

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/354117675>

STM32 ADC TUTORIAL with application to real-time control

Technical Report · August 2021

DOI: 10.13140/RG.2.2.17875.30249/1

CITATIONS

0

READS

706

3 authors:



Mohsen Fallah

Scania

22 PUBLICATIONS 51 CITATIONS

[SEE PROFILE](#)



Seyyed Alireza Davodi Navokh

Ferdowsi University Of Mashhad

3 PUBLICATIONS 0 CITATIONS

[SEE PROFILE](#)



Mehran Mozaffari-Jovein

Ferdowsi University Of Mashhad

2 PUBLICATIONS 0 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



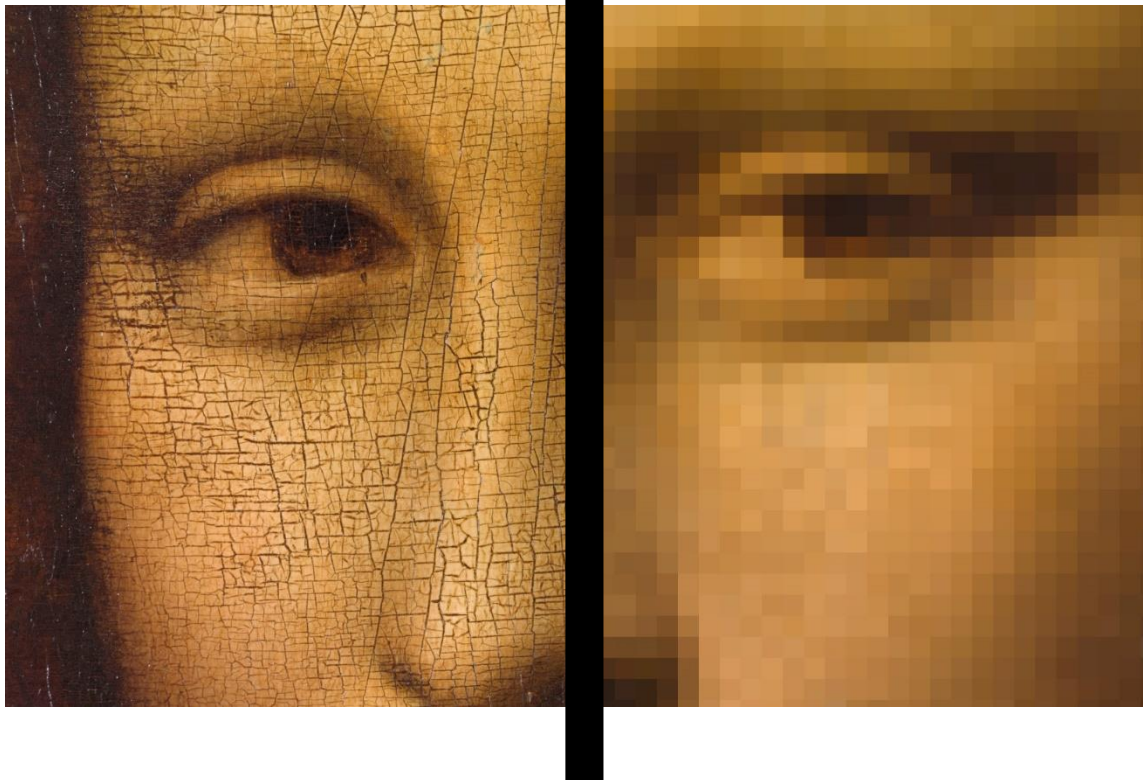
Adaptive Active Control of Boring Bar Chatter [View project](#)



Simulation and control of voice coil actuator (VCA) [View project](#)

STM32 ADC TUTORIAL

| with application to real-time control |



| Mohsen Fallah , Alireza Davodi , Mehran Mozaffari |









Fallah M, Davodi A, Mozaffari M. STM32 ADC TUTORIAL: with application to real-time control.
Mashhad: Ferdowsi University of Mashhad, CAD/CAM Laboratory; 2021. 65 p. Report No.: 2.

[DOI: 10.13140/RG.2.2.17875.30249/1](https://doi.org/10.13140/RG.2.2.17875.30249/1)

1. Introduction

This tutorial focuses on the fundamentals of communicating with the (Analog-to-Digital Conversion) ADC module of STM32 boards. The specific question of this tutorial is how to perform the ADC / DAC operations in both single and continuous modes with maximum sampling frequency in real-time mode. The presented algorithm implements a real-time phase inversion on the input signal read by the ADC module. A dummy PID controller is added to the developed codes to demonstrate the benchmark application of ARM CMSIS DSP library to the readers. The practical application of ADC module for single-channel real-time control is well-documented in this tutorial. What you will learn is how to:

-  Download the STM32 software packages
-  Compile ARM CMSIS 4.5.0 DSP library in STM32CubeIDE
-  Select between different modes of ADC module: Single or Continuous
-  Configure and utilize TIM / ADC / DAC / GPIO modules of the microcontroller
-  Program NUCLEO-F746ZG in STM32CubeIDE using C programming language
-  Examine the real-time performance of developed codes

All the texts highlighted in blue will direct you to the appropriate content that assist you to enrich your understanding while reading this tutorial. In order to improve the content of this tutorial, you can easily share your suggestions, ideas and improved codes or algorithms with us by commenting on this document's webpage or easily contacting us via email by sending your queries to: mohsen.fallah@gmail.com.

2. STM32 Software Packages

The ST has introduced the **STM32Cube ecosystem** that is a combination of software tools and embedded software libraries. It includes a wide range of PC software tools addressing all the needs of a complete project. Inside the STM32Cube ecosystem, you have access to the following software packages:

Table 1 Basic software packages inside STM32Cube ecosystem

STM32CubeMX	This easy-to-use graphical user interface software generates initialization C code for Cortex-M cores and generates the Linux device tree source for Cortex-A cores.
STM32CubeIDE	It is an Integrated Development Environment. Based on open-source solutions like Eclipse or the GNU C/C++ toolchain, this IDE includes compilation reporting features and advanced debug features. It also integrate additional features present in other tools from the ecosystem, such as the HW and SW initialization and code generation from STM32CubeMX.
STM32CubeProgrammer	It provides an easy-to-use and efficient environment for reading, writing and verifying devices and external memories via a wide variety of available communication media (JTAG, SWD, UART, USB DFU, I2C, SPI, CAN).
STM32CubeMonitor	Powerful monitoring tools that help developers fine-tune the behavior and performance of their applications in real-time.

In this tutorial, we utilized the **STM32CubeIDE** in order to program, compile and debug the code on the STM32 board. With STM32CubeIDE, there is no need to download another development environment such as **MDK-Arm Keil**. For each STM32 series, **STM32Cube MCU and MPU packages** offer all the required embedded software bricks to operate the available set of STM32 peripherals. Here, we have used the STM32 Nucleo-144 development board with STM32F746ZG MCU. Therefore, It is essential to download the **STM32Cube MCU Package for STM32F7 series**.

The template of the code is generated in C language by the **STM32CubeMX** that is included in the STM32CubeIDE software package. All selected peripheral devices of the STM32 board are initialized inside this template code. The activation or deactivation of these peripherals are done by using appropriate commands from **STM32F7 HAL Drivers**.

Apart from the STM32CubeMonitor, one can easily monitor run-time variables via the **STM Studio** that offers a range of useful monitoring and visualization tools for STM32 microcontrollers. It is also recommended to install the **STSW Link**. In addition, for those who are interested to deploy their application models in MATLAB and Simulink to STM32 MCUs, they can download the **STM32 embedded target for MATLAB and Simulink**.

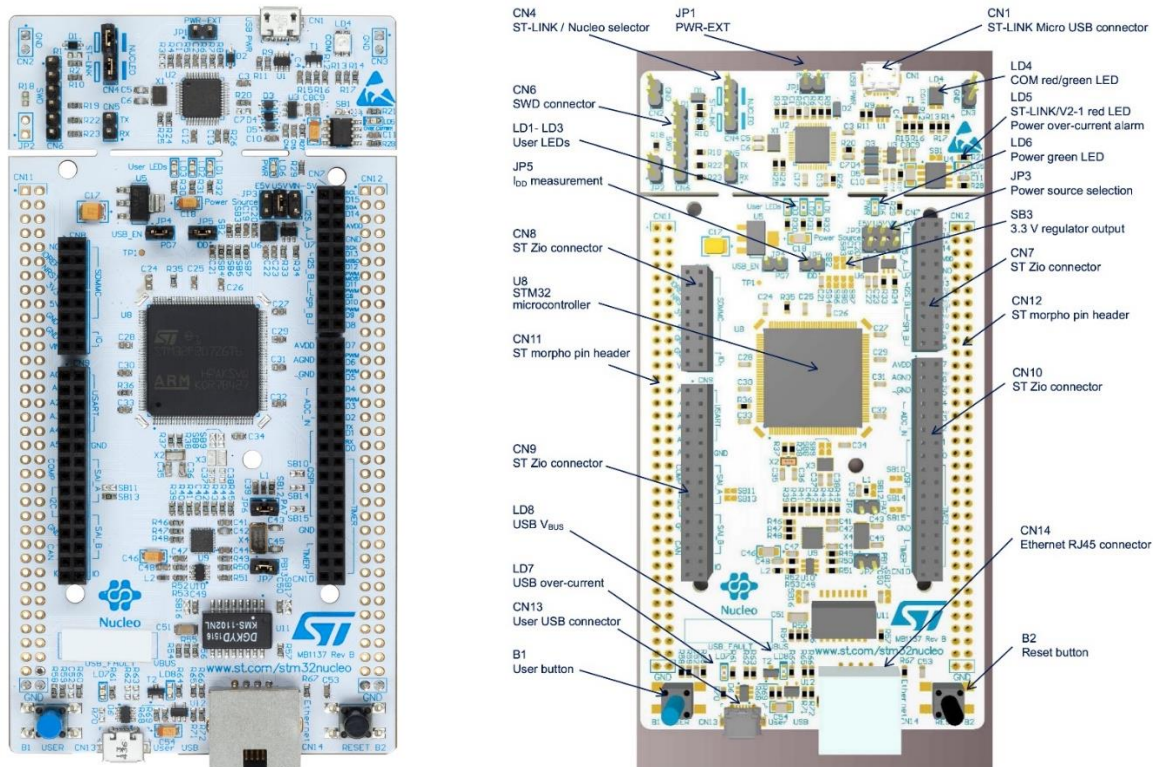


Fig. 1 The STM32 Nucleo-144 board ([NUCLEO-F746ZG](#))

3. STM32 Nucleo-144 development board with STM32F746ZG MCU

In this tutorial, we used one of the high-performance development boards from the STM32 family as shown in **Fig. 1**. The **STM32 Nucleo-144 board** (NUCLEO-F746ZG) provides

an affordable and flexible way for users to try out new concepts and build prototypes with STM32 microcontrollers, choosing from the various combinations of performance, power consumption and features.

4. Compile ARM CMSIS 4.5.0 DSP library in STM32CubeIDE

In the system identification and real-time control applications, it is always required to process the digital signals before sending them to the DAC module or after reading them from the ADC module. Therefore, the ARM CMSIS DSP library is always an indispensable part of the C program that is being developed inside the STM32CubeIDE. An open-source version of the ARM CMSIS library is always available on the GitHub. Here, we have downloaded and used the mature version of the [ARM CMSIS library 4.5.0](#).

After installing the STM32CubeIDE, launch the software from the link created on your desktop. The software asks you to select a directory as the workspace (*), where all the project files are generated. After a few seconds, the STM32CubeIDE Home window pops up. In this window, click on Start new STM32 Project (*). At this step the software tries to connect to the internet in order to download the latest files from the ST website. You may skip this step by disabling your internet connection for a few seconds.

Afterwards, another window opens that ask you to select the STM32 target. In the **Part Number** window type the name of the specific MCU for the STM32 board. Here, just type **STM32F746ZG** so that the software shows all the active products (*). Afterwards, on the MCU list select the **STM32F746ZGTx** and click on the Next button (*).

The next window sets up the basic features of the STM32 project (*) such as the project name, targeted language (**C**), targeted binary type (**Executable**), targeted project type (**STM32Cube**). After defining the project name, click on the Next button. In the subsequent window (*), you are able to set the firmware package name and version as well as the code generator option (**Copy only the nessecary library files**) and click on the Finish button. Finally, the STM32CubeIDE project window will be launched after a few seconds (*).

In order to access the available software packages, click on **Help > Manage Embedded Software Packages (*)**. You can find the available STM32Cube MCU Packages for the STM32F7 family in the opened window (*). If no package is available, you can either download the latest package by clicking on the From Url button or install the previously downloaded package by clicking on the From local button.

After downloading the ARM CMSIS library 4.5.0, copy the **arm_math.h** file from the library path ... \ **CMSIS** \ **Include** to the project path ... \ **Core** \ **Inc** (*). Then, create a folder named as **libs** in the main path of project directory (*). Afterwards, copy the file **libarm_cortexM7l_math** from the library path ... \ **CMSIS** \ **Lib** \ **GCC** (*) to the **libs** folder in the project path (*).

In the Project Explorer window (*), right click on the project title at the top and then click on the Properties tab or simply press **Alt + Enter** on the keyboard. Click on **C / C++ Build > Setting** in the opened window. On the Tool Settings tab (*), find and click on the **MCU GCC Linker > Libraries**. In Libraries (-l) section (*), click on the Add button and write the expression **arm_cortexM7l_math** in the opened window. Additionally, in Library search path (-L) section (*), click on the Add button and define the path to the **libs** folder in the project path.

On the Tool Settings tab, click on MCU Setting. Open the Floating-point ABI drop menu and select **Mix HW / SW implementation** (*). Finally, click on the Apply and Close button. In the Project Explorer window, click on **Core > Src > main.c**. This is the main file for the C program that will generally include all the user functions in any project. For initialization of the STM32F7 board peripherals, some standard functions will be added to this file as soon as we select and configure the required modules for our application via STM32CubeMX. One remaining step for the activation of the DSP library is to add two commands to the **main.c** file as shown below (*):


```

/* Private includes -----*/
/* USER CODE BEGIN Includes */
#define ARM_MATH_CM7
#include "arm_math.h"
/* USER CODE END Includes */

```

Please keep in mind that the user code should be written in-between specific lines inside the *main.c* file. For example, all user include files should be written between the `/* USER CODE BEGIN Includes */` and `/* USER CODE END Includes */`. The final step is to click on the **Project > Build All** so that all the project files are compiled by the IDE (*). All files will be built without any errors or warnings related to the addition of the DSP library to project directory.

5. ADC single and continuous modes

The ADC module of the STM32F7 board is able to convert the input data with 6-bits, 8-bits, 10-bits or 12-bits configurable resolution. The ADC operation can be either performed in single or continuous mode. One conversion per trigger is done in single mode, while successive conversions are launched in continuous mode by the ADC clock or a timer module after the first trigger is done. The single conversion mode is generally used for lower frequency operations, such as reading analog input data from the keypad, joystick or potentiometer, while the continuous conversion mode is recommended for higher frequency operations, such as reading analog data from the feedback sensors used in a controlled system. In order to maximize the speed of ADC operation, one can activate the Direct Memory Access (DMA) mode to facilitate the faster transfer of data from peripheral to memory. In addition, the STM32 NUCLEO-F746ZG board, in particular, has three ADC modules each of which has 16 input channels named by IN0 to IN15. Therefore, the ADC operation can be done on multiple channels, for example, to read the online data of multiple sensors connected to the STM32 board.

The speed at which the STM32 board interacts with the ADC and DAC modules is directly dependent on the time constant of the system under observation. In other words, the speed

with which the measured process variable responds to the changes in the controller output. In this tutorial, we have implemented the ADC operation in both single and continuous modes for execution of a single-channel ADC followed by some dummy mathematical computations, representing the processing of digital data by a PID controller. The processed signal is finally sent to the DAC module in order to complete the transfer path of the digital data inside the software code. The aim of this investigation is to determine the maximum frequency of a benchmark STM32F7 board for performing the single-channel real-time control operation. The conversion, processing and transfer of data from the ADC module to the DAC module is managed inside the callback function of a TIM module in order to control the time step of the entire control operation precisely. We have also examined the maximum feasible speeds of ADC and DAC peripherals in this dummy control scenario.

6. Configure TIM / ADC / DAC / GPIO modules

In this section, we configure the basic modules required for the control application. The essential parameters settings for the activation of TIM / ADC / DAC / GPIO modules on the STM32F7 device are defined in details. You should pay attention to the details of this section, since we are able to perform either of the ADC or DAC operations in single or continuous modes, which leads to four different versions of the real-time control code as provided in this tutorial (please refer to **Section 7**). Extra settings should be adjusted in order to perform the ADC / DAC operations in continuous mode. The speed of ADC operation in continuous mode is controlled by the ADC (peripheral) clock, while the speed of DAC operation in continuous mode is controlled by the TIM6 (timer) clock. Finally, the entire algorithm is executed inside the TIM2 global interrupt callback function. Therefore, the time step of the real-time control operation is dictated by the **Prescaler** and **Period** values of TIM2. However, the speed of single-channel ADC / DAC operations in continuous mode can exceed the frequency of control operation, as it is illustrated in this tutorial.

These four codes with different operation modes (either single or continuous) use different HAL functions and parameters settings to activate the ADC / DAC modules. Yet, they can generate similar results. In order to activate the STM32 board peripherals, you should click on the file with *.ioc extension* from the Project Explorer window so that the MCU of STM32 board shows up on the right hand side (*). From now on, we are using the STM32CubeMX software, which is integrated into the STM32CubeIDE software, for configuring the parameters of board modules including TIM, ADC, DAC and GPIO.

6.1. RCC Parameters

In order to activate the High Speed Clock (HSE), we should select the *System Core > RCC* from the Pinout and Configuration tab (*). In the RCC Mode window, set the HSE to *Crystal / Ceramic Resonator* (*). We do not make any other changes to the RCC Configuration window and leave all the parameters to their default values. The HSE Clock is very stable, and it is recommended to activate it for system identification and real-time control applications.

6.2. Clock Configuration

Then, open the Clock Configuration Tab to adjust the clock parameters via the STM32 clock tree. In this tab, you should make four changes to the clock tree. Firstly, set the Input frequency for the HSE to **8 MHz**. Secondly, select **HSE** from PLL Source Mux. Thirdly, select the **PLLCLK** from the System Clock Mux. Finally, set the HCLK frequency to the maximum value for the STM32F7 board, here, **216 MHz**. Click on the Resolve Clock Issues button on the top of this tab so that the software can find an admissible solution (*). As a result of these modifications to the STM32 clock tree, the following frequencies are adjusted for the APB1 and APB2 buses, as given in **Table 2**.

The DAC module uses the APB1 peripheral clock, while the ADC module uses the APB2 peripheral clock. On the other hand, the TIM2 and TIM6, which are used in this tutorial, utilize the APB1 timer clocks.

Table 2 Maximum frequency for APB1 / APB2 clocks

Buses Clocks	Value	Unit
APB1 peripheral clocks	54	[MHz]
APB1 timer clocks	108	[MHz]
APB2 peripheral clocks	108	[MHz]
APB2 timer clocks	216	[MHz]

6.3. TIM2 / TIM6 Parameters

In this project, we have activated two different timers on the STM32 board, namely the TIM2 and TIM6. The former is used to adjust the time step of the real-time control operation, while the latter is used to trigger the DAC operation in continuous mode. Therefore, you do not need to activate TIM6 if you would like to perform the DAC operation in single mode. However, the activation of TIM2 is necessary in all the developed codes. In order to activate the TIM2 module, we should firstly select the **Timers > TIM2** from the Pinout and Configuration tab (*). TIM2 is a general purpose timer. In the TIM2 Mode window, set the Clock Source to **Internal Clock** (*). In the Parameter Setting tab under Configuration window, set the **Counter Settings > Prescaler (16 bits)** to **1-1** and set the **Counter Settings > Counter Period (32 bits)** to **108-1**. The **Counter Settings > auto-reload preload** should be set to **Enable** (*). Finally, switch to the NVIC Settings tab under Configuration window and **Enable** the TIM2 global interrupt (*). These settings are the least timer settings that should be adjusted in all control codes.

Now you should activate TIM6 module in case you would like to perform the DAC operation in continuous mode. To do so, we should firstly choose the **Timers > TIM6** from the Pinout and Configuration tab (*). TIM6 is a basic timer that is specifically recommended for DAC operation. In fact, TIM6 is internally connected to the DAC module and is able to drive it through its trigger output. In the TIM6 Mode window, set the timer

status to **Activated** (*). In the Parameter Setting tab under Configuration window, set the **Counter Settings > Prescaler (16 bits)** to **1-1** and set the **Counter Settings > Counter Period (16 bits)** to **108-1**. The **Counter Settings > auto-reload preload** should be set to **Enable** (*). In addition, the **Trigger Output (TRGO) Parameters > Trigger Event Selection TRGO** should be set to **Update Event** (*). Finally, switch to the NVIC Settings tab under Configuration window and **Enable** the TIM6 global interrupt (*).

6.4. ADC Parameters

The activation of the ADC module is described in this section. Please consider that the NUCLEO board has three separate ADC modules each of which can support up to 16 input channels. Here, the focus of this tutorial is on the single-channel ADC operation using two different modes, i.e. single and continuous. To do so, we should firstly select the **Analog > ADC3** from the Pinout and Configuration tab (*). In the ADC3 Mode window, select the **IN3** which corresponds to pin PA3 on the MCU of STM32 board (*). In the Parameter Setting tab under Configuration window, only one change is required if you would like to execute the ADC operation in single mode. The **ADC_Regular_ConversionMode > Rank > Sampling Time** should be set to **15 Cycles** (*). Please note that the conversion of analog data with 12-bits of resolution requires at least 15 clock cycles. If we select multiple channels for ADC operation, the priority of these ADC channels can be defined under the **Rank**. Currently, the **ADC_Regular_ConversionMode > Number Of Conversion** is **1** and only one ADC channel is selected (*). Therefore, the first and only rank of the ADC operation is dedicated to **Channel 3** or equivalently pin PA3 on the MCU of STM32 board. These settings are sufficient for ADC operation in single mode. However, if the ADC operation is favored to be executed in continuous mode, the following modifications should be also done. In the Parameter Setting tab under Configuration window the **ADC_Settings > Scan Conversion Mode** and the **ADC_Settings > Continuous Conversion Mode** should be **Enabled** (*). Also the **ADC_Settings > DMA Continuous Requests** should be **Enabled** (*). But before that, the DMA feature should be activated for the ADC module. To do so, switch to the DMA Settings tab under Configuration window and click on the Add button.

Then, select **ADC3** from the drop menu (*). On the DMA Request Settings set the Mode to **Circular**. The Data Width is set to **Half Word** as the STM32 board ADC module has a maximum resolution of 12-bits (*). Finally, switch to the NVIC Settings tab under Configuration window and **Enable** the ADC3 global interrupt (*).

6.5. DAC Parameters

In this section, we have described the activation of the DAC module. Firstly, we should select the **Analog > DAC** from the Pinout and Configuration tab (*). The NUCLEO board has two separate channels for its DAC module. In the DAC Mode window, select the **OUT1 Configuration** (*). This DAC channel corresponds to pin PA4 on the MCU of STM32 board. In the Parameter Setting tab under Configuration window, there is no need to make any other changes if you would like to execute the DAC operation in single mode. The **DAC Out1 Settings > Output Buffer** is set by default to **Enable**. However, if you decide to execute the DAC operation in continuous mode, set the following configurations as well. The **DAC Out1 Settings > Trigger** should be set to **Timer 6 Trigger Out event** (*). In addition, switch to the DMA Settings tab under Configuration window and click on the Add button. Then, select **DAC1** from the drop menu (*). On the DMA Request Settings set the Mode to **Circular**. The Data Width is set to **Half Word** as the STM32 board DAC module has a maximum resolution of 12-bits (*).

6.6. GPIO Parameters

In this section, we have activated three GPIO pins that are connected to the on-board LEDs of the STM32 NUCLEO board. Move the pointer to the PB0 pin of the MCU and press the left button of your mouse. This pin is located on the bottom edge of the MCU. Then select **GPIO_Output** from the appeared drop menu (*). This will activate the green LED on the board. Similarly, move the pointer to the PB7 pin located on the right edge of the MCU and repeat the same steps to activate the blue LED on the board (*). Finally, move to the top edge of the MCU and find the PB14 corresponding to the red LED on the board. Repeat the same steps to activate this LED as well (*).

6.7. Generate Code

All the parameters settings are now adjusted correctly. Please note that we introduced some extra settings for the cases in which either each or both of the ADC and DAC operations are executed in continuous mode by the activation of DMA settings. In order to generate the standard code that adds the initialization functions for the activated modules and pins to the *main.c* file, click on **Project > Generate Code** from the menu bar (*). In order to create a report about the project, you can simply click on **Project > Generate Report** (*). A report file with *.pdf extension* is added to the project directory that includes a description of all the settings you have made for the activation of peripherals and timers in the STM32CubeMX environment.

7. Program NUCLEO-F746ZG in STM32CubeIDE using C language

In this section, we will take a close look to the template and sections of the C codes provided for this tutorial. All four versions of the code are described in details. Minor modifications distinct different versions of the code from one another, however, these minor variations represent different methods of activating the ADC / DAC modules on the STM32 board that can be used by the users.

7.1 Template of C code

The standard template of the code in the *main.c* file is explained in this section. The *main.c* file is where we combine our user defined codes with the codes automatically generated for initialization of the STM32 board peripherals. We tend to add our user defined codes in specific locations throughout this standard template. An overall look to the code template reveals that different blocks of the code are written before, inside and after the `int main(void)` function. As shown in **Fig. 2**, the blue boxes are where we add the blocks of user defined code, while the orange boxes are the locations where the software automatically puts the blocks of code related to initialization of the STM32 peripherals.

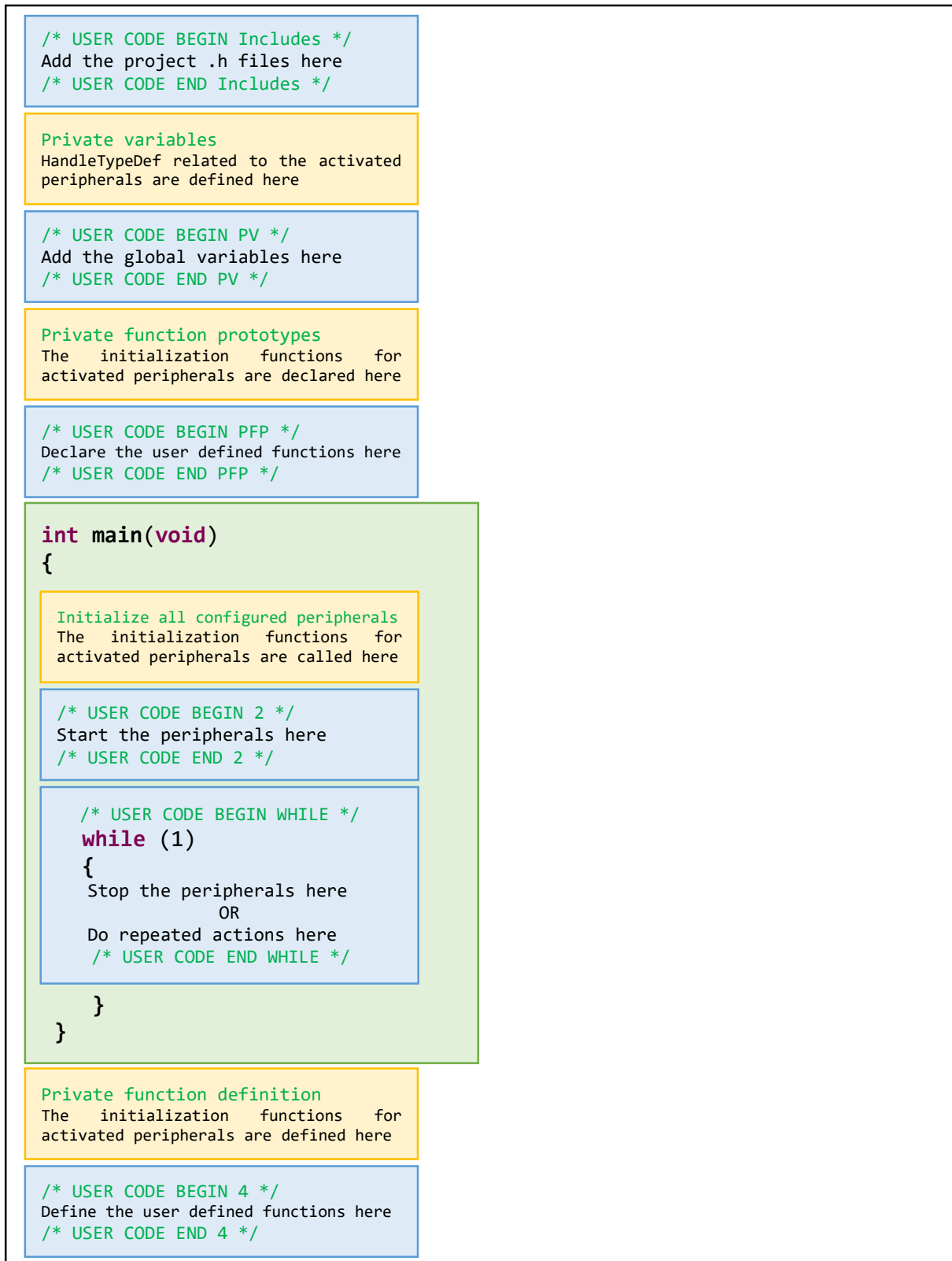


Fig. 2 Illustration of standard template for the C code in *main.c* file

The sections of the code related to initialization of the STM32 peripherals are automatically generated after one selects all the required parameters settings in the STM32CubeMX environment and clicks on the **Project > Generate Code** from the menu bar (*). However, we are responsible to add all the user defined variables and functions in appropriate locations throughout this standard template. Firstly, the include files of the project should be added between `/* USER CODE BEGIN Includes */` and `/* USER CODE END Includes */`. Here, we only use the ARM math functions, and therefore we should include its header file, *arm_math.h*, at the beginning of the code. Secondly, the definition of all global variables with different data types and sizes should be presented between `/* USER CODE BEGIN PV */` and `/* USER CODE END PV */`. All the defined variables can be called either inside the user defined functions or `int main(void)` function. We may initialize some variables here, and later modify their values inside the functions defined by user. The variables originally defined inside any function are considered as local, and cannot be accessed by other user defined functions. The third step is to declare the prototype of user defined functions between `/* USER CODE BEGIN PFP */` and `/* USER CODE END PFP */`. Additionally, the definitions of these functions are presented right after the `int main(void)` function between `/* USER CODE BEGIN 4 */` and `/* USER CODE END 4 */`.

In the `int main(void)` function, the user code can be written either outside or inside the infinite loop. We usually start the TIM / ADC / DAC modules of STM32 board before the start of infinite loop between `/* USER CODE BEGIN 2 */` and `/* USER CODE END 2 */`. However, we may also initialize some variables or create lookup tables and do some required mathematical computations here. Finally, the exit condition for stoppage of the STM32 modules can be presented inside the infinite loop between `/* USER CODE BEGIN WHILE */` and `/* USER CODE END WHILE */`. Generally, any repetitive computation may be executed inside the standard infinite `while` (1) loop. Especially, if the time step of code execution inside this infinite loop is not an essential requirement for the implementation of the whole code. However, in real-time control applications, we usually tend to have a strict control over the time step of executing repetitive actions. These actions usually include

online reading of the feedback data from ADC module, filtering of input data through the controller block and sending of modified data as command signal to the DAC module. In order to do so, we usually use the callback functions of TIM / ADC / DAC modules, which are repetitively called at certain time intervals or after the completion of a single or a series of ADC or DAC. In this tutorial, the transfer path of digital data from the ADC module to the DAC module takes place inside the callback function of a TIM module, which is responsible for controlling the time step of real-time control operation.

7.2 Algorithm of the developed codes

The developed codes can be used as benchmark examples of how we can read, modify and write digital data in real-time for developing either open-loop or closed-loop control applications by using the capabilities of STM32 boards. What takes place at the core of these codes are similar. The illustration of single-channel real-time dummy control algorithm is depicted in **Fig. 3**.

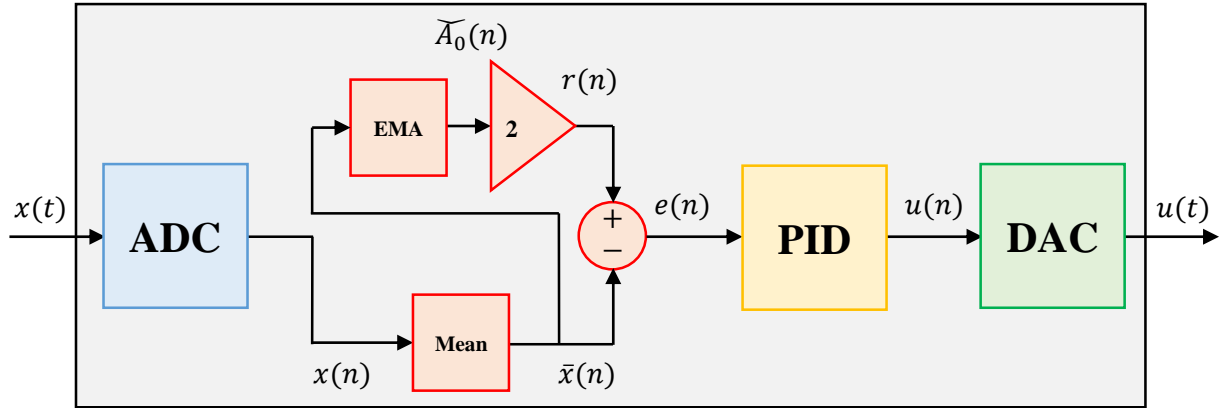


Fig. 3 Illustration of single-channel real-time dummy control algorithm

The analog input signal read from the ADC module can be generally composed of a time varying component, $Ag(h(t))$, plus a DC component, A_0 . According to **Eq. 1**, we assume that $h(t)$ is a harmonic function with linearly time dependent frequency, $f(t)$, that changes from F_s to F_f over T_c seconds. A is the constant amplitude of the time dependent component of input signal and φ_0 is the constant initial phase of the harmonic function:

$$\begin{aligned}
x(t) &= Ag(h(t)) + A_0 \\
h(t) &= \cos(2\pi f(t)t + \varphi_0). \quad f(t) = F_s + \frac{\alpha}{2}t. \quad \alpha = \frac{F_f - F_s}{T_c}
\end{aligned} \tag{1}$$

The shape of the input signal can be either harmonic, triangular or square based upon the definition of $g(t)$:

$$g(t) = \begin{cases} t & \text{harmonic} \\ \frac{2}{\pi} \sin^{-1}(t) & \text{triangular} \\ \text{sgn}(t) & \text{square} \end{cases} \tag{2}$$

When the analog signal passes through the ADC module, it turns into its equivalent digital value $x(n)$, which in a 12-bits conversion lays between 0 and 4095. In the presented algorithm, $x(n)$ is firstly stored in a predefined buffer array that memorizes the current and past values of the input signal. Here, the buffer array is suggested to have a maximum size of five, which means that the it can store the last five values of input signal read from the ADC module. It is then filtered by the `arm_mean_f32` function, which computes the mean value of input signal over the buffer array length, $\bar{x}(n)$. This averaging of input signal can assist to reduce the noise or undesirable oscillations of the input signal. Especially, when the sampling rate of input signal is much higher than its maximum frequency content. In order not to filter the frequency content of the input signal, it is recommended to get at least 20 samples across one period of the wave with the highest frequency. That is why the averaging of input signal is done with a short buffer size. For the special case of unit buffer size, as used in the developed codes, it is obvious that $\bar{x}(n) = x(n)$. Therefore, the `arm_mean_f32` function only does a trivial modification on the digital signal.

According to **Fig. 3**, the average value of signal, $\bar{x}(n)$, is then passes through the Exponential Moving Average (EMA) function in order to estimate the DC component of the input signal. The EMA function tries to predict the DC component of the signal, $\widetilde{A}_0(n)$, based upon a recursive formula that considers a fraction of the signal's current value as well as a history of its past values, as follows:

$$\widetilde{A}_0(n) = \gamma \bar{x}(n) + (1 - \gamma) \widetilde{A}_0(n - 1). \quad \widetilde{A}_0(0) = 2048 \quad (3)$$

The coefficient γ represents the degree of weighting decrease, which is a factor between 0 and 1. A higher value of γ discounts older observations faster. Here, its value is considered to be 0.0002 so that the EMA fastly converges to the DC component of the input signal. The initial estimation of signal's DC component, $\widetilde{A}_0(0)$, can be considered as any value between 0 and 4095. Here, it is assumed to be equal to 2048. The instantaneous value for the reference signal is taken as twice the instantaneous value of $\widetilde{A}_0(n)$, and the error signal is thus generated by comparing the feedback signal $\bar{x}(n)$ with the reference signal $r(n)$ as follows:

$$e(n) = r(n) - \bar{x}(n) \quad (4)$$

From what just has been said, we know that:

$$\begin{aligned} x(n) &= Ag(h(n)) + A_0 \\ \bar{x}(n) &= x(n) \\ r(n) &= 2\widetilde{A}_0(n) \approx 2A_0 \end{aligned} \quad (5)$$

therefore, **Eq. 4** can be rewritten as follows:

$$e(n) \approx 2A_0 - (Ag(h(n)) + A_0) \approx A_0 - Ag(h(n)) \quad (6)$$

which indicates that $e(n)$ is the anti-phase signal constructed directly from the original input signal $x(n)$. In other words, $e(n)$ has the same magnitude as $x(n)$, but its phase difference is 180 degrees (π radians). Finally, the error signal, $e(n)$, is passed through a PID controller with unit proportional gain, $K_p = 1$, and zero integral / derivative gains, $K_i = K_d = 0$, by using the `arm_pid_f32` function:

$$u(n) = \left(K_p + \frac{K_i T_s z}{z - 1} + \frac{K_d N(z - 1)}{(1 + NT_s)z - 1} \right) e(n) \quad (7)$$

The given discrete-time PID controller is defined by using the backward Euler method for both the integral and derivative terms. The specific description of `arm_pid_f32` function's algorithm can be found on Keil website (*). Again, this dummy PID control function does a trivial modification on $e(n)$ to construct the command signal $u(n)$, which will be then sent to the DAC module. As a result, the analog output signal, $u(t)$, read from the DAC module has an opposite phase in comparison to the analog input signal, $x(t)$, sent to the ADC module. The idea of signal phase inversion is clearly shown in **Fig. 4**.

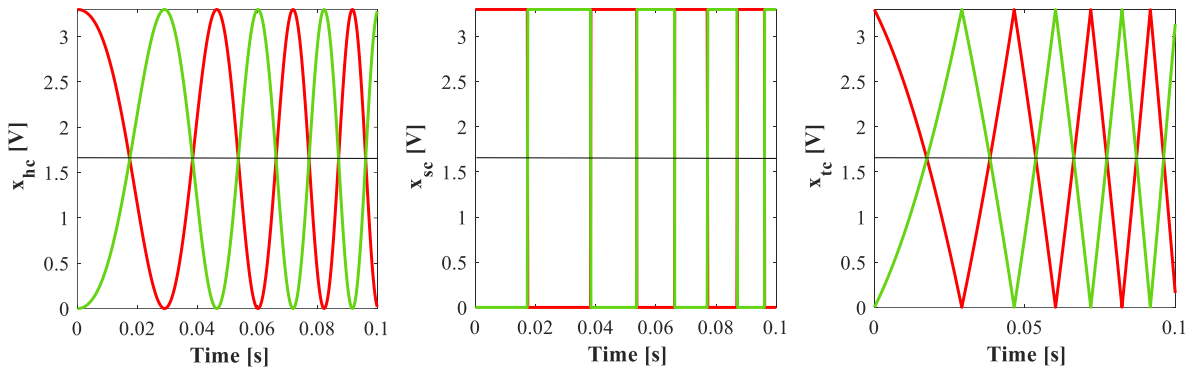


Fig. 4 Comparison of the input chirp signal $x(t)$ and the antiphase command signal $u(t)$

In fact, what the presented dummy control algorithm does is a real-time procedure for phase reversal or phase inversion of the input signal. The concept of phase reversal is commonly used in the context of active noise control, where the objective of real-time control is to generate an anti-noise signal that is in opposite phase with the original noise signal. The destructive interference of these noise and anti-noise signals results in perfect cancellation of unwanted noise. Here, we used some prepared functions from the ARM CMSIS DSP library such as `arm_mean_f32` and `arm_pid_f32`, which are generally used in a closed-loop control application. However, the output of these functions are the same as their input at every instance of time, and therefore these functions do not modify the instantaneous values of the digital signal in any way. That's why we call the presented algorithm as a real-time dummy control operation, which in fact only performs the phase inversion on the input signal.

7.3 Single conversion ADC / single conversion DAC

The first version of dummy control code deals with the primary mode of communication with the ADC / DAC modules, which is the single conversion mode. The code begins with the definition of global variables required for adjusting the parameters of TIM / ADC / DAC modules plus the introduction of variables required for the definition of the dummy PID controller.

The dummy control operation takes place inside the callback function of a TIM module. Since TIM2 is used as the specific timer that controls the frequency of entire operation, the variable *MyClkFreq* is equal to the maximum frequency of the APB1 timer clocks, which is equal to **108 MHz** (please refer to **Table 2**). The **Period Counter** or its equivalent variable, *MyPeriod*, was set to **108-1**, and the value of **Prescaler** or its equivalent variable, *MyPrescaler*, was set to **20-1**. These values are selected so that the entire control algorithm can be executed in real-time. This means that the maximum frequency of control operation is equal to **50 kHz** according to the following formula:

$$F_{TIM} = \frac{MyClkFreq}{(MyPeriod + 1) \times (MyPrescaler + 1)}. \quad dt = \frac{1}{F_{TIM}} \quad (8)$$

The time step of control operation is simply equal to the inverse of the timer's frequency, F_{TIM} , and if we would like to execute the control operation in real-time mode, we should be sure that all the conversions and computations performed on the digital signal should be completed faster than the passage of one time step dt . Our observations show that by using the STM32 board, the single-channel control operation can be performed in real-time mode with a maximum frequency of 50 kHz (by using TIM2 module). Enthusiast readers may investigate the possibility of increasing the speed of control operation beyond 50 kHz by using an advanced timer from APB2 bus, i.e. TIM1 or TIM8, that utilizes a higher frequency clock in comparison to TIM2. If we increase the timer's frequency by decreasing the value of *MyPrescaler* below the recommended value, the control algorithm will be

executed with a significant amount of time delay. Therefore, the parameters of TIM2 are kept constant in all versions of the developed codes.

```
/* USER CODE BEGIN PV */
// TIM PARAMETERS
uint8_t chvar; // Check Variable
uint32_t MyClkFreq = 108000000; // Fixed APB1 Clock Frequency [Hz]
uint32_t MyPrescaler = 20-1; // Minimum Prescaler Value [cycles]
uint32_t MyPeriod = 108-1; // Fixed Period Value [cycles]
uint32_t TIMcount = 0; // Timer Counter
uint32_t TRESET = 0; // Timer Reset Index
double t = 0; // Time Variable [s]
double tstop = 20; // Finish Time [s]
// ADC PARAMETERS
#define buffer_size 1 // Buffer Size for ADC Buffer Array
uint8_t data_ready = 0; // ADC Status Index [0 = not ready , 1 = ready]
uint32_t ADCValue[1] = {0}; // Current ADC Value
uint32_t BUFcount = 0; // Buffer Counter
float32_t ADCBuffer[buffer_size] = {0}; // ADC Buffer Array
float32_t feedbackValue = 0; // Current Feedback Value
// CONTROL PARAMETERS
#define PID_GAIN_KP 1 // Proportional Gain Value
#define PID_GAIN_KI 0 // Integral Gain Value
#define PID_GAIN_KD 0 // Derivative Gain Value
float KP; // Proportional Gain Variable
float KI; // Integral Gain Variable
float KD; // Derivative Gain Variable
arm_pid_instance_f32 MyPID; // PID Structure
float32_t PID_in , PID_out; // PID Input and Output Values
float32_t referenceValue; // Reference Value
float32_t errorValue = 0; // Current Error Value
float32_t signalMean = 2048; // Initial Mean Value [Here = 1.65 volt]
// DAC PARAMETERS
uint32_t DACValue[1] = {0}; // Current DAC Value
float32_t commandValue = 0; // Current Command Value
/* USER CODE END PV */
```

The next step is to define the user function prototypes, as shown below. The code includes eleven private functions that will be later introduced after the `int main(void)` function. These user functions are declared right after the declaration of initialization functions for the STM32 modules (please refer to **Fig. 2**).

```
/* USER CODE BEGIN PFP */
void MY_BLINK(void); // Function : Blinking Green LED //
void MY_INPUT_CHECK(void); // Function : Checks Variables and Initialize Controller //
void MY_CONTROL_START(void); // Function : Starts TIM-ADC-DAC Operations //
void MY_TIM_UPDATE(uint32_t* TCOUNT); // Function : Update Timer Counter //
void MY_ADC_UPDATE(void); // Function : Reads the Current Data from ADC Channel //
void MY_DUMMY_CONTROL(void); // Function : Passes the Data through PID Controller //
void MY_DAC_UPDATE(void); // Function : Writes the Next Data to DAC channel //
// Function : Computes Exponential Moving Average //
void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN);
uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT); // Function : Saturate DAC Output //
void TEST_MY_CODE (void); // Function : Test the Correct Execution of Code //
void MY_CONTROL_STOP(double* TIMVAL); // Function : Stops TIM-ADC-DAC Operations //
/* USER CODE END PFP */
```

The first block of code inside the `int main(void)` function is used to initialize the PID structure by setting the constant PID gains as well as starting all the required peripherals of the STM32 board.

```
/* USER CODE BEGIN 2 */
///// CHECK INPUT AND INITIALIZE CONTROLLER /////
MY_BLINK( );
MY_INPUT_CHECK( ); // Check Input Variables
MY_BLINK( );
///// START THE TIM-ADC-DAC MODULES /////
MY_CONTROL_START( ); // Starts the Control Operation
/* USER CODE END 2 */
```

The second block of code inside the `int main(void)` function is written inside the infinite `while (1)` loop. It is used to test the execution of code during the control operation and stopping all the STM32 peripherals as soon as the time of control operation finishes.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    ///// FINISH THE TIM-ADC-DAC MODULES /////
    TEST_MY_CODE( ); // Test the Execution of Code
    MY_CONTROL_STOP(&t); // Stops the Control Operation
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

The description of user defined functions are presented after the `int main(void)` function. Some of these user defined functions are the most important ones that their definitions may slightly vary from code to code. Firstly, `void MY_CONTROL_START(void)` is the function that starts the TIM / ADC / DAC modules of the STM32 board with the predefined settings. Similarly, `void MY_CONTROL_STOP(double* TIMVAL)` is the function that stops the TIM / ADC / DAC modules after the code has been executed for `tstop` seconds. Additionally, the `void MY_TIM_UPDATE(uint32_t* TCOUNT)` function updates the timer counter variable, `TIMcount`, and computes the time of control operation by updating the time variable, `t`.

Secondly, the function `void MY_ADC_UPDATE(void)` is responsible for converting the instantaneous values of analog signal to its corresponding digital values as well as storing the converted digital signal in the `ADCBuffer` array. Every iteration that `void MY_ADC_UPDATE(void)` is executed, the `HAL_ADC_Start(&hadc3)` command is used to start the ADC module followed by the `HAL_ADC_GetValue(&hadc3)` command which reads the input data from the single ADC channel. In the single conversion mode, the task of ADC module is completed after reading a single data point, and therefore the conversion of analog signal does not continue.

Thirdly, the function `void MY_DUMMY_CONTROL(void)` is written to implement the dummy control algorithm as described in **Section 7.2**. Inside this function, the digital signal passes through some prepared functions from the ARM CMSIS DSP library such as `arm_mean_f32` and `arm_pid_f32`. The output of PID controller, `PID_out`, is directly computed by filtering the input of PID controller, `PID_in`, with the `&MyPID` structure that stores the constant PID gains. This is done by using the command `PID_out = arm_pid_f32(&MyPID, PID_in)`. In addition, the function `void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN)` is used to estimate the DC component of the input signal. The required mathematical computations are performed in the `void MY_DUMMY_CONTROL(void)` function to construct the output antiphase signal from the input signal.

Finally, the function `void MY_DAC_UPDATE(void)` is responsible for restricting the values of output signal prior to sending them to the DAC module by using the `HAL_DAC_SetValue(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACValue[0])` command. Since the DAC module is capable of performing 12-bits conversion, the output digital values should not be greater than 4095. That is why the output of digital PID controller should be saturated by passing through the `uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT)`. Every iteration that `void MY_DAC_UPDATE(void)` is executed, a new output value is written to the DAC module by using the `HAL_DAC_SetValue` command. It is obvious that when we perform the ADC / DAC operations in single mode, the frequency of interaction with these modules is equal to the frequency of control operation governed by the TIM2 module, i.e. 50 kHz.

```
/* USER CODE BEGIN 4 */
///// Function : Blinking Green LED //////////////////////////////////////
void MY_BLINK(void)
{
    for (int ii = 0 ; ii < 5 ; ii++)
    {
        HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_0);
        HAL_Delay(200);
    }
}
```

```

    }
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);
}
//// Function : Checks Variables and Initialize Controller //////////
void MY_INPUT_CHECK(void)
{
    // Check Timer Parameters //
    chvar = 0; // Check Variable
    (MyClkFreq != 108000000) ? (MyClkFreq = 108000000 , chvar = 1) : 0;
    (MyPrescaler < 20-1) ? (MyPrescaler = 20-1 , chvar = 1) : 0;
    (MyPeriod != 108-1) ? (MyPeriod = 108-1 , chvar = 1) : 0;
    TRESET = MyClkFreq / (MyPrescaler + 1) /(MyPeriod + 1); // Timer Reset Index
    if (chvar == 1)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_SET);
        HAL_Delay(1000);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_RESET);
    }
    // Initialize Controller Parameters //
    KP = PID_GAIN_KP;
    KI = PID_GAIN_KI;
    KD = PID_GAIN_KD;
    MyPID.Kp = KP; // Set the Proportional Gain
    MyPID.Ki = KI; // Set the Integral Gain
    MyPID.Kd = KD; // Set the Derivative Gain
    arm_pid_init_f32(&MyPID, 1); // Initialize the Controller
}
//// Function : Starts TIM-ADC-DAC Operations //////////////////////////////////////
void MY_CONTROL_START(void)
{
    htim2.Init.Prescaler = MyPrescaler;
    htim2.Init.Period = MyPeriod;
    HAL_TIM_Base_Init(&htim2);
    HAL_TIM_Base_Start_IT(&htim2); // Start TIM2 Module in Interrupt Mode
    //HAL_ADC_Start(&hadc3) ;// Start the ADC Module
    HAL_DAC_Start(&hdac,DAC_CHANNEL_1); // Start the DAC Module
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 , GPIO_PIN_SET);
}

```

```

}
//// Function : Update Timer Counter //////////////////////////////////
void MY_TIM_UPDATE(uint32_t* TCOUNT)
{
    *TCOUNT += 1; // Update Timer Counter
    // Reset Timer Counter and Update Time Variable
    (*TCOUNT == TRESET) ? (*TCOUNT = 0 , t += 1) : 1;
}
//// Function : Reads the Current Data from ADC Channel //////////
void MY_ADC_UPDATE(void)
{
    HAL_ADC_Start(&hadc3); // Start the ADC3 Module
    ADCValue[0] = HAL_ADC_GetValue(&hadc3); // Get Current ADC Value
    BUFcount += 1; // Update Buffer Counter
    ADCBuffer[BUFcount-1] = (float32_t)ADCValue[0]; // Fill ADC Buffer Array
    (BUFcount == buffer_size) ? (BUFcount = 0) : 1; // Reset Buffer Counter
    data_ready = 1; // Update ADC Status Index
}
//// Function : Passes the Data through PID Controller //////////
void MY_DUMMY_CONTROL(void)
{
    // Compute Mean Value of ADC Buffer Array
    arm_mean_f32(ADCBuffer, (uint32_t)buffer_size, &feedbackValue);
    // Compute Mean Value of Input Signal (referenceValue)
    MY_EXP_MA(0.0002, feedbackValue, &signalMean);
    referenceValue = 2 * signalMean; // Reference Value
    errorValue = referenceValue - feedbackValue; // Compute Error Value
    PID_in = errorValue; // Set PID Input
    PID_out = arm_pid_f32(&MyPID, PID_in); // Compute PID Output
}
//// Function : Writes the Next Data to DAC channel //////////
void MY_DAC_UPDATE(void)
{
    commandValue = PID_out; // Set Command Value
    DACValue[0] = MY_LIMIT_OUTPUT((uint32_t)commandValue); // Set Next DAC Value
    // Write the Next Value to DAC Channel

```

```

        HAL_DAC_SetValue(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACValue[0]);
    }

    /// Function : Computes Exponential Moving Average ///
    void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN)
    {
        *SIGMEAN = gamma * SIGVAL + (1 - gamma) * (*SIGMEAN);
    }

    /// Function : Saturate DAC Output ///
    uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT)
    {
        return (INPUT > 4095) ? (4095) : (INPUT);
    }

    /// Function : Test the Correct Execution of Code ///
    void TEST_MY_CODE(void)
    {
        if (DACValue[0] == 4095) // ADCValue[0] == 4095 (Connected to 3.3 V - VCC)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
        }
        else if (DACValue[0] == 0) // ADCValue[0] == 0 (Connected to 0 V - GND)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET);
        }
        else // 0 < ADCValue[0] < 4095 (Not Connected to either GND or VCC)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_14, GPIO_PIN_RESET);
        }
    }

    /// Function : Stops TIM-ADC-DAC Operations ///
    void MY_CONTROL_STOP(double* TIMVAL)
    {
        if (*TIMVAL == tstop)
        {
            HAL_TIM_Base_Stop_IT(&htim2); // Stop the TIM2 Module in Interrupt Mode
        }
    }

```



```

        HAL_ADC_Stop(&hadc3); // Stop the ADC3 Module
        HAL_DAC_Stop(&hdac, DAC_CHANNEL_1); // Stop the DAC Module
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_7 | GPIO_PIN_14 , GPIO_PIN_RESET);
    }
}

///// Function : HAL_TIM_PeriodElapsedCallback //////////////////////////////////
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    UNUSED(htim);
    MY_TIM_UPDATE(&TIMcount); // Update Timer Counter
    MY_ADC_UPDATE( ); // Update ADC Operation - Read Current Data
    MY_DUMMY_CONTROL( ); // Update Control Operation - Process Data
    MY_DAC_UPDATE( ); // Update DAC - Write Next Data
}

////////////////////////////////////
/* USER CODE END 4 */

```

Regarding the other user defined functions, the `void MY_BLINK(void)` function only sets and resets the green LED on the board at the beginning of code execution. The function `void MY_INPUT_CHECK(void)` checks the defined parameters for the TIM module, and if the user inputs override the recommended values, this function will modify the inputs. At the end of this function, all LEDs will be turned on for a second as a signal to the user that the TIM parameters were not adjusted properly and have been corrected accordingly. This function also sets the values of proportional, integral and derivative, {*Kp*, *Ki*, *Kd*}, gains inside the `&MyPID` structure by using the command `arm_pid_init_f32(&MyPID, 1)`. In addition, the `void TEST_MY_CODE(void)` is a function that is repeatedly called inside the infinite loop. If we connect the input ADC pin to the GND pin, it will turn the green LED on as a signal to the user that the ADC channel is currently connected to the lowest voltage level (0 V). On the other hand, if we connect the input ADC pin to the VCC pin, it will turn the red LED on to signify the user that the ADC channel is connected to the highest voltage level (3.3 V).

Last but not least, the `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function, which is originally defined in `stm32f7xx_hal_tim.c`, is the callback function that is continuously executed during each iteration of time. It manages the real-time execution of the entire control operation. Inside this function, four of the user defined functions are executed repetitively in order to perform the code's algorithm properly as presented in **Fig. 3**. The `MY_TIM_UPDATE(&TIMcount)` updates the control time and it is followed by the `MY_ADC_UPDATE()` that reads the input data from the ADC channel. The `MY_DUMMY_CONTROL()` modifies the input data according to the developed dummy control algorithm and finally `MY_DAC_UPDATE()` sends the appropriate output data to the DAC channel. All these actions should be executed faster than the time increment selected for the control time step (please refer to **Eq. 8**).

7.4 Single conversion ADC / continuous conversion DAC

The second version of dummy control code is developed to investigate the possibility of communicating with the DAC module in continuous mode, while the ADC module is called in single conversion mode. The aim is to find the maximum speed with which we are able to interact with the DAC module while performing a single-channel real-time control operation. The code begins with the definition of global variables required for adjusting the parameters of TIM / ADC / DAC modules plus the introduction of variables required for the definition of the dummy PID controller.

```
/* USER CODE BEGIN PV */
// TIM PARAMETERS
uint8_t chvar; // Check Variable
uint32_t MyClkFreq = 108000000; // Fixed APB1 Clock Frequency [Hz]
uint32_t MyPrescaler = 20-1; // Minimum Prescaler Value [cycles]
uint32_t MyPeriod = 108-1; // Fixed Period Value [cycles]
uint32_t TIMcount = 0; // Timer Counter
uint32_t TRESET = 0; // Timer Reset Index
double t = 0; // Time Variable [s]
double tstop = 20; // Finish Time [s]
```

```

// ADC PARAMETERS
#define buffer_size 1 // Buffer Size for ADC Buffer Array
uint8_t data_ready = 0; // ADC Status Index [0 = not ready , 1 = ready]
uint32_t ADCValue[1] = {0}; // Current ADC Value
uint32_t BUFcount = 0; // Buffer Counter
float32_t ADCBuffer[buffer_size] = {0}; // ADC Buffer Array
float32_t feedbackValue = 0; // Current Feedback Value
// CONTROL PARAMETERS
#define PID_GAIN_KP 1 // Proportional Gain Value
#define PID_GAIN_KI 0 // Integral Gain Value
#define PID_GAIN_KD 0 // Derivative Gain Value
float KP; // Proportional Gain Variable
float KI; // Integral Gain Variable
float KD; // Derivative Gain Variable
arm_pid_instance_f32 MyPID; // PID Structure
float32_t PID_in , PID_out; // PID Input and Output Values
float32_t referenceValue; // Reference Value
float32_t errorValue = 0; // Current Error Value
float32_t signalMean = 2048; // Initial Mean Value [Here = 1.65 volt]
// DAC PARAMETERS
uint32_t DACValue[1] = {0}; // Current DAC Value
float32_t commandValue = 0; // Current Command Value
/* USER CODE END PV */

```

In order to activate the DAC module in continuous mode, we have to activate TIM6 to trigger the DAC in continuous mode. Please take a look at **Section 6.3** and **Section 6.5** in order to remember the specific settings associated with the activation of DAC module with DMA feature. As a result of the extra settings done, the parameters configured for initialization of DAC module inside `static void MX_DAC_Init(void)` function will be modified in comparison to the previous code. Additionally, an initialization function named `static void MX_DMA_Init(void)` is added to the other functions, which enables the DMA feature of the DAC module that facilitate faster transfer of data from memory to peripheral.

Similar to the previous code, the dummy control operation takes place inside the callback function of TIM2 module. MyPeriod was set to **108-1** and MyPrescaler was set to **20-1** so

that the maximum frequency of control operation is equal to **50 kHz** according to **Eq. 8**. However, now that we have activated TIM6 for triggering the DAC module, we can adjust its *Period Counter* and *Prescaler* values differently in order to interact with the DAC module at a faster rate than the frequency of control operation. The next step is to define the user function prototypes, as shown below. These user functions are declared right after the declaration of initialization functions for the STM32 modules.

```
/* USER CODE BEGIN PFP */
void MY_BLINK(void); // Function : Blinking Green LED //
void MY_INPUT_CHECK(void); // Function : Checks Variables and Initialize Controller //
void MY_CONTROL_START(void); // Function : Starts TIM-ADC-DAC Operations //
void MY_TIM_UPDATE(uint32_t* TCOUNT); // Function : Update Timer Counter //
void MY_ADC_UPDATE(void); // Function : Reads the Current Data from ADC Channel //
void MY_DUMMY_CONTROL(void); // Function : Passes the Data through PID Controller //
void MY_DAC_UPDATE(void); // Function : Writes the Next Data to DAC channel //
// Function : Computes Exponential Moving Average //
void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN);
uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT); // Function : Saturate DAC Output //
void TEST_MY_CODE (void); // Function : Test the Correct Execution of Code //
void MY_CONTROL_STOP(double* TIMVAL); // Function : Stops TIM-ADC-DAC Operations //
/* USER CODE END PFP */
```

The first block of code inside the `int main(void)` function is used to initialize the PID structure by setting the constant PID gains as well as starting all the required peripherals of the STM32 board.

```
/* USER CODE BEGIN 2 */
///// CHECK INPUT AND INITIALIZE CONTROLLER /////
MY_BLINK( );
MY_INPUT_CHECK( ); // Check Input Variables
MY_BLINK( );
///// START THE TIM-ADC-DAC MODULES /////
MY_CONTROL_START( ); // Starts the Control Operation
/* USER CODE END 2 */
```

The second block of code inside the `int main(void)` function is written inside the infinite `while (1)` loop. It is used to test the execution of code during the control operation and stopping all the STM32 peripherals as soon as the time of control operation finishes.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    ///// FINISH THE TIM-ADC-DAC MODULES /////
    TEST_MY_CODE( ); // Test the Execution of Code
    MY_CONTROL_STOP(&t); // Stop the Control Operation
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

So far, the presented blocks of the code were very similar to the previous code. However, the main differences arise in the definitions of some user defined functions, which their descriptions are presented after the `int main(void)` function. In the following paragraphs, we enumerate the changes associated with the activation of DAC module in continuous mode (with DMA feature).

Firstly, `void MY_CONTROL_START(void)` is defined to start the TIM / ADC / DAC modules of the STM32 board with the predefined settings. The TIM2 was started in interrupt mode as it was specifically used to control the frequency of real-time operation by repetitively calling the `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function. In this function, however, we also set the `htim6.Init.Prescaler` to **4-1** and set the `htim6.Init.Period` to **108-1**. Since TIM6 is located on the APB1 bus, its maximum frequency is equal to **108 MHz** (please refer to **Table 2**). Therefore, in this code the frequency of TIM6 for triggering the DAC module was set to **250 kHz** according to **Eq. 8**, which is fivefold higher than the frequency of control operation. Please note that the TIM6 was started in normal (non-interrupted) mode. Faster DAC speed or frequency means that the output or command signal is transferred to the DAC peripheral at a faster rate than the

speed of real-time control operation. Finally, we should use the appropriate command `HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, DACValue, 1, DAC_ALIGN_12B_R)` in order to start the DAC module in continuous mode.

Secondly, inside the `void MY_DAC_UPDATE(void)`, the instantaneous value of the output signal is directly assigned to the `DACValue[0]` without calling the `HAL_DAC_SetValue` function like in the previous code, as the DAC module continuously transfers the current value of `DACValue[0]` to the predefined DAC channel with the remarkable speed of 250 kHz. Finally, the TIM6 and DAC modules should be stopped inside the `void MY_CONTROL_STOP(double* TIMVAL)` after the complete execution of control operation, which results in minor modifications of code syntax in comparison to the previous code.

```
/* USER CODE BEGIN 4 */
//// Function : Blinking Green LED //////////////////////////////////////
void MY_BLINK(void)
{
    for (int ii = 0 ; ii < 5 ; ii++)
    {
        HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_0);
        HAL_Delay(200);
    }
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);
}
//// Function : Checks Variables and Initialize Controller //////////
void MY_INPUT_CHECK(void)
{
    // Check Timer Parameters //
    chvar = 0; // Check Variable
    (MyClkFreq != 108000000) ? (MyClkFreq = 108000000 , chvar = 1) : 0;
    (MyPrescaler < 20-1) ? (MyPrescaler = 20-1 , chvar = 1) : 0;
    (MyPeriod != 108-1) ? (MyPeriod = 108-1 , chvar = 1) : 0;
    TRESET = MyClkFreq / (MyPrescaler + 1) / (MyPeriod + 1); // Timer Reset Index
    if (chvar == 1)
```

```

{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_SET);
    HAL_Delay(1000);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_RESET);
}

// Initialize Controller Parameters //
KP = PID_GAIN_KP;
KI = PID_GAIN_KI;
KD = PID_GAIN_KD;
MyPID.Kp = KP; // Set the Proportional Gain
MyPID.Ki = KI; // Set the Integral Gain
MyPID.Kd = KD; // Set the Derivative Gain
arm_pid_init_f32(&MyPID, 1); // Initialize the Controller
}

//// Function : Starts TIM-ADC-DAC Operations //////////////////////////////////
void MY_CONTROL_START(void)
{
    htim2.Init.Prescaler = MyPrescaler;
    htim2.Init.Period = MyPeriod;
    HAL_TIM_Base_Init(&htim2);
    HAL_TIM_Base_Start_IT(&htim2); // Start TIM2 Module in Interrupt Mode
    htim6.Init.Prescaler = 4-1; // Maximize DAC DMA Speed [ 3 < ... < MyPrescaler ]
    htim6.Init.Period = MyPeriod;
    HAL_TIM_Base_Init(&htim6);
    HAL_TIM_Base_Start(&htim6); // Start TIM6 Module for DAC
    //HAL_ADC_Start(&hadc3) ;// Start the ADC Module
    // Start the DAC Module in DMA Mode
    HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, DACValue, 1, DAC_ALIGN_12B_R);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 , GPIO_PIN_SET);
}

//// Function : Update Timer Counter //////////////////////////////////
void MY_TIM_UPDATE(uint32_t* TCOUNT)
{
    *TCOUNT += 1; // Update Timer Counter
    // Reset Timer Counter and Update Time Variable
    (*TCOUNT == TRESET) ? (*TCOUNT = 0 , t += 1) : 1;
}

```



```

///// Function : Reads the Current Data from ADC Channel //////////////////////////////////
void MY_ADC_UPDATE(void)
{
    HAL_ADC_Start(&hadc3); // Start the ADC3 Module
    ADCValue[0] = HAL_ADC_GetValue(&hadc3); // Get Current ADC Value
    BUFcount += 1; // Update Buffer Counter
    ADCBuffer[BUFcount-1] = (float32_t)ADCValue[0]; // Fill ADC Buffer Array
    (BUFcount == buffer_size) ? (BUFcount = 0) : 1; // Reset Buffer Counter
    data_ready = 1; // Update ADC Status Index
}

///// Function : Passes the Data through PID Controller //////////////////////////////////
void MY_DUMMY_CONTROL(void)
{
    // Compute Mean Value of ADC Buffer Array
    arm_mean_f32(ADCBuffer, (uint32_t)buffer_size, &feedbackValue);
    // Compute Mean Value of Input Signal (referenceValue)
    MY_EXP_MA(0.0002, feedbackValue, &signalMean);
    referenceValue = 2 * signalMean; // Reference Value
    errorValue = referenceValue - feedbackValue; // Compute Error Value
    PID_in = errorValue; // Set PID Input
    PID_out = arm_pid_f32(&MyPID, PID_in); // Compute PID Output
}

///// Function : Writes the Next Data to DAC channel //////////////////////////////////
void MY_DAC_UPDATE(void)
{
    commandValue = PID_out; // Set Command Value
    DACValue[0] = MY_LIMIT_OUTPUT((uint32_t)commandValue); // Write the Next
Value to DAC Channel
}

///// Function : Computes Exponential Moving Average //////////////////////////////////
void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN)
{
    *SIGMEAN = gamma * SIGVAL + (1 - gamma) * (*SIGMEAN);
}

///// Function : Saturate DAC Output //////////////////////////////////
uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT)

```

```

{
    return (INPUT > 4095) ? (4095) : (INPUT);
}

//// Function : Test the Correct Execution of Code //////////////////////////////////
void TEST_MY_CODE(void)
{
    if (DACValue[0] == 4095) // ADCValue[0] == 4095 (Connected to 3.3 V - VCC)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
    }
    else if (DACValue[0] == 0) // ADCValue[0] == 0 (Connected to 0 V - GND)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET);
    }
    else // 0 < ADCValue[0] < 4095 (Not Connected to either GND or VCC)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_14, GPIO_PIN_RESET);
    }
}

//// Function : Stops TIM-ADC-DAC Operations //////////////////////////////////
void MY_CONTROL_STOP(double* TIMVAL)
{
    if (*TIMVAL == tstop)
    {
        HAL_TIM_Base_Stop_IT(&htim2); // Stop the TIM2 Module in Interrupt Mode
        HAL_TIM_Base_Stop(&htim6); // Stop the TIM6 Module
        HAL_ADC_Stop(&hadc3); // Stop the ADC3 Module
        HAL_DAC_Stop_DMA(&hdac, DAC_CHANNEL_1); // Stop the DAC Module in DMA Mode
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_7 | GPIO_PIN_14, GPIO_PIN_RESET);
    }
}

//// Function : HAL_TIM_PeriodElapsedCallback //////////////////////////////////
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{

```

```

    UNUSED(htim);
    MY_TIM_UPDATE(&TIMcount); // Update Timer Counter
    MY_ADC_UPDATE( ); // Update ADC Operation - Read Current Data
    MY_DUMMY_CONTROL( ); // Update Control Operation - Process Data
    MY_DAC_UPDATE( ); // Update DAC - Write Next Data
}
////////////////////////////////////
/* USER CODE END 4 */

```

The other user defined functions are quite similar to the ones defined in the previous code. Please remember that the `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function is at the heart of the control algorithm which calls the user defined functions corresponding to the update of time variable, `MY_TIM_UPDATE(&TIMcount)`, reading the input data from the ADC module in single mode, `MY_ADC_UPDATE()`, performing the real-time phase inversion of the input signal or the so-called dummy control computations, `MY_DUMMY_CONTROL()`, as well as sending the output data to the DAC module in continuous mode, `MY_DAC_UPDATE()`. Please keep in mind that all these actions should be implemented faster than the fraction of time corresponding to the frequency of TIM2 module so that we can expect that the algorithm is executed in real-time.

7.5 Continuous conversion ADC / single conversion DAC

The third version of dummy control code is developed to investigate the possibility of communicating with the ADC module in continuous mode, while the DAC module is called in single conversion mode. The aim is to find the maximum speed with which we are able to interact with the ADC module while performing a single-channel real-time control operation. The code begins with the definition of global variables required for adjusting the parameters of TIM / ADC / DAC modules plus the introduction of variables required for the definition of the dummy PID controller. One specific modification is made to the input variables. A variable named as `sConfig`, which is originally defined as a local variable inside the ADC initialization function `static void MX_ADC3_Init(void)`, is

introduced here as a global variable. This is due to the fact that one of the user defined functions will later call this variable during the execution of code.

In order to activate the ADC module in continuous mode, we have to adjust some extra settings including the activation of DMA for the DAC module as described in **Section 6.4**. As a result of those settings, an initialization function named `static void MX_DMA_Init(void)` is added to the other functions, which enables the DMA feature for the ADC module that facilitate faster transfer of data from peripheral to memory. In addition, the parameters configured for initialization of ADC module inside the `static void MX_ADC3_Init(void)` function will be modified in comparison to the previous codes.

```
/* USER CODE BEGIN PV */
// Comment the following line in the Function: static void MX_ADC3_Init(void)
ADC_ChannelConfTypeDef sConfig = {0}; // The sConfig should be Defined Globally
// TIM PARAMETERS
uint8_t chvar; // Check Variable
uint32_t MyClkFreq = 108000000; // Fixed APB1 Clock Frequency [Hz]
uint32_t MyPrescaler = 20-1; // Minimum Prescaler Value [cycles]
uint32_t MyPeriod = 108-1; // Fixed Period Value [cycles]
uint32_t TIMcount = 0; // Timer Counter
uint32_t TRESET = 0; // Timer Reset Index
double t = 0; // Time Variable [s]
double tstop = 20; // Finish Time [s]
// ADC PARAMETERS
#define buffer_size 1 // Buffer Size for ADC Buffer Array
uint8_t data_ready = 0; // ADC Status Index [0 = not ready , 1 = ready]
uint32_t ADCValue[1] = {0}; // Current ADC Value
uint32_t BUFcount = 0; // Buffer Counter
float32_t ADCBuffer[buffer_size] = {0}; // ADC Buffer Array
float32_t feedbackValue = 0; // Current Feedback Value
// CONTROL PARAMETERS
#define PID_GAIN_KP 1 // Proportional Gain Value
#define PID_GAIN_KI 0 // Integral Gain Value
#define PID_GAIN_KD 0 // Derivative Gain Value
float KP; // Proportional Gain Variable
```

```

float KI; // Integral Gain Variable
float KD; // Derivative Gain Variable
arm_pid_instance_f32 MyPID; // PID Structure
float32_t PID_in , PID_out; // PID Input and Output Values
float32_t referenceValue; // Reference Value
float32_t errorValue = 0; // Current Error Value
float32_t signalMean = 2048; // Initial Mean Value [Here = 1.65 volt]
// DAC PARAMETERS
uint32_t DACValue[1] = {0}; // Current DAC Value
float32_t commandValue = 0; // Current Command Value
/* USER CODE END PV */

```

Similar to the presented algorithms, the dummy control operation takes place inside the callback function of TIM2 module. MyPeriod was set to **108-1** and MyPrescaler was set to **20-1** so that the maximum frequency of control operation is equal to **50 kHz** according to **Eq. 8**. However, one may activate a timer module from APB2 bus, such as TIM1 or TIM8, in order to trigger the ADC module by activating the TRGO feature of the timer. We leave this as an exercise for the enthusiast readers. Here, we have used the default APB2 peripheral (ADC) clock to trigger the ADC module in continuous mode at a faster rate than the frequency of control operation. The next step is to define the user function prototypes, as shown below. These user functions are declared right after the declaration of initialization functions for the STM32 modules.

```

/* USER CODE BEGIN PFP */
void MY_BLINK(void); // Function : Blinking Green LED //
void MY_INPUT_CHECK(void); // Function : Checks Variables and Initialize Controller //
void MY_CONTROL_START(void); // Function : Starts TIM-ADC-DAC Operations //
void MY_TIM_UPDATE(uint32_t* TCOUNT); // Function : Update Timer Counter //
void MY_ADC_UPDATE(void); // Function : Reads the Current Data from ADC Channel //
void MY_DUMMY_CONTROL(void); // Function : Passes the Data through PID Controller //
void MY_DAC_UPDATE(void); // Function : Writes the Next Data to DAC channel //
// Function : Computes Exponential Moving Average //
void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN);
uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT); // Function : Saturate DAC Output //

```

```

void TEST_MY_CODE (void); // Function : Test the Correct Execution of Code //
void MY_CONTROL_STOP(double* TIMVAL); // Function : Stops TIM-ADC-DAC Operations //
/* USER CODE END PFP */

```

The first block of code inside the `int main(void)` function is used to initialize the PID structure by setting the constant PID gains as well as starting all the required peripherals of the STM32 board.

```

/* USER CODE BEGIN 2 */
///// CHECK INPUT AND INITIALIZE CONTROLLER /////
MY_BLINK( );
MY_INPUT_CHECK( ); // Check Input Variables
MY_BLINK( );
///// START THE TIM-ADC-DAC MODULES /////
MY_CONTROL_START( ); // Starts the Control Operation
/* USER CODE END 2 */

```

The second block of code inside the `int main(void)` function is written inside the infinite `while (1)` loop. It is used to test the execution of code during the control operation and stopping all the STM32 peripherals as soon as the time of control operation finishes.

```

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    ///// FINISH THE TIM-ADC-DAC MODULES /////
    TEST_MY_CODE( ); // Test the Execution of Code
    MY_CONTROL_STOP(&t); // Stop the Control Operation
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */

```

Before expressing the user defined functions, we must make a slight change to the initialization function for the ADC module. Inside the `static void MX_ADC3_Init(void)`

function, we should disable the local definition of variable `sConfig`, since it was deliberately expressed as a global variable at the beginning of the code.

```
static void MX_ADC3_Init(void)
{
    /* USER CODE BEGIN ADC3_Init 0 */
    /* USER CODE END ADC3_Init 0 */
    // ADC_ChannelConfTypeDef sConfig = {0};
    /* USER CODE BEGIN ADC3_Init 1 */
    /* USER CODE END ADC3_Init 1 */
    /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number
    of conversion) */
    hadc3.Instance = ADC3;
    hadc3.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    hadc3.Init.Resolution = ADC_RESOLUTION_12B;
    hadc3.Init.ScanConvMode = ADC_SCAN_ENABLE;
    hadc3.Init.ContinuousConvMode = ENABLE;
    hadc3.Init.DiscontinuousConvMode = DISABLE;
    hadc3.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc3.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadc3.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc3.Init.NbrOfConversion = 1;
    hadc3.Init.DMAContinuousRequests = ENABLE;
    hadc3.Init.EOCSelection = ADC_EOC_SEQ_CONV;
    if (HAL_ADC_Init(&hadc3) != HAL_OK)
    { Error_Handler(); }
    /** Configure for the selected ADC regular channel its corresponding rank in the sequencer
    and its sample time. */
    sConfig.Channel = ADC_CHANNEL_3;
    sConfig.Rank = ADC_REGULAR_RANK_1;
    sConfig.SamplingTime = ADC_SAMPLETIME_15CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK)
    { Error_Handler(); }
    /* USER CODE BEGIN ADC3_Init 2 */
    /* USER CODE END ADC3_Init 2 */
}
```

So far, the presented blocks of the code were very similar to the previous codes. However, the main differences arise in the definitions of some user defined functions, which their descriptions are presented after the `int main(void)` function. In the following paragraphs, we enumerate the changes associated with the activation of ADC module in continuous mode (with DMA feature).

Firstly, `void MY_CONTROL_START(void)` is defined to start the TIM / ADC / DAC modules of the STM32 board with the predefined settings. The TIM2 was started in interrupt mode as it was specifically used to control the frequency of real-time operation by repetitively calling the `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function. In this function, however, we also set `sConfig.SamplingTime` to `ADC_SAMPLETIME_3CYCLES` in order to maximize the speed of ADC module. It should be noted that the maximum frequency of the APB2 peripheral clock is **108 MHz**. The operating frequency of ADC module is computed according to the following formula:

$$F_{ADC} = \frac{ADC_{ClkFreq}}{(t_{SAMPLING} + t_{CONVERSION}) * ADC_{Prescaler}} \quad (9)$$

$$t_{CONVERSION} = 15 \text{ cycles}, \quad t_{SAMPLING} = 3 \text{ cycles}$$

$$ADC_{ClkFreq} = 108 \text{ MHz}, \quad ADC_{Prescaler} = 4$$

please note that the ADC module has a default clock prescaler that its value is dependent on the settings done in the Clock Configuration tab (please refer to **Section 6.2**). The value of the ADC clock prescaler is defined under the `hadc3.Init.ClockPrescaler` inside the `static void MX_ADC3_Init(void)` function. Additionally, $t_{CONVERSION}$ is the time required for performing a 12-bits ADC, which by default is equal to **15 cycles**. Therefore, in this code the operating frequency of ADC module was set to **1.5 MHz** according to **Eq. 9**, which is thirtyfold higher than the frequency of control operation. Faster ADC speed or frequency means that the input or feedback signal is read from the ADC peripheral at a faster rate than the speed of real-time control operation. Since the ADC peripherals are located on APB2 bus, while the DAC and TIM2 modules are located on APB1 bus, we are able to communicate with the ADC module at a much faster rate in comparison to the DAC

module. The hardware resources of the STM32 board are limited. Therefore, one should closely pay attention to the hardware architecture of the board before selecting the most appropriate modules for a specific control or measurement operation. Because it may yield a direct effect on the maximum allowable speed of that operation. Finally, we should utilize the appropriate command `HAL_ADC_Start_DMA(&hadc3, (uint32_t*)ADCValue, 1)` in order to start the ADC module in continuous mode.

Secondly, inside the `void MY_ADC_UPDATE(void)`, the instantaneous value of the input signal is directly stored in the `ADCValue[0]` by using the `HAL_ADC_GetValue(&hadc3)` function without the need to call the `HAL_ADC_Start(&hadc3)` function like in the previous codes, as the ADC module continuously transfers the input data to the `ADCValue[0]` from the predefined ADC channel with the staggering speed of 1.5 MHz. Finally, the ADC module should be stopped inside the `void MY_CONTROL_STOP(double* TIMVAL)` after the complete execution of control operation, which results in minor modifications of code syntax in comparison to the previous codes.

```
/* USER CODE BEGIN 4 */
//// Function : Blinking Green LED //////////////////////////////////////
void MY_BLINK(void)
{
    for (int ii = 0 ; ii < 5 ; ii++)
    {
        HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_0);
        HAL_Delay(200);
    }
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);
}
//// Function : Checks Variables and Initialize Controller //////////
void MY_INPUT_CHECK(void)
{
    // Check Timer Parameters //
    chvar = 0; // Check Variable
    (MyClkFreq != 108000000) ? (MyClkFreq = 108000000 , chvar = 1) : 0;
}
```

```

(MyPrescaler < 20-1) ? (MyPrescaler = 20-1 , chvar = 1) : 0;
(MyPeriod != 108-1) ? (MyPeriod = 108-1 , chvar = 1) : 0;
TRESET = MyClkFreq / (MyPrescaler + 1) / (MyPeriod + 1); // Timer Reset Index
if (chvar == 1)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_SET);
    HAL_Delay(1000);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_RESET);
}
// Initialize Controller Parameters //
KP = PID_GAIN_KP;
KI = PID_GAIN_KI;
KD = PID_GAIN_KD;
MyPID.Kp = KP; // Set the Proportional Gain
MyPID.Ki = KI; // Set the Integral Gain
MyPID.Kd = KD; // Set the Derivative Gain
arm_pid_init_f32(&MyPID, 1); // Initialize the Controller
}

//// Function : Starts TIM-ADC-DAC Operations //////////////////////////////////
void MY_CONTROL_START(void)
{
    htim2.Init.Prescaler = MyPrescaler;
    htim2.Init.Period = MyPeriod;
    HAL_TIM_Base_Init(&htim2);
    HAL_TIM_Base_Start_IT(&htim2); // Start TIM2 Module in Interrupt Mode
    // Maximize ADC DMA Speed [ 3CYCLES < .... < 480CYCLES ]
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    HAL_ADC_ConfigChannel(&hadc3, &sConfig);
    // Start the ADC Module in DMA Mode
    HAL_ADC_Start_DMA(&hadc3, (uint32_t*)ADCValue, 1) ;
    HAL_DAC_Start(&hdac, DAC_CHANNEL_1); // Start the DAC Module
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 , GPIO_PIN_SET);
}

//// Function : Update Timer Counter //////////////////////////////////
void MY_TIM_UPDATE(uint32_t* TCOUNT)
{
    *TCOUNT += 1; // Update Timer Counter
}

```

```

        (*TCOUNT == TRESET) ? (*TCOUNT = 0 , t += 1) : 1; // Reset Timer Counter and
Update Time Variable
    }
    //// Function : Reads the Current Data from ADC Channel //////////
    void MY_ADC_UPDATE(void)
    {
        ADCValue[0] = HAL_ADC_GetValue(&hadc3); // Get Current ADC Value
        BUFcount += 1; // Update Buffer Counter
        ADCBuffer[BUFcount-1] = (float32_t)ADCValue[0]; // Fill ADC Buffer Array
        (BUFcount == buffer_size) ? (BUFcount = 0) : 1; // Reset Buffer Counter
        data_ready = 1; // Update ADC Status Index
    }
    //// Function : Passes the Data through PID Controller //////////
    void MY_DUMMY_CONTROL(void)
    {
        // Compute Mean Value of ADC Buffer Array
        arm_mean_f32(ADCBuffer, (uint32_t)buffer_size, &feedbackValue);
        // Compute Mean Value of Input Signal (referenceValue)
        MY_EXP_MA(0.0002, feedbackValue, &signalMean);
        referenceValue = 2 * signalMean; // Reference Value
        errorValue = referenceValue - feedbackValue; // Compute Error Value
        PID_in = errorValue; // Set PID Input
        PID_out = arm_pid_f32(&MyPID, PID_in); // Compute PID Output
    }
    //// Function : Writes the Next Data to DAC channel //////////
    void MY_DAC_UPDATE(void)
    {
        commandValue = PID_out; // Set Command Value
        // Write the Next Value to DAC Channel
        DACValue[0] = MY_LIMIT_OUTPUT((uint32_t)commandValue);
        // Write the Next Value to DAC Channel
        HAL_DAC_SetValue(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, DACValue[0]);
    }
    //// Function : Computes Exponential Moving Average //////////
    void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN)
    {

```

```

        *SIGMEAN = gamma * SIGVAL + (1 - gamma) * (*SIGMEAN);
    }
    //// Function : Saturate DAC Output ///////////////////////////////////
    uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT)
    {
        return (INPUT > 4095) ? (4095) : (INPUT);
    }
    //// Function : Test the Correct Execution of Code ///////////////////
    void TEST_MY_CODE(void)
    {
        if (DACValue[0] == 4095) // ADCValue[0] == 4095 (Connected to 3.3 V - VCC)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
        }
        else if (DACValue[0] == 0) // ADCValue[0] == 0 (Connected to 0 V - GND)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET);
        }
        else // 0 < ADCValue[0] < 4095 (Not Connected to either GND or VCC)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_14, GPIO_PIN_RESET);
        }
    }
    //// Function : Stops TIM-ADC-DAC Operations ///////////////////////////////////
    void MY_CONTROL_STOP(double* TIMVAL)
    {
        if (*TIMVAL == tstop)
        {
            HAL_TIM_Base_Stop_IT(&htim2); // Stop the TIM2 Module in Interrupt Mode
            HAL_ADC_Stop_DMA(&hadc3); // Stop the ADC3 Module in DMA Mode
            HAL_DAC_Stop(&hdac, DAC_CHANNEL_1); // Stop the DAC Module
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_7 | GPIO_PIN_14, GPIO_PIN_RESET);
        }
    }
}

```

```

////// Function : HAL_TIM_PeriodElapsedCallback ///////////////////////////////////
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    UNUSED(htim);
    MY_TIM_UPDATE(&TIMcount); // Update Timer Counter
    MY_ADC_UPDATE( ); // Update ADC Operation - Read Current Data
    MY_DUMMY_CONTROL( ); // Update Control Operation - Process Data
    MY_DAC_UPDATE( ); // Update DAC - Write Next Data
}

//////////////////////////////////////
/* USER CODE END 4 */

```

The other user defined functions are quite similar to the ones defined in the previous codes. Please remember that the `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function is at the heart of the control algorithm which calls the user defined functions corresponding to the update of time variable, `MY_TIM_UPDATE(&TIMcount)`, reading the input data from the ADC module in continuous mode, `MY_ADC_UPDATE()`, performing the real-time phase inversion of the input signal or the so-called dummy control computations, `MY_DUMMY_CONTROL()`, as well as sending the output data to the DAC module in single mode, `MY_DAC_UPDATE()`. Please keep in mind that all these actions should be implemented faster than the fraction of time corresponding to the frequency of TIM2 module so that we can expect that the algorithm is executed in real-time.

7.6 Continuous conversion ADC / continuous conversion DAC

The last version of dummy control code is developed to investigate the possibility of communicating with both of the ADC / DAC modules in continuous mode, while a single-channel control operation is executed in real-time inside the callback function of TIM2. The objective is to find the maximum speed with which we are able to interact with the TIM / ADC / DAC modules while performing a single-channel real-time control operation. All the pervious modifications in **Code 2** and **Code 3** (please refer to **Section 7.4** and **Section 7.5**) are included in this code. Therefore, you should already have a bright understanding of what is going to be described in this section.

The code begins with the definition of global variables required for adjusting the parameters of TIM / ADC / DAC modules plus the introduction of variables required for the definition of the dummy PID controller. One specific modification is made to the input variables. A variable named as sConfig, which is originally defined as a local variable inside the ADC initialization function `static void MX_ADC3_Init(void)`, is introduced here as a global variable.

```
/* USER CODE BEGIN PV */
// Comment the following line in the Function: static void MX_ADC3_Init(void)
ADC_ChannelConfTypeDef sConfig = {0}; // The sConfig should be Defined Globally
// TIM PARAMETERS
uint8_t chvar; // Check Variable
uint32_t MyClkFreq = 108000000; // Fixed APB1 Clock Frequency [Hz]
uint32_t MyPrescaler = 25-1; // Minimum Prescaler Value [cycles]
uint32_t MyPeriod = 108-1; // Fixed Period Value [cycles]
uint32_t TIMcount = 0; // Timer Counter
uint32_t TRESET = 0; // Timer Reset Index
double t = 0; // Time Variable [s]
double tstop = 20; // Finish Time [s]
// ADC PARAMETERS
#define buffer_size 1 // Buffer Size for ADC Buffer Array
uint8_t data_ready = 0; // ADC Status Index [0 = not ready , 1 = ready]
uint32_t ADCValue[1] = {0}; // Current ADC Value
uint32_t BUFcount = 0; // Buffer Counter
float32_t ADCBuffer[buffer_size] = {0}; // ADC Buffer Array
float32_t feedbackValue = 0; // Current Feedback Value
// CONTROL PARAMETERS
#define PID_GAIN_KP 1 // Proportional Gain Value
#define PID_GAIN_KI 0 // Integral Gain Value
#define PID_GAIN_KD 0 // Derivative Gain Value
float KP; // Proportional Gain Variable
float KI; // Integral Gain Variable
float KD; // Derivative Gain Variable
arm_pid_instance_f32 MyPID; // PID Structure
float32_t PID_in , PID_out; // PID Input and Output Values
```

```

float32_t referenceValue; // Reference Value
float32_t errorValue = 0; // Current Error Value
float32_t signalMean = 2048; // Initial Mean Value [Here = 1.65 volt]
// DAC PARAMETERS
uint32_t DACValue[1] = {0}; // Current DAC Value
float32_t commandValue = 0; // Current Command Value
/* USER CODE END PV */

```

In order to activate the ADC / DAC modules in continuous mode, we have to adjust some extra settings including the activation of DMA for the ADC / DAC modules as described in **Section 6.4** and **Section 6.5**. In addition, we should also activate TIM6 to trigger the DAC in continuous mode as explained in **Section 6.3**. As a result of all these settings, an initialization function named `static void MX_DMA_Init(void)` is added to the other functions, which enables the DMA feature for the ADC / DAC modules that facilitate faster transfer of data from peripheral to memory and vice versa. In addition, the parameters configured for initialization of ADC / DAC modules inside the `static void MX_ADC3_Init(void)` and `static void MX_DAC_Init(void)` functions will be modified.

Similar to the presented algorithms, the dummy control operation takes place inside the callback function of TIM2 module. MyPeriod was set to **108-1** and MyPrescaler was set to **25-1** so that the maximum frequency of control operation is equal to **40 kHz** according to **Eq. 8**. The speed of control operation is slightly reduced as we had activated the ADC / DAC modules with DMA feature. This decision had a negative impact on the maximum frequency of TIM2 module with which we perform the single-channel control operation in real-time mode. The next step is to define the user function prototypes, as shown below. These user functions are declared right after the declaration of initialization functions for the STM32 modules.

```

/* USER CODE BEGIN PFP */
void MY_BLINK(void); // Function : Blinking Green LED //
void MY_INPUT_CHECK(void); // Function : Checks Variables and Initialize Controller //
void MY_CONTROL_START(void); // Function : Starts TIM-ADC-DAC Operations //

```

```

void MY_TIM_UPDATE(uint32_t* TCOUNT); // Function : Update Timer Counter //
void MY_ADC_UPDATE(void); // Function : Reads the Current Data from ADC Channel //
void MY_DUMMY_CONTROL(void); // Function : Passes the Data through PID Controller //
void MY_DAC_UPDATE(void); // Function : Writes the Next Data to DAC channel //
// Function : Computes Exponential Moving Average //
void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN);
uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT); // Function : Saturate DAC Output //
void TEST_MY_CODE (void); // Function : Test the Correct Execution of Code //
void MY_CONTROL_STOP(double* TIMVAL); // Function : Stops TIM-ADC-DAC Operations //
/* USER CODE END PFP */

```

The first block of code inside the `int main(void)` function is used to initialize the PID structure by setting the constant PID gains as well as starting all the required peripherals of the STM32 board.

```

/* USER CODE BEGIN 2 */
///// CHECK INPUT AND INITIALIZE CONTROLLER /////
MY_BLINK( );
MY_INPUT_CHECK( ); // Check Input Variables
MY_BLINK( );
///// START THE TIM-ADC-DAC MODULES /////
MY_CONTROL_START( ); // Starts the Control Operation
/* USER CODE END 2 */

```

The second block of code inside the `int main(void)` function is written inside the infinite `while (1)` loop. It is used to test the execution of code during the control operation and stopping all the STM32 peripherals as soon as the time of control operation finishes.

```

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    ///// FINISH THE TIM-ADC-DAC MODULES /////
    TEST_MY_CODE( ); // Test the Execution of Code
    MY_CONTROL_STOP(&t); // Stop the Control Operation
/* USER CODE END WHILE */

```



```

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

```

So far, the presented blocks of the code were very similar to the previous codes. However, the main differences arise in the definitions of some user defined functions, which their descriptions are presented after the `int main(void)` function. Before expressing the modifications made in the user defined functions, we must make a slight change to the initialization function for the ADC module. Inside the `static void MX_ADC3_Init(void)` function, we should disable the local definition of variable `sConfig`, as illustrated on **Page 42**. In the following paragraphs, we enumerate the changes associated with the activation of ADC / DAC modules in continuous mode (with DMA feature).

Firstly, `void MY_CONTROL_START(void)` is defined to start the TIM / ADC / DAC modules of the STM32 board with the predefined settings. The TIM2 was started in interrupt mode as it was specifically used to control the frequency of real-time operation by repetitively calling the `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function. In this function, however, we also set `sConfig.SamplingTime` to `ADC_SAMPLETIME_84CYCLES` in order to maximize the speed of ADC module. As a result, the operating frequency of ADC module was set slightly above **270 kHz** according to *Eq. 9*, which is almost eightfold higher than the frequency of control operation. Furthermore, we also adjust the `htim6.Init.Prescaler` to **5-1** and set the `htim6.Init.Period` to **108-1**. Therefore, the frequency of TIM6 for triggering the DAC module was set to **200 kHz** according to *Eq. 8*, which is fivefold higher than the frequency of control operation. Please note that the TIM6 was started in normal (non-interrupted) mode. Finally, we utilize the appropriate command `HAL_ADC_Start_DMA(&hadc3, (uint32_t*)ADCValue, 1)` in order to start the ADC module in continuous mode. Similarly, the DAC module is started in continuous mode by using the `HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, DACValue, 1, DAC_ALIGN_12B_R)` command.

Secondly, inside the `void MY_ADC_UPDATE(void)`, the instantaneous value of the input signal is directly stored in the `ADCValue[0]` by using the `HAL_ADC_GetValue(&hadc3)` function without the need to call the `HAL_ADC_Start(&hadc3)` function, as the ADC module continuously transfers the input data to the `ADCValue[0]` from the predefined ADC channel with the remarkable speed of 270 kHz. Thirdly, inside the `void MY_DAC_UPDATE(void)`, the instantaneous value of the output signal is directly assigned to the `DACValue[0]` without calling the `HAL_DAC_SetValue` function, as the DAC module continuously transfers the current value of `DACValue[0]` to the predefined DAC channel with the remarkable speed of 200 kHz. Finally, the TIM6, ADC and DAC modules should be stopped inside the `void MY_CONTROL_STOP(double* TIMVAL)` after the complete execution of control operation, which results in minor modifications of code syntax in comparison to the previous codes.

```
/* USER CODE BEGIN 4 */
//// Function : Blinking Green LED //////////////////////////////////////
void MY_BLINK(void)
{
    for (int ii = 0 ; ii < 5 ; ii++)
    {
        HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_0);
        HAL_Delay(200);
    }
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_0,GPIO_PIN_RESET);
}
//// Function : Checks Variables and Initialize Controller //////////
void MY_INPUT_CHECK(void)
{
    // Check Timer Parameters //
    chvar = 0; // Check Variable
    (MyClkFreq != 108000000) ? (MyClkFreq = 108000000 , chvar = 1) : 0;
    (MyPrescaler < 25-1) ? (MyPrescaler = 25-1 , chvar = 1) : 0;
    (MyPeriod != 108-1) ? (MyPeriod = 108-1 , chvar = 1) : 0;
    TRESET = MyClkFreq / (MyPrescaler + 1) / (MyPeriod + 1); // Timer Reset Index
}
```

```

    if (chvar == 1)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_SET);
        HAL_Delay(1000);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14 | GPIO_PIN_0 , GPIO_PIN_RESET);
    }
    // Initialize Controller Parameters //
    KP = PID_GAIN_KP;
    KI = PID_GAIN_KI;
    KD = PID_GAIN_KD;
    MyPID.Kp = KP; // Set the Proportional Gain
    MyPID.Ki = KI; // Set the Integral Gain
    MyPID.Kd = KD; // Set the Derivative Gain
    arm_pid_init_f32(&MyPID, 1); // Initialize the Controller
}
//// Function : Starts TIM-ADC-DAC Operations ///////////////////////////////////
void MY_CONTROL_START(void)
{
    htim2.Init.Prescaler = MyPrescaler;
    htim2.Init.Period = MyPeriod;
    HAL_TIM_Base_Init(&htim2);
    HAL_TIM_Base_Start_IT(&htim2); // Start TIM2 Module in Interrupt Mode
    htim6.Init.Prescaler = 5-1; // Maximize DAC DMA Speed [ 4 < ... < MyPrescaler ]
    htim6.Init.Period = MyPeriod;
    HAL_TIM_Base_Init(&htim6);
    HAL_TIM_Base_Start(&htim6); // Start TIM6 Module for DAC
    // Maximize ADC DMA Speed [ 84CYCLES < .... < 480CYCLES ]
    sConfig.SamplingTime = ADC_SAMPLETIME_84CYCLES;
    HAL_ADC_ConfigChannel(&hadc3, &sConfig);
    // Start the ADC Module in DMA Mode
    HAL_ADC_Start_DMA(&hadc3, (uint32_t*)ADCValue, 1);
    // Start the DAC Module in DMA Mode
    HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, DACValue, 1, DAC_ALIGN_12B_R);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 , GPIO_PIN_SET);
}
//// Function : Update Timer Counter ///////////////////////////////////
void MY_TIM_UPDATE(uint32_t* TCOUNT)

```

```

{
    *TCOUNT += 1; // Update Timer Counter
    (*TCOUNT == TRESET) ? (*TCOUNT = 0 , t += 1) : 1; // Reset Timer Counter and
Update Time Variable
}

//// Function : Reads the Current Data from ADC Channel ///////////
void MY_ADC_UPDATE(void)
{
    ADCValue[0] = HAL_ADC_GetValue(&hadc3); // Get Current ADC Value
    BUFcount += 1; // Update Buffer Counter
    ADCBuffer[BUFcount-1] = (float32_t)ADCValue[0]; // Fill ADC Buffer Array
    (BUFcount == buffer_size) ? (BUFcount = 0) : 1; // Reset Buffer Counter
    data_ready = 1; // Update ADC Status Index
}

//// Function : Passes the Data through PID Controller ///////////
void MY_DUMMY_CONTROL(void)
{
    // Compute Mean Value of ADC Buffer Array
    arm_mean_f32(ADCBuffer, (uint32_t)buffer_size, &feedbackValue);
    // Compute Mean Value of Input Signal (referenceValue)
    MY_EXP_MA(0.0002, feedbackValue, &signalMean);
    referenceValue = 2 * signalMean; // Reference Value
    errorValue = referenceValue - feedbackValue; // Compute Error Value
    PID_in = errorValue; // Set PID Input
    PID_out = arm_pid_f32(&MyPID, PID_in); // Compute PID Output
}

//// Function : Writes the Next Data to DAC channel ///////////
void MY_DAC_UPDATE(void)
{
    commandValue = PID_out; // Set Command Value
    DACValue[0] = MY_LIMIT_OUTPUT((uint32_t)commandValue); // Write the Next
Value to DAC Channel
}

//// Function : Computes Exponential Moving Average ///////////
void MY_EXP_MA(double gamma, float32_t SIGVAL, float32_t* SIGMEAN)
{

```

```

        *SIGMEAN = gamma * SIGVAL + (1 - gamma) * (*SIGMEAN);
    }

    /// Function : Saturate DAC Output //////////////////////////////////
    uint32_t MY_LIMIT_OUTPUT(uint32_t INPUT)
    {
        return (INPUT > 4095) ? (4095) : (INPUT);
    }

    /// Function : Test the Correct Execution of Code //////////////////
    void TEST_MY_CODE(void)
    {
        if (DACValue[0] == 4095) // ADCValue[0] == 4095 (Connected to 3.3 V - VCC)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);
        }
        else if (DACValue[0] == 0) // ADCValue[0] == 0 (Connected to 0 V - GND)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET);
        }
        else // 0 < ADCValue[0] < 4095 (Not Connected to either GND or VCC)
        {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_14, GPIO_PIN_RESET);
        }
    }

    /// Function : Stops TIM-ADC-DAC Operations //////////////////////////////////
    void MY_CONTROL_STOP(double* TIMVAL)
    {
        if (*TIMVAL == tstop)
        {
            HAL_TIM_Base_Stop_IT(&htim2); // Stop the TIM2 Module in Interrupt Mode
            HAL_TIM_Base_Stop(&htim6); // Stop the TIM6 Module
            HAL_ADC_Stop_DMA(&hadc3); // Stop the ADC3 Module in DMA Mode
            HAL_DAC_Stop_DMA(&hdac, DAC_CHANNEL_1); // Stop the DAC Module in DMA Mode
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_7 | GPIO_PIN_14, GPIO_PIN_RESET);
        }
    }

```

```

}
//// Function : HAL_TIM_PeriodElapsedCallback //////////////////////////////////
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    UNUSED(htim);
    MY_TIM_UPDATE(&TIMcount); // Update Timer Counter
    MY_ADC_UPDATE( ); // Update ADC Operation - Read Current Data
    MY_DUMMY_CONTROL( ); // Update Control Operation - Process Data
    MY_DAC_UPDATE( ); // Update DAC - Write Next Data
}
////////////////////////////////////
/* USER CODE END 4 */

```

The other user defined functions are quite similar to the ones defined in the previous codes. Please remember that the `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function is at the heart of the control algorithm which calls the user defined functions corresponding to the update of time variable, `MY_TIM_UPDATE(&TIMcount)`, reading the input data from the ADC module in continuous mode, `MY_ADC_UPDATE()`, performing the real-time phase inversion of the input signal or the so-called dummy control computations, `MY_DUMMY_CONTROL()`, as well as sending the output data to the DAC module in continuous mode, `MY_DAC_UPDATE()`. Please keep in mind that all these actions should be implemented faster than the fraction of time corresponding to the frequency of TIM2 module so that we can expect that the algorithm is executed in real-time.

8. Examine the real-time performance of developed codes

We presented a detailed description for the various versions of the real-time control code in **Section 7**. All these codes implement the same dummy control algorithm for phase inversion of the input signal in real-time mode as described in **Section 7.2**. However, the speeds at which we interact with the TIM / ADC / DAC can be different. We have selected these frequencies in order to showcase the maximum capacity of the NUCLEO-F746ZG board for implementation of a single-channel real-time control operation. The comparison between various versions of the code are clearly given in **Table 3**. You should consider

that the maximum frequency of control operation is directly dependent on the number of mathematical computations done inside the `void MY_DUMMY_CONTROL(void)` function. This function can be replaced by your own user defined function in order to perform a specific open-loop or closed-loop control operation. To keep the frequency of control action to the maximum allowable value, the code inside the controller function should be written in an efficient manner. This can be done by using the most appropriate functions available from the ARM CMSIS DSP library as well as using efficient computational algorithms that are implemented inside the user defined functions.

Table 3 Maximum frequency of communicating with TIM / ADC / DAC modules during the single-channel real-time control operation

Code versions	F_{TIM} [kHz]	F_{ADC} [kHz]	F_{DAC} [kHz]
Code 1 (Section 7.3)	50	50	50
Code 2 (Section 7.4)	50	50	250
Code 3 (Section 7.5)	50	1500	50
Code 4 (Section 7.6)	40	270	200

The experimental setup used for verifying the real-time performance of the developed codes is illustrated in **Fig. 5**. Two NUCLEO-F746ZG boards are required for this experimentation. The first STM32 board is programmed to generate a chirp signal with the sampling frequency that is equal to the frequency of control operation, i.e. 40 kHz or 50 kHz. The chirp generation algorithm is well-documented in [STM32 DAC TUTORIAL](#) that is available to the enthusiast readers. This chirp signal is applied as the input signal to the ADC module of the second STM32 board, which is programmed to implement the real-time phase inversion algorithm.

In order to monitor the input and output signals, the input chirp signal sent to the ADC module as well as the output signal read from the DAC module of the second STM32 board

is captured with 100 kHz sampling frequency by using a two-channel digital USB oscilloscope, HANTEK 6022B, as shown in **Fig. 5**.

The time domain plots of input / output signals for **Code 1** to **Code 4** are illustrated in **Fig. 6** to **Fig. 9**, respectively. All chirp signals had the same amplitude A [V], DC component A_0 [V], frequency content, $F_s - F_f$ [Hz] and period, T_c [s]. Our observations show that we are capable of reversing the phase of input signal in real-time mode by using the developed algorithm, including the four versions of code as described in **Section 7**. The performance of all presented codes are quite satisfactory as the comparison between the input and output signals clearly shows that phase of input signal is exactly reversed by 180 degrees (π radians). This real-time phase inversion obviously occurs irrespective of the signal's shape or frequency content. Please keep in mind that the sampling frequency of input chirp signal is kept equal to the frequency of control operation, as governed by TIM2 module.

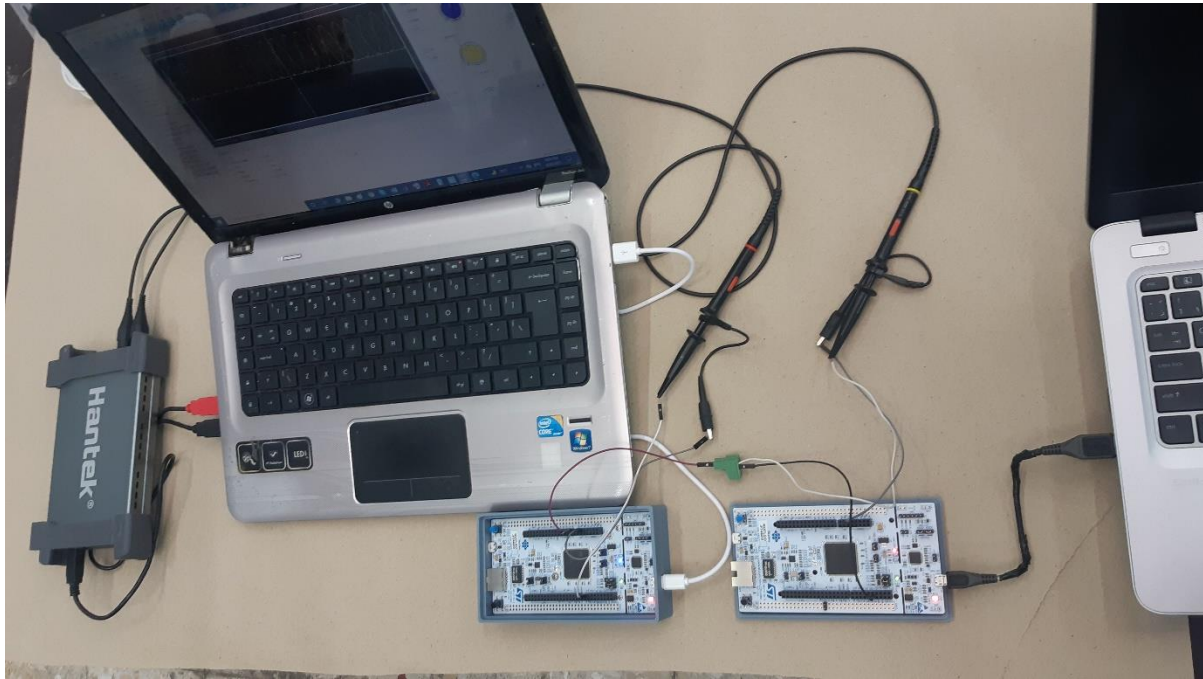


Fig. 5 Illustration of the experimental setup that includes two NUCLEO-F746ZG boards as well as a two-channel digital USB oscilloscope – one of the STM32 boards is programmed to generate the chirp signal, and the other STM32 board is programmed to implement the real-time control code (phase inversion)

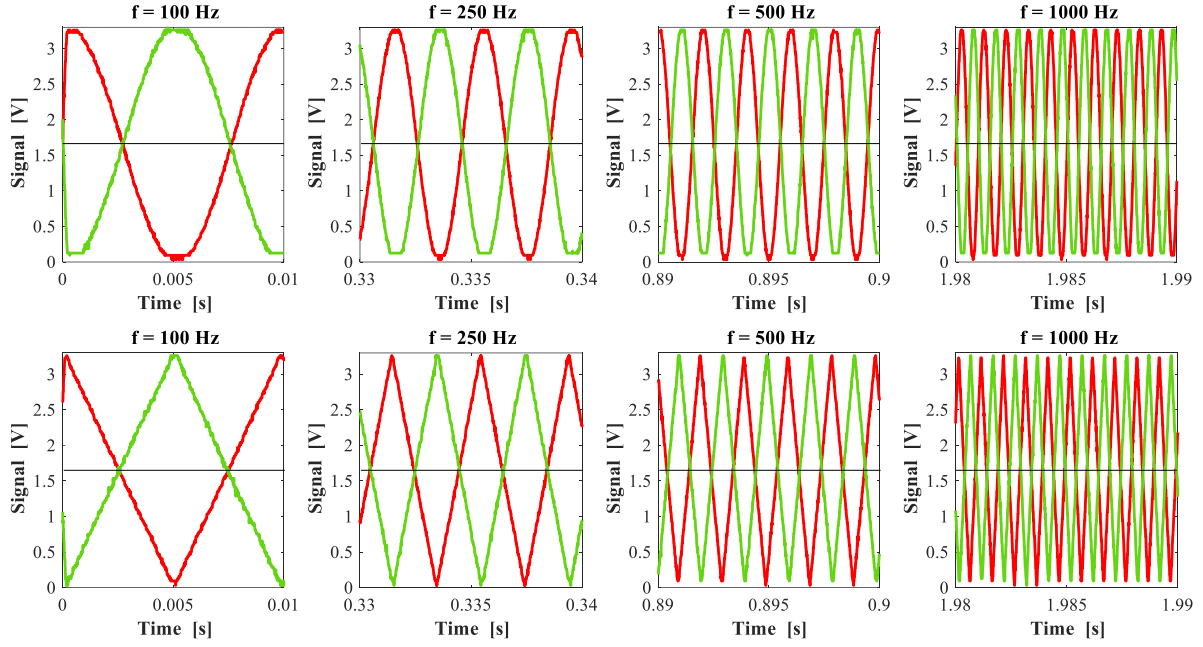


Fig. 6 Comparison of the input ADC signal (red) and the antiphase DAC signal (green) – input chirp parameters $\{A = 1.65, A_0 = 1.65, F_s = 100, F_f = 1000, T_c = 2, N_c = 50\}$ – Code 1

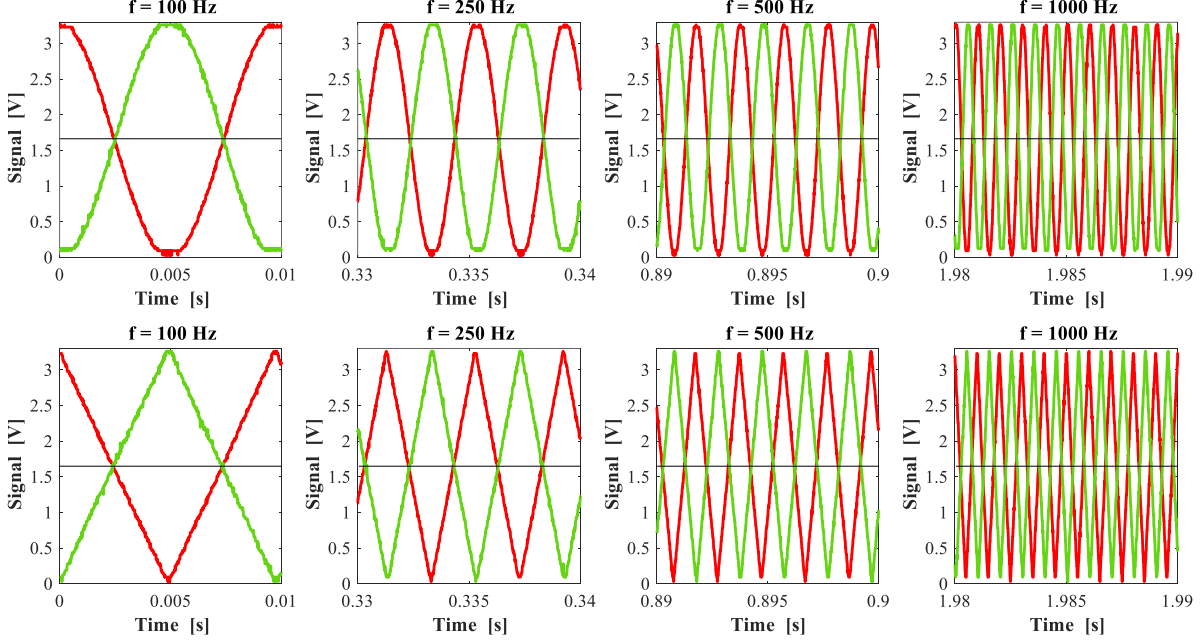


Fig. 7 Comparison of the input ADC signal (red) and the antiphase DAC signal (green) – input chirp parameters $\{A = 1.65, A_0 = 1.65, F_s = 100, F_f = 1000, T_c = 2, N_c = 50\}$ – Code 2

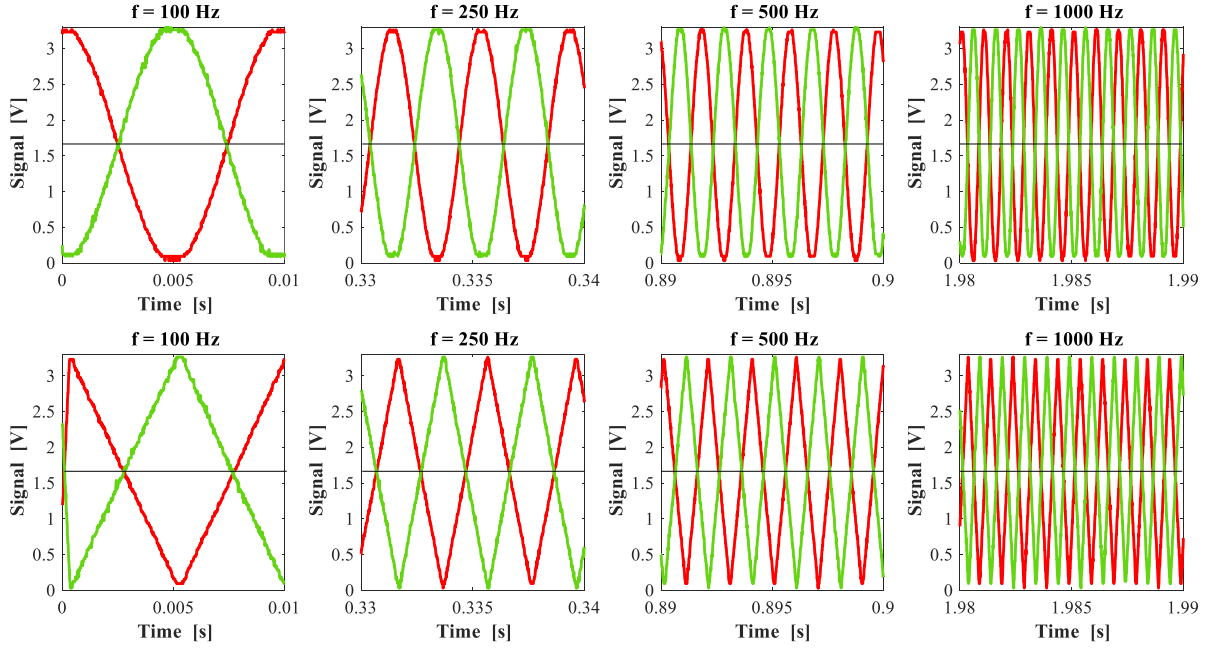


Fig. 8 Comparison of the input ADC signal (red) and the antiphase DAC signal (green) – input chirp parameters $\{A = 1.65, A_0 = 1.65, F_s = 100, F_f = 1000, T_c = 2, N_c = 50\}$ – Code 3

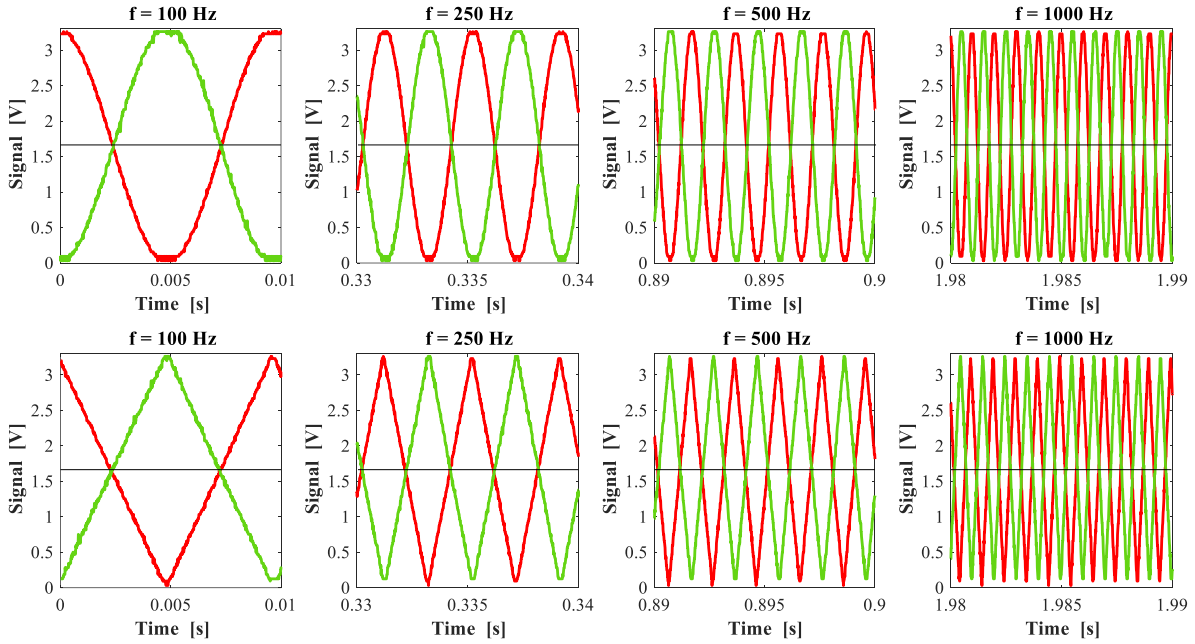


Fig. 9 Comparison of the input ADC signal (red) and the antiphase DAC signal (green) – input chirp parameters $\{A = 1.65, A_0 = 1.65, F_s = 100, F_f = 1000, T_c = 2, N_c = 40\}$ – Code 4

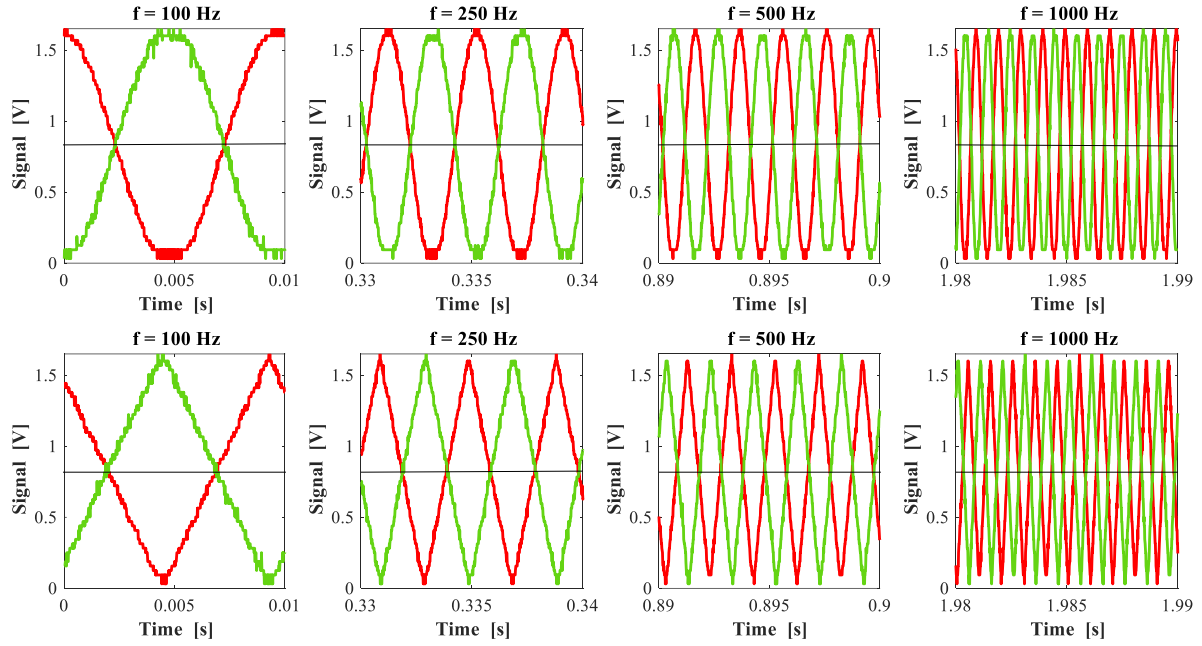


Fig. 10 Comparison of the input ADC signal (red) and the antiphase DAC signal (green) – input chirp parameters $\{A = 0.825, A_0 = 0.825, F_s = 100, F_f = 1000, T_c = 2, N_c = 50\}$ – Code 1

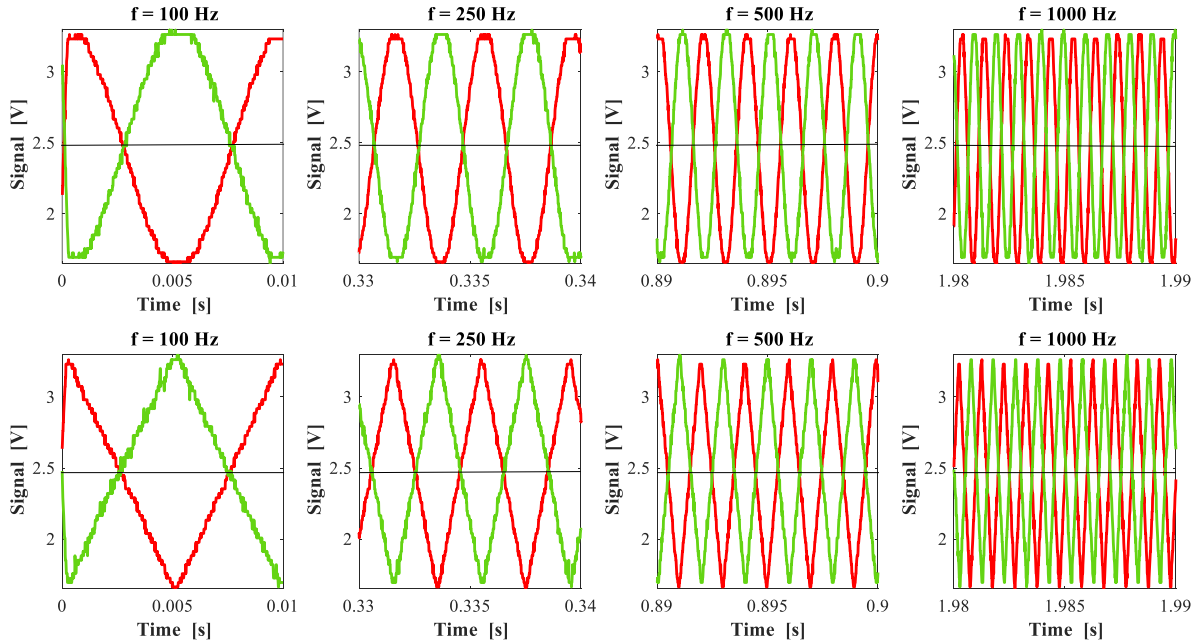


Fig. 11 Comparison of the input ADC signal (red) and the antiphase DAC signal (green) – input chirp parameters $\{A = 0.825, A_0 = 2.475, F_s = 100, F_f = 1000, T_c = 2, N_c = 50\}$ – Code 1

In order to illustrate the real-time performance of developed codes in reversing the phase of input signals with different amplitude A [V] and DC component A_0 [V] values, four experiments are done whose results are clearly illustrated in **Fig. 10** and **Fig. 11**. The phase inversion algorithm quickly tracks the DC component of the input signal and successfully reverses the signal about the DC component.

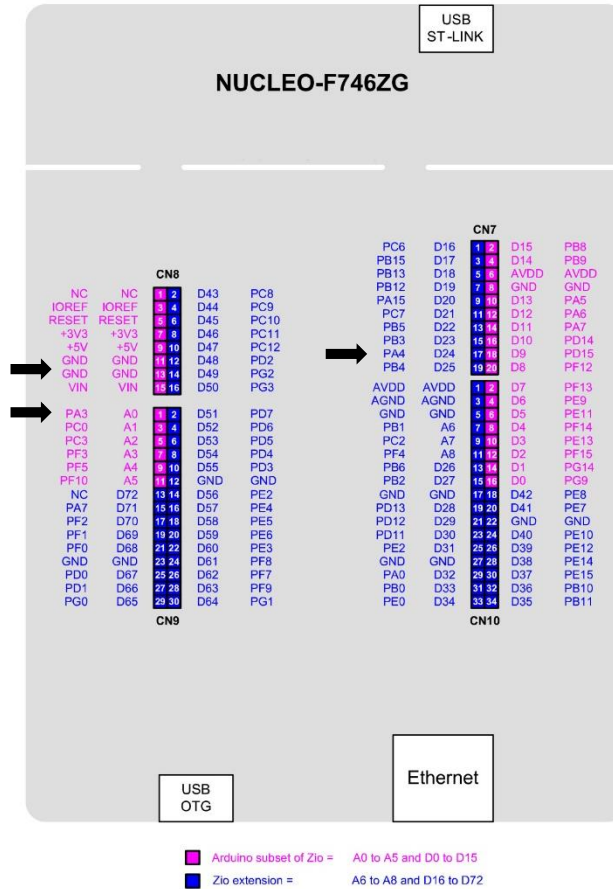


Fig. 12 Hardware layout and configuration for the STM32 NUCLEO board

In order to connect the signal generator to the second STM32 NUCLEO board, we should properly connect the wire delivering the input signal to the ADC module of the board. Please note that the STM32 board has three ADC modules each of which has 16 input channels named by IN0 to IN15. However, only some of these ADC channels are directly accessible from CN9 and CN10 connectors. The list of accessible ADC pins is given in

Table 4. In this tutorial we activated ADC3_IN3, which corresponds to the pin PA3 of the MCU that is connected to the connector pin 1 of CN9.

Table 4 List of ADC pins directly accessible from CN9 and CN 10 connectors

CN# / Pin#	STM32 Pin	Pin Function	CN# / Pin#	STM32 Pin	Pin Function
CN9 / pin1	PA3	ADC123_IN3	CN9 / pin11	PF10	ADC3_IN8
CN9 / pin3	PC0	ADC123_IN10	CN10 / pin7	PB1	ADC12_IN9
CN9 / pin5	PC3	ADC123IN_13	CN10 / pin9	PC2	ADC123IN_12
CN9 / pin7	PF3	ADC3_IN9	CN10 / pin11	PF4	ADC3_IN14
CN9 / pin9	PF5	ADC3_IN15	-	-	-

The MCU pins associated with DAC OUT1 and GND are required to illustrate the output signal received from the DAC module on the digital USB oscilloscope. The pin PA4 is associated with DAC OUT1 and can be easily accessed from connector pin 17 located on CN7. Similarly, the GND pins, PD2 and PG2, are attached to the connector pins 11 and 13 of CN8. All these pins are clearly marked on **Fig. 12**.

9. Conclusions






In this tutorial, we specifically described the application of ADC module in STM32F7 boards. We tried to explain the procedure of developing four different real-time algorithms for inversion of input signal's phase, while activating, adjusting and interacting with the TIM / ADC / DAC / GPIO modules of the STM32 NUCLEO-F746ZG board. The developed dummy control operation utilizes prepared functions from the ARM CMSIS DSP library including the conventional PID controller in order to demonstrate the application of ready-to-use functions. In addition, some user defined functions are also written in the body of the C code to implement the required numerical algorithms in an efficient manner. The ADC / DAC operations were performed in both single and continuous modes, where the entire data conversions and computations occurs inside the callback function of a TIM module. We were able to achieve a real-time performance with

a maximum sampling frequency of 40 kHz up to 50 kHz that is quite remarkable. Please note that the Simulink Desktop Real-Time (SLDRT), as an example of a commercially available solution, supports real-time performance up to 1 kHz with Simulink, and up to 20 kHz with Simulink Coder (using External Mode). However, the current solution based upon the usage of STM32 boards guarantees a maximum sampling frequency of 50 kHz when interacting with the Input / Output modules. The activation of DMA feature for the ADC / DAC modules allowed us to increase the speed of communication with the Input / Output modules beyond the sampling frequency of control operation. In this tutorial, we focused on the single-channel real-time control operation. However, the DMA feature will also allow us to read multiple data from multiple ADC channels or write multiple data to the output DAC channel.



10. References

In order to study and learn more about the other modules and features of the STM32F7 boards, you may read or watch the following benchmark references:

10.1 Official ST Documents

-  [Description of STM32F7 HAL and Low-layer drivers \(UM1905\)](#)
-  [STM32 Nucleo-144 boards \(UM1974\)](#)
-  [STM32F75xxx and STM32F74xxx advanced Arm®-based 32-bits MCUs \(RM0385\)](#)
-  [STM32F746xx Datasheet](#)
-  [How to get the best ADC accuracy in STM32 microcontrollers \(AN2834\)](#)

10.2 STM32 Books

-  [Mastering STM32](#)
-  [Programming with STM32](#)

10.3 STM32 Videos

-  [Mutex Embedded Education](#)