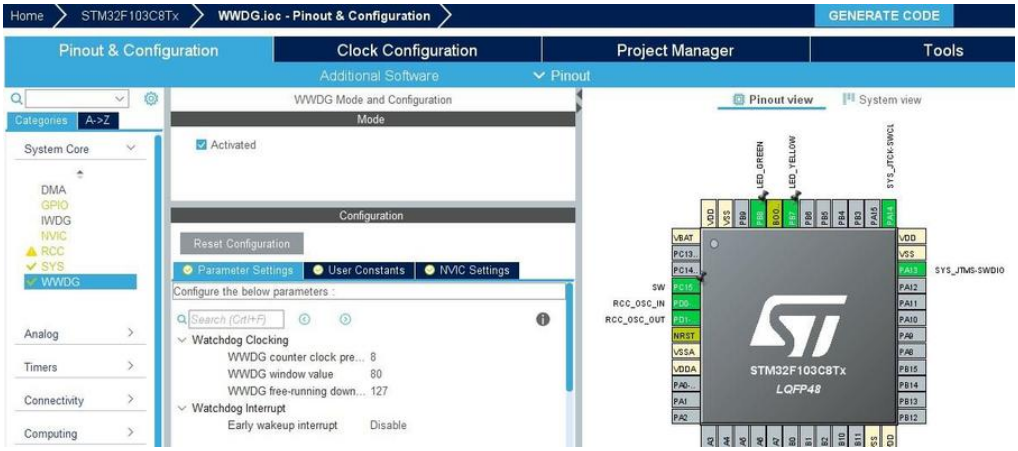

Figure 12.6: Activating WWDG timer, LED_GREEN, LED_YELLOW, and SW

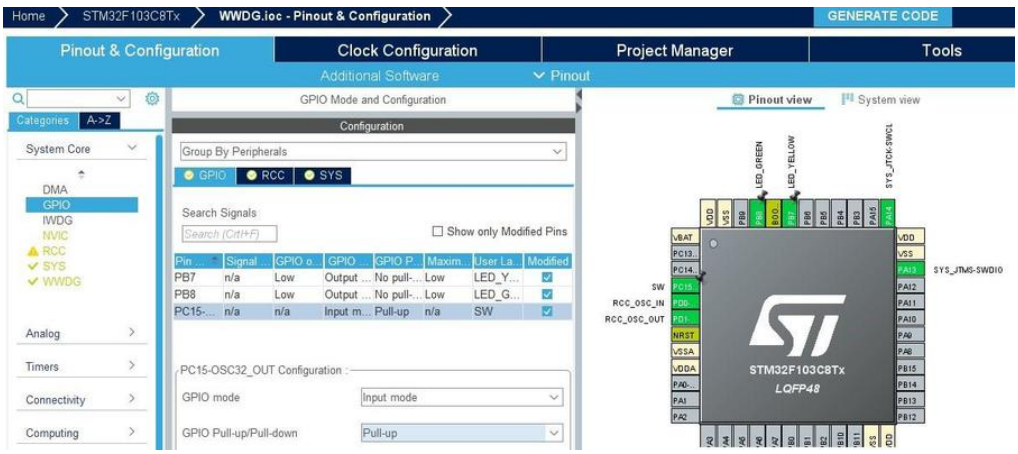

Figure 12.7: Defining SW as input and pull up mode

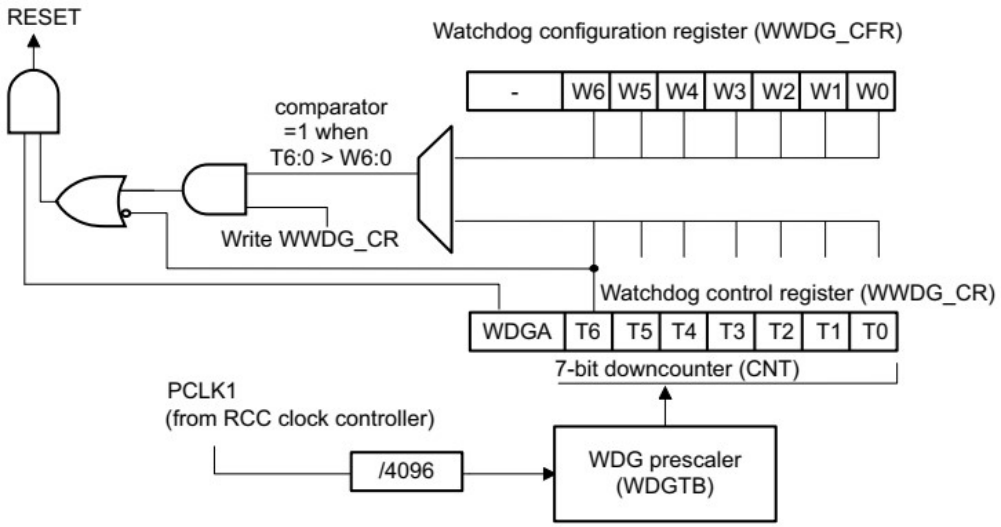

Figure 12.8: WWDG block diagram from RM0008 Reference Manual file

From Figures 12.5, 12.6, and 12.8 (PCLK1 is divided to 4096 and then it is applied to the WDG Prescaler block), we can calculate the WWDG frequency and period as follows:

WWDG frequency = 36 MHz/(4096×8) ≈ 1099 Hz, WWDG period = 1/1099 ≈ 910 µs

Window Start = 910 µs × (127 – 80) ≈ 43 ms

Window Timeout = 910 µs × (127 – 63) ≈ 58 ms

Finally, we can generate code in the SW4STM32 software environment as illustrated in Figure 12.9. The code inside the while loop in the main function is illustrated in Figure 12.10.

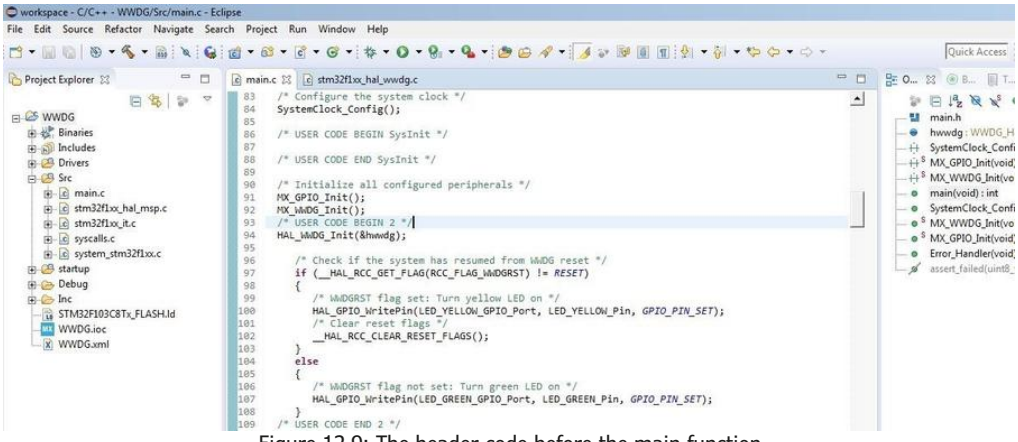

Figure 12.9: The header code before the main function

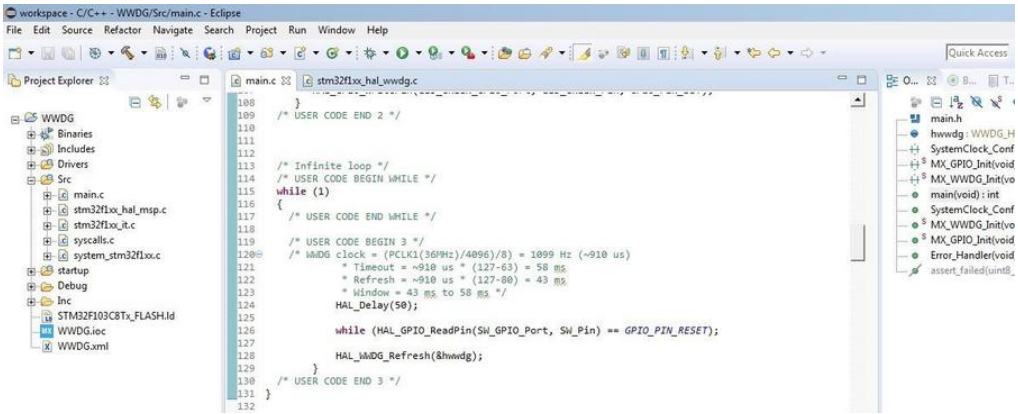

Figure 12.10: Code inside the while loop in the main function

Right-click the project and select 'Build Project' to compile the project and generate hex code for programming the microcontroller. When running the program, the yellow LED will be on. By pressing the reset button, the yellow LED turns off and the green LED will switch on. If the SW button is pressed, the green LED will switch off and the yellow LED on. If we change HAL_Delay(50) to HAL_Delay(42) and then compile and run the program by pressing the reset button, the green LED will be momentarily switched on. By releasing the reset button, the green LED will switch off and the yellow LED will switch on again.

12.3 • IWDG Timer Project Settings

In the following example, we evaluate the IWDG timer operation. The same as before, activate SW, LED_GREEN, and LED_YELLOW. Then enable the external crystal resonator pins. Finally, from the system core, and IWDG subsection, we activate the mode. From the parameter settings tab, set the IWDG counter clock Prescaler to 32 and the IWDG down-counter reload value as 4095 as shown in Figure 12.11. From the Clock Configuration tab, we can see the IWDG clock frequency is 40 kHz as shown in Figure 12.12.

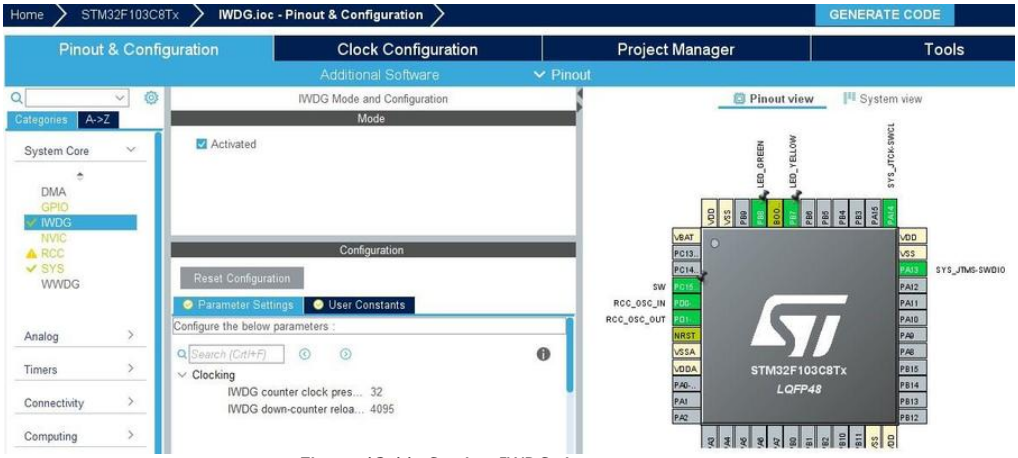

Figure 12.11: Setting IWDG timer parameters

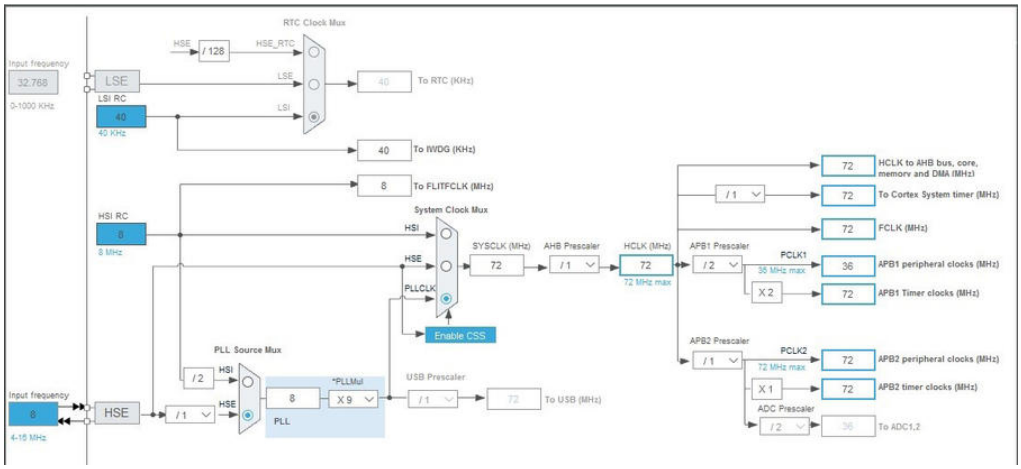

Figure 12.12: The IWDG clock frequency

From Figures 12.11 and 12.12 we have the following equations:

IWDG counter period = 1/(40 kHz × 32) = 0.8 ms
IWDG counter reload time = 4095 × 0.8 ms ≈ 3.2 s

From the Project Manager tab, we can then select the SW4STM32 software as the compiler and generate the code. The header code of the main.c file is shown in Figure 12.13. The code inside the while loop is demonstrated in Figure 12.14.

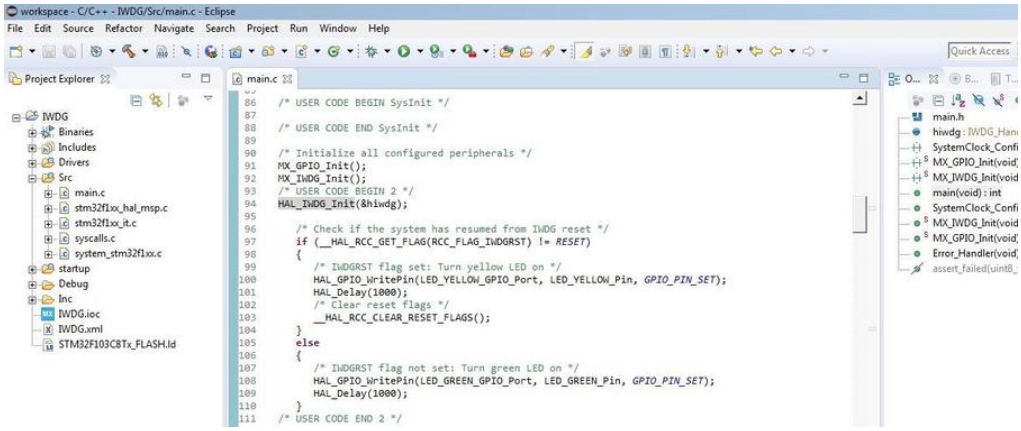

Figure 12.13: The header code of main.c

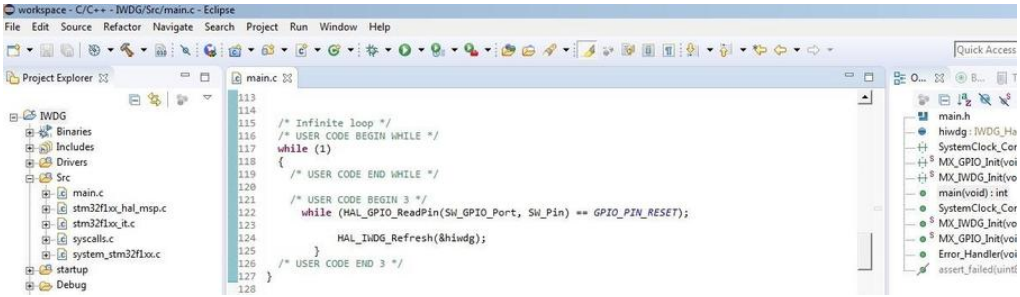

Figure 12.14: The code inside the while loop

We can then compile the project by right-clicking the project name and selecting the 'Build Project' icon. After successfully compiling the project, the hex file generated can be used for programming the microcontroller. By executing the program, the yellow LED will switch on. If the reset button is pressed, the green LED will switch on. By pressing the SW button for more than 3.2s, the green LED will turn off and the yellow LED turns on.

12.4 • Summary

The watchdog timer is an important aspect of microcontrollers. It is used to reset the CPU when it hangs. STM32 microcontrollers have two watchdog timer units which are called the Independent Watchdog (IWDG) and Window Watchdog (WWDG) timers. In this chapter, WWDG and IWDG timer project settings were discussed in detail.

Chapter 13 • Inter-Integrated Circuit (I²C) in STM32 Microcontrollers

13.1 • Introduction

The Inter-Integrated Circuit (I²C) is a serial bus protocol consisting of two signal lines called SCL and SDL lines. These are used for communication with different devices. SCL stands for the Serial Clock line and this signal is always generated by the master device. SDL stands for the Serial Data Line. The SDL signal is generated by either the master or the devices connected to it with the I²C protocol. Both SCL and SDL lines are in open-drain mode when there is no transfer between I²C interface devices. In the following example, we will configure the STM32 microcontroller as a simple I²C slave device. Communication is then established with the I²C slave device from another STM32 microcontroller which is an I²C master device. The block diagram of the I²C protocol example is shown in Figure 13.1.

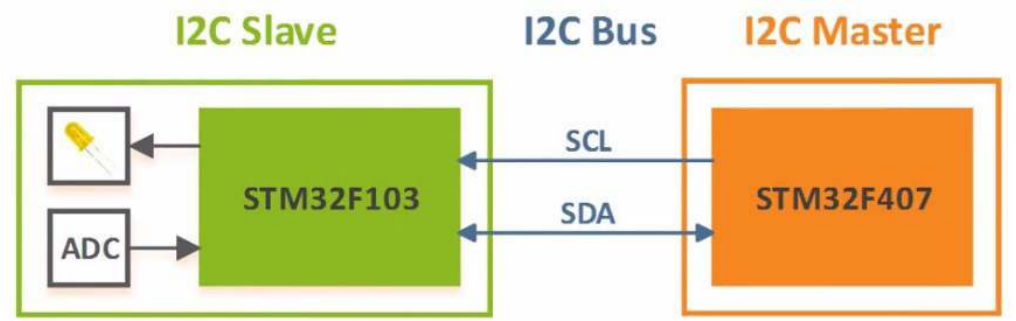

Figure 13.1: The block diagram of the I2C protocol example

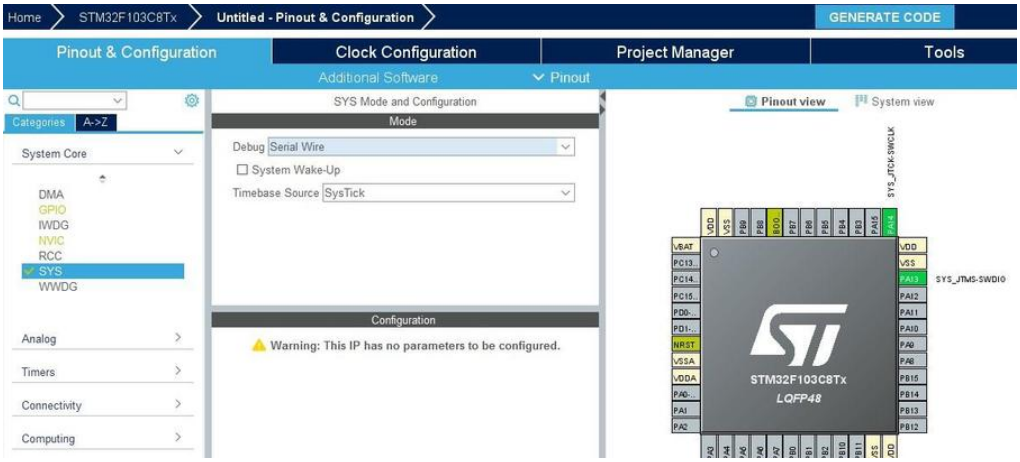

Figure 13.2: Setting Debug as Serial Wire

13.2 • I²C Settings for Slave Microcontroller

First, we set the I²C slave microcontroller. From the System Core section and SYS subsection, select debug as the Serial Wire as depicted in Figure 13.2. Then, from the RCC subsection, we enable the external crystal resonator as illustrated in Figure 13.3. From the connectivity section and I²C² subsection, select I²C as demonstrated in Figure 13.4. From the Analogue section and ADC1 subsection, enable IN6 as shown in Figure 13.5. The parameter settings tab for ADC1 (IN6) is shown in Figure 13.6. Parameter Settings for I²C² are depicted in Figure 13.7. Finally, from the Project Manager tab, select 'SW4STM32' as the compiler and generate the code.

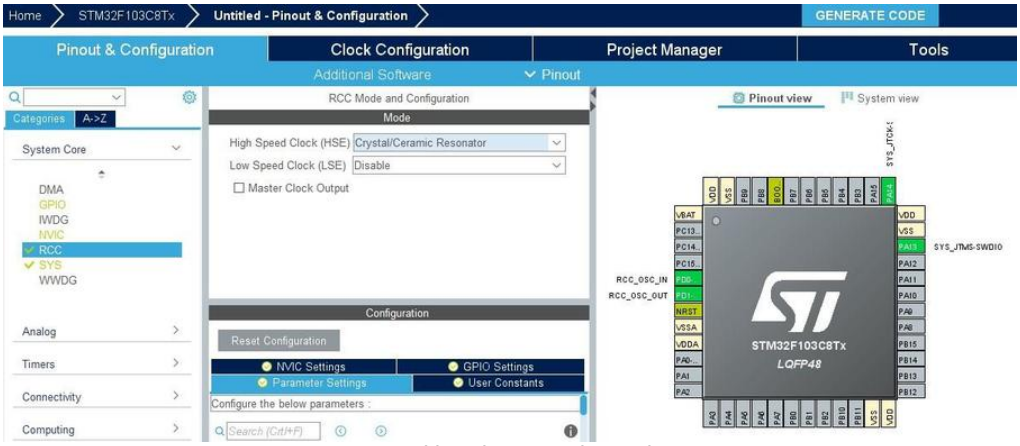

Figure 13.3: Enabling the external crystal resonator

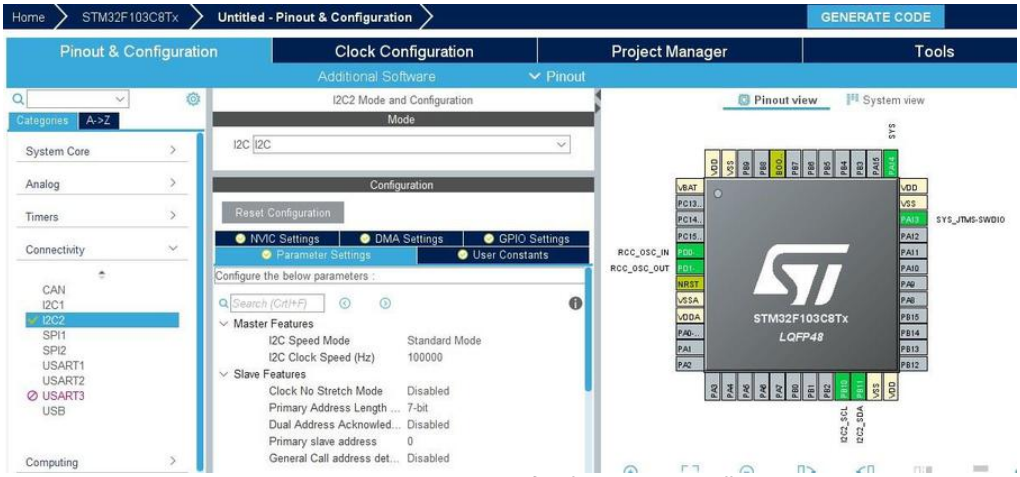

Figure 13.4: Activating I2C2 for slave microcontroller

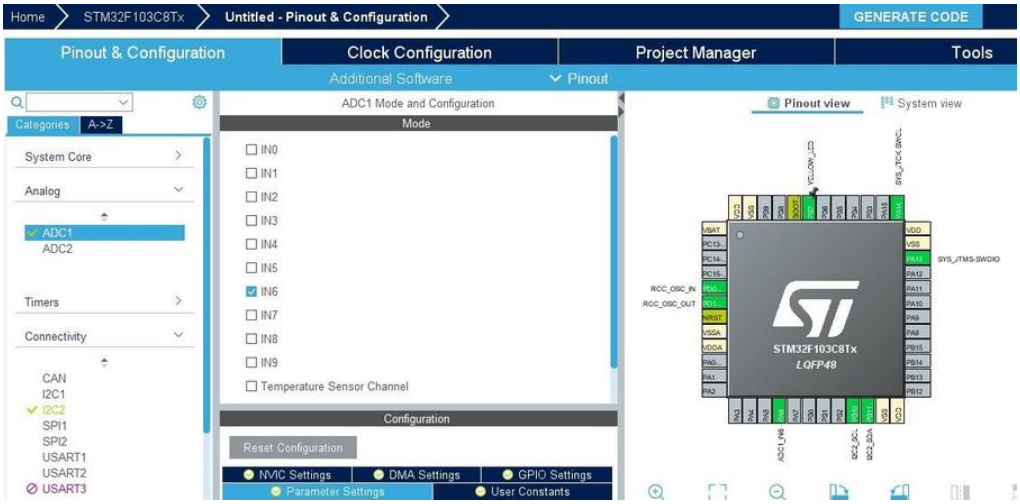

Figure 13.5: Enabling ADC1 (IN6)

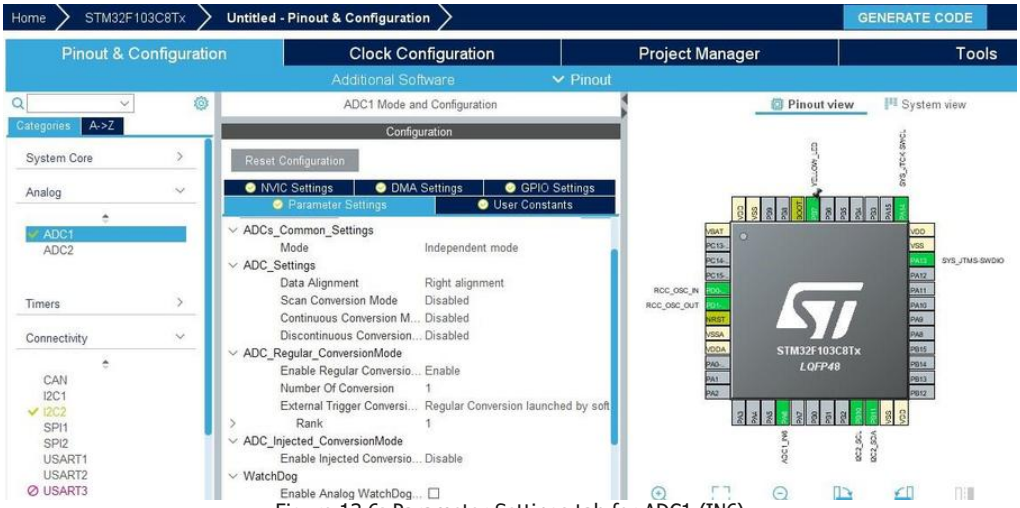

Figure 13.6: Parameter Settings tab for ADC1 (IN6)

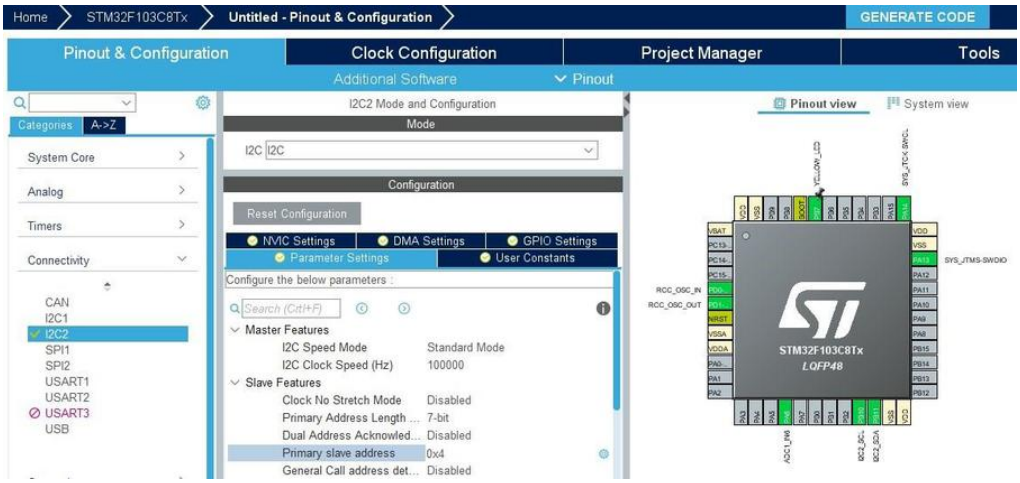

Figure 13.7: The Parameter Settings for I2C2

The header code of main.c is illustrated in Figure 13.8. The code inside the while loop is shown in Figures 13.9 and 13.10.

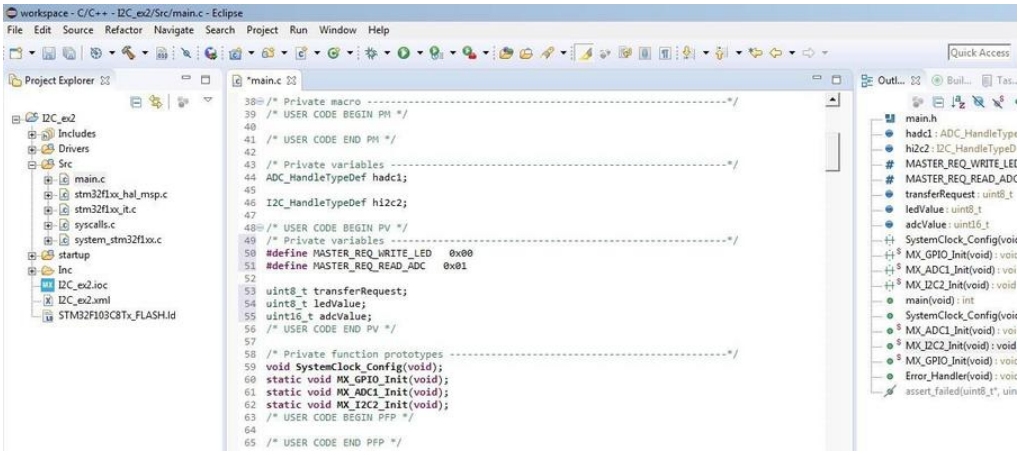

Figure 13.8: The header code of the main.c file

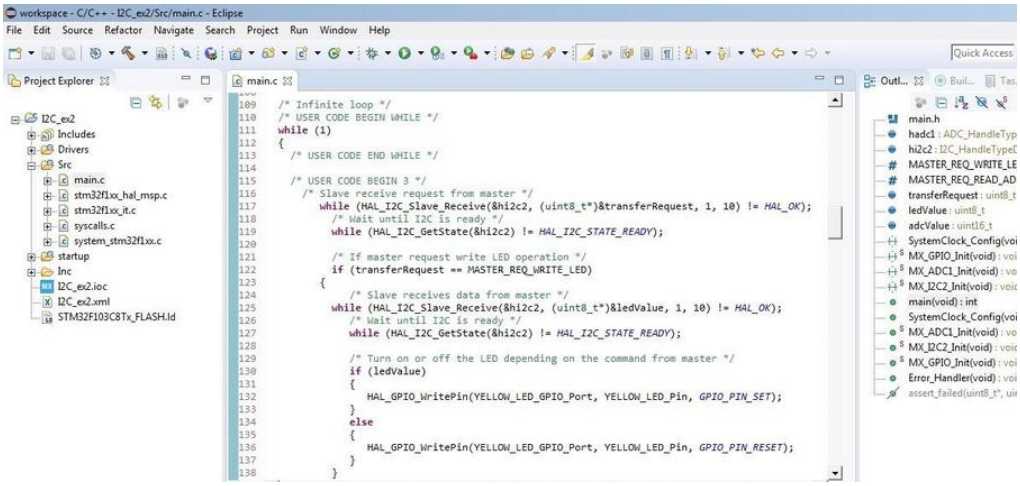

Figure 13.9: Code inside the while loop (part1)

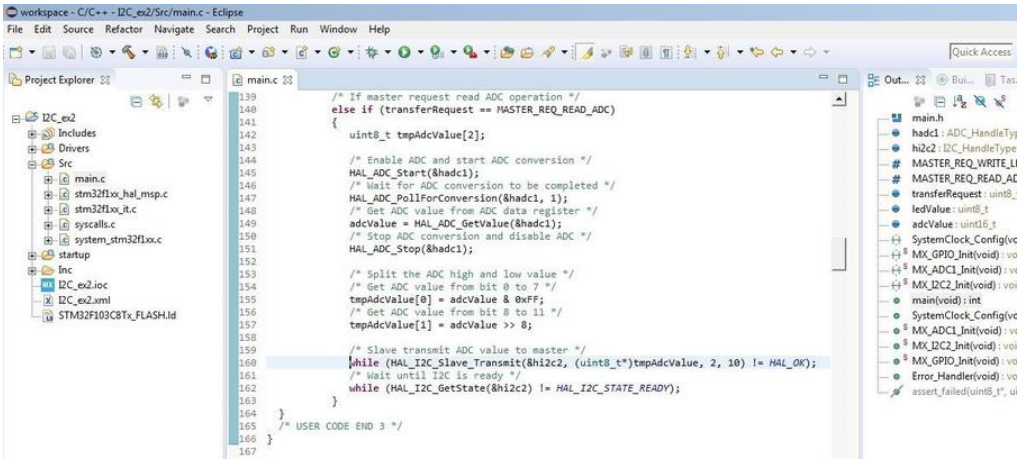

Figure 13.10: Code inside the while loop (part2)

Right-click on the project and select 'Build Project' to compile the project and generate the hex code for programming the I²C slave microcontroller.

13.3 ● I²C Settings for Master Microcontroller

We can now set the I²C master microcontroller. From the System Core section and SYS subsection, select Debug as the serial wire as shown in Figure 13.11. From the connectivity section and I²C[1] subsection, select I²C and Default parameter settings as shown in Figure 13.12. Finally, from the Project Manager tab, select SW4STM32 as the compiler and generate the code.

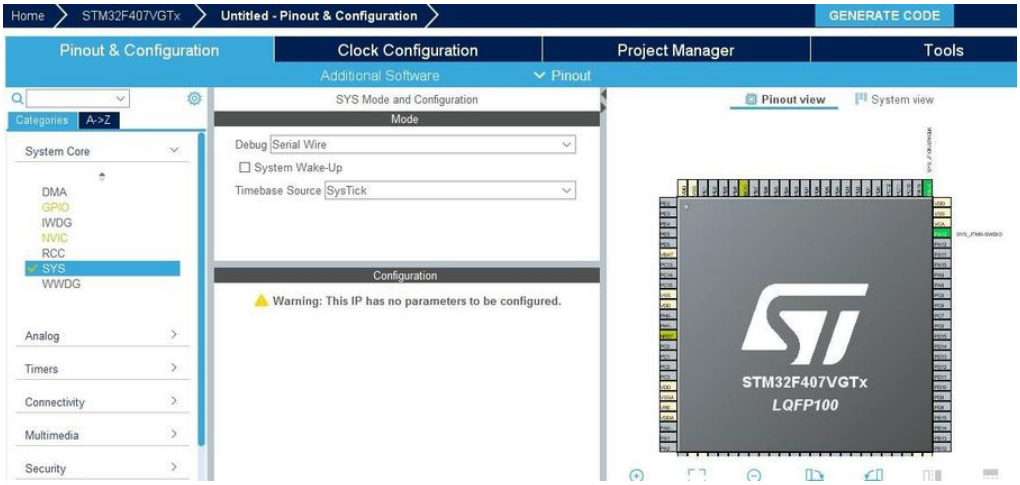

Figure 13.11: Setting Debug as Serial Wire

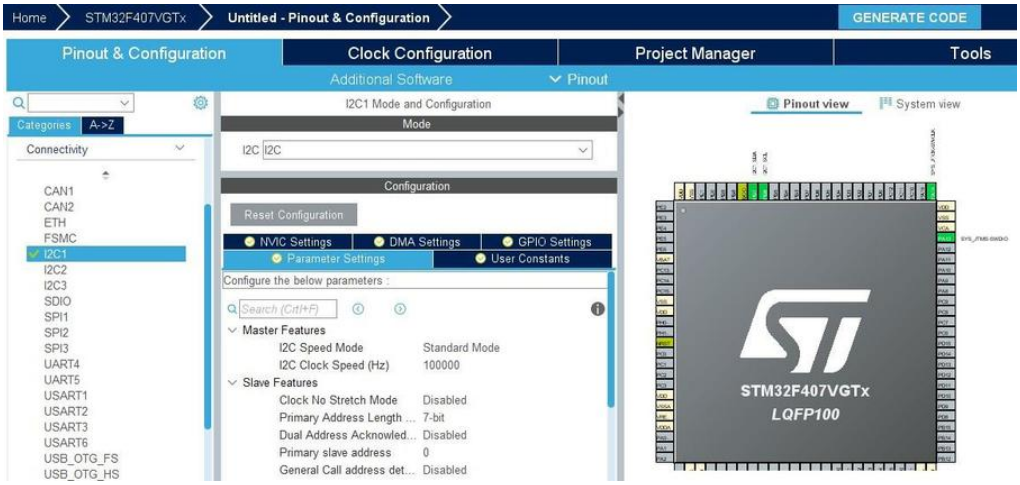

Figure 13.12: Activating I2C1 for the master microcontroller

The header code of the main.c file is illustrated in Figure 13.13. The code inside the while loop is shown in Figures 13.14 and 13.15 respectively.

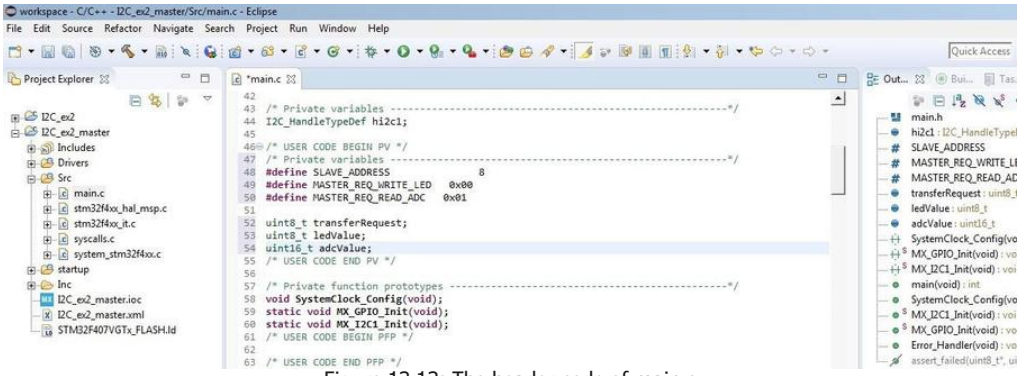

Figure 13.13: The header code of main.c

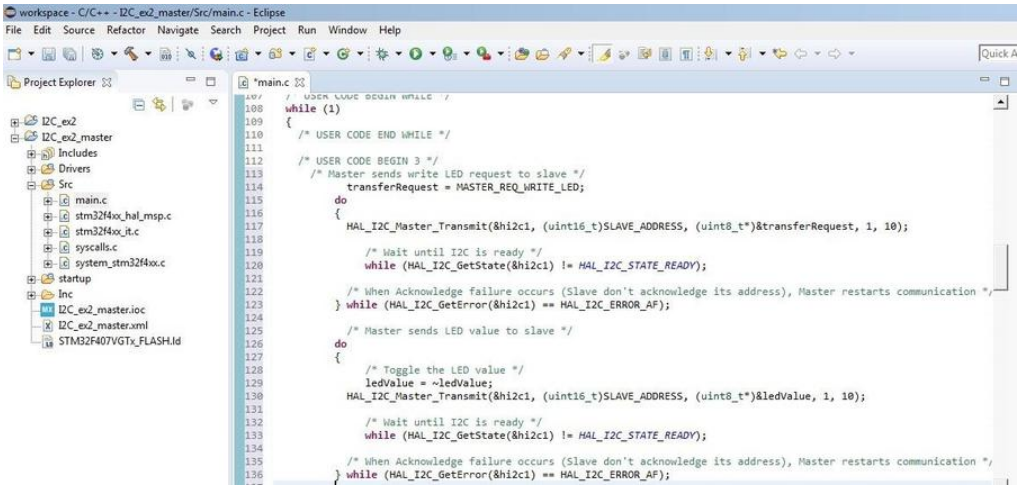

Figure 13.14: Code inside the while loop (part1)

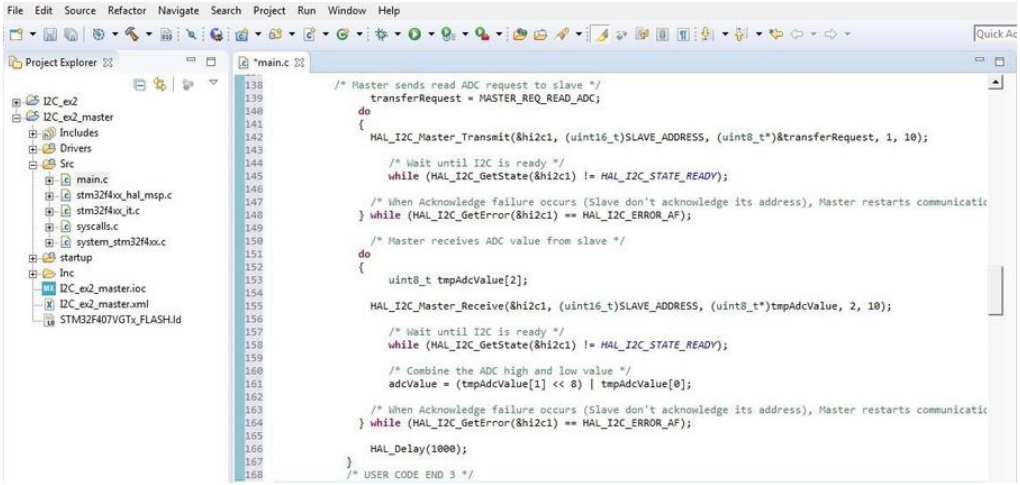

Figure 13.15: Code inside the while loop (part2)

Right-click on the project and select ‘Build Project’ to compile the project and generate the hex code for programming the I²C master microcontroller.

13.4 • Summary

The Inter-Integrated Circuit (I²C) is a serial bus protocol consisting of two signal lines called SCL and SDL lines. These are used for communication. SCL stands for the ‘Serial Clock Line’ and this signal is always generated by the master device. SDL stands for Serial Data Line and the SDL signal is generated either by the master or devices connected to it with I²C protocol. Both SCL and SDL lines are in open-drain mode when there is no transfer between I²C interface devices. In this chapter, the I²C protocol settings for master and slave microcontrollers were explained in detail.

Chapter 14 • Direct Memory Access (DMA) in STM32 Microcontrollers

14.1 • Introduction

In many microcontroller projects, reading and writing data is required. Reading data can be from a peripheral unit like an ADC and then writing the values to RAM. On the other hand, it may be a requirement to send data using the SPI protocol, moreover, to read data from RAM and continuously write to the SPI data register. By using the processor, a significant amount of processing time is lost. To avoid hogging the CPU processing time, most advanced microcontrollers, have a Direct Memory Access (DMA) controller. Data transfers using DMA are implemented between memory locations without the need for CPU. The block diagram of DMA is shown in Figure 14.1.

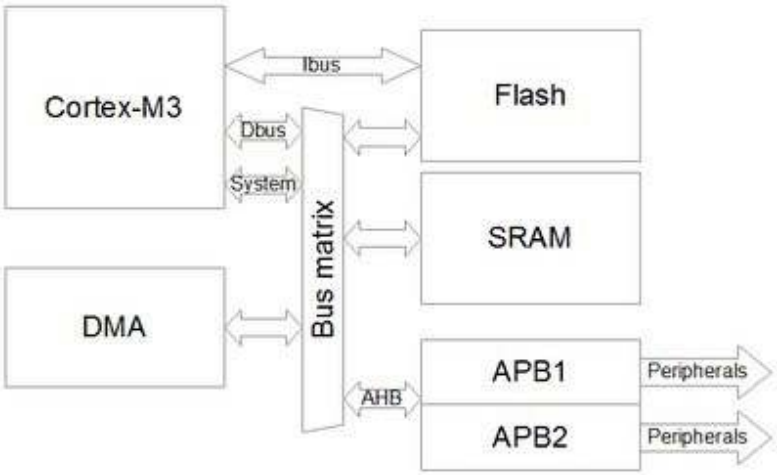

Figure 14.1: Block diagram of DMA

Consider the following examples to demonstrate the DMA applications in STM32 microcontroller programming.

14.2 • DMA Settings for ADC

From the system core section and RCC subsection, activate the external crystal resonator as shown in Figure 14.2. From the Analogue section and ADC1 subsection, enable IN0, and IN1. From the Parameter Settings tab, set the required parameters and activate pin PB0 (GPIO_Output) as illustrated in Figure 14.3. From ADC1 of the Analogue section and the DMA Settings tab, click the 'Add' button and apply parameters as shown in Figure 14.4. From the Clock Configuration tab, adjust the suitable clock frequencies as shown in Figure 14.5.

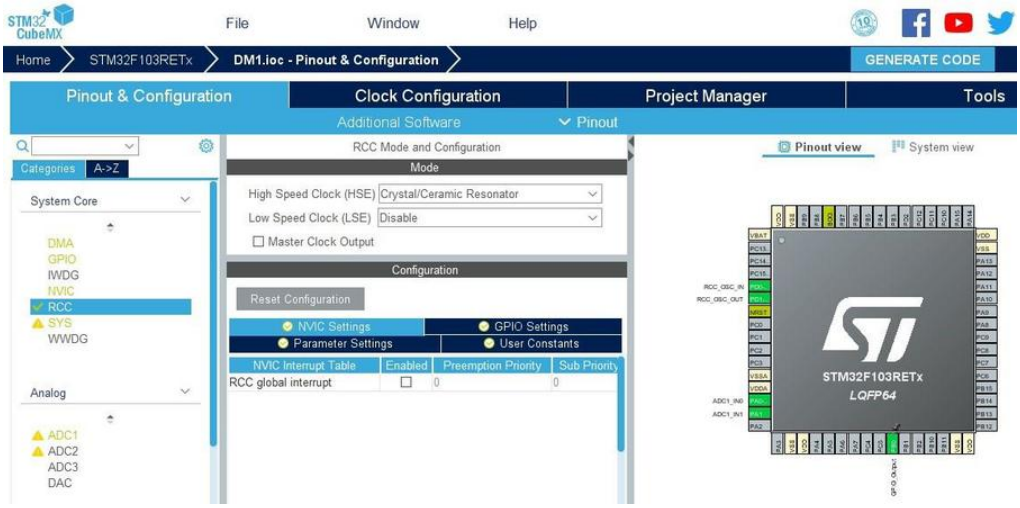

Figure 14.2: Activating the external crystal resonator

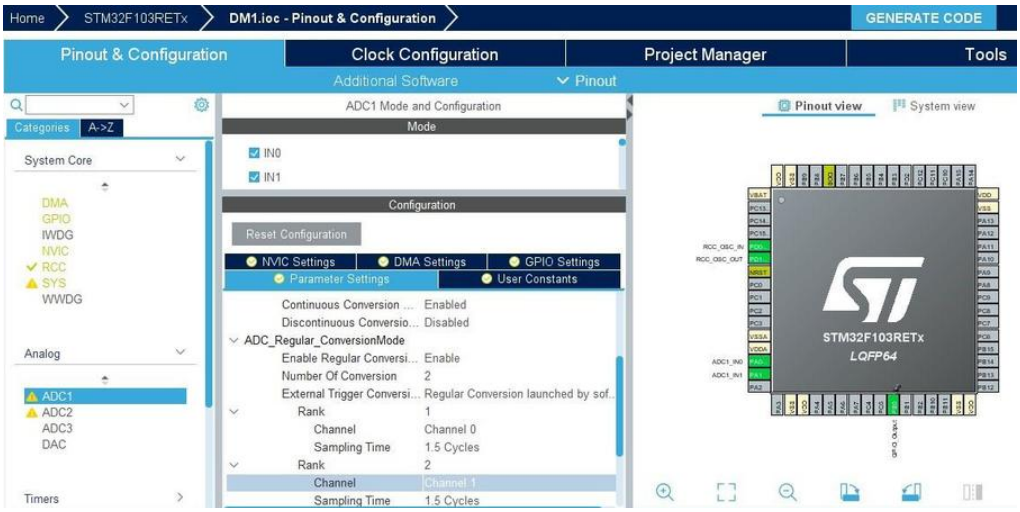

Figure 14.3: Enabling analogue pins and the digital output pin

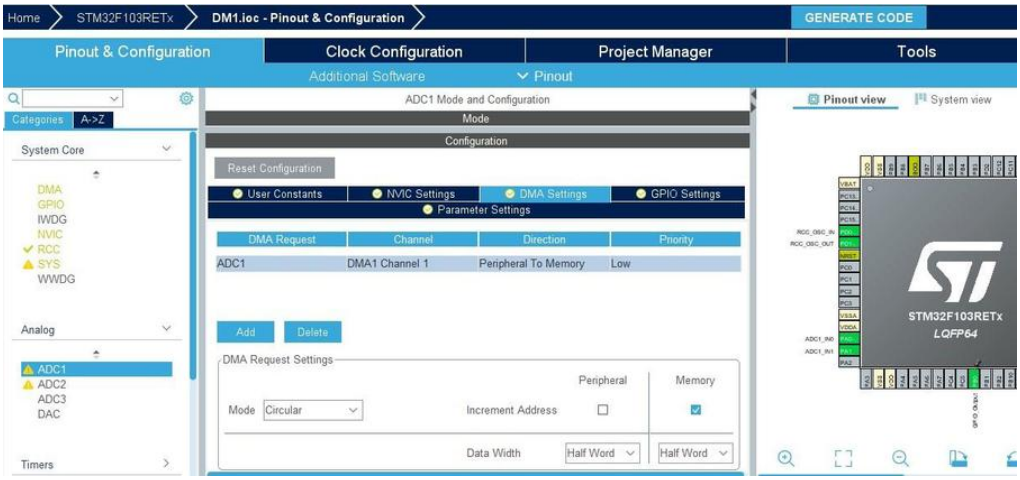

Figure 14.4: DMA Settings for ADC1

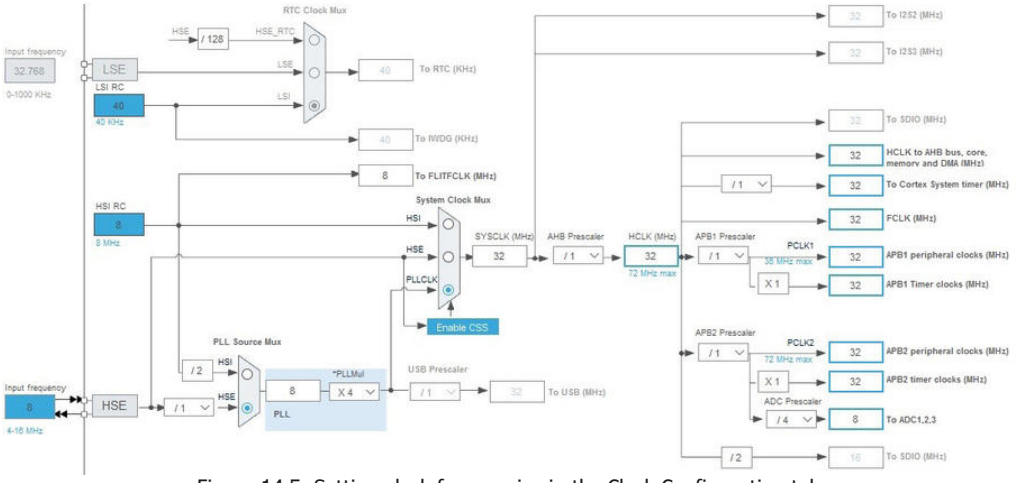

Figure 14.5: Setting clock frequencies in the Clock Configuration tab

From the Project Manager tab, select MDK-ARM V5 as the compiler and generate code in Keil uVision. The header code of the main.c file is demonstrated in Figure 14.6.

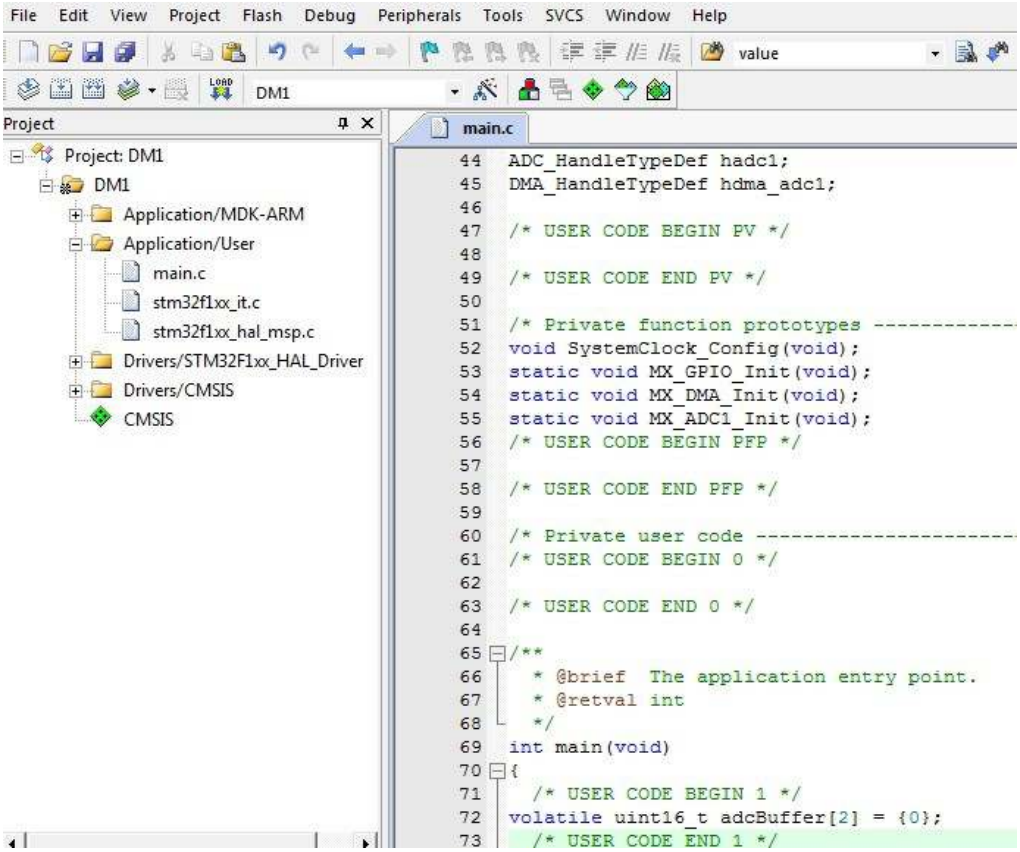

Figure 14.6: The header code of main.c

The code inside the while loop is illustrated in Figure 14.7. Finally, we can compile the project and generate a hex code for programming the microcontroller.

14.3 • Timer, ADC, and DMA Project Settings

The purpose of the next example is to set up a periodic timer which starts an ADC measurement in the background using DMA. Since the TIM3 clock frequency is 8 MHz, to generate events at 10Hz, a Prescaler of 800-1 is used with a counter period of 1000-1. After the Prescaler, the frequency is 10 kHz. The counter reaches 1000 after 0.1 seconds, after which the value is reloaded. An important setting here is the “Trigger Event Selection: Update Event” which will be used to trigger the ADC. From the timers section and TIM3 subsection, select the clock source as the internal clock and undertake the required Parameter Settings as shown in Figure 14.8.

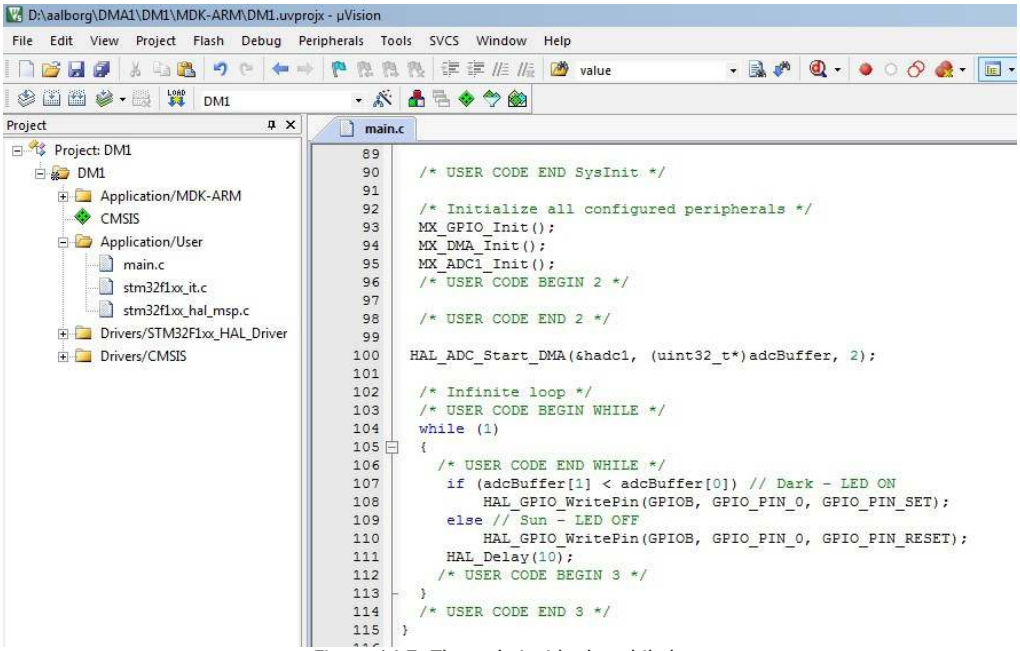

Figure 14.7: The code inside the while loop

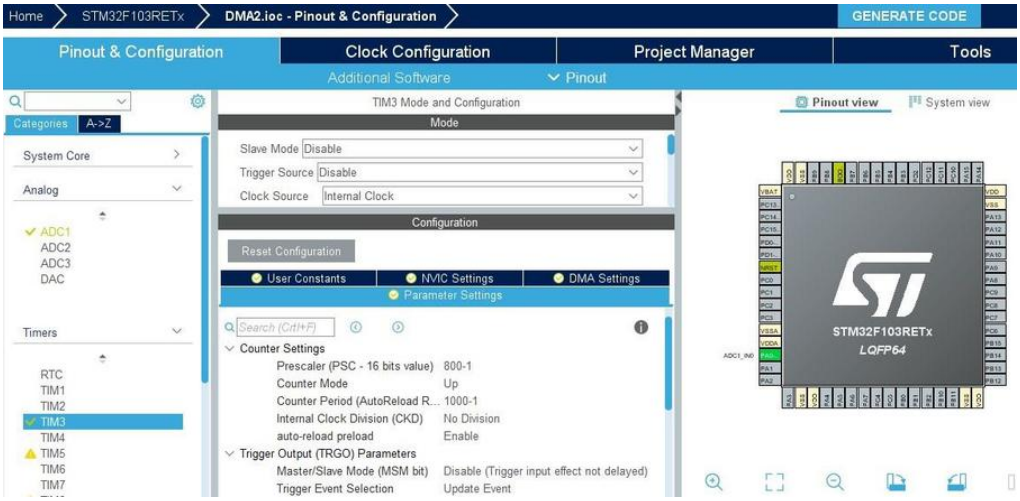

Figure 14.8: TIM3 Parameter Settings

From the Analogue section and ADC1 subsection, enable IN0. From the Parameter Settings tab, set the required parameters as illustrated in Figure 14.9. From ADC1 of the Analogue section and DMA Settings tab, click the Add button and apply parameters as shown in Figure 14.10. From the Project Manager tab, select 'MDK-ARM V5' as the compiler and generate the code in Keil uVision. The header code of main.c is demonstrated in Figure 14.11.

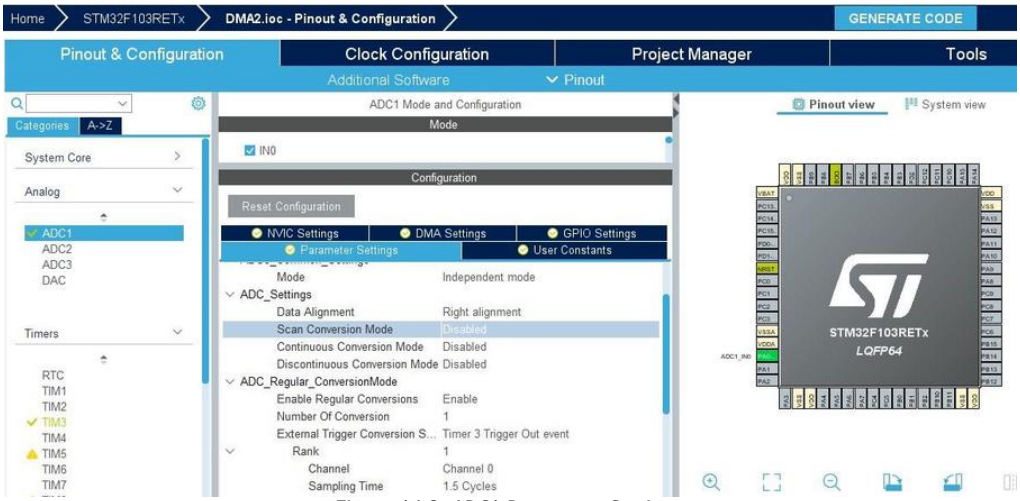

Figure 14.9: ADC1 Parameter Settings

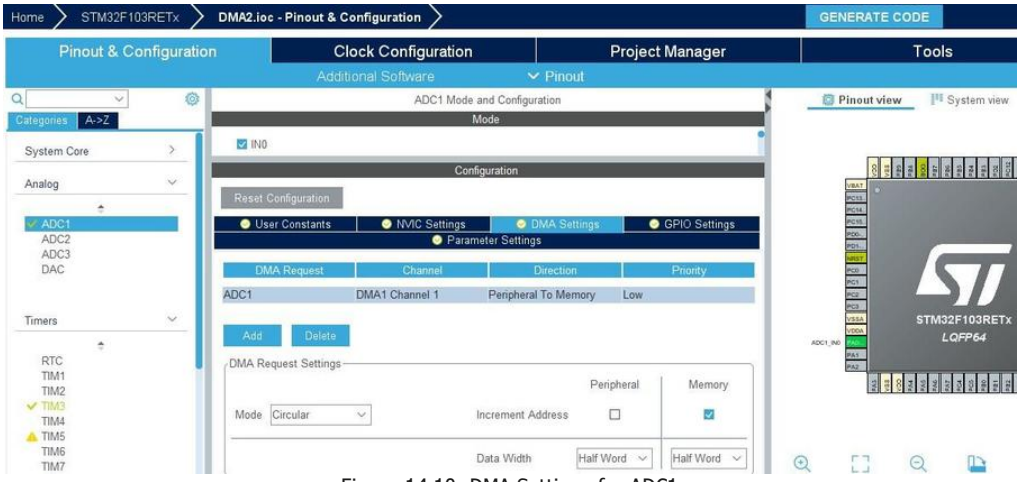

Figure 14.10: DMA Settings for ADC1

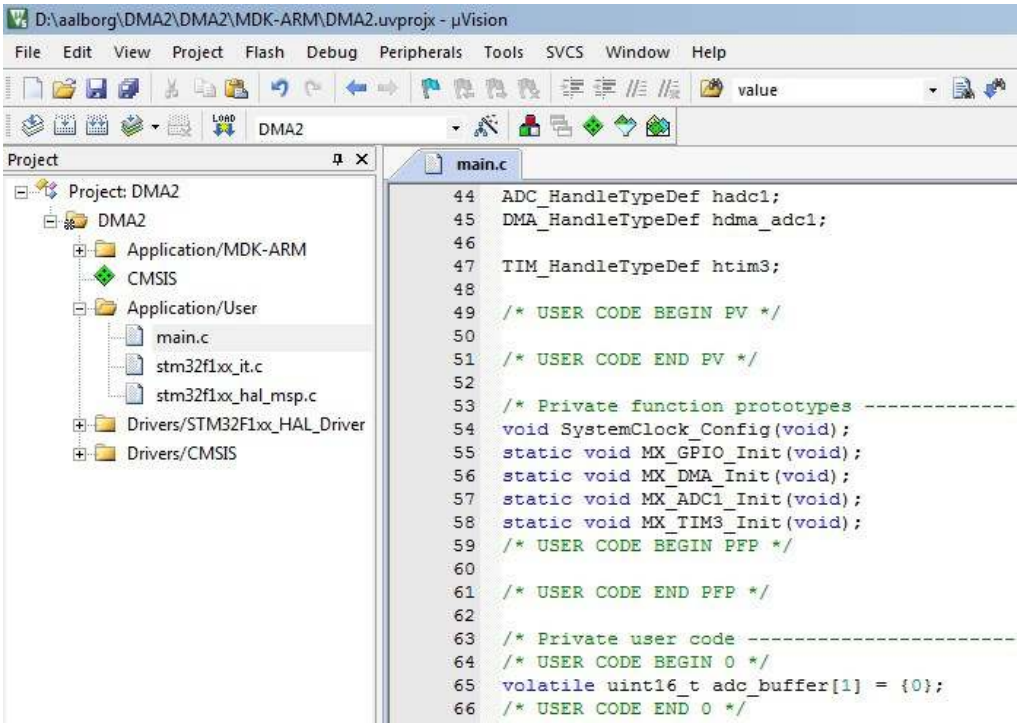

Figure 14.11: The header code of main.c

The code before the while loop and inside the main function is illustrated in Figure 14.12. Somewhere down in the main.c file we add the interrupt function for a completed ADC conversion as shown in Figure 14.13. We can finally compile the project and generate a hex code for programming the microcontroller.

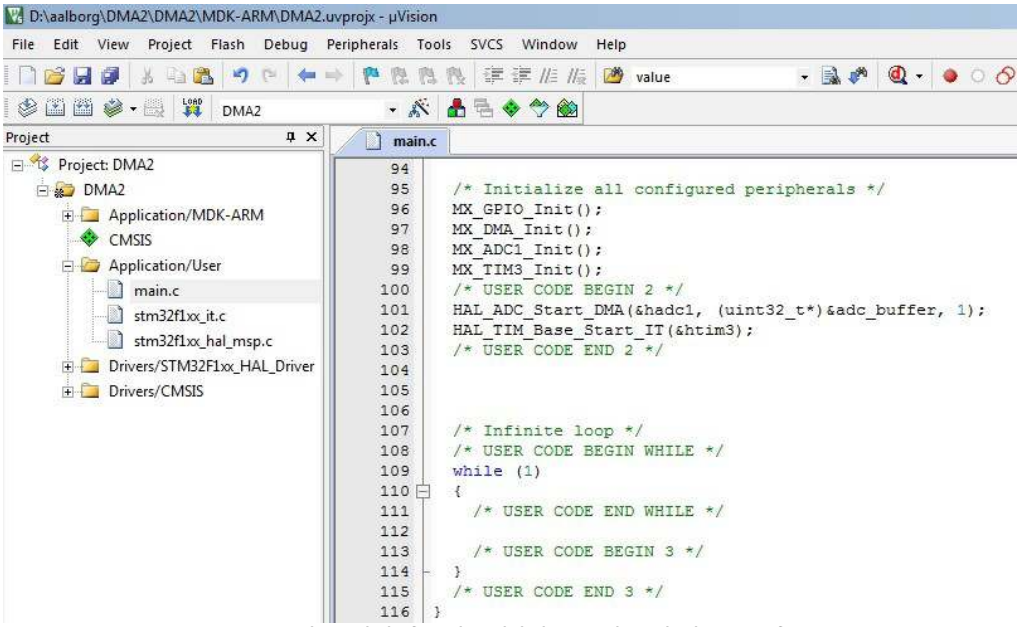

Figure 14.12: The code before the while loop and inside the main function

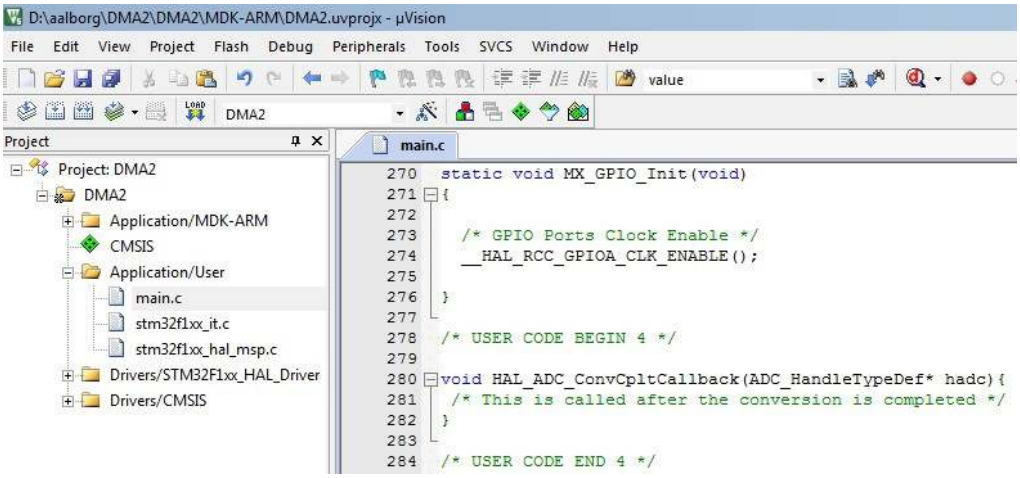

Figure 14.13: Adding the interrupt function for a completed ADC conversion

14.4 • DMA and USART Project Settings

In the final example, we will use DMA on the USART protocol. Also, make sure not to activate the USART1 interrupt. Cubemx will initiate DMA. If both interrupt and DMA are activated, the program will work as we want. After performing DMA configuration we will now configure USART. In Figure 14.14, set USART1 Mode as Asynchronous. The Parameter Settings tab for the USART1 protocol is shown in Figure 14.15.

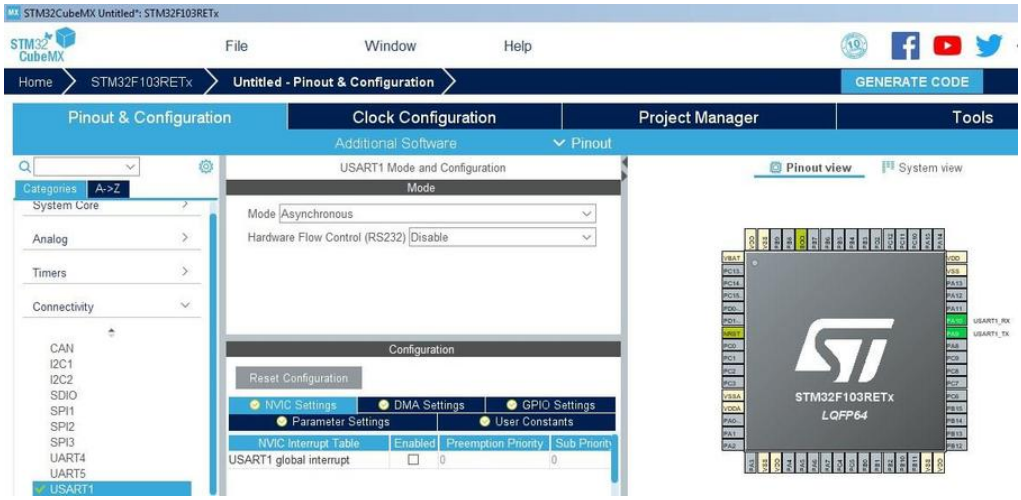

Figure 14.14: Setting USART1 Mode as asynchronous

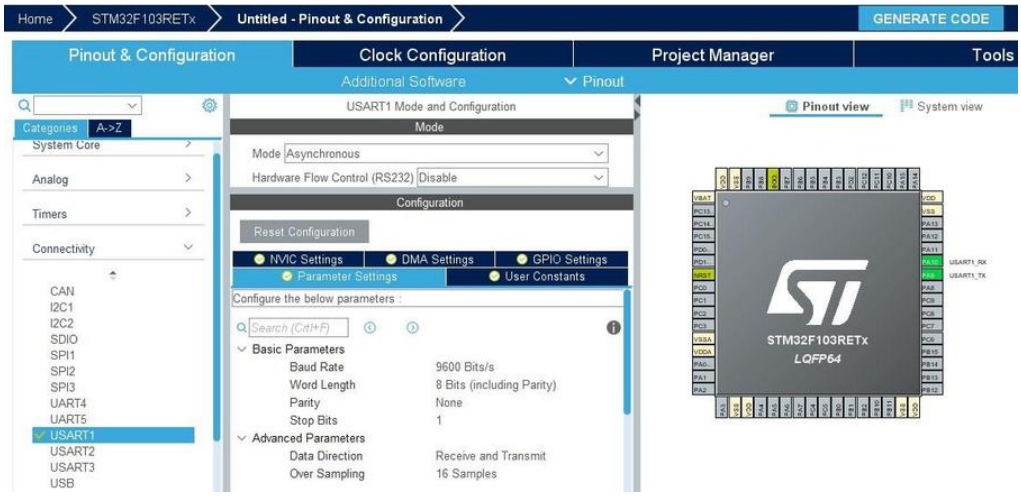

Figure 14.15: Parameter Settings for USART1

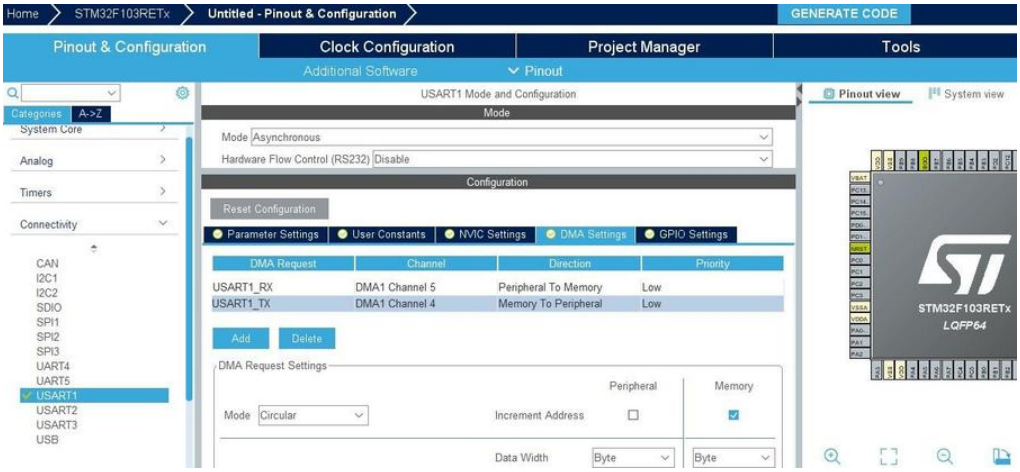

Figure 14.16: Adding DMA for USART1_RX and USART1_TX pins

From the DMA Settings tab, add DMA for the USART1_RX and USART1_TX pins as shown in figure 14.16. From the NVIC Settings of USART1 subsection, the DM1 global interrupts for USART1_RX and USART1_TX pins are enabled as shown in Figure 14.17. From the Project Manager tab, select MDK-ARM V5 as the compiler and generate the code in Keil uVision. The header code of the main.c file is shown in Figure 14.18.

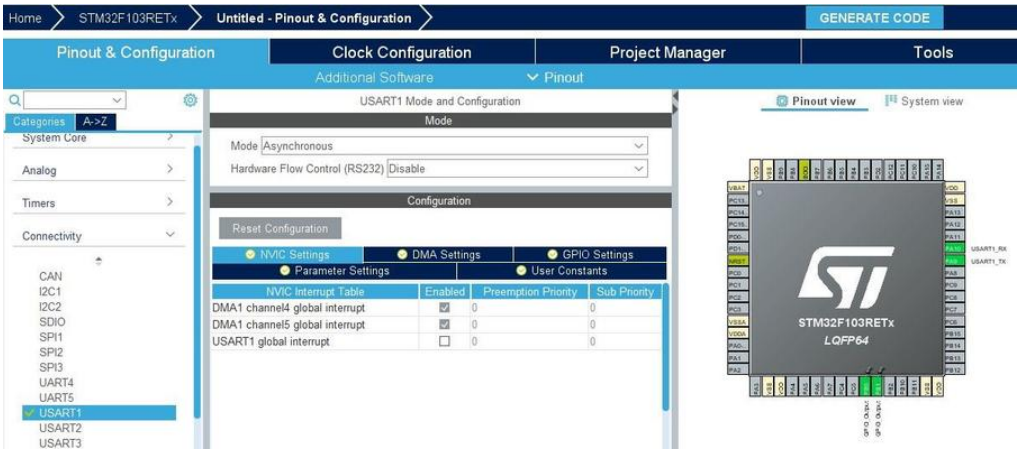

Figure 14.17: Enabling DM1 global interrupts

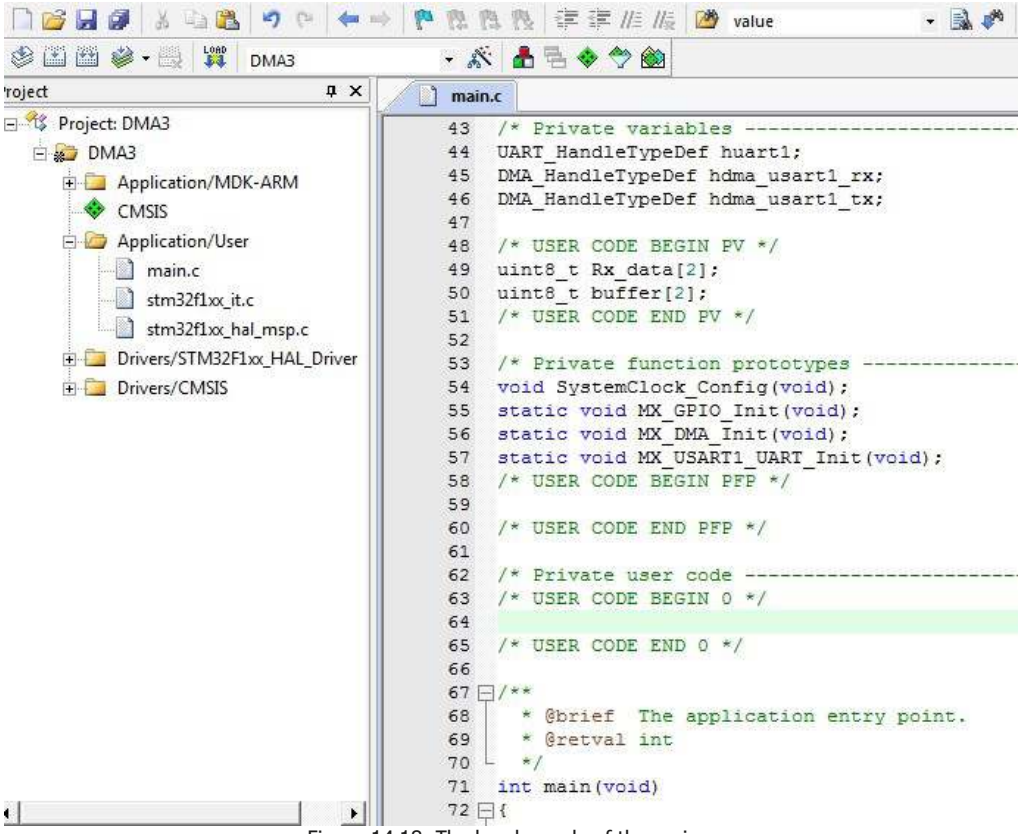

Figure 14.18: The header code of the main.c

The code inside the while loop is shown in Figure 14.19. At the end, somewhere down in the main.c file, add the interrupt function for a completed USART conversion as shown in Figure 14.20. We can finally compile the project and generate a hex code for programming the microcontroller.

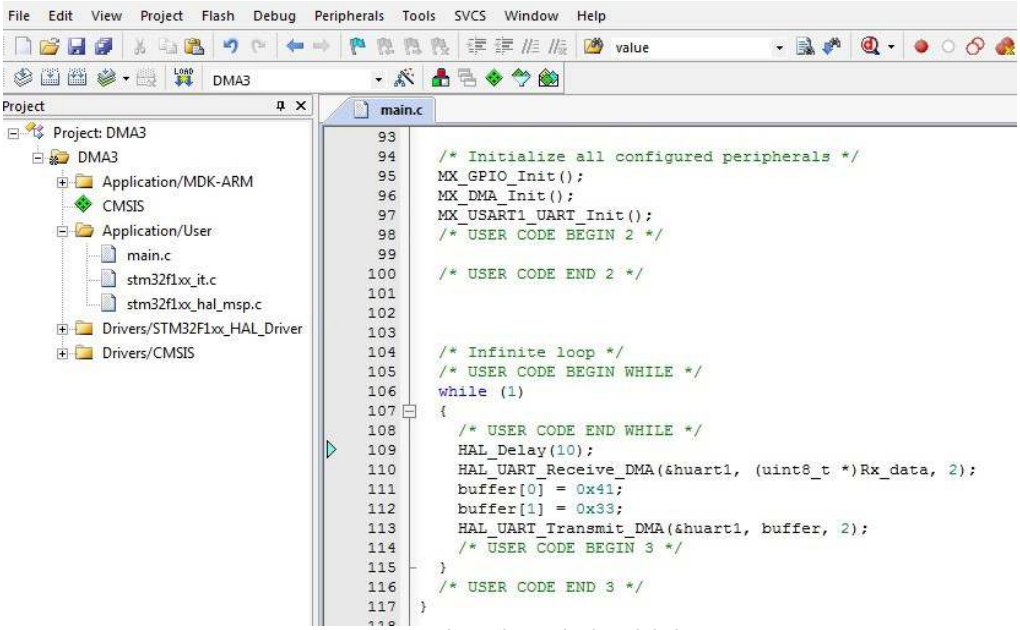

Figure 14.19: The code inside the while loop

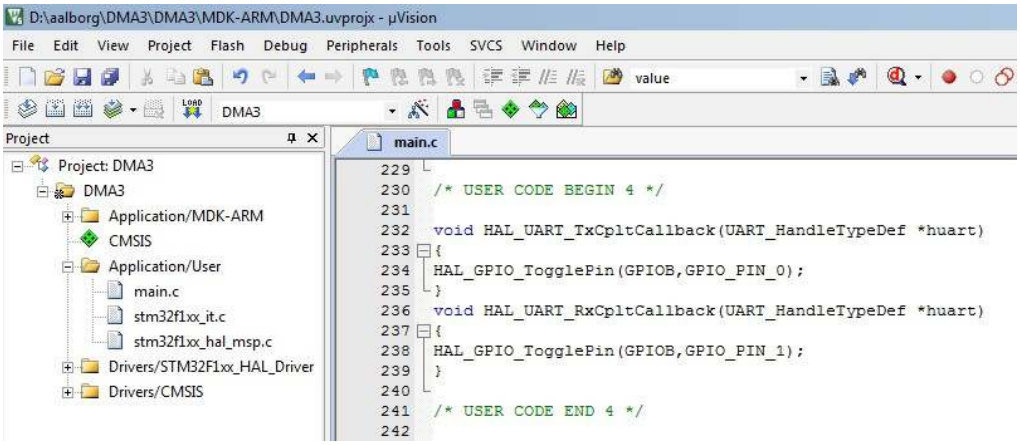

Figure 14.20: Adding interrupt functions for a completed USART conversion

14.5 • Summary

By using processor memory, a significant amount of processing time is lost. To avoid holding the CPU processing time, most advanced microcontrollers have a Direct Memory Access (DMA) controller. Data transfers using DMA are implemented between memory locations without the need for a CPU. In this chapter, DMA settings for different projects were discussed in detail. We evaluated three projects to demonstrate DMA applications in STM32 microcontroller programming.

Chapter 15 • Real-Time Operating System (RTOS) in STM32 Microcontrollers

15.1 • Introduction

Real-time systems come in a wide variety of implementations. This chapter focuses on how to use a real-time OS (RTOS) to create a real-time application on an STM32 microcontroller. Any system that has a deterministic response to a given event can be considered "real-time." If a system is considered to fail when it doesn't meet a timing requirement, it must be real-time. There are many different ways of achieving real-time behaviour. Hardware-based real-time systems can cover anything from analogue filters, closed-loop control, and simple state machines to complex video codecs. When implemented with power saving in mind, ASICs can be made to consume less power than MCU-based solutions. In general, hardware has the advantage of performing operations in parallel and instantly, as opposed to a single-core MCU, which only gives the illusion of parallel processing. The firmware that runs a scheduling kernel on an MCU is RTOS-based firmware. The introduction of the scheduler and some RTOS-primitives allows tasks to operate under the illusion they have the processor to themselves. Using an RTOS enables the system to remain responsive to the most important events while performing other complex tasks in the background. Software running on a full OS that contains a memory management unit (MMU) and central processing unit (CPU) is considered RTOS-based software. Applications that are implemented with this approach can be highly complex, requiring many different interactions between various internal and external systems. The advantage of using a full OS is all of the capability that comes along with it—both hardware and software. FreeRTOS is one of the most popular RTOS implementations for MCUs and is very widely available. It has been around for over 15 years and has been ported to dozens of platforms.

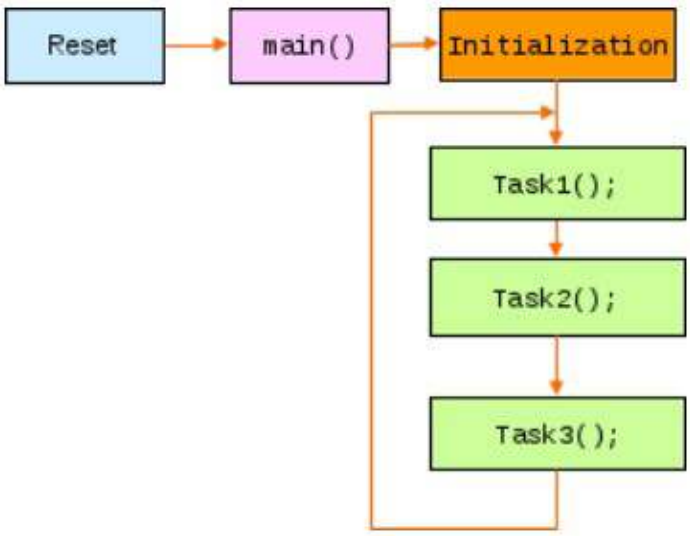

Figure 15.1: Flowchart of a multitasking programming model

The flowchart of a multitask programming model is shown in Figure 15.1.

15.2 • FreeRTOS Project Settings

In the following example, we are going to toggle two LEDs at different rates using FreeRTOS multitask programming model using STM32 microcontrollers. Firstly, from the system core section and SYS subsection, select the timebase Source as TIM1 as shown in Figure 15.2. We then enable the external crystal resonator as illustrated in Figure 15.3.

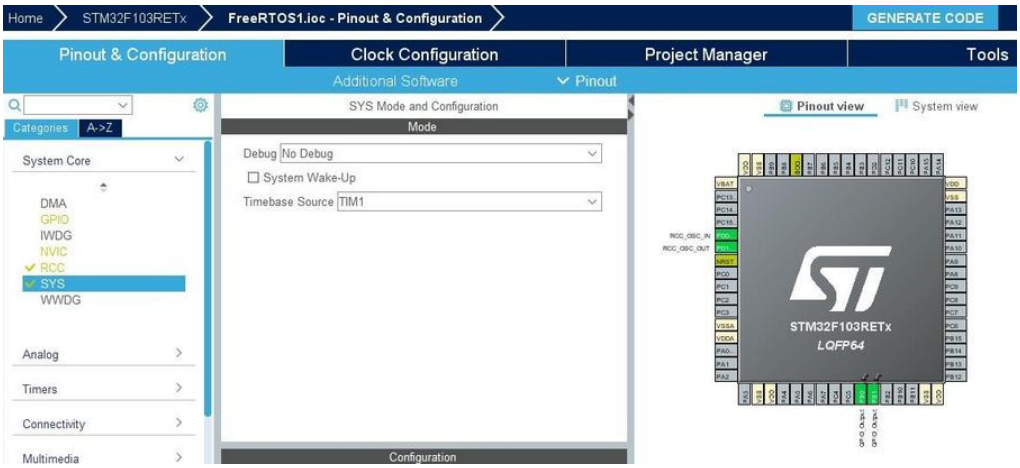

Figure 15.2: Selecting TIM1 for Timebase Source

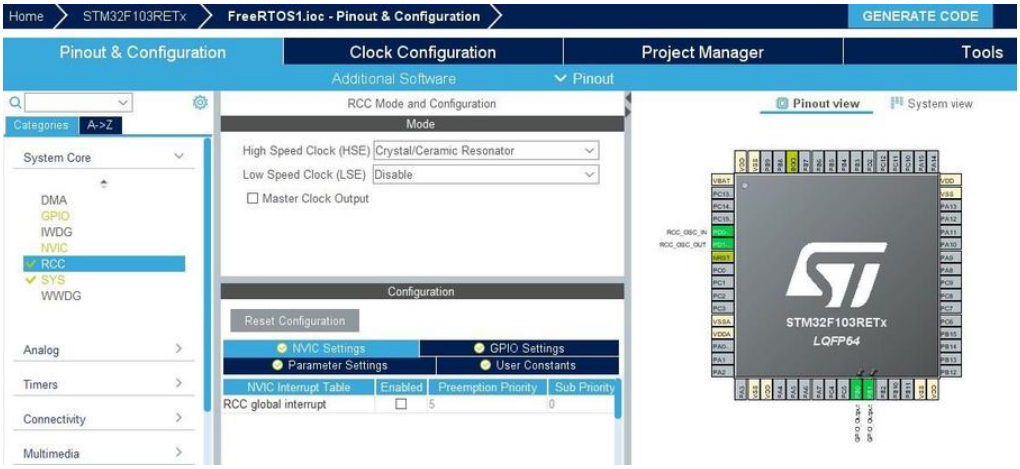

Figure 15.3: Enabling External Crystal Resonator

We also set the PB0 and PB1 pins as GPIO_Output and apply GPIO settings as demonstrated in Figure 15.4. From the middleware section and FREERTOS subsection, select the interface as CMSIS_V1 and keep the default settings for Config Parameters as shown in Figure 15.5.

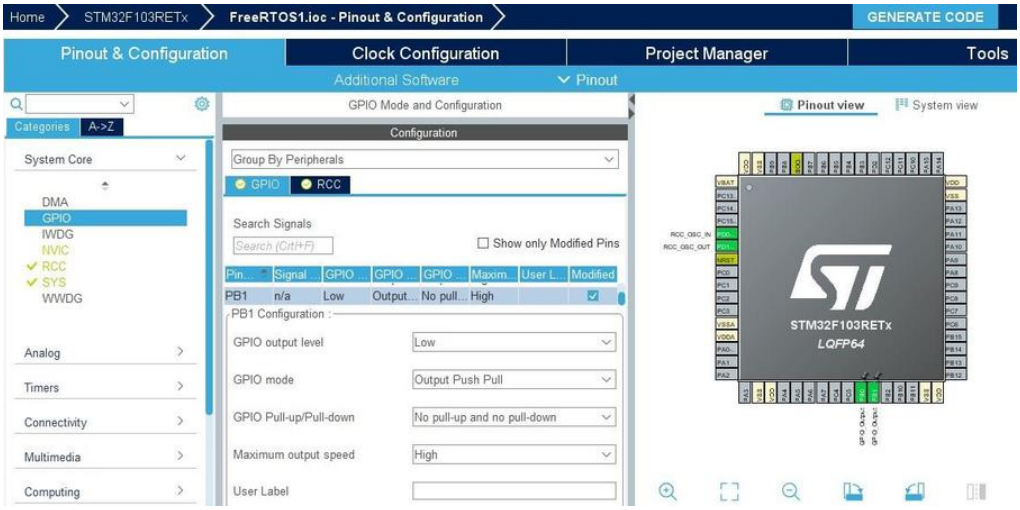

Figure 15.4: Setting PB0 and PB1 pins as GPIO_Output

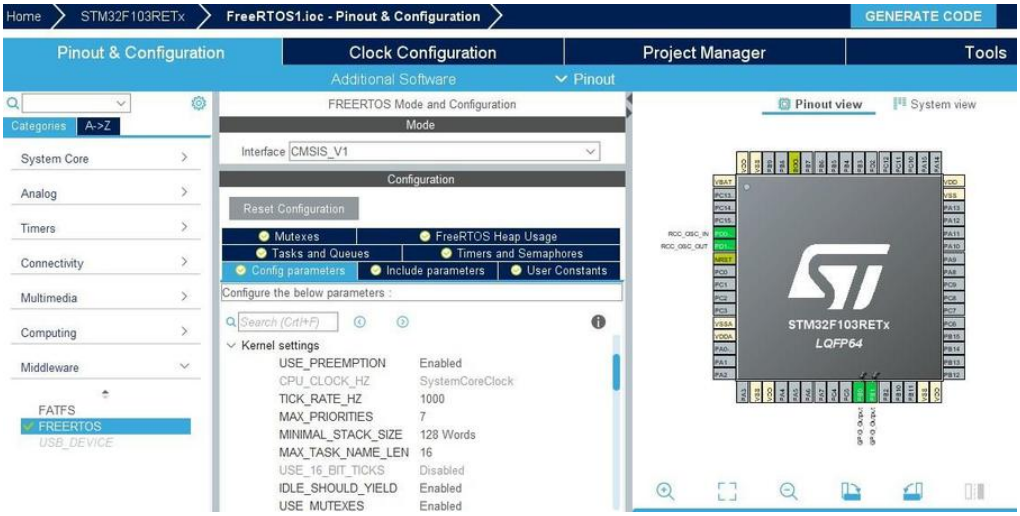

Figure 15.5: Selecting CMSIS_V1 as Interface and default settings for Config parameters

Finally, we set the Clock Configuration tab as shown in Figure 15.6. From the Project Manager tab, select 'MDK-ARM V5' as the compiler and generate the code in Keil uVision. The header code of main.c before the main function is shown in Figure 15.7. Code inside the main function and before while loop is illustrated in Figure 15.8. The code inside the two tasks for toggling two LEDs at different rates is shown in Figure 15.9. Finally, we can compile the project and generate a hex code for programming the microcontroller.

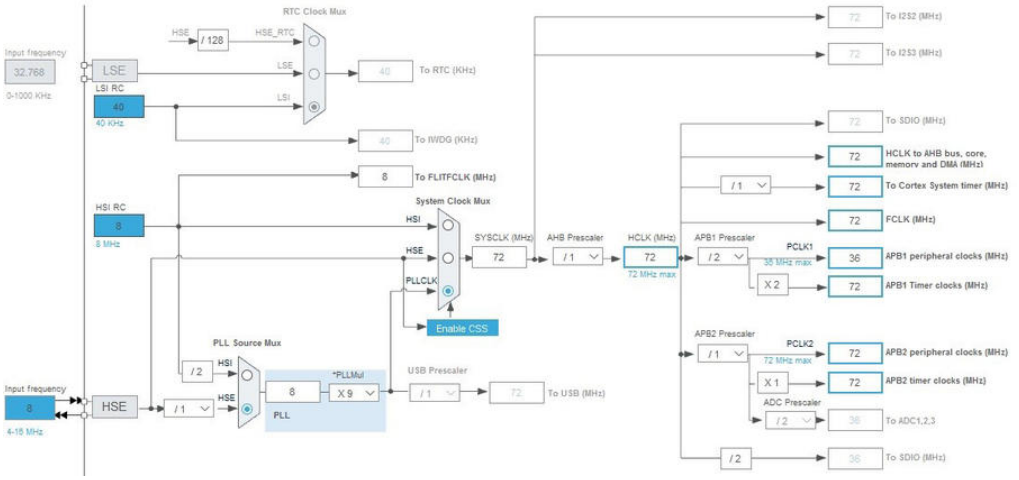

Figure 15.6: Setting the Clock Configuration tab

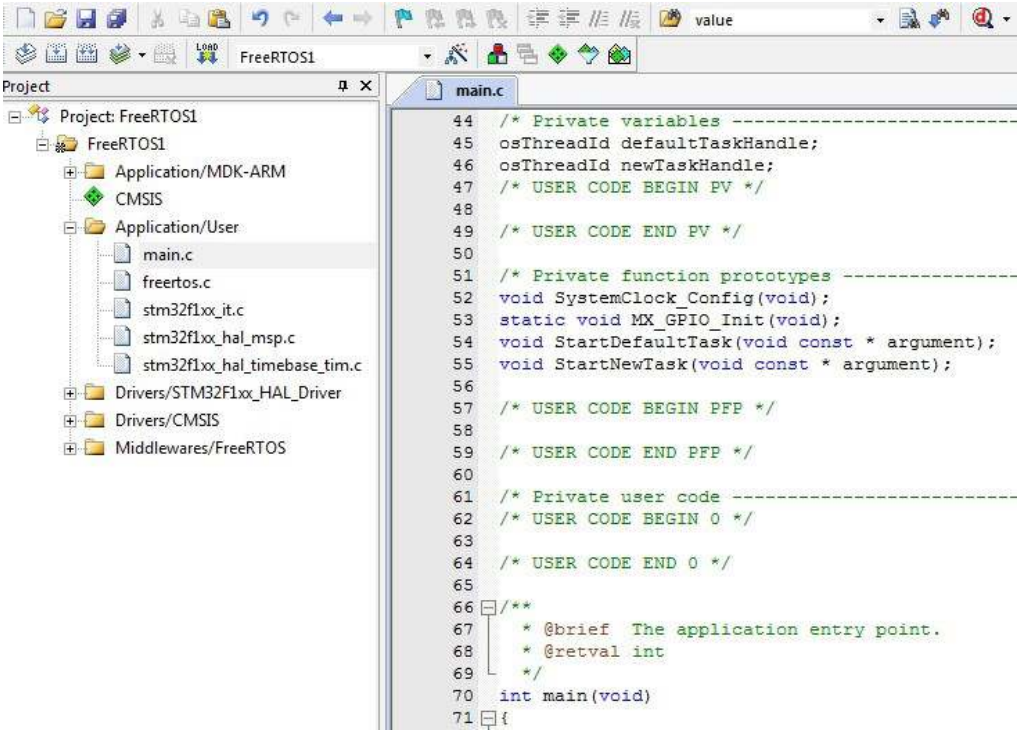

Figure 15.7: The header code before the main function

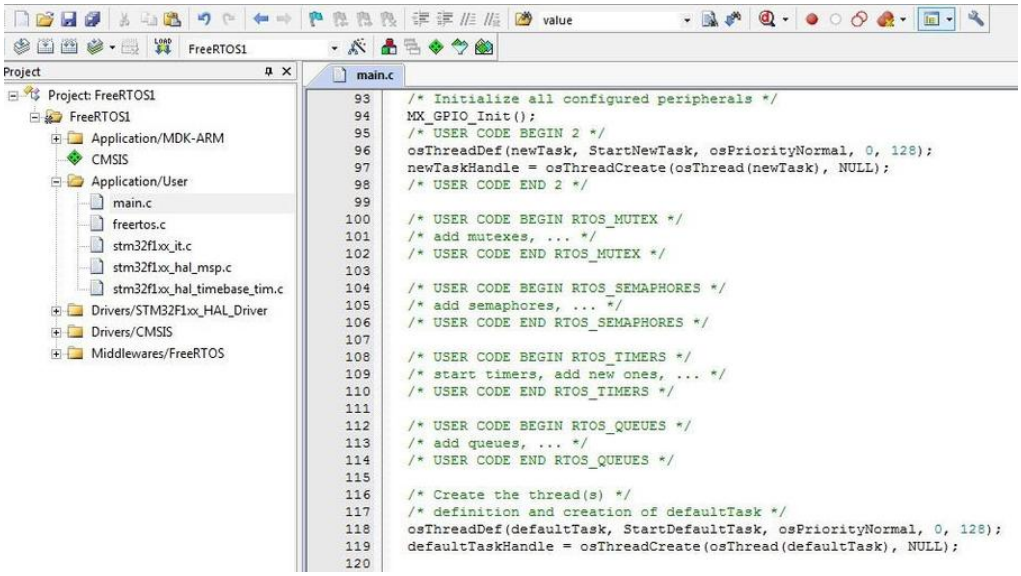

Figure 15.8: Code inside the main function

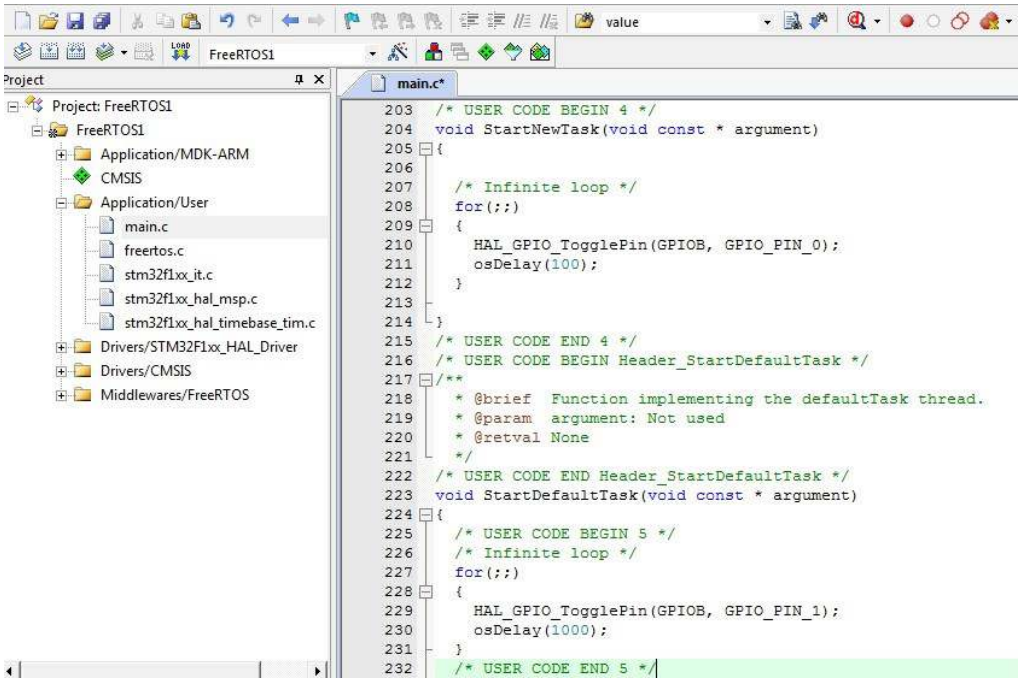

Figure 15.9: Code inside the two tasks for toggling LEDs at different rates

15.3 • Summary

Real-time systems come in a wide variety of implementations and use cases for real-time operations. In this chapter, we explained how to use a real-time OS (RTOS) to create a real-time application on an STM32 microcontroller. FreeRTOS is one of the most popular RTOS implementations for MCUs and is widely supported. It has been around for over 15 years, and has been ported to dozens of platforms.

Chapter 16 ● STM32 Microcontrollers Programming using STM32duino

16.1 ● Introduction

Arduino is an open-source electronics platform based on a simple hardware and software environment. Arduino boards can read inputs - for example, a light on a sensor, pressing a button, or message - and turn it into an output - activating a motor, turning on an LED, etc. Arduino boards can do what they do by programming the microcontroller on the board. The Arduino IDE is used for this. Arduino has been used in thousands of projects, from controlling everyday objects to complex scientific instruments. Arduino is open-source and there is a large support community. Arduino was born at the Ivrea Interaction Design Institute as an easy tool for fast prototyping, aimed at users without a background in electronics and programming. As soon as it was applied extensively worldwide, the Arduino board began to adapt to new needs and challenges. All Arduino boards are completely open-source, helping users build experimental boards independently and eventually adapt them to their particular needs. The Arduino IDE is open-source and is continually growing via a vast worldwide user-base.

The plcLib library presented by Walter Ditch for Arduino can be applied for sequential function chart (SFC) programming. For multitasking projects, the FreeRTOS library is also available for Arduino boards. In this chapter, an industrial automation example with an STM32 board is evaluated with SFC as a state machine programming method which is used extensively for programmable logic controller (PLC) programming. STM32 microcontrollers are from the STMicroelectronics microcontroller families. Therefore all existing methods to program an ARM chip can be used for the STM32 board. A frequently used IDE is the Keil ARM MDK. Other options include IAR workbench, Atollic TrueStudio, MicroC Pro ARM, Crossworks ARM, Ride 7, PlatformIO+STM32, etc.

The appeal of the STM32 microcontroller is its capacity to be programmed using the Arduino IDE which has many readily available libraries. In this chapter, we will use the Arduino IDE to program STM32 microcontrollers.

16.2 ● Installing STM32duino in Arduino IDE

In accordance with Figure 16.1 for installing the STM32duino library in the Arduino IDE environment, click the File tab and select preferences. In the preferences window, tick the compilation and output boxes (show verbose output during) to generate hex code for programming after successful compilation. Furthermore, in additional boards manager URL enter the following web link:

https://github.com/stm32duino/BoardManagerFiles/raw/master/STM32/package_stm_index.json

Click the OK button as shown in Figure 16.2. From the Tools tab and Board section, select the ‘boards manage’ icon as shown in Figure 16.3. In the boards manager window, choose

type as contributed as shown in Figure 16.4. Select STM32 Cores by STMicroelectronics and install by pressing the install button as shown in Figure 16.5. After installation, an INSTALLED message appears as shown in Figure 16.6.

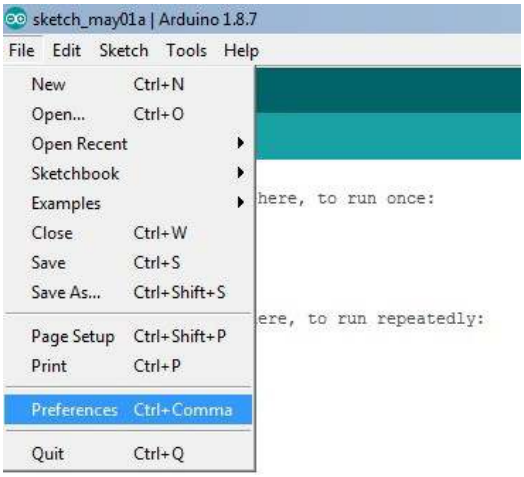

Figure 16.1: Selecting Preferences in the File tab

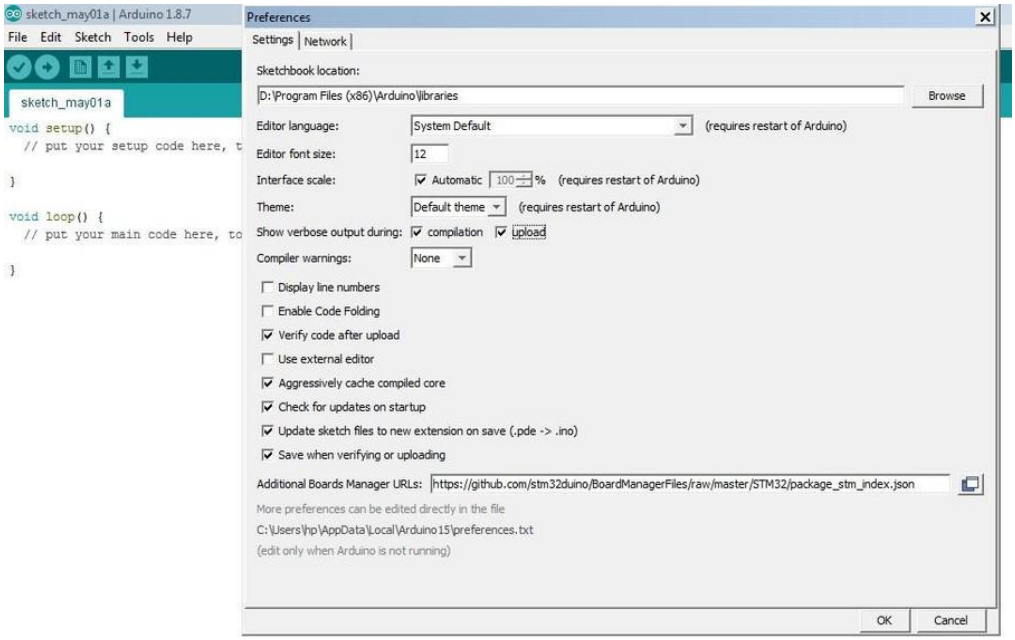

Figure 16.2: Preferences dialog settings

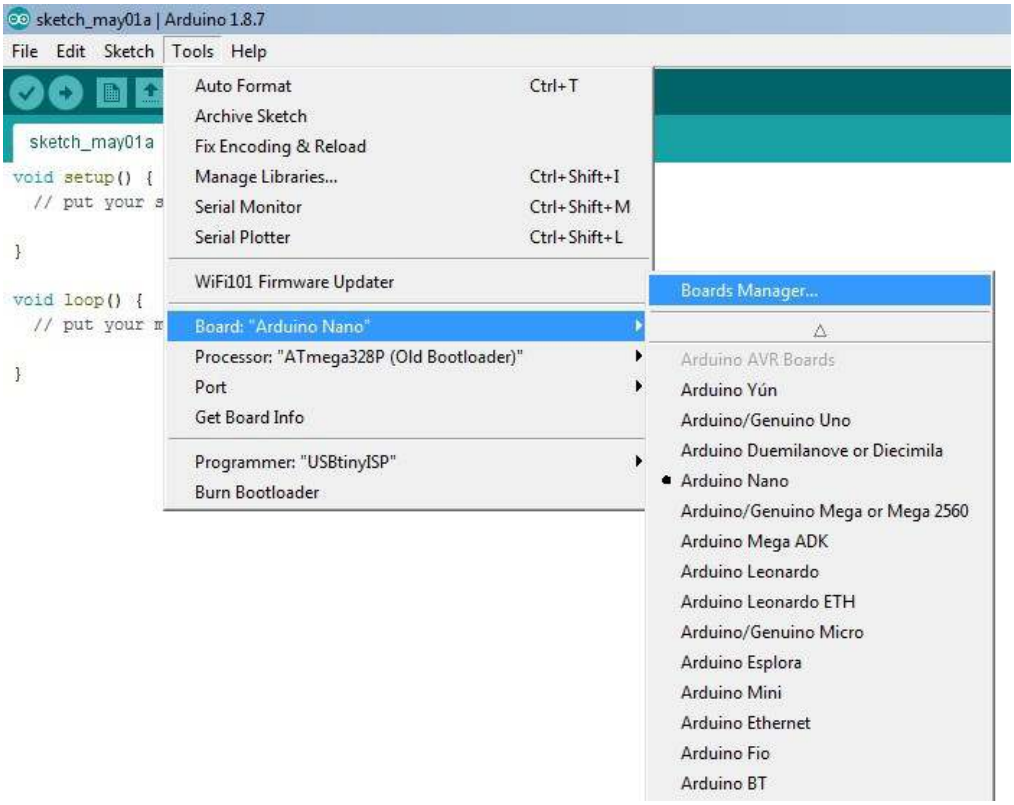

Figure 16.3: Selecting Boards Manager in the Tools tab

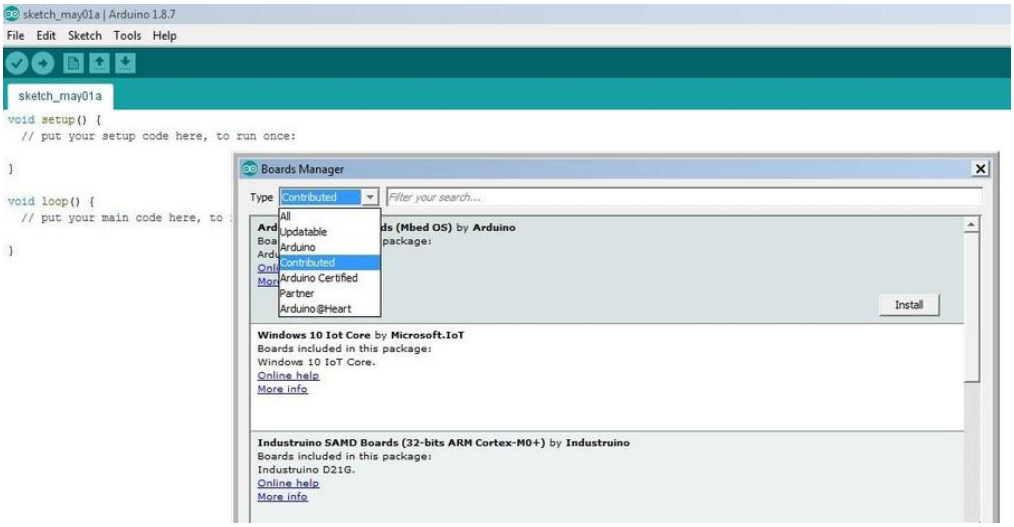

Figure 16.4: Selecting Type as Contributed in the Boards Manager window

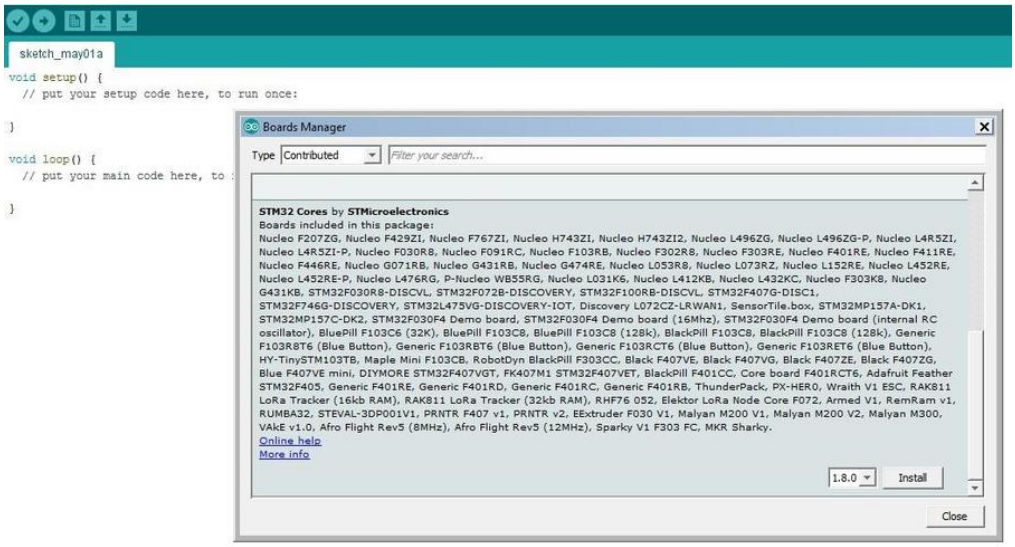

Figure 16.5: Installing STM32 Cores by STMicroelectronics

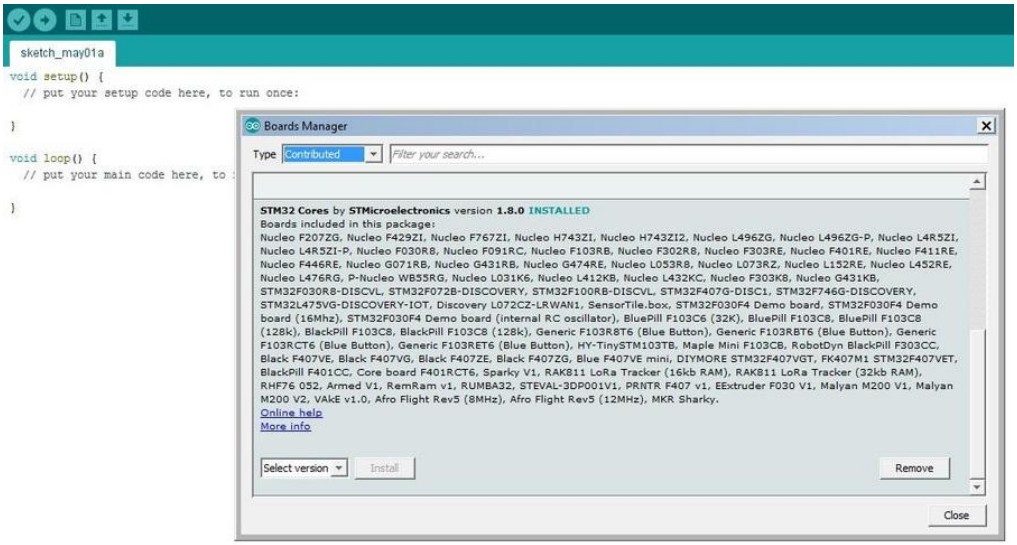

Figure 16.6: INSTALLED message after installation

We can now check the installed STM32 boards by clicking on the Tools tab, and selecting the board and choosing, for example, the Generic STM32F1 series as shown in Figure 16.7. If we click on the Tools tab again, we can see the Generic STM32F1 series board in the board part number section as illustrated in Figure 16.8. For flashing an LED connected to the PC13 pin of the STM32 blue pill board, the program is demonstrated in Figure 16.9. From the Sketch tab, select Verify/Compile to compile the project and generate the hex code for programming the microcontroller as shown in Figure 16.10. The hex code generated and

its accessing route message are shown in Figure 16.11.

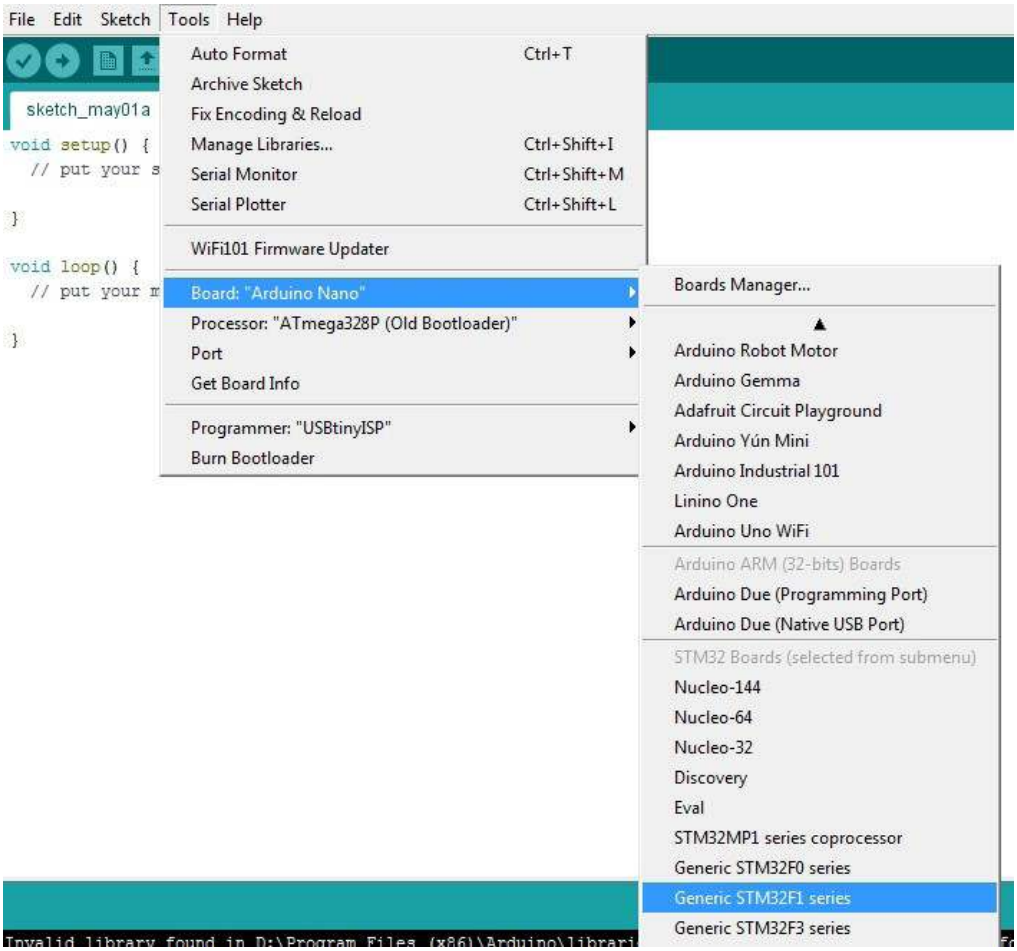

Figure 16.7: Choosing the Generic STM32F1 series from the board section

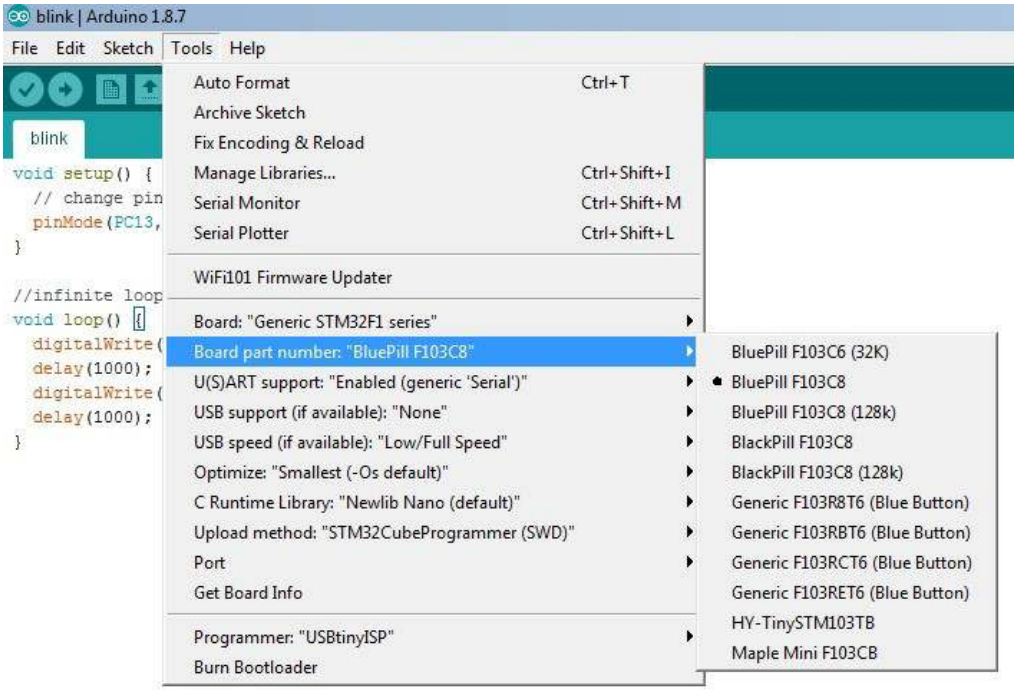

Figure 16.8: Selecting board part number in the Tools tab

Figure 16.9: Flashing an LED program

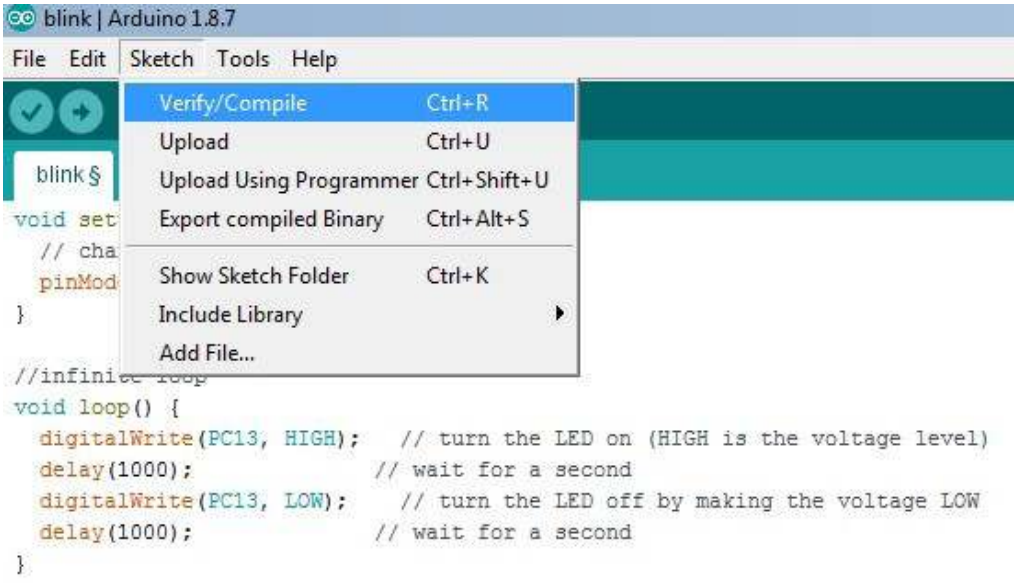

Figure 16.10: Compiling the program

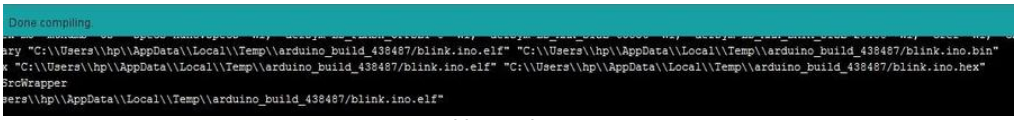

Figure 16.11: Generated hex code accessing route message

As another example of reading an analogue value from the PB0 pin, we can write a program as shown in Figure 16.12. After a successful compilation of the program, the hex file generated (for programming the STM32 microcontroller) route message appears as shown in Figure 16.13.

Figure 16.12: Reading an analogue value program

Figure 16.13: Generated hex code accessing route message

16.3 • Automation Control System Programming

We are now going to program an industrial automation example with an STM32 board using SFC as a state machine programming method called plcLib (PLC library for Arduino IDE). This is used extensively for programmable logic controller (PLC) programming. The order of performance of the system shown in Figure 16.14 is as follows:

- #The box is in LS1 position (LS1 closed).
- By pressing the start button, the conveyor motor starts and the box moves to position A (LS1 opens).
- The conveyor moves the box to position A and then stops (the position is detected by

eight off-to-on pulses from the encoder to up counter).

- After a 10-second delay, the conveyor starts to move and the box moves to LS2 and stops (LS2 closes).
- An emergency stop pushbutton is used to stop the process at any time.
- If the process is stopped by the emergency stop push button, the timer and counter will automatically reset.

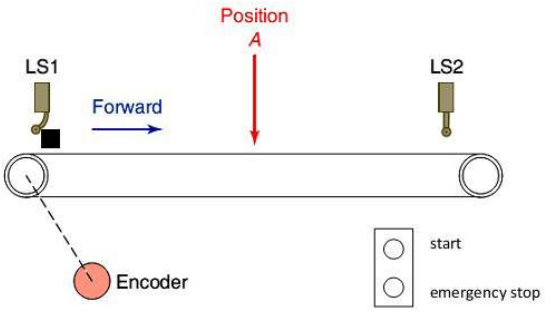

Figure 16.14: An industrial automation system

The SFC diagram which is also called the state transition diagram is demonstrated in Figure 16.15. Further information about drawing SFC diagrams for controlling industrial automation systems and other programming methods is given in the book entitled “Advanced PLC Programming” authored by Majid Pakdel. The following code is used in the Arduino IDE environment to implement the above automation system with SFC programming. The default pins which have been defined in the plcLib library are not suitable for controlling this automation system. We need 5 inputs and one output so use CustomIO in the library examples to define pins as necessary in this automation control example. The CustomIO. ino file is as follows.

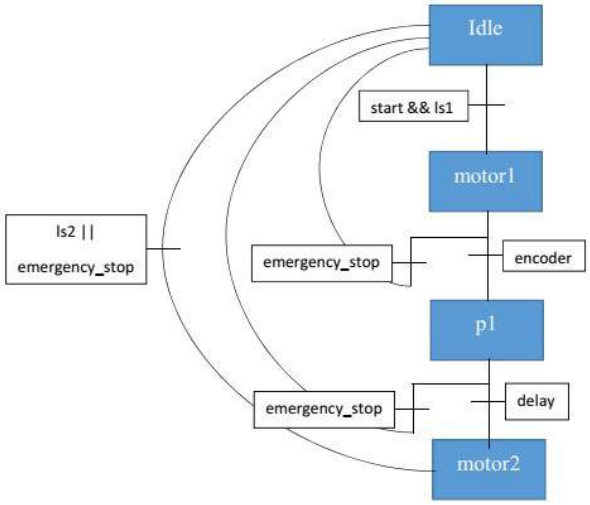

Figure 16.15: The SFC (state transition) diagram of the system

```
#define noPinDefs
#include <plcLib.h>
/* Programmable Logic Controller Library for the Arduino and Compatibles
   Custom I/O - Define your own inputs and outputs from scratch
   Begin by including the line '#define noPinDefs' at the start, which prevents
   creation of standard inputs and output names X0, X1, ..., Y0, Y1, ... etc.
   (This option is available in plcLib V1.2.0 or greater.)
   The 'setupPLC();' command is not needed if you are using custom inputs
   and outputs.
   You can then define your own pin names, as required. Any 'unsigned integer'
   variables are treated as user defined variables, while 'integers' refer to
   local pin numbers. For example: -
        "unsigned int X1, Y1;" creates X1 and Y1 as user defined variables
        "int X0=A0, Y0=3, Y2=6;" creates X0, Y0 and Y2 as local pin names
   The next step is to define pins as either inputs or outputs by using the
   "pinMode(pin, mode)" command. Sample configuration code has been placed in
   the 'customIO' function which is available from the IO tab and called from
   within the 'setup()' function below.
   To enable remote monitoring of inputs and outputs, begin by enabling the
   serial port in the setup(); section. E.g. "Serial.begin(9600);"
      Next, include the serial monitor command in the loop() section. E.g.
        "serialMonitor("YourCircuitBoardNameHere");"
   Software and Documentation:
   https://github.com/wditch/plcLib
*/
unsigned int Idle = 1;        // Create variables
unsigned int motor1 = 0;
unsigned int p1 = 0;
unsigned int motor2 = 0;
unsigned int cout = 0;
int X0=PB3, X1=PB4, X2=PB5, X3=PB6, X4=PB7, Y0=PB8;     // Create local pins
//X0=start, X1=ls1, X2=ls2, X3=encoder, X4=emergency_stop, Y0=motor1||motor2
Counter ctr(8); // Final count = 8, starting at zero
//unsigned long TIMER0 = 0; // Define variable used to hold timer 0 elapsed //time
unsigned long DELAY0 = 0;
void setup() {
  customIO();              // Setup input and output pin directions (See IO tab)
 // Serial.begin(9600);  // Enable serial port (needed for serial IO monitor)
}
void loop() {
  in(Idle); // Read Start-up state
  andBit(X0); // AND with Step 1 transition input
  andBit(X1);
  set(motor1); // Activate Step 1
  reset(Idle); // Cancel Start-up state
  in(motor1);
```

```
  andBit(X4);
  ctr.clear();
  set(Idle);
  reset(motor1);
  in(motor1);
  andBit(X3);
  //timerOn(TIMER0, 10); // 10 ms switch debounce delay
  ctr.countUp(); // Count up
  ctr.upperQ(); // Display Count Up output on Y1
  out(cout);
  andBit(cout); // Read Input 0
  set(p1); // Activate Step 1
  reset(motor1); // Cancel Start-up state
  in(p1);
  andBit(X4);
  ctr.clear();
  set(Idle);
  reset(p1);
  in(p1);
  timerOn(DELAY0, 10000);
  set(motor2); // Activate Step 1
  reset(p1); // C

  in(motor2);
  andBit(X4);
  ctr.clear();
  set(Idle);
  reset(motor2);
  in(motor2);
  andBit(X2);
  ctr.clear();
  set(Idle);
  reset(motor2);
  in(motor1);
  orBit(motor2);
  out(Y0);
  //serialMonitor("YourCircuitBoardNameHere");  // Enable remote I/O monitoring //
via the serial port
}
```

The IO.ino file is given as follows:

```
void customIO() {
  // Input pin directions
  pinMode(X0, INPUT_PULLDOWN);
  pinMode(X1, INPUT_PULLDOWN);
  pinMode(X2, INPUT_PULLDOWN);
  pinMode(X3, INPUT_PULLDOWN);
  pinMode(X4, INPUT_PULLDOWN);
  // Output pin directions
  pinMode(Y0, OUTPUT);
  // Note that Xs and Ys are variables so
  // no pinMode definitions are required
}
```

16.4 • Multitasking in PLC Programming

For multitasking projects, we can install the FreeRTOS library for STM32duino as shown in Figure 16.16. Alternatively, we can also download the STM32duino FreeRTOS library from:

https://www.arduinolibraries.info/libraries/stm32duino-free-rtos

From the Sketch tab and Include Library icon, select the Add.ZIP library, as illustrated in Figure 16.17. In the following example, we are going to implement two tasks: blinking an LED and controlling the automation system demonstrated in Figure 16.14. Therefore we use both the plcLib and STM32FreeRTOS libraries in the project. The CustomIO.ino file is as follows:

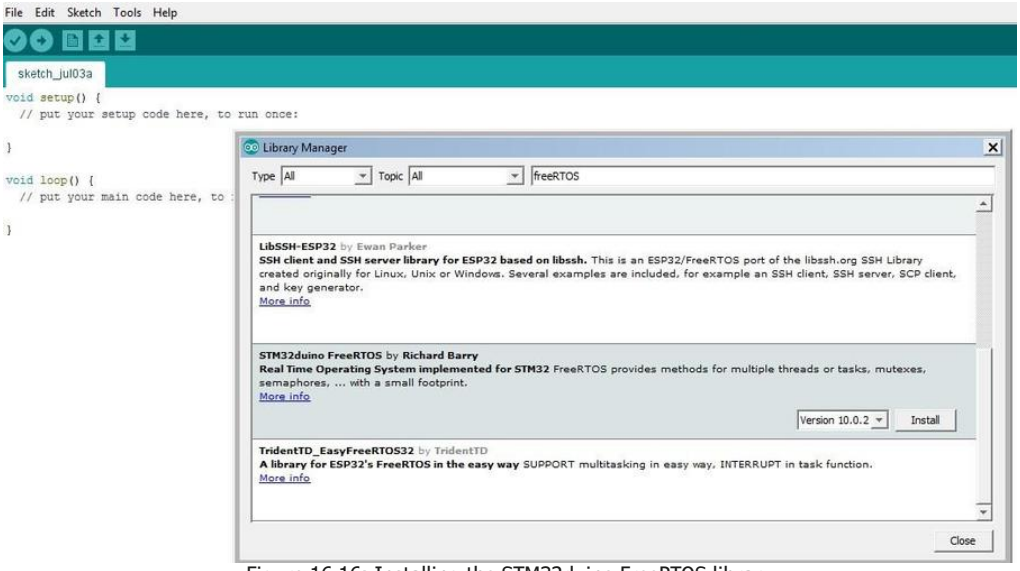

Figure 16.16: Installing the STM32duino FreeRTOS library

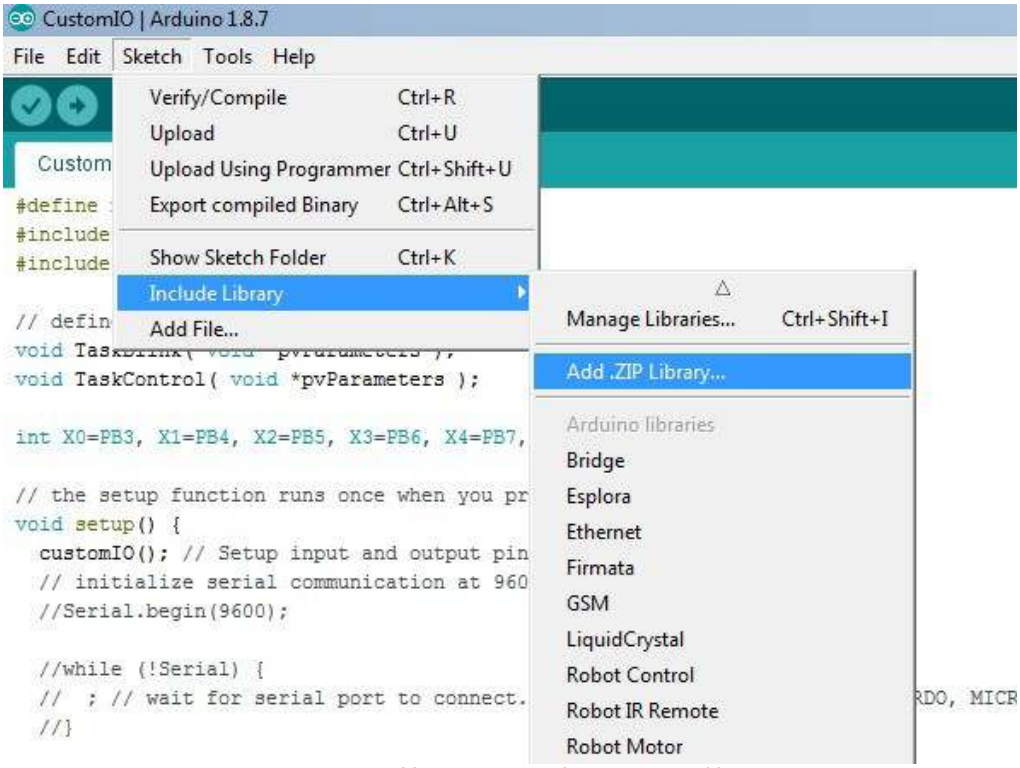

Figure 16.17: Adding .ZIP STM32duino FreeRTOS library

```
#define noPinDefs
#include <plcLib.h>
#include <STM32FreeRTOS.h>
// define two tasks for Blink & Control
void TaskBlink( void *pvParameters );
void TaskControl( void *pvParameters );

int X0=PB3, X1=PB4, X2=PB5, X3=PB6, X4=PB7, Y0=PB8; // Create local pins

// the setup function runs once when you press reset or power the board
void setup() {
  customIO(); // Setup input and output pin directions (See IO tab)
  // initialize serial communication at 9600 bits per second:
  //Serial.begin(9600);

  //while (!Serial) {
  //  ; // wait for serial port to connect. Needed for native USB, on LEONARDO,
MICRO, YUN, and other 32u4 based boards.
  //}
```

```
  // Now set up two tasks to run independently.
  xTaskCreate(
    TaskBlink
    ,  (const portCHAR *)"Blink"   // A name just for humans
    ,  128  // This stack size can be checked & adjusted by reading the Stack
Highwater
    ,  NULL
    ,  1  // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0
being the lowest.
    ,  NULL );

  xTaskCreate(
    TaskControl
    ,  (const portCHAR *)"Control"
    ,  128  // Stack size
    ,  NULL
    ,  1  // Priority
    ,  NULL );

  // start scheduler
  vTaskStartScheduler();
  //Serial.println("Insufficient RAM");
  while(1);
}

void loop()
{
  // Empty. Things are done in Tasks.
}
/*--------------------------------------------------*/
/*---------------------- Tasks ---------------------*/
/*--------------------------------------------------*/

void TaskBlink(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the UNO, LEONARDO, MEGA,
and ZERO
  it is attached to digital pin 13, on MKR1000 on pin 6. LED_BUILTIN takes care
  of use the correct LED pin whatever is the board used.
```

```
  The MICRO does not have a LED_BUILTIN available. For the MICRO board please
substitute
  the LED_BUILTIN definition with either LED_BUILTIN_RX or LED_BUILTIN_TX.
  e.g. pinMode(LED_BUILTIN_RX, OUTPUT); etc.

  If you want to know what pin the on-board LED is connected to on your Arduino
model, check
  the Technical Specs of your board  at https://www.arduino.cc/en/Main/Products

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald

  modified 2 Sep 2016
  by Arturo Guadalupi
*/
  // initialize digital LED_BUILTIN on pin 13 as an output.
  pinMode(PC13, OUTPUT);

  for (;;) // A Task shall never return or exit.
  {
    digitalWrite(PC13, HIGH);   // turn the LED on (HIGH is the voltage level)
    vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one second
    digitalWrite(PC13, LOW);    // turn the LED off by making the voltage LOW
    vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one second
  }
}

void TaskControl(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  unsigned int Idle = 1; // Create variables
  unsigned int motor1 = 0;
  unsigned int p1 = 0;
  unsigned int motor2 = 0;
  unsigned int cout = 0;
  int X0=PB3, X1=PB4, X2=PB5, X3=PB6, X4=PB7, Y0=PB8; // Create local pins
  //X0=start, X1=ls1, X2=ls2, X3=encoder, X4=emergency_stop, Y0=motor1||motor2
  Counter ctr(8); // Final count = 8, starting at zero
//unsigned long TIMER0 = 0; // Define variable used to hold timer 0 elapsed //time
  unsigned long DELAY0 = 0;
  //customIO(); // Setup input and output pin directions (See IO tab)
  for (;;)
  {
```

```
    in(Idle); // Read Start-up state
    andBit(X0); // AND with Step 1 transition input
    andBit(X1);
    set(motor1); // Activate Step 1
    reset(Idle); // Cancel Start-up state
    in(motor1);
    andBit(X4);
    ctr.clear();
    set(Idle);
    reset(motor1);
    in(motor1);
    andBit(X3);
    //timerOn(TIMER0, 10); // 10 ms switch debounce delay
    ctr.countUp(); // Count up
    ctr.upperQ(); // Display Count Up output on Y1
    out(cout);
    andBit(cout); // Read Input 0
    set(p1); // Activate Step 1
    reset(motor1); // Cancel Start-up state
    in(p1);
    andBit(X4);
    ctr.clear();
    set(Idle);
    reset(p1);
    in(p1);
    timerOn(DELAY0, 10000);
    set(motor2); // Activate Step 1
    reset(p1); // C
    in(motor2);
    andBit(X4);
    ctr.clear();
    set(Idle);
    reset(motor2);
    in(motor2);
    andBit(X2);
    ctr.clear();
    set(Idle);
    reset(motor2);
    in(motor1);
    orBit(motor2);
    out(Y0);
  }
}
```

The IO.ino file is given below.

```
void customIO() {
```

```
  // Input pin directions
pinMode(X0, INPUT_PULLDOWN);
pinMode(X1, INPUT_PULLDOWN);
pinMode(X2, INPUT_PULLDOWN);
pinMode(X3, INPUT_PULLDOWN);
pinMode(X4, INPUT_PULLDOWN);
// Output pin directions
pinMode(Y0, OUTPUT);
  // Note that X1 and Y1 are variables so
  // no pinMode definitions are required
}
```

16.5 ● Summary

In this chapter, an industrial automation example with an STM32 board was implemented using a sequential function chart as a finite state machine programming method. This is used extensively for programmable logic controller (PLC) programming. The plcLib library presented by Walter Ditch for Arduino IDE was used for sequential function chart (SFC) programming. For undertaking multitasking projects, the FreeRTOS library was included in STM32duino (Arduino IDE environment) and simultaneous operations of two different tasks were successfully implemented. Therefore, low-cost PLCs with simple programming methods can be undertaken using STM32 microcontrollers.

Chapter 17 • Finite State Machine (FSM) Programming in STM32 Microcontrollers

17.1 • Introduction

A common design technique of industrial automation control systems is the venerable finite state machine (FSM). Industrial automation control designers use this programming method to break complex problems into manageable states and transitions. FSM programming is a model of a hypothetical machine made of one or more states and transitions. Only a single state can be active at any time and therefore the machine should transition from one state to another in order to perform different actions as required. The FSM model consists of three main parts, as shown in Figure 17.1.

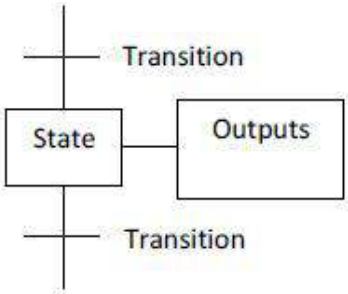

Figure 17.1: Main components of the FSM model

The initial state in the FSM model is represented by two nested rectangles. The initial state relates to the state from which the system starts, as shown in Figure 17.2.

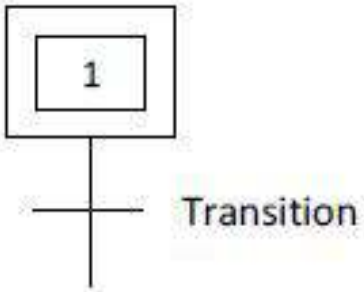

Figure 17.2: Initial state of the FSM model

Finite State Machine is a graphical method for describing program structure with modular concurrent control systems. FSM only describes the structured organization of program modules and the programs themselves must be written in one of the available programming languages.

The Basic FSM Structures are as follows:

A) States

Each state is a control module that can be programmed. There are two types of states: **initial** and **regular**. The initial state is performed on the first run and after the reset, and the regular state occurs when the transition logic is enabled. When a state is disabled, its state is initialized and only active states are evaluated during scanning the program.

B) Transitions

Each transition is a state-controlled control module that ultimately evaluates a transition variable. When the transition variable is correct, the following states are activated and the preceding states are disabled. Only transitions followed by active states are evaluated. A transition can be a simple variable value or combination of logical variables.

Also, the program control structures are as below:

A) Selective/Alternative Branch:

This structure is shown in Figure 17.3. If T1 and T2 are simultaneously activated, priority will be left (T1) and the next state will be S2. If more than one transition variable is true, the priority is left to right. Only one branch is active at a time. If S4 is enabled and T3 is true, S6 is enabled and S4 disabled.

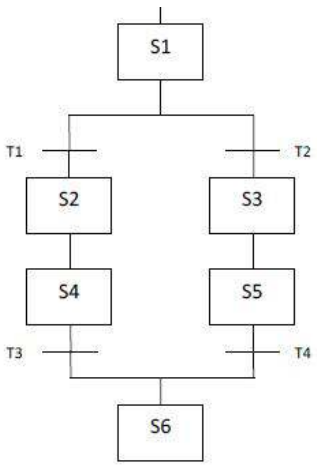

Figure 17.3: Selective/Alternative Branch

B) Simultaneous/Parallel Branch:

This structure is shown in Figure 17.4 and is the structure used for concurrent control programming. If S1 is active and T1 is true, S2 and S3 are enabled and S1 is disabled. If

S4 and S5 are active and T2 is true, S6 is activated and S4 and S5 are disabled.

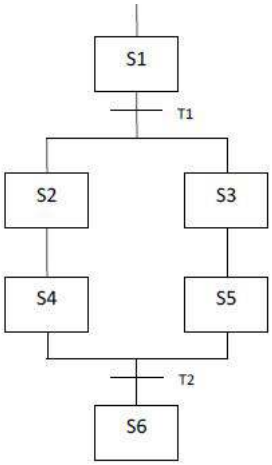

Figure 17.4: Simultaneous/Parallel Branch

17.2 • FSM programming using mikroC PRO for ARM

We are now going to implement FSM programming on STM32 microcontrollers. In this chapter, we will demonstrate how to write FSM programming using mikroC PRO for ARM and Mbed online compiler software environments. Consider the following industrial automation system as described below (This industrial automation example was before evaluated in chapter 16 and is given here again to write its control programming using the FSM method).

The order of performance of the system shown in Figure 17.5 is as follows:

- The box is in LS1 position (LS1 closed).
- By pressing the start button, the conveyor motor starts and the box moves to position A (LS1 opens).
- The conveyor moves the box to position A and stops (the position is detected by eight off-to-on pulses from the encoder to up counter).
- After a 10-second delay, the conveyor starts to move and the box moves to LS2 and stops (LS2 closes).
- An emergency stop pushbutton is used to stop the process at any time.
- If the process is stopped by the emergency stop push button, the timer and counter will reset automatically.

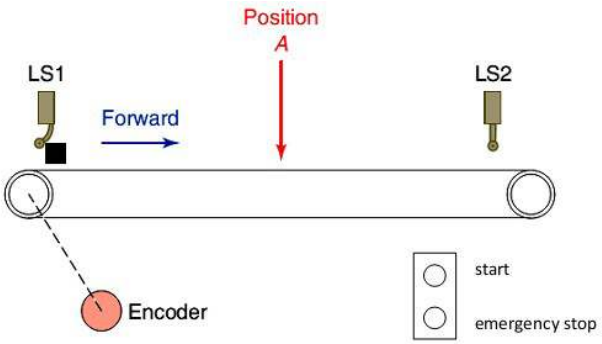

Figure 17.5: An industrial automation system

The finite state machine (FSM) diagram which is also called the state transition diagram is shown in Figure 17.6.

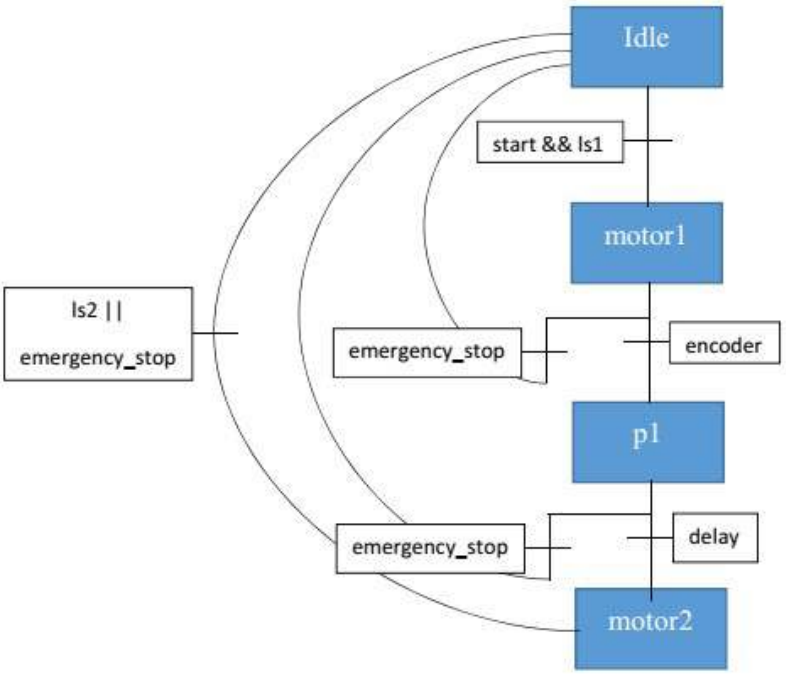

Figure 17.6: The FSM (state transition) diagram of the system

mikroC PRO for ARM is a C compiler for writing fast multimedia applications for ARM Cortex M3 and M4 devices using the mikroC programming environment. By clicking on the New Project icon, a window pops up as shown in Figure 17.7. In the New Project Wizard window, select 'standard project' as the project type and click on the next button as shown in Figure 17.8.

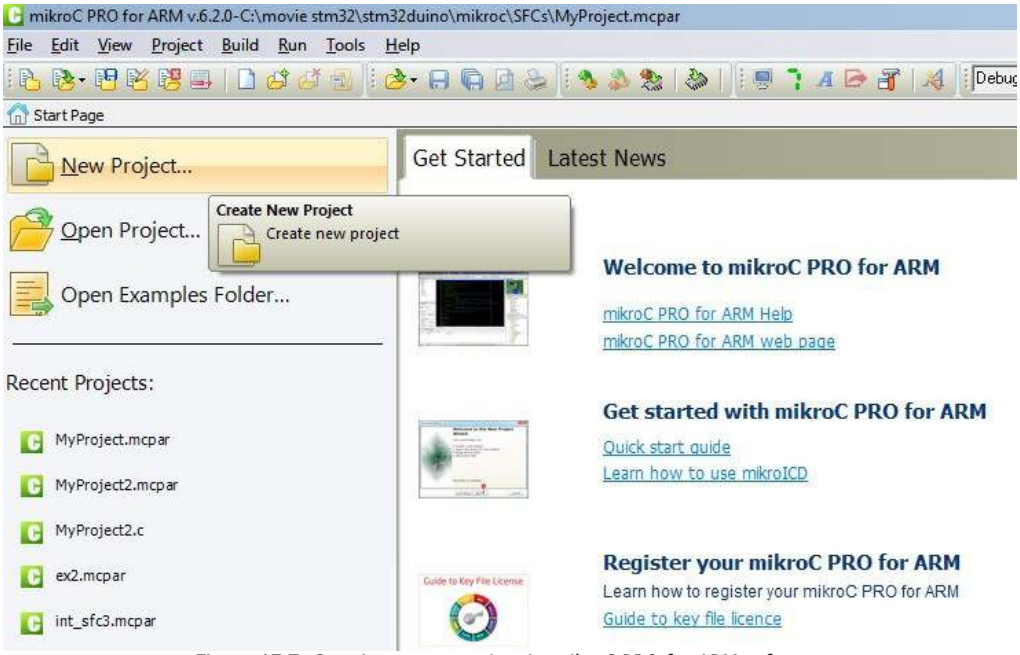

Figure 17.7: Creating a new project in mikroC PRO for ARM software

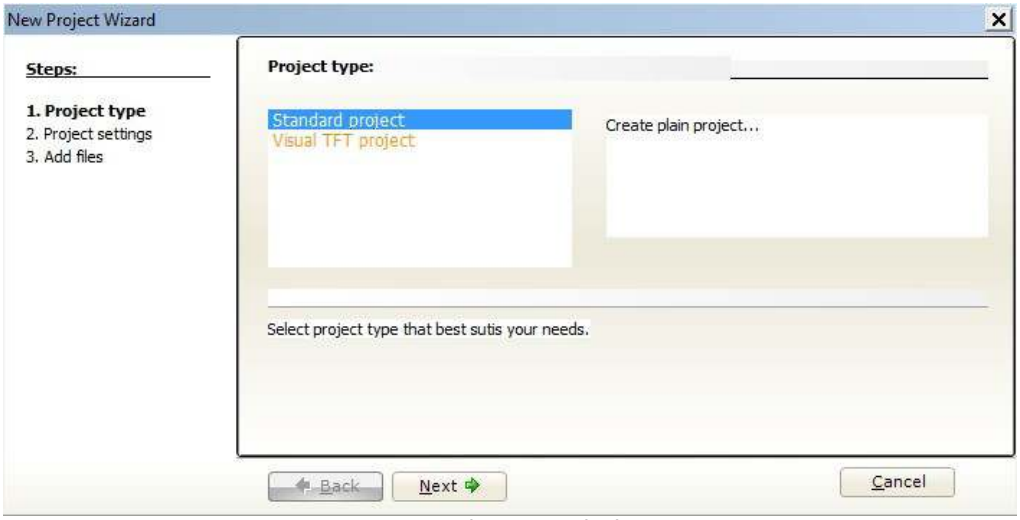

Figure 17.8: Selecting Standard project

As demonstrated in Figure 17.9, choose a project name and folder and then select the STM32F103 microcontroller (blue pill STM32F103C8 header board). Set the device clock to 8 MHz. Click the next button. In the next window, add the header (*.h) or source (*.c) files to the project. Click on the finish button to complete creating the project as shown in Figure 17.10.

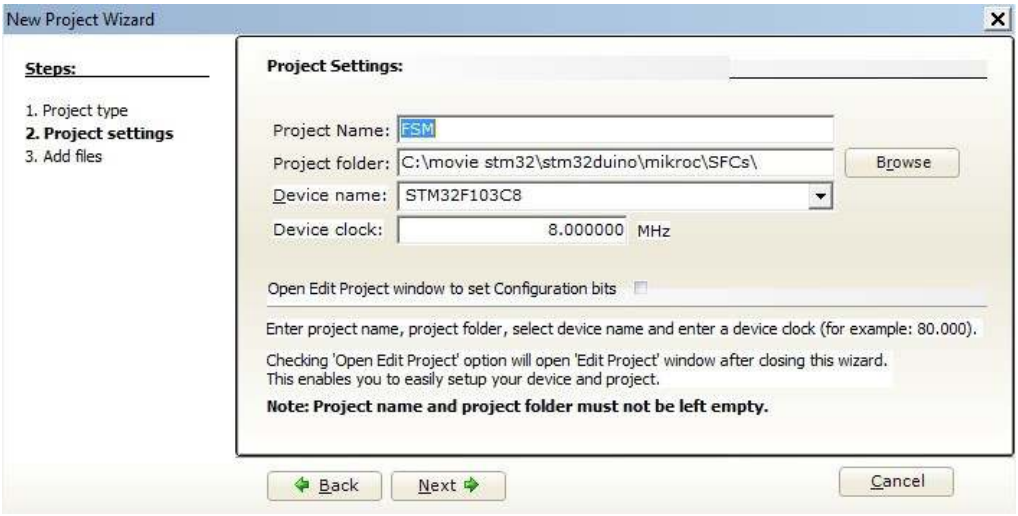

Figure 17.9: Project Settings

Figure 17.10: Adding header or source files to the project

Finite state machine programming is usually written with switch…case statements. The FSM C code using mikroC PRO for ARM software for controlling the industrial automation system shown in Figure 17.5 are as follows:

```
#define start GPIOB_IDR.B4
#define ls1 GPIOB_IDR.B5
#define ls2 GPIOB_IDR.B6
#define encoder GPIOB_IDR.B7
#define emergency_stop GPIOB_IDR.B8
#define motor GPIOB_ODR.B9

void main()
{
  unsigned int Count;                              // Initialize counter
  char state = 1;
  GPIO_Digital_Output(&GPIOB_BASE, _GPIO_PINMASK_9);         // Conf PB8 as digital
output

  //GPIO_Digital_Input(&GPIOB_BASE, _GPIO_PINMASK_4);
  //GPIO_Digital_Input(&GPIOB_BASE, _GPIO_PINMASK_5);
  //GPIO_Digital_Input(&GPIOB_BASE, _GPIO_PINMASK_6);
  //GPIO_Digital_Input(&GPIOB_BASE, _GPIO_PINMASK_7);
  //GPIO_Digital_Input(&GPIOB_BASE, _GPIO_PINMASK_8);

  GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_4,
              _GPIO_CFG_DIGITAL_INPUT | _GPIO_CFG_PULL_UP); // Conf PB4 as digital
input
  GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_5,
              _GPIO_CFG_DIGITAL_INPUT | _GPIO_CFG_PULL_UP); // Conf PB5 as digital
input
  GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_6,
              _GPIO_CFG_DIGITAL_INPUT | _GPIO_CFG_PULL_UP); // Conf PB6 as digital
input
  GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_7,
              _GPIO_CFG_DIGITAL_INPUT | _GPIO_CFG_PULL_UP); // Conf PB7 as digital
input
  GPIO_Config(&GPIOB_BASE, _GPIO_PINMASK_8,
              _GPIO_CFG_DIGITAL_INPUT | _GPIO_CFG_PULL_UP); // Conf PB8 as digital
input

  for(;;)                                          // DO Forever
  {
       switch (state)
       {
          case 1: {                                //Idle
             Count = 0;
             motor = 0;
             if(start == 0 && ls1 == 0) state = 2;
             break;
             }
```

```
        case 2: {                                   //motor1
           motor = 1;
           while(encoder == 0);
           Count++;             // Increment counter
           Delay_ms(10);        //debouncing time
           while(encoder == 1);
           if(Count == 8) state = 3;
           if(emergency_stop == 0) state = 1;
           break;
           }
        case 3: {                                   //p1
           motor = 0;
           if(emergency_stop == 0) state = 1;
           Delay_ms(10000);
           state = 4;
           break;
           }
        case 4: {                                  //motor2
           motor = 1;
           if(emergency_stop == 0 || ls2 == 0) state = 1;
           break;
           }
      }
  }
}
```

Now we can build the project and generate a bin or hex file for programming the microcontroller as depicted in Figure 17.11.

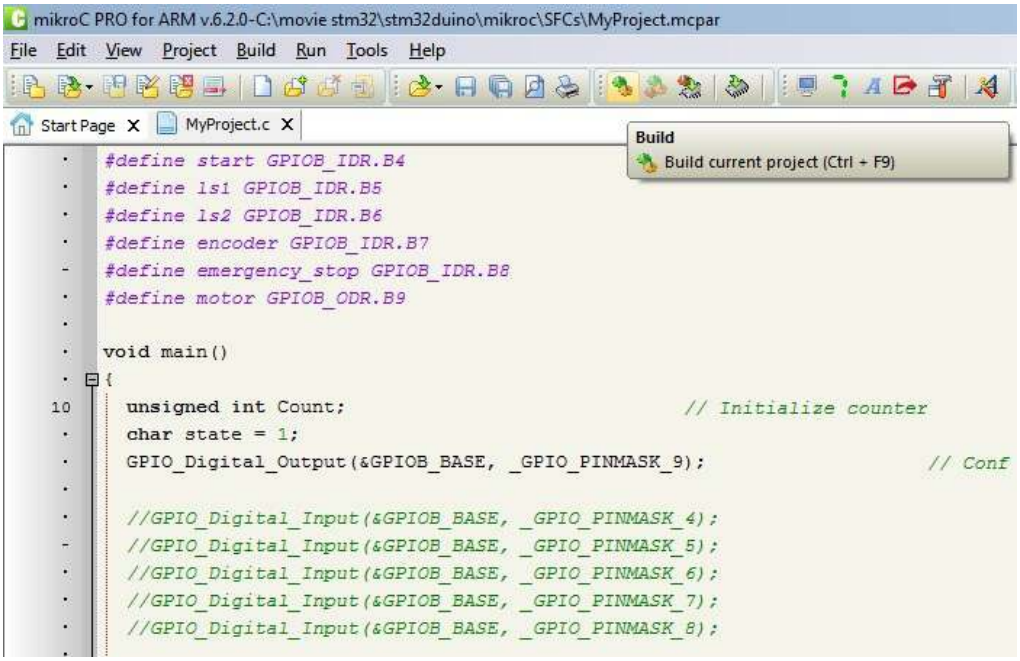

Figure 17.11: Build the project and generate bin or hex file

17.3 • FSM Programming using Mbed Online Compiler

Now we are going to implement the same industrial automation control system using the Mbed online compiler. In the Mbed online compiler, you can jump directly into application development with Mbed OS without installing anything. The weblink of Mbed online compiler is as follows:

https://ide.mbed.com/compiler

For creating a new project, right-click on 'My Programs' in the program workspace and select 'New Program' as shown in Figure 17.12. In the create new program window, select the NUCLEO-F103RB as a platform (this platform is used for programming of STM32F103 microcontrollers), the template as mbed OS Blinky LED HelloWorld, and a Program Name (FSM). When done, click the OK button as illustrated in Figure 17.13. In the Mbed online compiler, like mikroC PRO, the finite state machine programming is usually written with switch…case statements. The FSM C code using Mbed online compiler for controlling the industrial automation system demonstrated in Figure 17.5 is as follows:

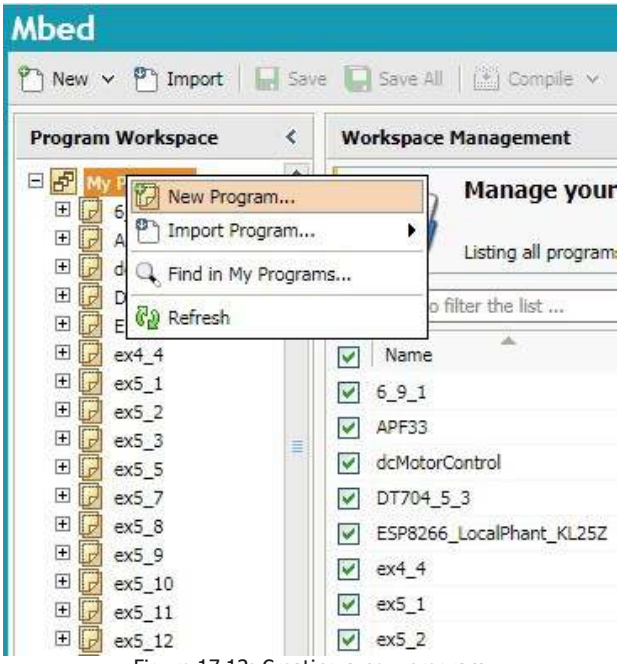

Figure 17.12: Creating a new program

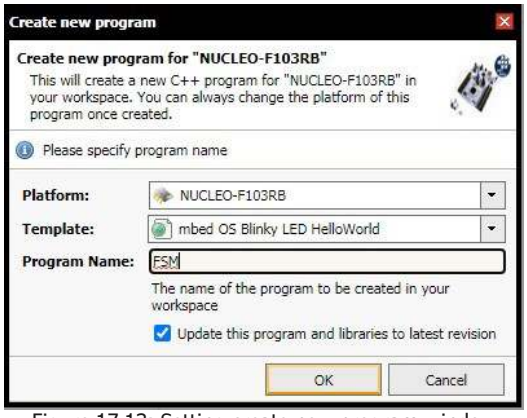

Figure 17.13: Setting create new program window

```
#include “mbed.h”
DigitalOut motor(PB_9);
DigitalIn start(PB_4);
DigitalIn ls1(PB_5);
DigitalIn ls2(PB_6);
DigitalIn encoder(PB_7);
DigitalIn emergency_stop(PB_8);

int main() {
  unsigned int Count;                     // Initialize counter
  char state = 1;
  start.mode(PullUp);
  ls1.mode(PullUp);
  ls2.mode(PullUp);
  encoder.mode(PullUp);
  emergency_stop.mode(PullUp);

while(1) {

     switch (state)
       {
          case 1: {                                 //Idle
             Count = 0;
             motor = 0;
             if(start == 0 && ls1 == 0) state = 2;
             break;
             }
          case 2: {                                  //motor1
             motor = 1;
             while(encoder == 0);
             Count++;                       // Increment counter
             while(encoder == 1);
             if(Count == 8) state = 3;
             if(emergency_stop == 0) state = 1;
             break;
             }
          case 3: {                                  //p1
             motor = 0;
             if(emergency_stop == 0) state = 1;
             wait(10);
             state = 4;
             break;
             }
          case 4: {                                 //motor2
             motor = 1;
             if(emergency_stop == 0 || ls2 == 0) state = 1;
```

```
            break;
            }

        }
}
}
```

We can then compile the program and generate a bin file for programming the microcontroller by clicking the compile button as shown in Figure 17.14.

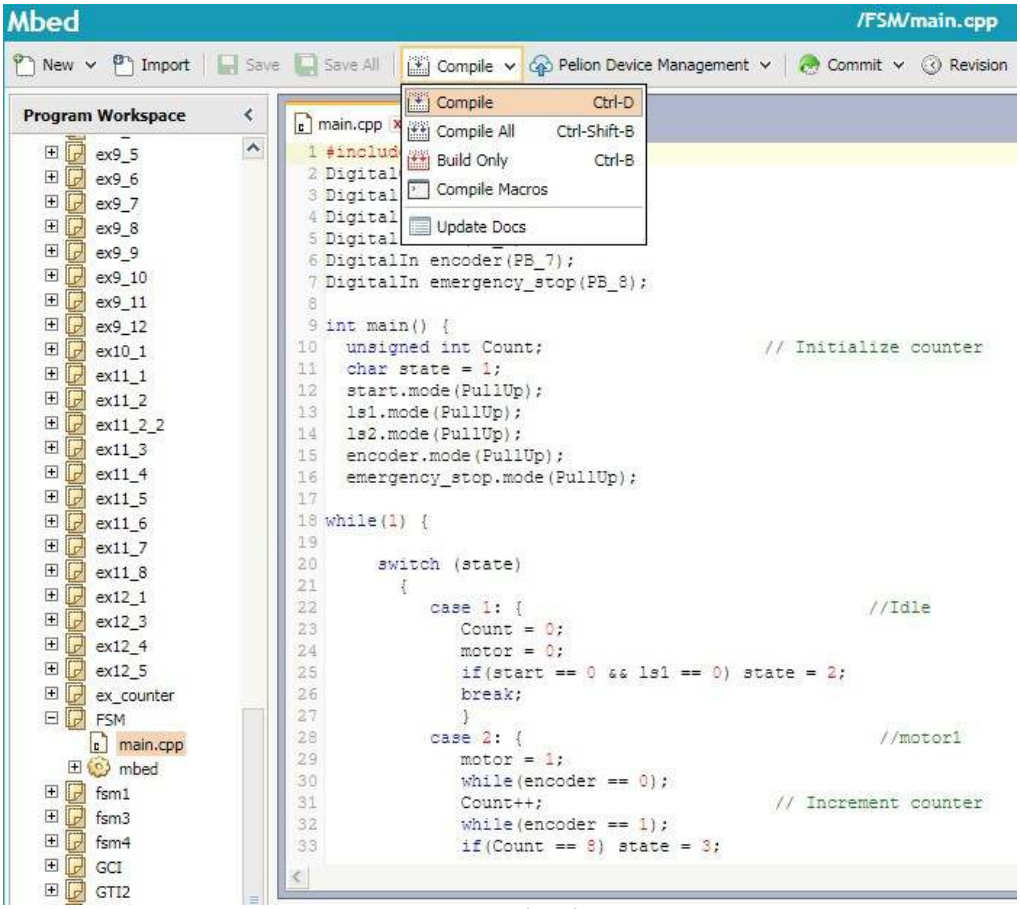

Figure 17.14: Compiling the program

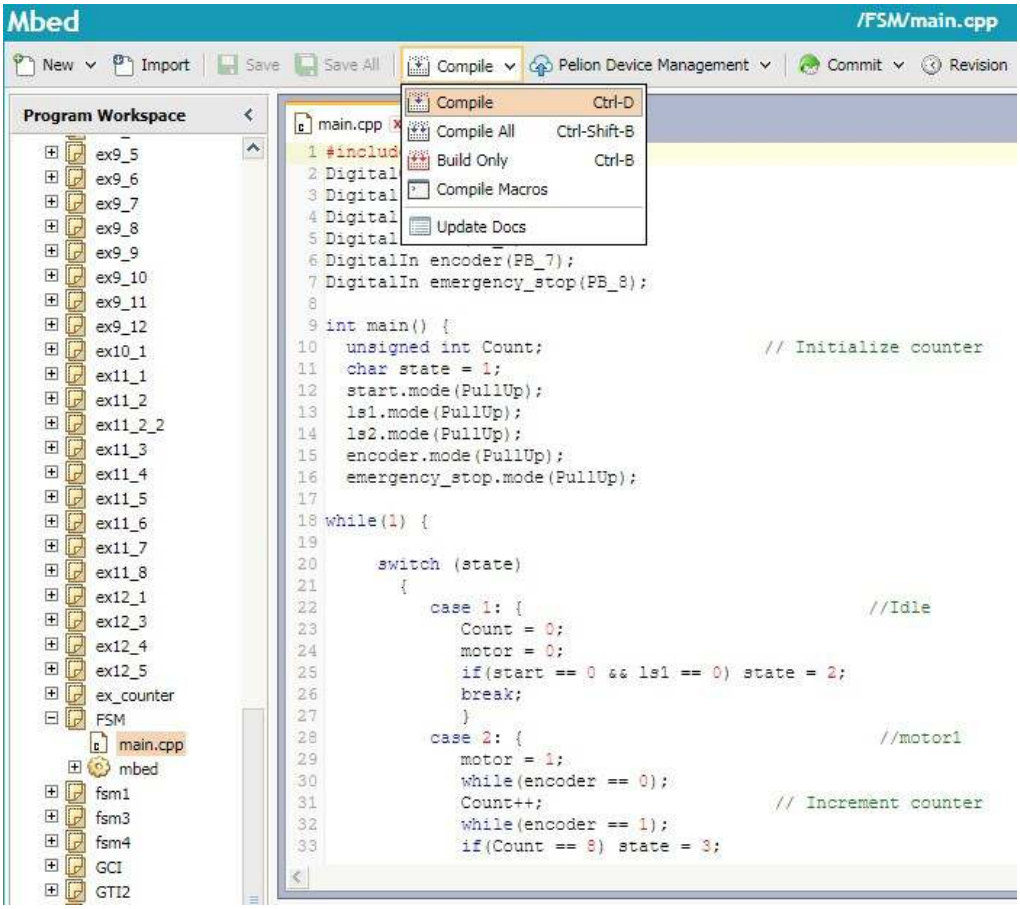

Figure 17.15: Success! message in the compile output for the program window

After successful compilation a 'Success!' message in the compile output for program window appears and a bin file is simultaneously generated as illustrated in Figure 17.15.

17.4 • FSM Modular Programming Using Functions

When the program gets longer and more complex, it may be better to include some of the reusable code into functions. Functions are also called subroutines or procedures. With functions, code can easily be reused, making "int main()" much more simple, hence, reducing programming complexity. For larger projects, we can also separate code into different files. This is called modular programming. We are going to write an FSM program for the industrial automation system shown in Figure 17.5 using functions and modular programming. We can add a new file from the Mbed online compiler by right-clicking the program and select 'New File….' The header file, i.e. the "*.h" file, is used mainly for declarations, such as compiler directives, variable declarations, and function prototypes. The '*.cpp' file is for implementing the functions. The header file "state_machine.h" is used to join multiple files together. The code for modular FSM programming of the mentioned industrial automation system is as follows:

```
//*************************************************************************
#include “state_machine.h”            //main.cpp
int main() {
    idle();
while(1) {
}
}
//*************************************************************************

//*************************************************************************
#ifndef STATE_MACHINE_H              //state_machine.h
#define STATE_MACHINE_H
#include “mbed.h”
void idle();
void motor1();
void p1();
void motor2();
#endif
//*************************************************************************

//*************************************************************************
#include “state_machine.h”          //state_machine.cpp
DigitalOut motor(PB_9);
DigitalIn start(PB_4);
DigitalIn ls1(PB_5);
DigitalIn ls2(PB_6);
DigitalIn encoder(PB_7);
DigitalIn emergency_stop(PB_8);

int pulseCount;

void idle() {
    start.mode(PullUp);
    ls1.mode(PullUp);
    pulseCount = 0;
    motor = 0;
    if(ls1 == 0 && start == 0) {
    motor1();
    }
    else {
    idle();
    }
}

void motor1() {
    encoder.mode(PullUp);
```

```
    emergency_stop.mode(PullUp);
    motor = 1;
    while(encoder == 0); // If button is pressed
    pulseCount++;      // Increment Count
    while(encoder == 1); // Wait until released
    if(emergency_stop == 0) {
    idle();
    }
    else if (pulseCount == 8) {
    p1();
    }
    else {
    motor1();
    }
}

void p1() {
    emergency_stop.mode(PullUp);
    motor = 0;
    if(emergency_stop == 0) {
    idle();
    }
    else {
    wait(10);
    motor2();
    }
}

void motor2() {
    emergency_stop.mode(PullUp);
    ls2.mode(PullUp);
    motor = 1;
    if(emergency_stop == 0 || ls2 == 0) {
    idle();
    }
    else {
    motor2();
    }
}

//*********************************************************************
```

The implemented FSM modular programming using functions in Mbed online compiler is shown in Figure 17.16.

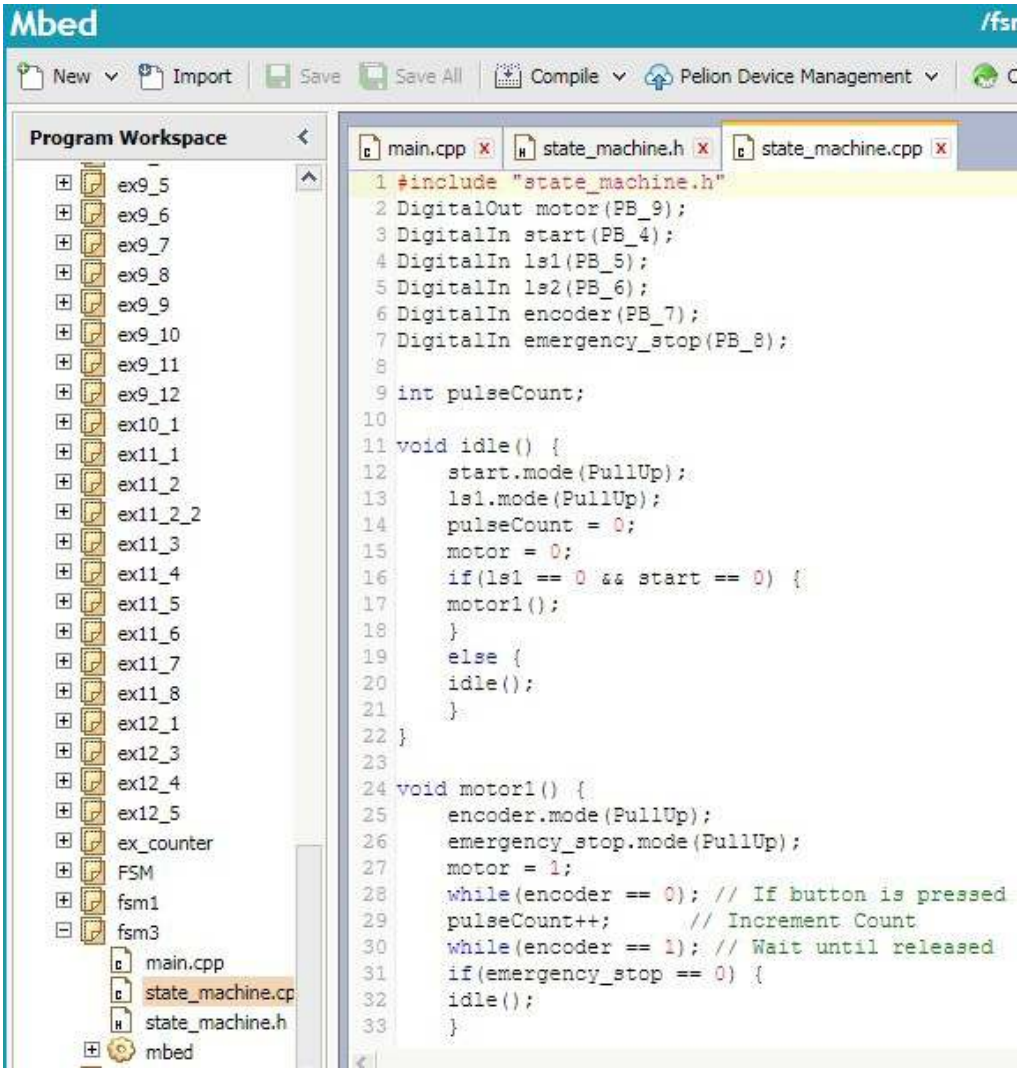

Figure 17.16: FSM modular programming using functions

17.5 ● Summary

A common design technique used in industrial automation control systems is the venerable finite state machine (FSM). Industrial automation control designers use this programming method to break complex problems into manageable states and transitions. In this chapter, the FSM programming method was explained in detail using mikroC PRO for ARM and the Mbed online compilers. The FSM modular programming method using functions was applied for the same project using the Mbed online compiler.

Chapter 18 • STM32 Microcontrollers Programming using MATLAB/ Simulink

18.1 • Introduction

Simulink is a graphical programming environment in MATLAB software used for modelling, simulating, and analyzing dynamical systems. It consists of graphical block diagram tools for simulation of different control systems. Simulink, as an add-on product to MATLAB, provides an interactive, graphical environment for the modelling and simulation of control systems. It enables the rapid construction of control design in detail with minimal effort.
For modelling, Simulink provides a graphical user interface (GUI) for building models as block diagrams. It includes a comprehensive library of predefined blocks that can be used for creating models of systems using drag-and-drop mouse operations. The user is rapidly able to produce a model in Simulink that would require hours to build in a laboratory environment. It supports linear and nonlinear systems, modelled in continuous time, sampled time, or a hybrid of the two. Finally, Simulink is integrated with MATLAB and data can be easily shared between different programs. The Simulink model templates also let you automatically configure the Simulink environment with the recommended settings for digital signal processing modeling. In this chapter, we are going to demonstrate how to generate code for STM32 microcontrollers using Simulink. In the first example, we want to generate a pulse with a specific frequency and period on a pin of the STM32 microcontroller using Simulink, STM32CubeMX, and Keil MDK. As another example, we will generate gate driving sinusoidal PWM (SPWM) pulses for a three-phase inverter on six pins of the STM32 microcontroller using Simulink, STM32CubeMX, and Keil MDK.

18.2 • Pulse Generation Programming using Simulink

Consider the Simulink block diagram as shown in Figure 18.1. The settings for the pulse generator and data type conversion blocks are depicted in Figures 18.2 and 18.3 respectively.

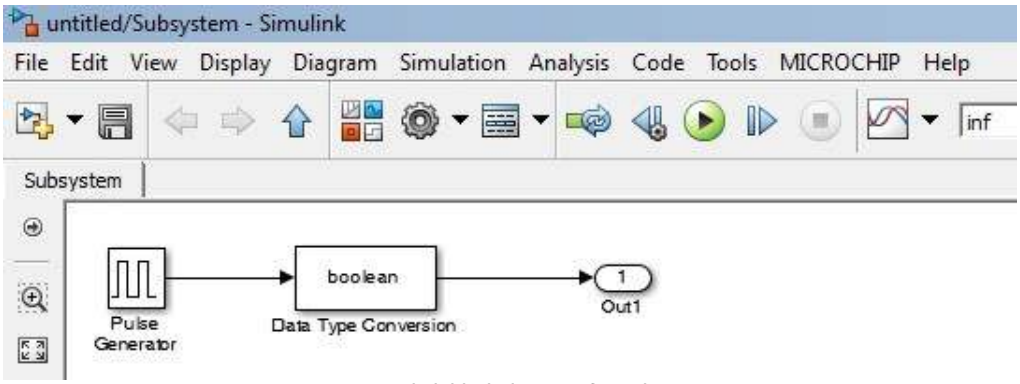

Figure 18.1: Simulink block diagram for pulse generation

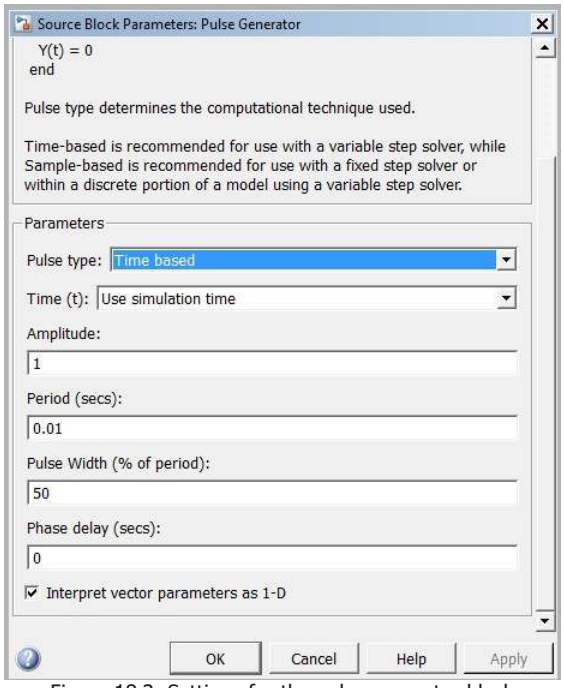

Figure 18.2: Settings for the pulse generator block

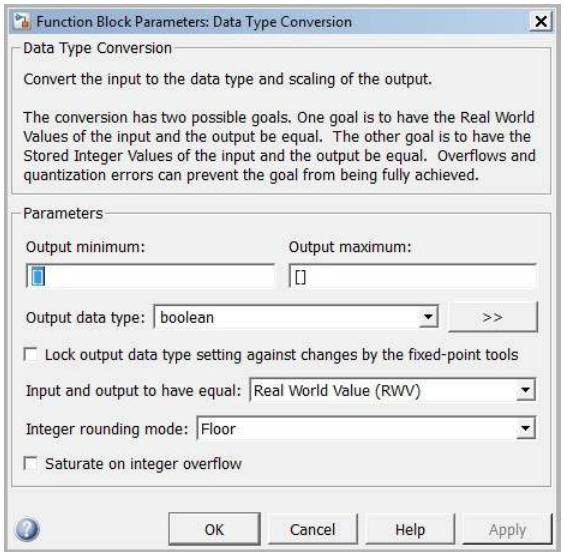

Figure 18.3: Settings for the data type conversion block

We then open the Configuration Parameters dialog box and default the Solver tab, changing stop time to inf as illustrated in Figure 18.4. In the Hardware Implementation tab, set the

device vendor to ARM Compatible and the device type to ARM Cortex as demonstrated in Figure 18.5.

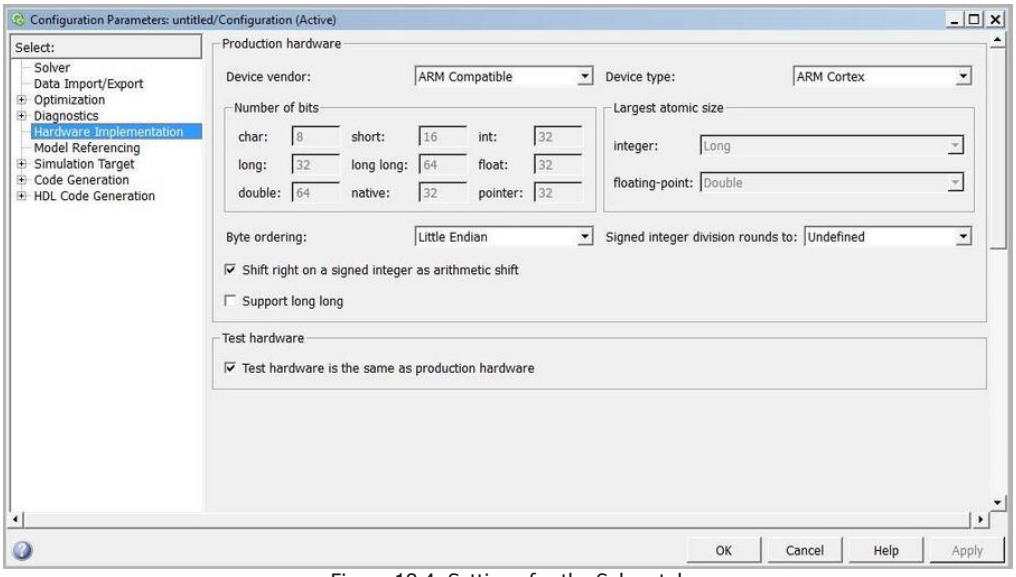

Figure 18.4: Settings for the Solver tab

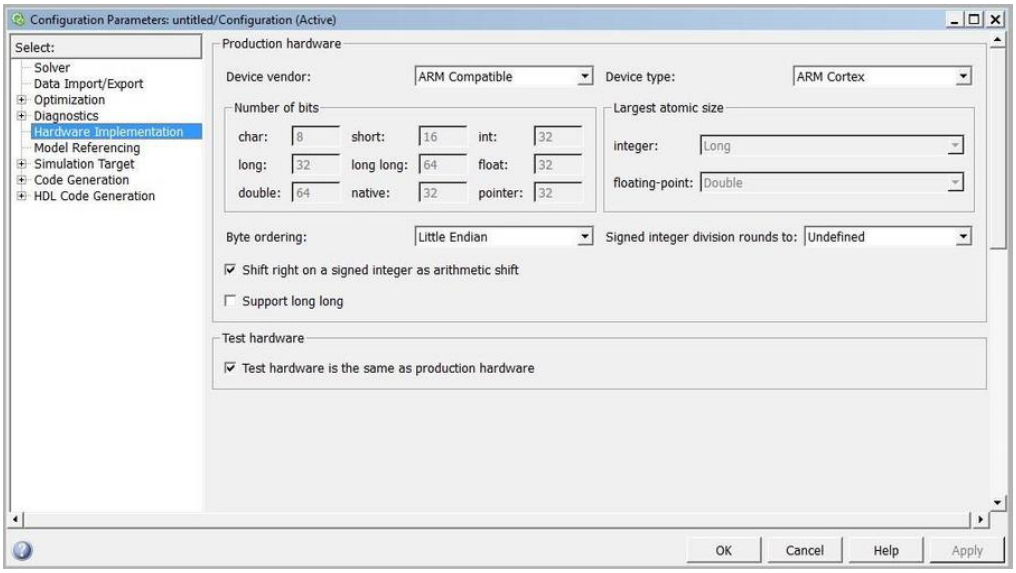

Figure 18.5: Settings for the hardware implementation tab

In the Code Generation tab, click the browse button and set the system target file to ert.tlc and the description as Embedded Coder. Also, set the toolchain to GNU Tools for ARM Embedded Processors as shown in Figure 18.6.

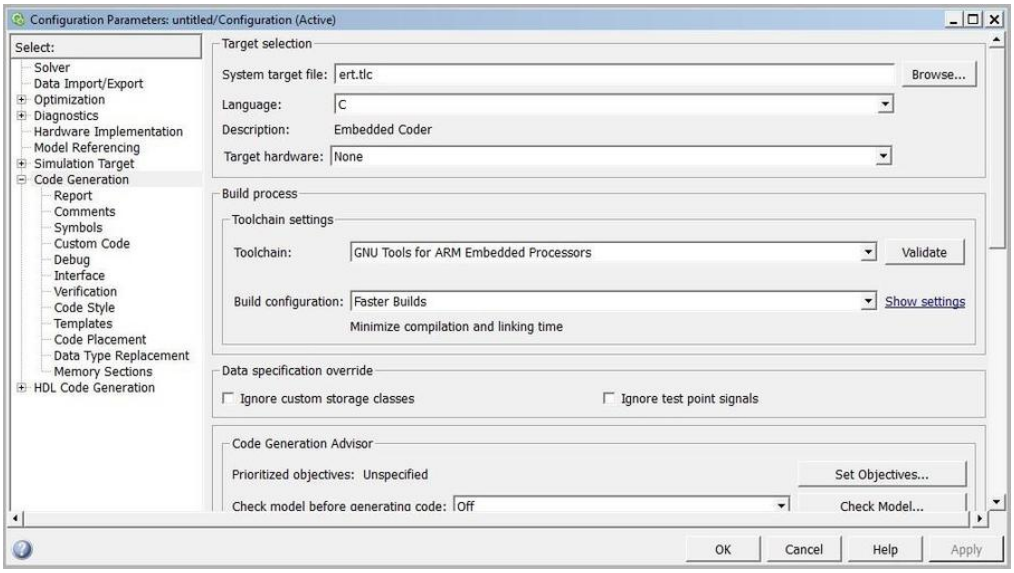

Figure 18.6: Settings for the Code Generation tab

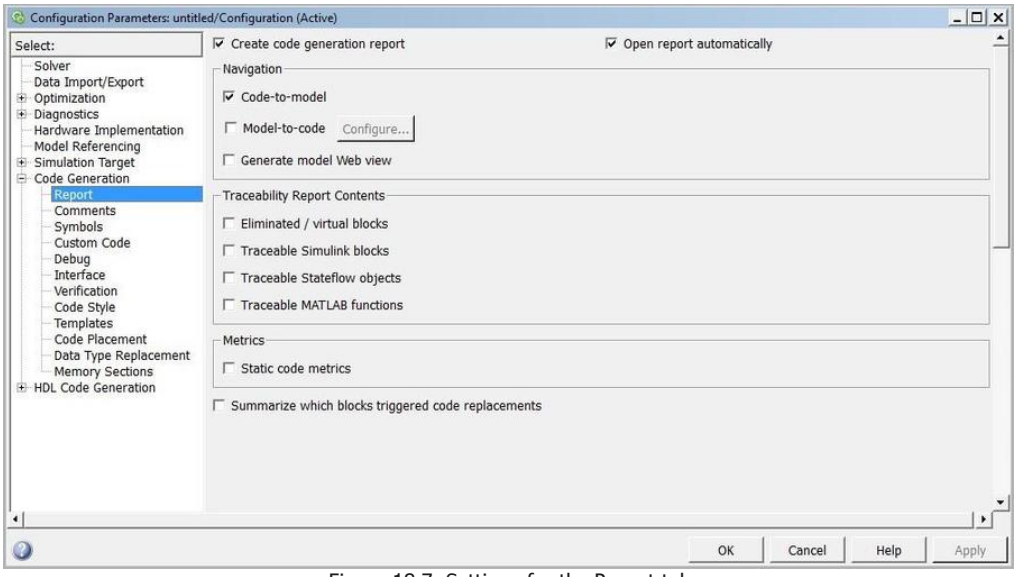

Figure 18.7: Settings for the Report tab

From the Code Generation Report tab, tick the create code generation report. The report and code-to-model boxes open automatically as depicted in Figure 18.7. The default settings for the Code Generation Interface tab are illustrated in Figure 18.8. Click OK, closing the configuration parameters dialog box. Select both the pulse generator and data type conversion blocks and create a subsystem with an output. Right-click on the subsystem and select C/C++ code. Build the 'This Subsystem' icon as demonstrated in Figure 18.9. Click

the build button in the build code for the subsystem dialog box as shown in Figure 18.10. A ‘subsystem build failed’ message pops up in the diagnostic viewer window as shown in Figure 18.11. As a solution, in the MATLAB command window, use the change directory (cd), selecting the Simulink code generation saving directory as illustrated in Figure 18.12. For viewing the code generation report, right-click the subsystem and select C/C++ code and the Open Subsystem Report icons respectively as demonstrated in Figure 18.13. The code generation report for the subsystem and Diagnostic Viewer windows (with a failure message which can be ignored) are shown in figure 18.14. The code generated is saved in the Subsystem_ert_rtw folder as shown in Figure 18.15.

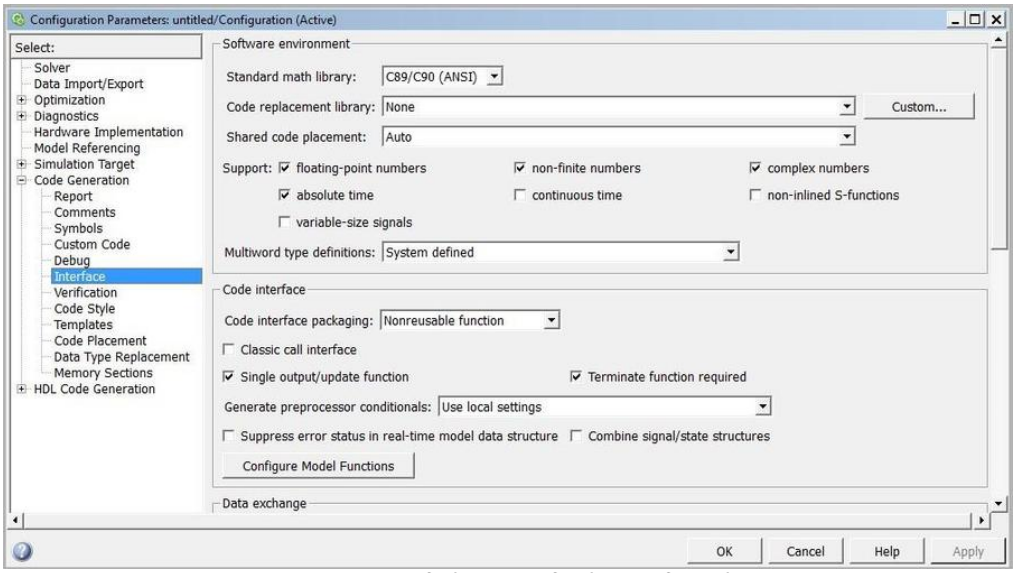

Figure 18.8: Default settings for the Interface tab

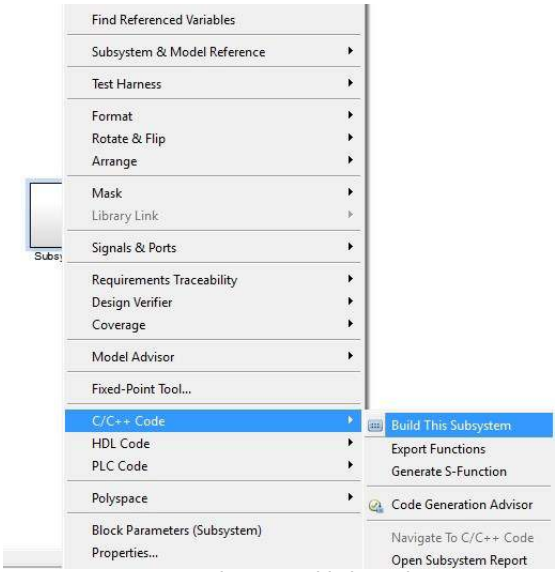

Figure 18.9: Selecting Build This Subsystem

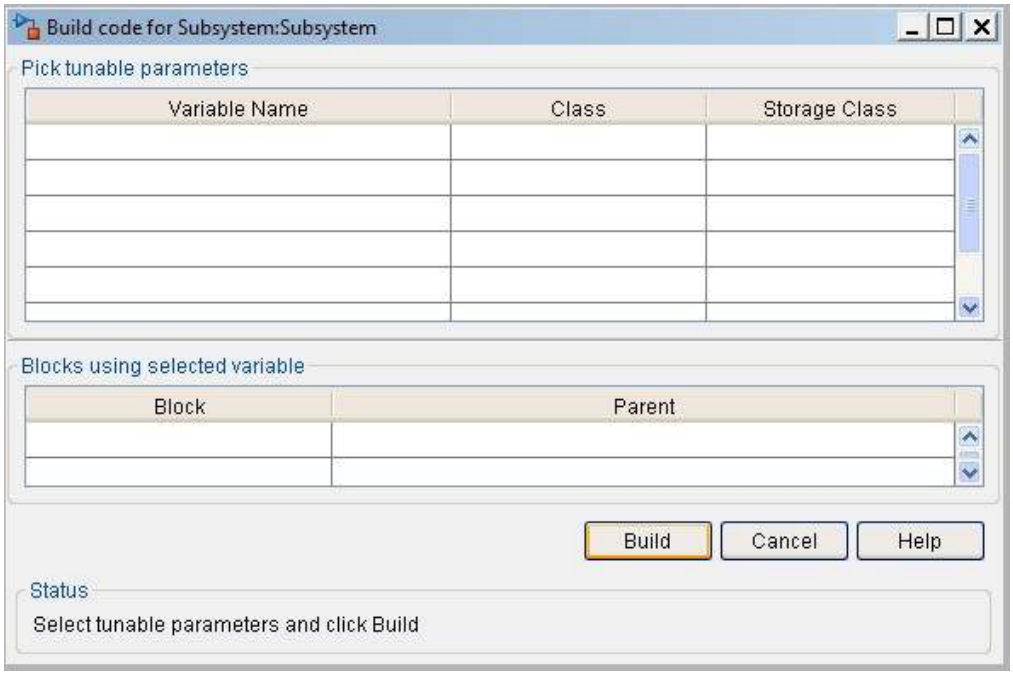

Figure 18.10: Build code for the subsystem dialog box

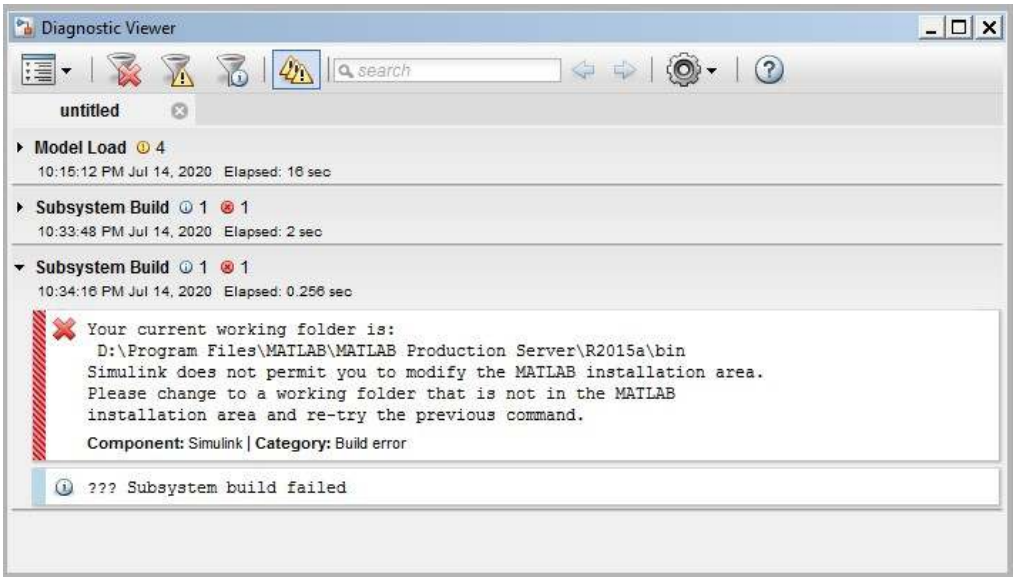

Figure 18.11: Subsystem build failed message

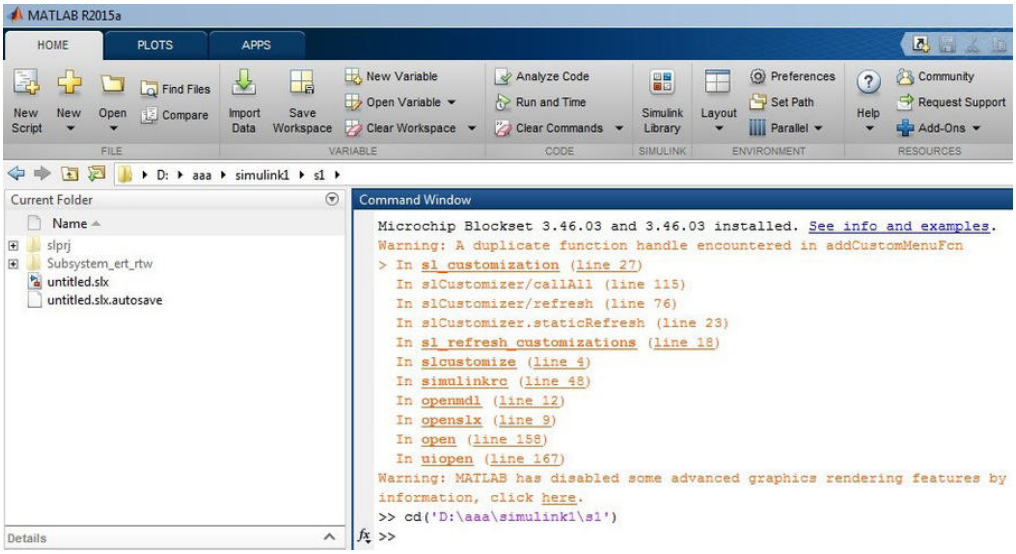

Figure 18.12: Using change directory (cd) command

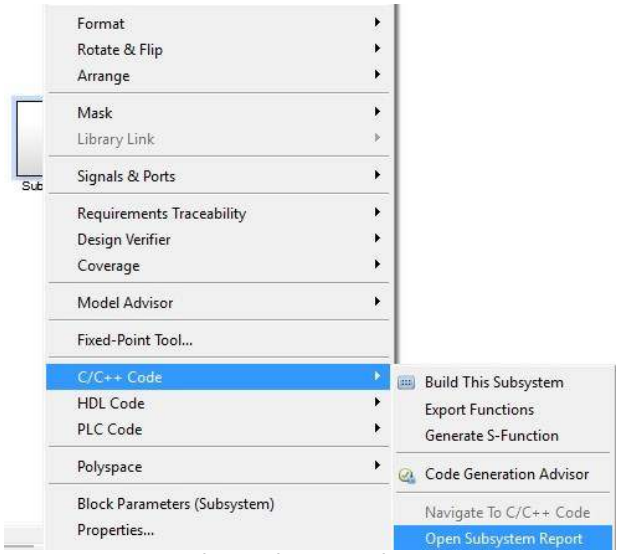

Figure 18.13: Selecting the Open Subsystem Report icon

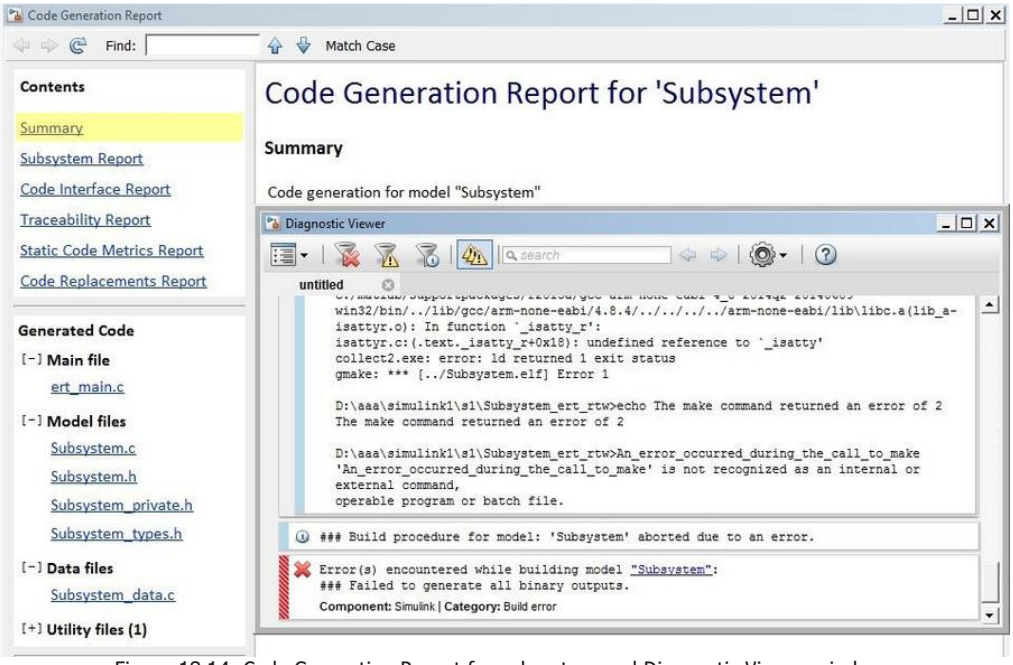

Figure 18.14: Code Generation Report for subsystem and Diagnostic Viewer windows

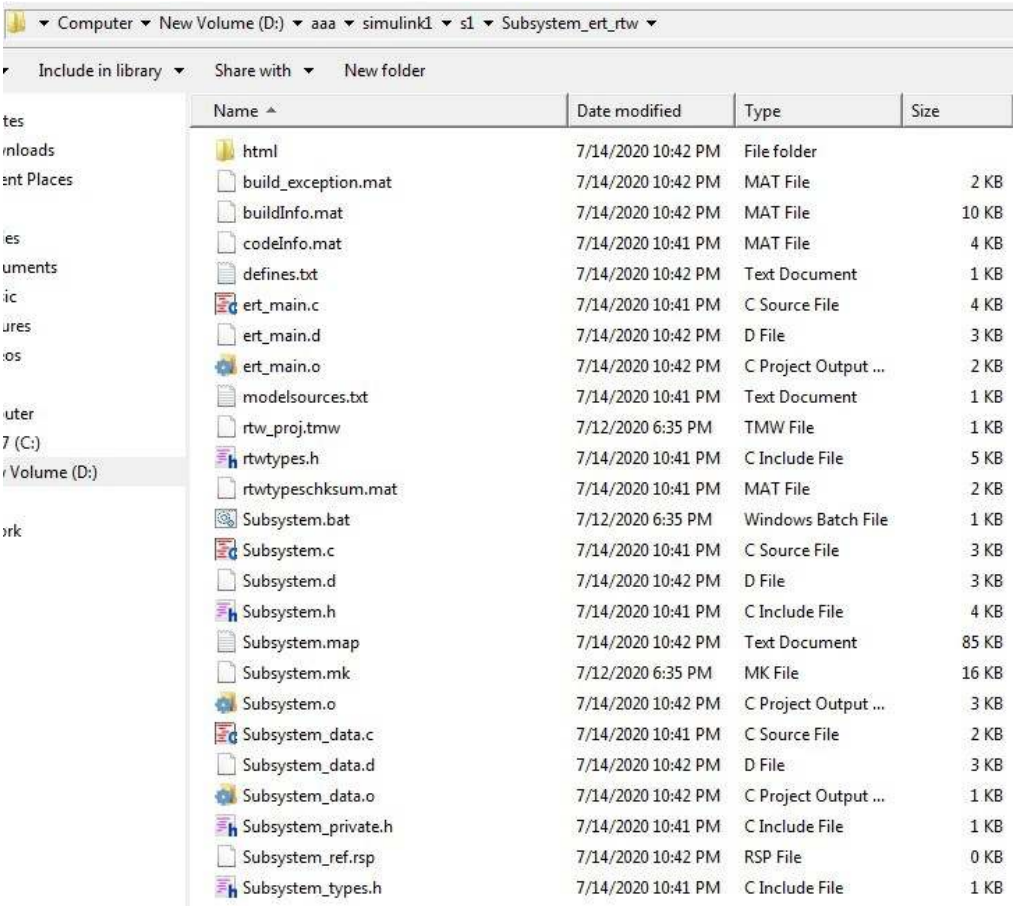

Figure 18.15: Generated code in the Subsystem_ert_rtw folder

Use STM32CubeMX to define a pin (PB9) of the STM32F103C8T6 microcontroller as GPIO_Output as shown in Figure 18.16. From the Project Manager tab, select Keil MDK-ARM as the Toolchain/IDE and click the GENERATE CODE button as demonstrated in Figure 18.17. Copy the (*.h) and (*.c) files inside the Subsystem_ert_rtw folder to the Src folder in the generated Keil MDK-ARM files directory as illustrated in Figure 18.18. In the generated Keil MDK-ARM software, right-click on the Application/User folder and select 'Add Existing Files' to Group 'Application/User' and select the added (*.h) and (*.c) files in the Src folder as shown in Figure 18.19. Copy the code inside the main.c file to ert_main.c in accordance with figures 18.20 and 18.21. Furthermore copy the remaining code of main.c file to the end of ert_main.c and delete the main.c file from the Application/User and Src folders. Add model outputs inside the rt_OneStep function as shown in Figure 18.22. Finally, call the rt_OneStep function inside the while loop as illustrated in Figure 18.23. We can now compile the project and generate hex code for programming the microcontroller.

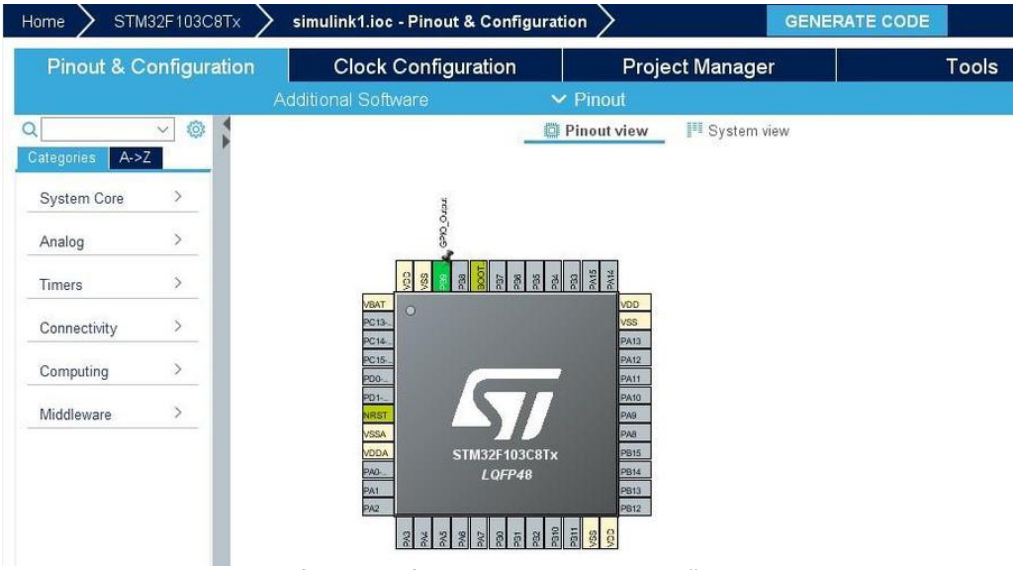

Figure 18.16: Defining PB9 of STM32F103C8T6 microcontroller as GPIO_Output

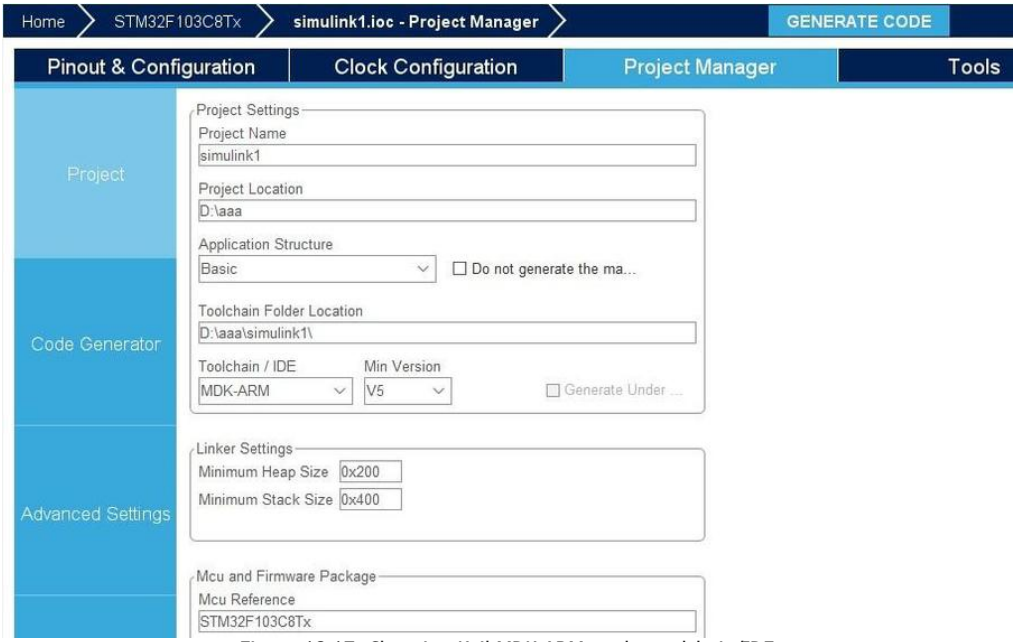

Figure 18.17: Choosing Keil MDK-ARM as the toolchain/IDE

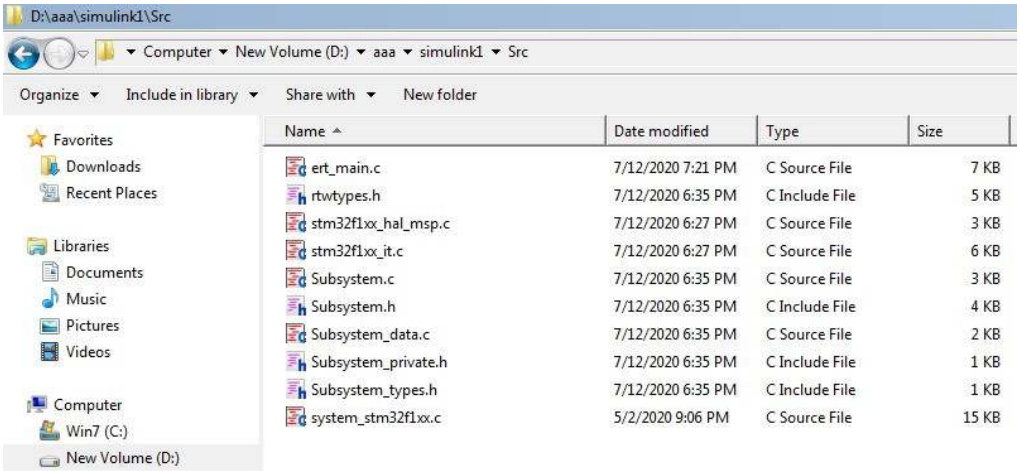

Figure 18.18: Copying the (*.h) and (*.c) files inside Subsystem_ert_rtw folder to Src folder

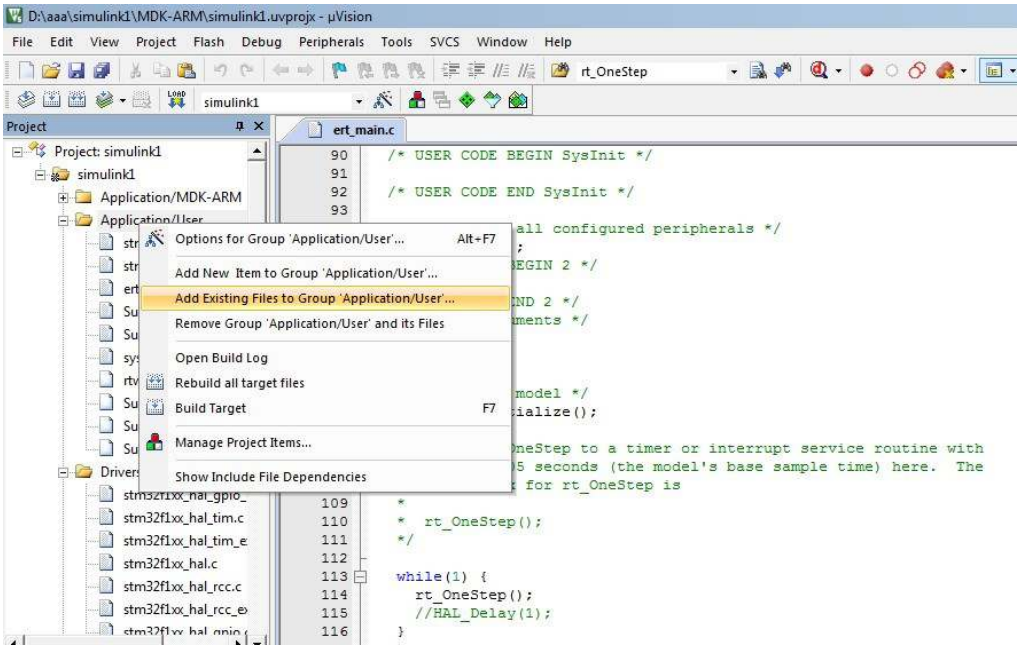

Figure 18.19: Selecting Add Existing Files to Group 'Application/User'....

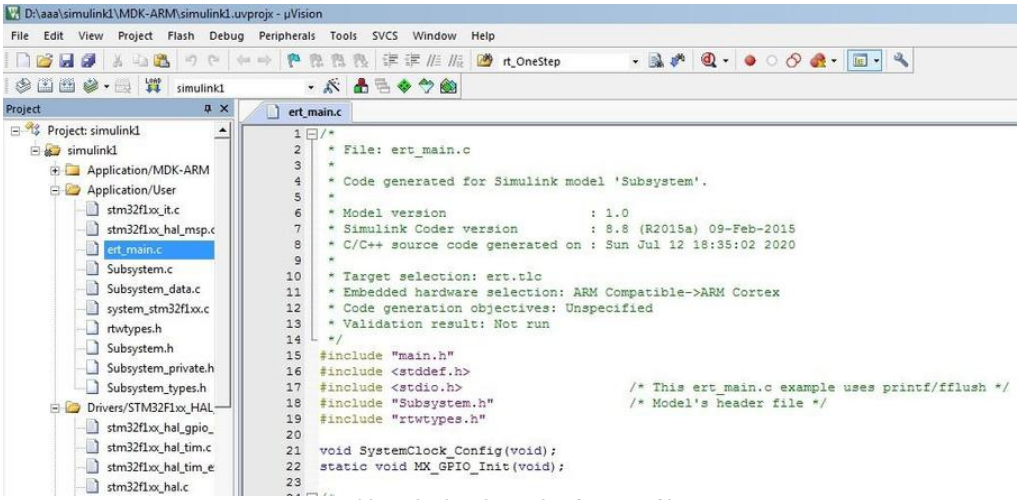

Figure 18.20: Adding the header code of main.c file to ert_main.c

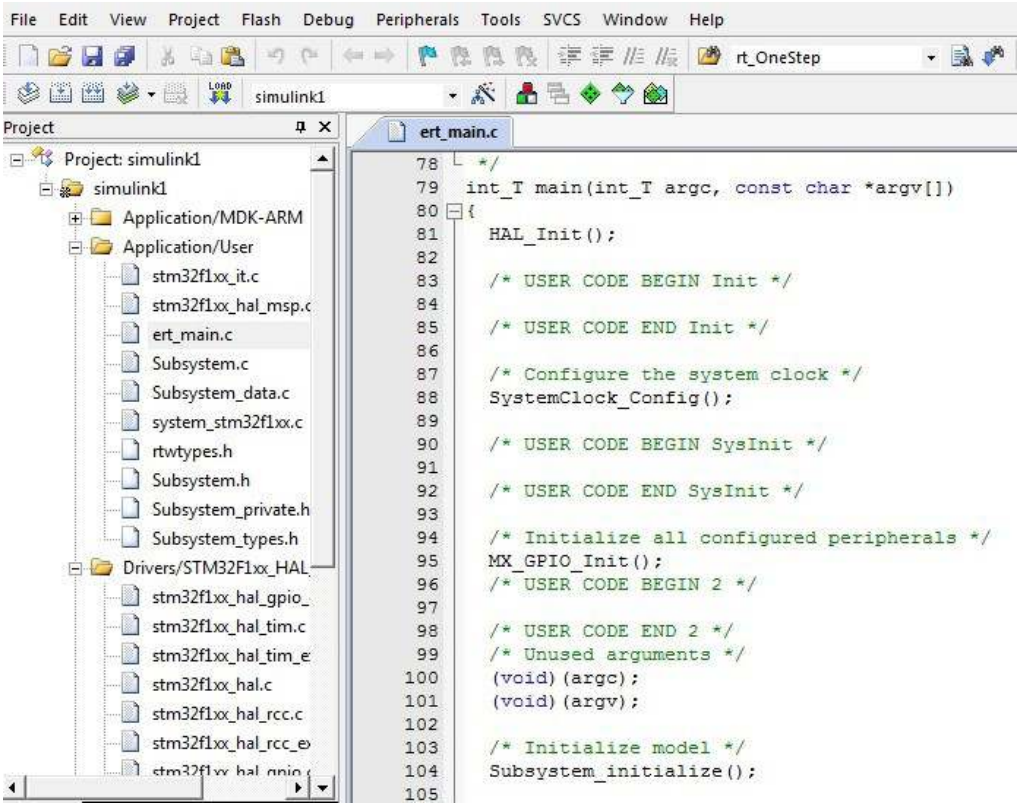

Figure 18.21: Adding main.c code inside the main function to ert_main.c

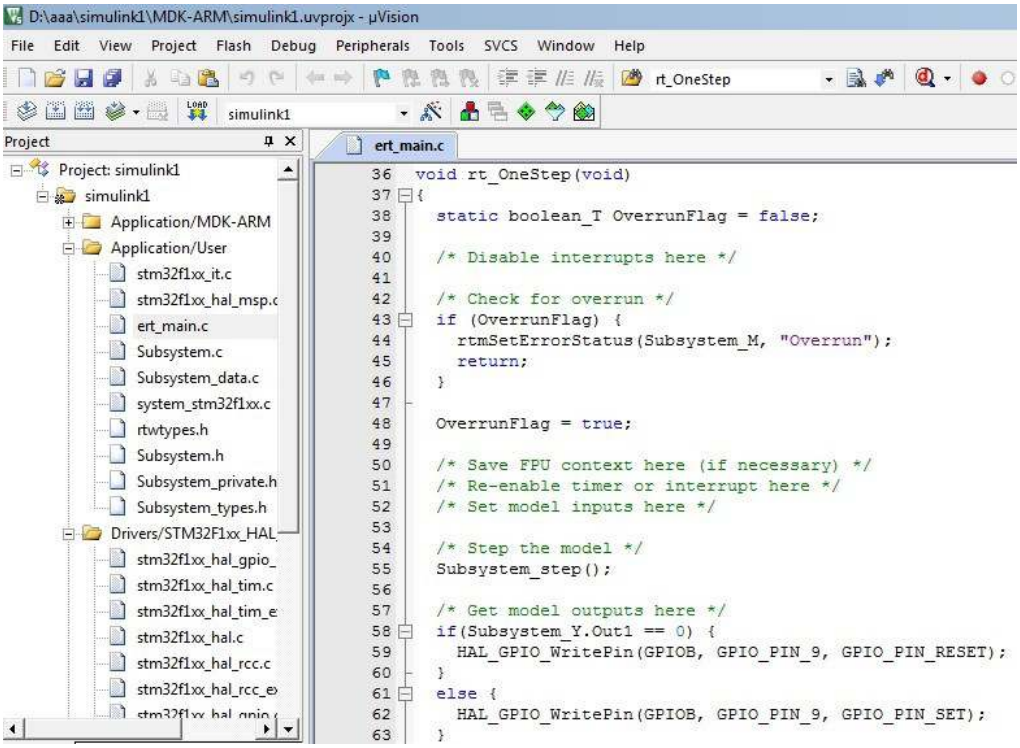

Figure 18.22: Adding model outputs

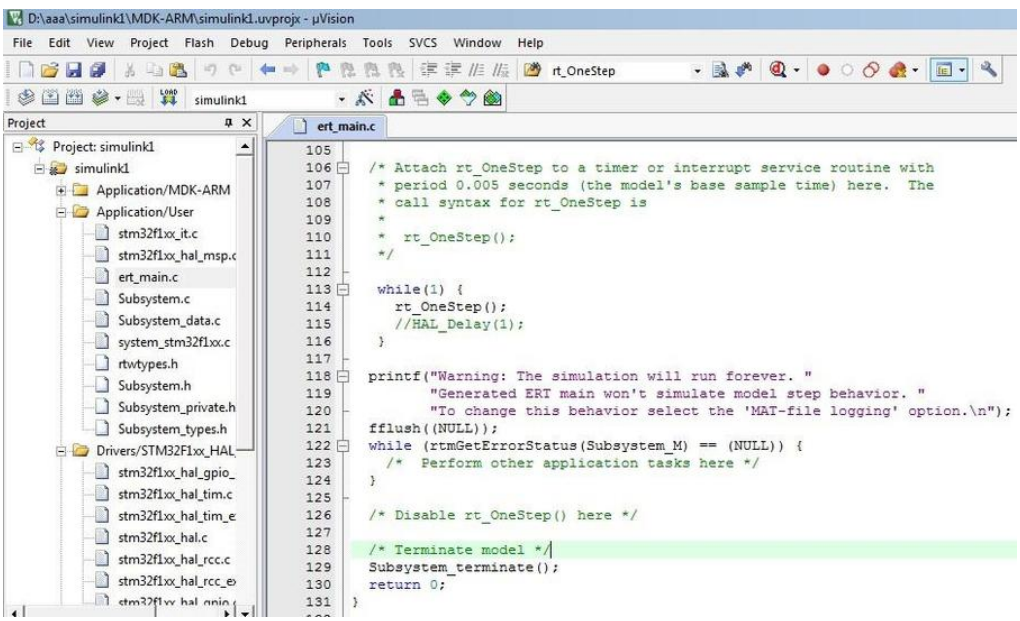

Figure 18.23: Calling rt_OneStep function inside while loop

18.3 • SPWM Generation Programming for Three Phase Inverter using Simulink

As another example, consider the Simulink model for a three-phase inverter as shown in Figure 18.24. The Simulink block diagrams for the sinusoidal PWM (SPWM) subsystem are depicted in Figure 18.25.

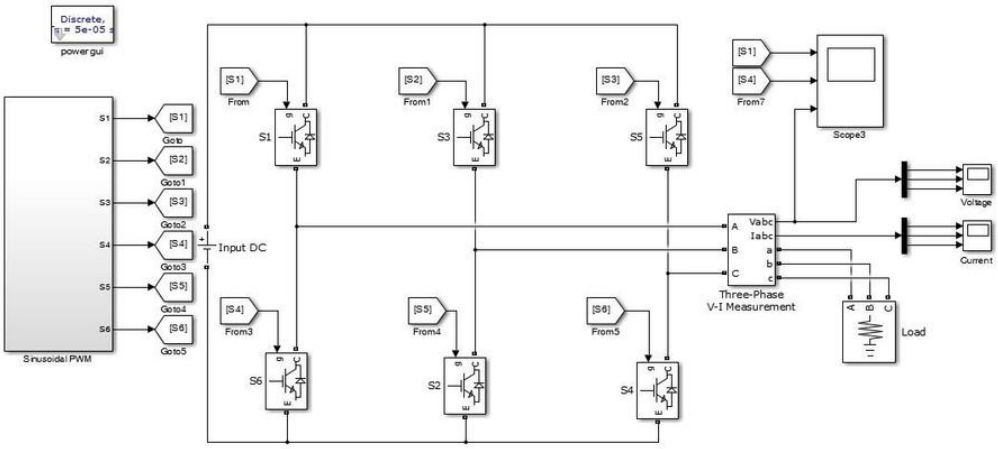

Figure 18.24: Simulink model for three-phase inverter

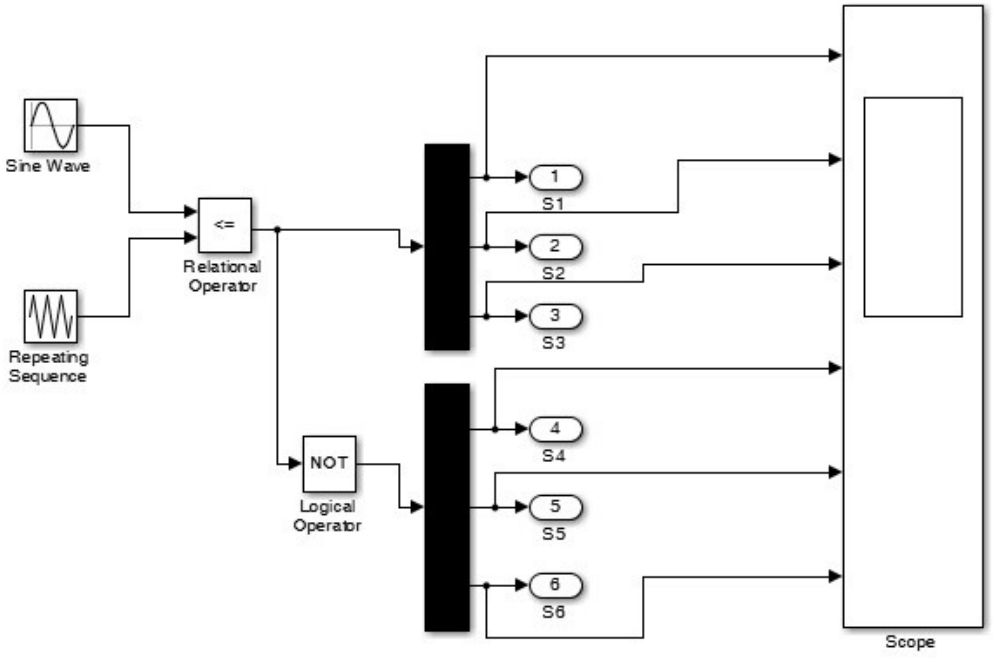

Figure 18.25: Sinusoidal PWM subsystem block diagrams

The sine wave and repeating sequence parameters are shown in Figures 18.26 and 18.27 respectively.

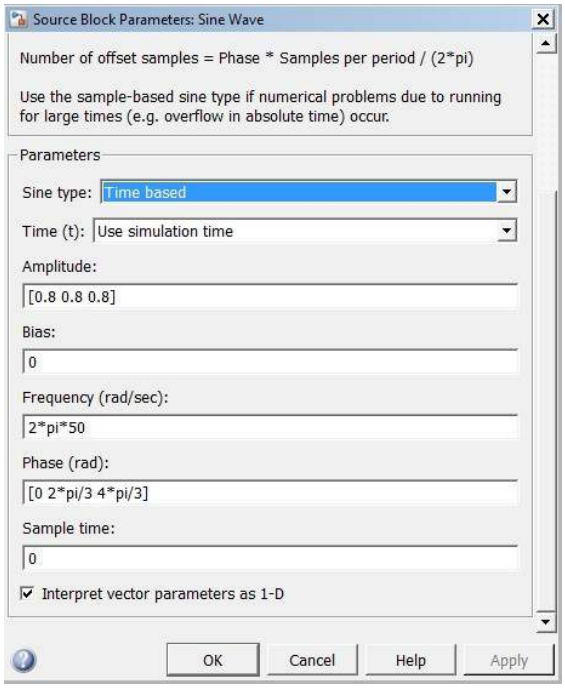

Figure 18.26: Sine wave parameters

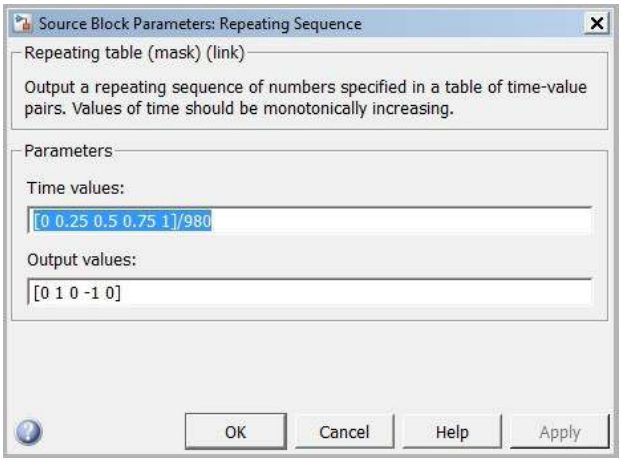

Figure 18.27: Repeating sequence parameters

The sinusoidal PWM gate driving signals (S1, S2, S3, S4, S5, and S6) is shown in Figure 18.28. We are going to generate such sinusoidal PWM signals on the STM32 microcontroller pins so can eliminate other blocks in Simulink three-phase inverter modelling, only considering the sinusoidal PWM subsystem as shown in Figure 18.29.

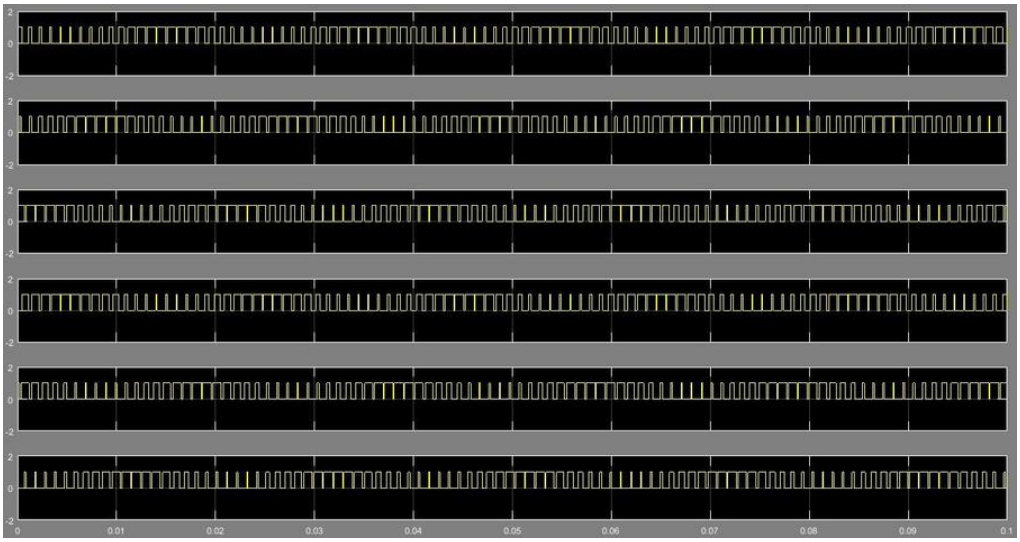

Figure 18.28: Sinusoidal PWM gate driving signals

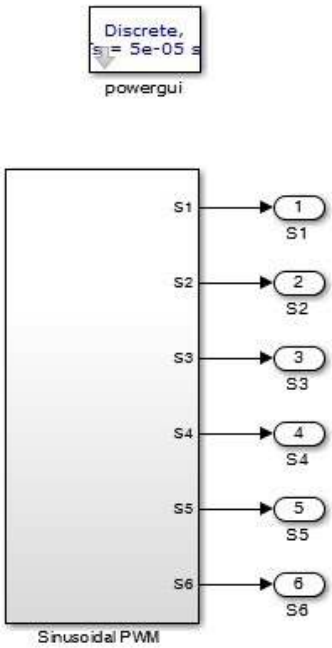

Figure 18.29: Sinusoidal PWM subsystem

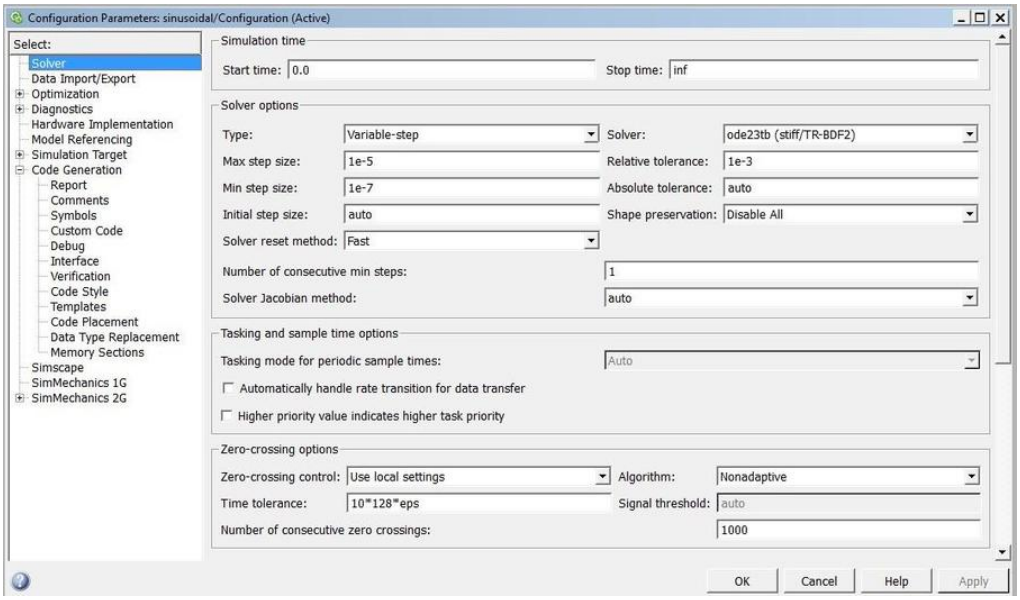

Figure 18.30: Settings for the Solver tab

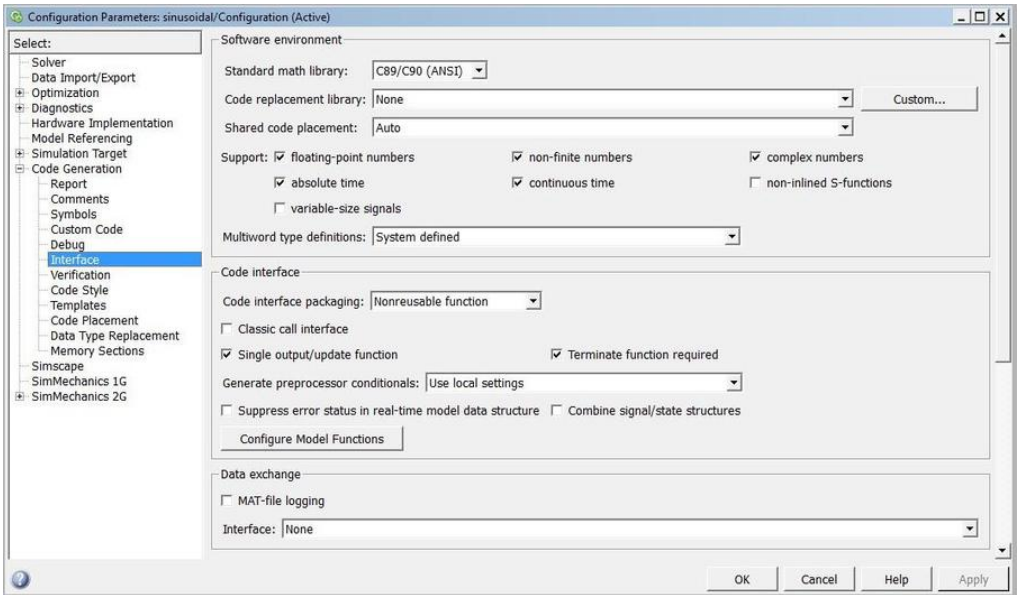

Figure 18.31: Settings for the Interface tab

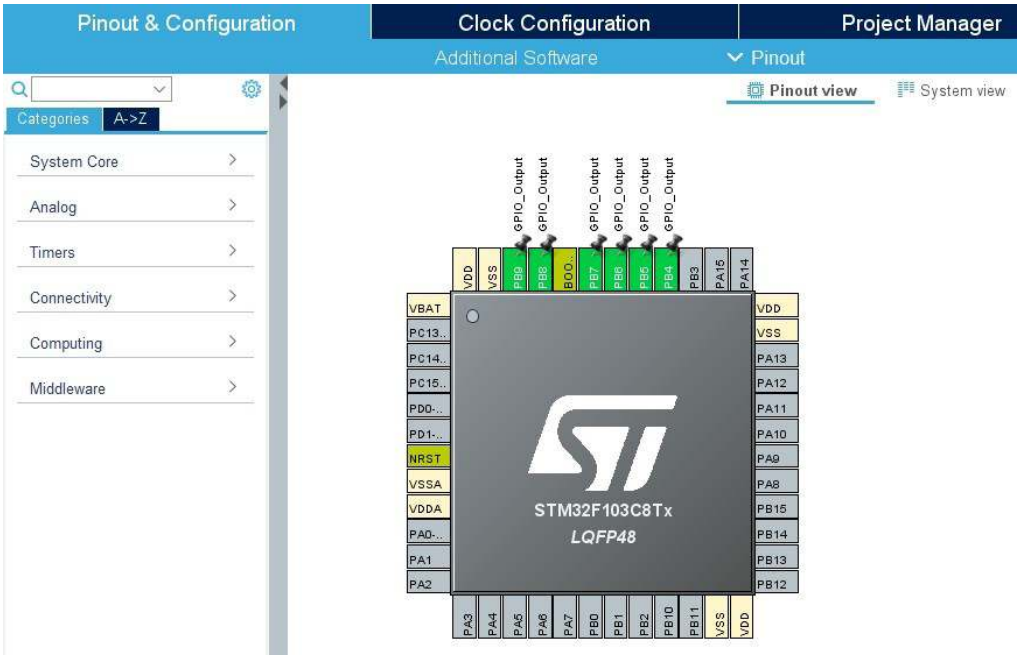

Figure 18.32: Defining PB4-PB9 of the STM32F103C8T6 microcontroller as GPIO_Outputs

Settings for the Solver and Interface tabs are shown in Figures 18.30 and 18.31 respectively. Other tab settings including hardware implementation, code generation, and report are the same as in the previous example. We use STM32CubeMX to define the PB4-PB9 pins of STM32F103C8T6 microcontroller as GPIO_Outputs as shown in Figure 18.32. From the Project Manager tab, select Keil MDK-ARM as the toolchain/IDE and click the 'GENERATE CODE' button to generate code in Keil MDK-ARM. Copy the (*.h) and (*.c) files inside the Sinusoidal0_ert_rtw folder to the Src folder in the generated Keil MDK-ARM file directory. In the generated Keil MDK-ARM software, right-click on the Application/User folder and select 'Add Existing Files' to the Application/Usergroup and further select the added (*.h) and (*.c) files in the Src folder. We then copy the code inside the main.c file to ert_main.c and delete the main.c file from the Application/User and Src folders. Add model outputs inside the rt_OneStep function and finally call the rt_OneStep function inside the while loop. To successfully compile the project, we will also need two files called rtw_continuous.h and rtw_solver.h. These should be added to the Src folder. Right-click on the application/user folder and select 'Add Existing Files' to the group 'Application/User', selecting the added rtw_continuous.h and rtw_solver.h files in the Src folder. We can now compile the project and generate the hex code for programming the microcontroller. The code of ert_main.c, rtw_continuous.h, and rtw_solver.h files is available in the book download section.

18.4 ● Summary

Simulink, as an add-on product to MATLAB provides an interactive graphical environment for the modelling and simulation of control systems. Users are rapidly able to produce

models in Simulink that would otherwise require hours to build in a laboratory environment. It supports linear and nonlinear systems, modelled in continuous or sampled time, or a hybrid of the two. In this chapter, we demonstrated how to generate code for STM32 microcontrollers using Simulink.

• Index

A

analogue to digital converter *52*, *61*, *62*, *68*
Arduino IDE *7*, *8*, *10*, *166*, *175*
ARM Cortex-M *5*, *8*, *11*, *12*, *17*

B

baud rate *107*
blinking an LED *170*
BOOT mode *13*
buffer *40*, *70*
Build Project *104*, *114*, *125*, *130*, *140*

C

C/C++ code *195*, *196*
clock *11*, *106*, *130*, *131*, *180*
clock configuration *24*, *78*, *107*, *118*, *141*
CMOS *40*
code execution *57*
Code Generation *27*, *194*, *195*, *199*
CoIDE *12*
communication protocols *9*, *12*, *106*, *116*, *125*
compiler *8*, *11*, *12*, *118*, *191*
Configuration section *22*, *53*, *54*, *78*, *86*
Counter Mode *86*, *91*, *92*
Counter Period *78*, *86*, *91*, *92*
Counters *6*, *8*, *76*
CPU *9*, *126*, *132*, *152*, *153*
CustomIO *167*, *170*

D

data type *192*, *193*, *195*
debug folder *85*, *112*
Direct Memory Access *6*, *9*, *141*, *152*

E

external crystal resonator *53*, *62*, *71*, *142*, *154*
External Interrupt *5*, *52*
external pulses *9*, *76*, *86*, *90*

F

finite state machine *10*, *175*, *176*, *184*, *191*
Flash Loader Demonstrator software *13*
Flowcode 8 *12*
FreeRTOS *7*, *153*, *154*, *159*, *170*, *175*
frequency *11*, *21*, *62*, *91*, *130*,
FSM model *176*

G

GPIO *21*, *22*, *32*, *43*, *54*, *209*
graphical user interface *192*

H

HAL *8*, *30*, *32*, *33*, *114*, *123*
header code *59*, *112*, *138*, *147*, *150*
hex code *90*, *96*, *114*, *137*, *209*
hex file *30*, *34*, *85*, *104*, *112*, *165*
HSE *21*, *107*, *118*

I

I2C *6*, *9*, *133*, *134*, *137*, *140*
IAR Embedded Workbench *8*, *11*, *28*, *45*
in-circuit serial programmer *11*
Independent Watchdog *9*, *132*
industrial automation *10*, *159*, *166*, *167*, *178*, *188*
Inter-Integrated Circuit *6*, *9*, *133*, *140*

J

JTAG *12*

K

Keil MDK ARM *8*, *11*
Keil uVision *51*, *74*, *75*, *150*, *155*

L

LCD *5*, *8*, *46*, *66*, *12*, *118*
LSE *21*, *97*, *98*, *99*

M

main.c *5*, *100*, *144*, *200*, *209*
master *116*, *118*, *120*, *137*, *140*
MATLAB *7*, *8*, *10*, *12*, *192*, *196*, *209*
Mbed online compiler *8*, *12*, *178*, *188*, *191*

MCU *8*, *18*, *20*, *70*, *153*
MDK-ARM V5 *143*, *145*, *150*, *155*
mikroC PRO *7*, *10*, *179*, *184*, *191*
Modular Programming *7*, *188*

N

NVIC *52*, *61*, *76*, *79*, *107*

P

PCB layout *15*, *17*
plcLib *159*, *167*, *170*, *175*
prescaler *76*, *77*, *78*
programmable logic controller *159*, *166*, *175*
Project Manager section *71*, *92*
Pulse Width Modulation *6*, *9*, *91*
pushbutton *41*, *43*, *45*, *167*, *178*
PWM signal *91*, *92*, *94*

R

RCC *97*, *127*, *134*, *141*
Real-Time Clock *6*, *9*, *97*, *105*
reference voltage *62*, *69*
resolution coefficient *62*
RTOS *7*, *9*, *153*, *158*
RX *106*, *107*, *108*, *150*, *173*

S

SCL *9*, *133*, *140*
SDL *9*, *133*, *140*
sequential function chart *159*, *175*
Serial Communication *6*, *9*, *106*
Serial Wire *12*, *127*, *133*, *134*, *138*
Serial Wire Debug *12*
seven-segment *38*, *39*, *40*, *41*
sine wave *206*
sinusoidal PWM *192*, *205*, *207*
slave *86*, *117*, *122*, *133*, *137*
Solver *193*, *194*, *208*, *209*
speed *9*, *40*, *116*, *125*
SPI protocol *9*, *116*, *117*, *125*, *141*
Src folder *47*, *89*, *200*, *202*, *209*
STM32CubeMX *5*, *45*, *52*, *69*, *200*
STM32duino *7*, *12*, *159*, *171*, *175*
STM32F1 *11*, *19*, *20*, *162*, *163*
STM32FreeRTOS *170*, *171*

Subsystem *195*, *196*, *199*, *200*, *202*
SW4STM32 *8*, *11*, *79*, *131*, *137*
switch…case statement *181*
system core *86*, *118*, *130*, *141*, *154*

T

Timers *6*, *8*, *76*, *86*, *91*
Toolchain *200*
Trigger *86*, *144*
TTL *13*, *15*, *40*, *106*
TX *106*, *150*, *173*

U

Universal Asynchronous Receiver/Transmitter *106*
Universal Synchronous/Asynchronous Receiver/Transmitter *106*

W

Watchdog Timer *6*, *9*, *126*
while loop *34*, *60*, *111*, *136*, *204*
Window Watchdog *9*, *126*, *132*

elektorbooks

Master the software tools behind the STM32 microcontroller

Advanced Programming with STM32 Microcontrollers

This book is project-based and aims to teach the software tools behind STM32 microcontroller programming. Author Majid Pakdel has developed projects using various different software development environments including Keil MDK, IAR Embedded Workbench, Arduino IDE and MATLAB. Readers should be able to use the projects as they are, or modify them to suit to their own needs. This book is written for students, established engineers, and hobbyists. STM32 microcontroller development boards including the STM32F103 and STM32F407 are used throughout the book. Readers should also find it easy to use other ARM-based development boards.

Advanced Programming with STM32 Microcontrollers includes:

- introduction to easy-to-use software tools for STM32
- accessing the features of the STM32
- practical, goal oriented learning
- complete code available online
- producing practical projects with ease

Topics cover:

- Pulse Width Modulation
- Serial Communication
- Watchdog Timers
- I²C
- Direct Memory Access (DMA)
- Finite State Machine Programming
- ADCs and DACs
- External Interupts
- Timers and Counters

Majid Pakdel was born in Mianeh, Iran in 1981. He received his BSc, MSc and PhD in electrical engineering from Amirkabir University of Technology, Isfahan University of Technology and the University of Zanjan respectively. He was a guest PhD student at Aalborg University in 2015-16.

Elektor International Media BV
www.elektor.com

9 783895 764103

elektor
design > share > sell