Development of complex embedded applications using STM32CubeIDE

Software Design for Embedded Systems

Additional course materials

Author: Hrvoje Džapo

Faculty of Electrical Engineering and Computing

University of Zagreb, 2022

Contents

1 Development tools

This guide will assume the following development environment:

- integrated development environment: STM32CubeIDE[1]
  - available for Windows and Linux OS
- hardware:
  - STM32F4DISCOVERY development kit (includes on-board ST-Link debugger)
  - external UART/USB dongle (for serial communication via VCP)
  - USB mini cable
  - additional connecting wires

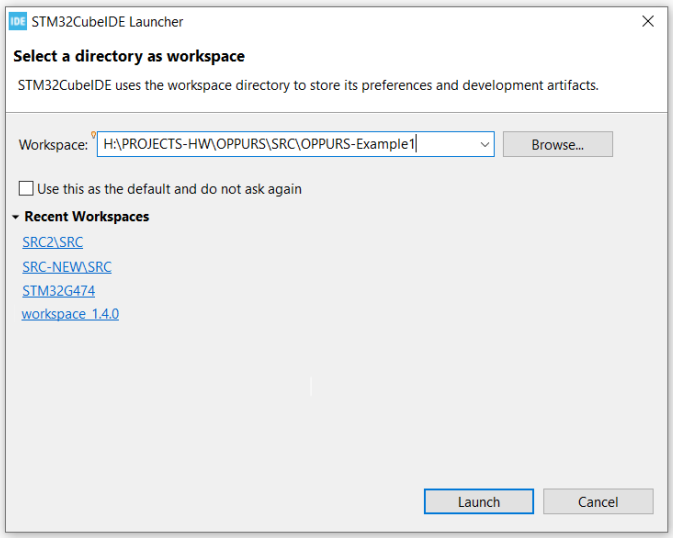
Please refer to the laboratory exercises and ST Microelectronics web pages for detailed instruction how to install STM32CubeIDE and additional device drivers (ST-Link debugger).

[1] Available from https://www.st.com/en/development-tools/stm32cubeide.html

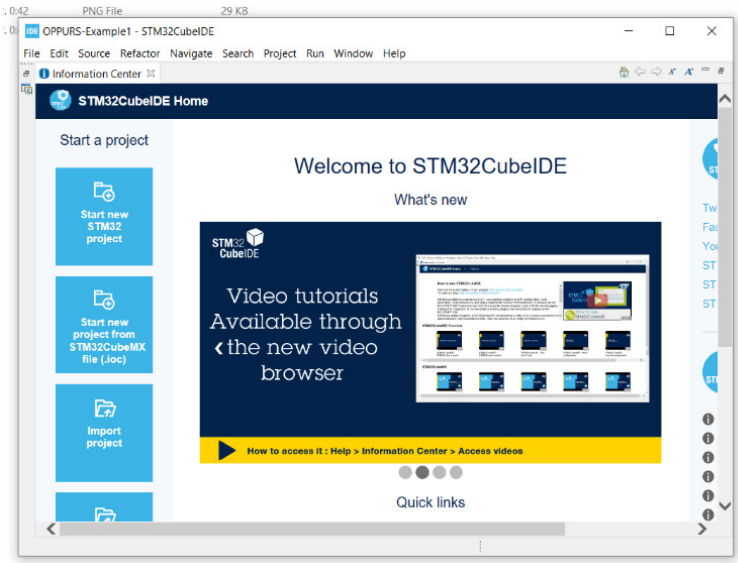
2 Creating HAL-based applications in STM32CubeIDE

2.1 Project creation process

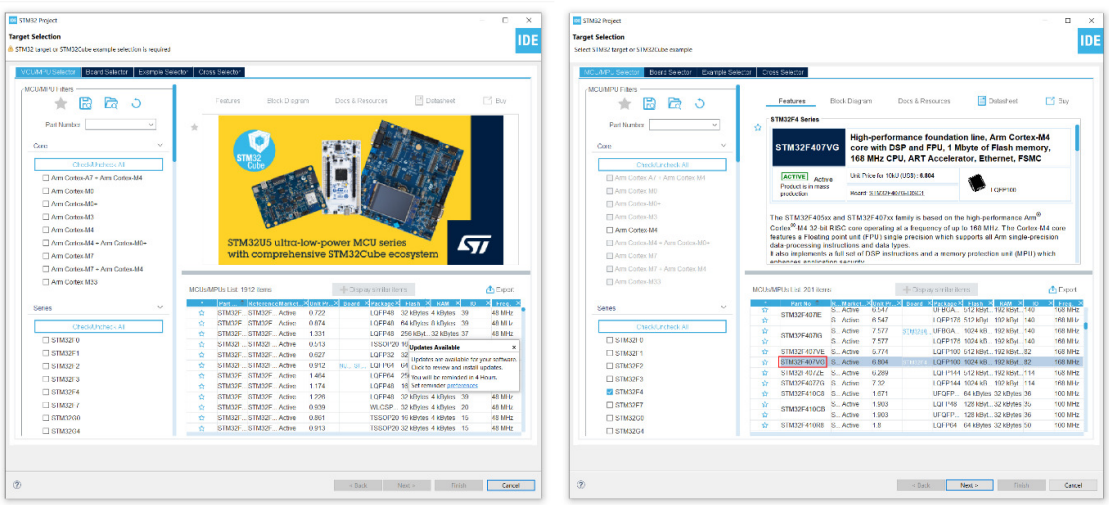
Select the directory where now workspace with project will be created:



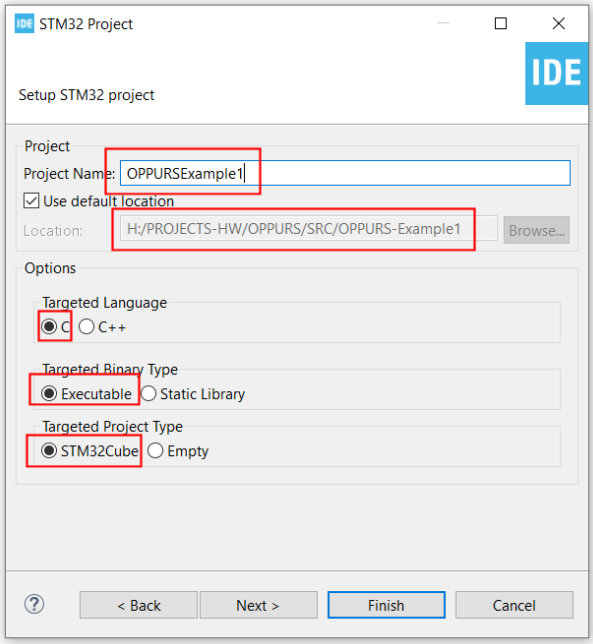
Select *Start new STM32 project*:



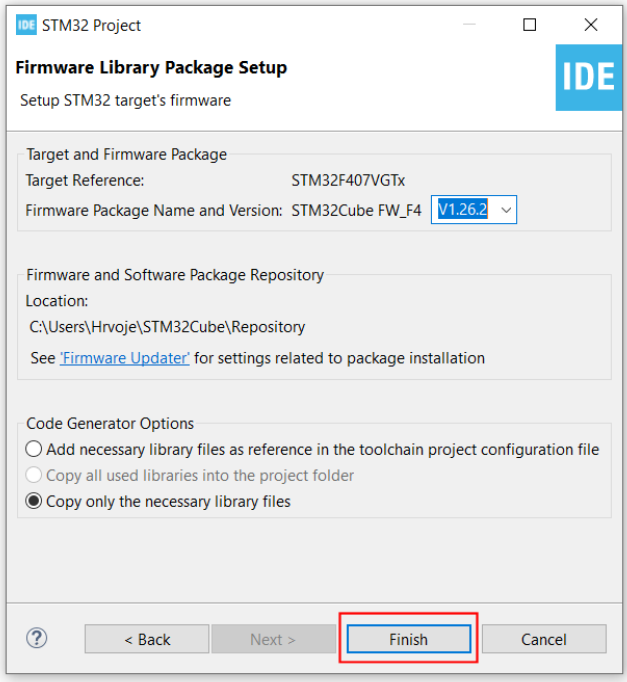
Then select target microcontroller (STM32F407VG in case of STM32F4DISCOVERY):



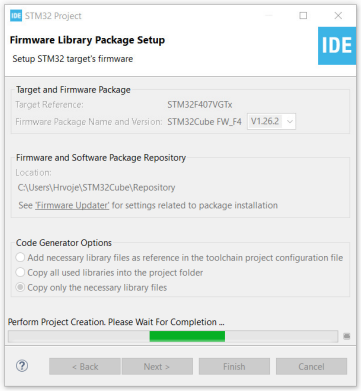
Select new project name (project folder will be created under workspace folder); select C programming language, executable type, STM32Cube (using STMicroelectronics provided HAL libraries):

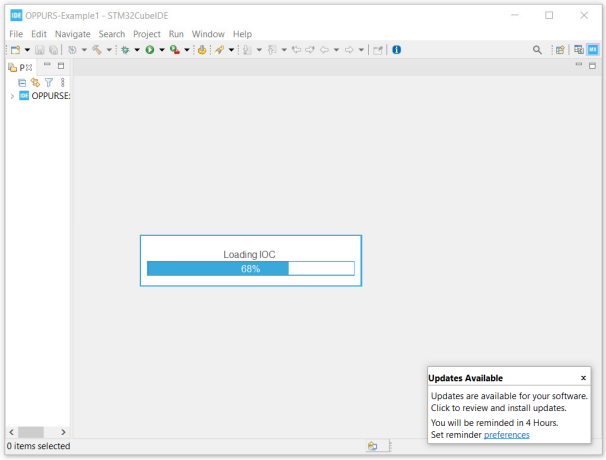


Finish the project creation process:

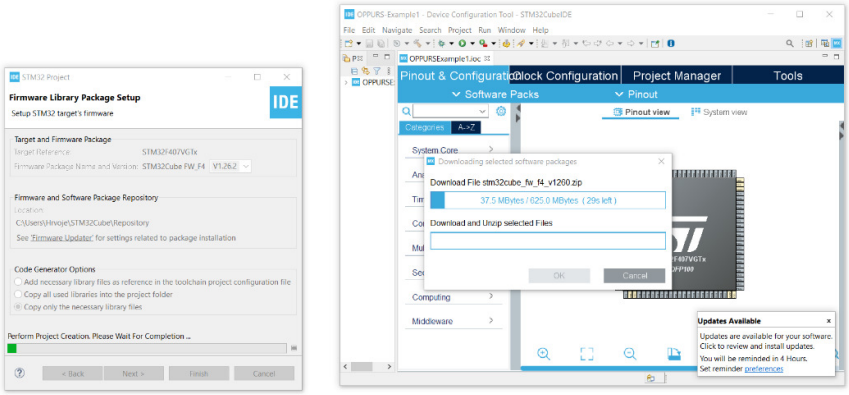


Wait for IDE (integrated development environment) to create necessary files:

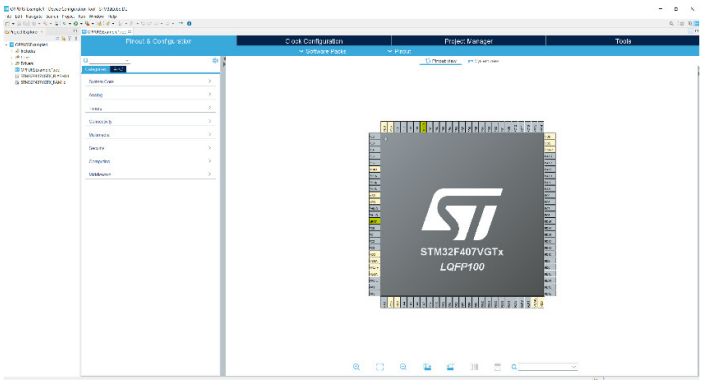




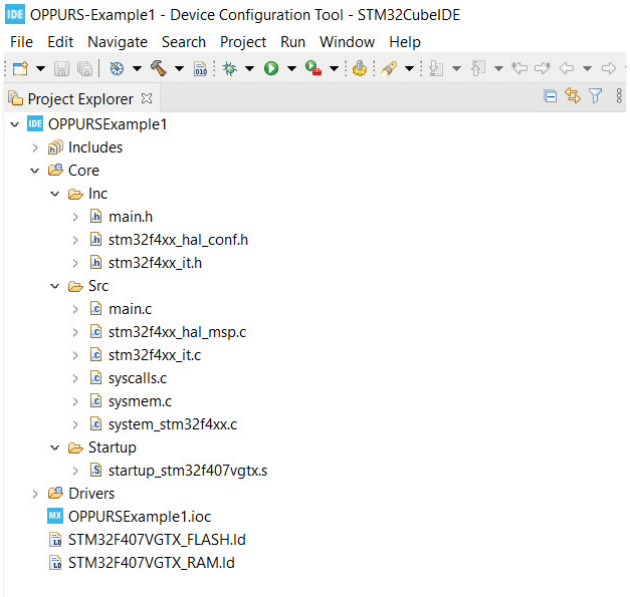
IDE will automatically install the software package for chosen microcontroller family if not already present in system installation – wait for the process to finish (in this case HAL libraries for STM32F4 family):

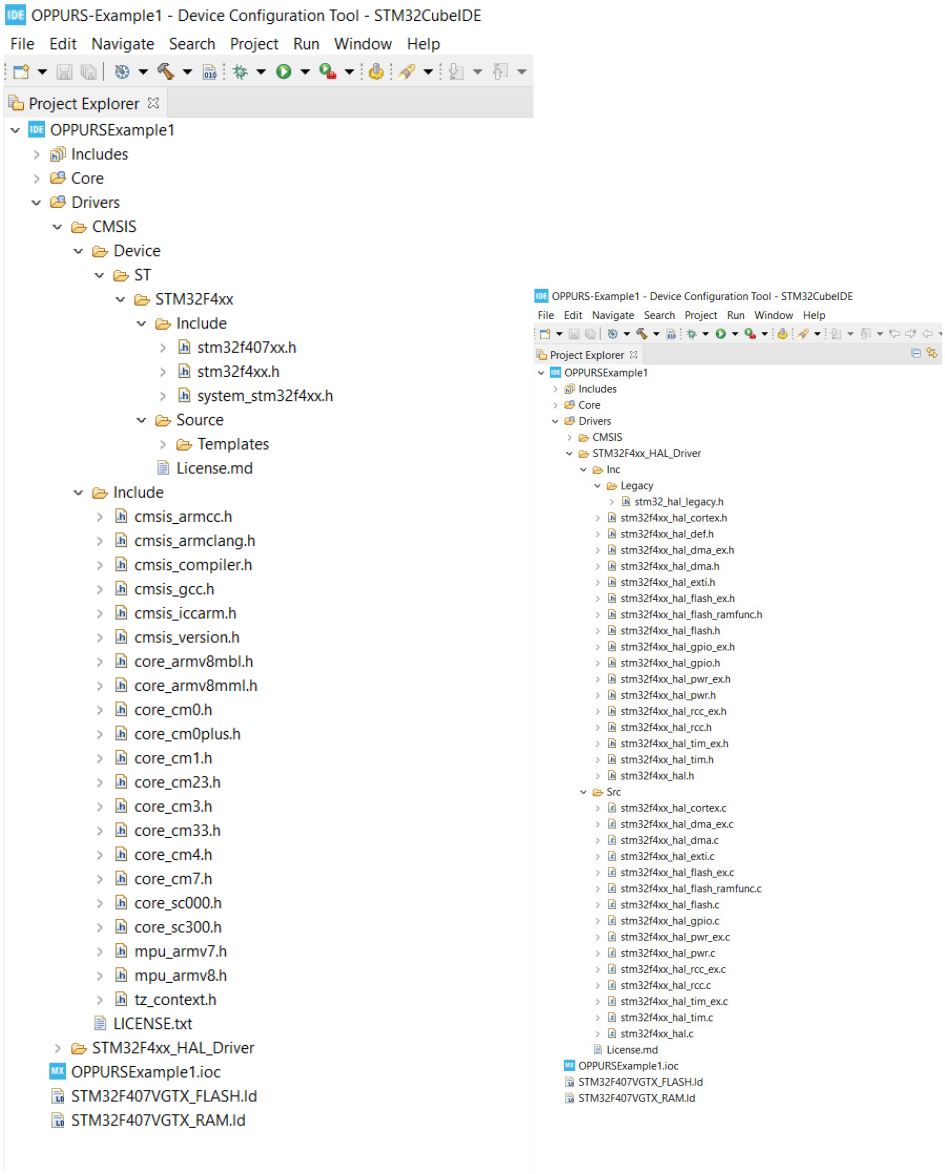


After project is set up the window will look like this:



The source tree looks like this:





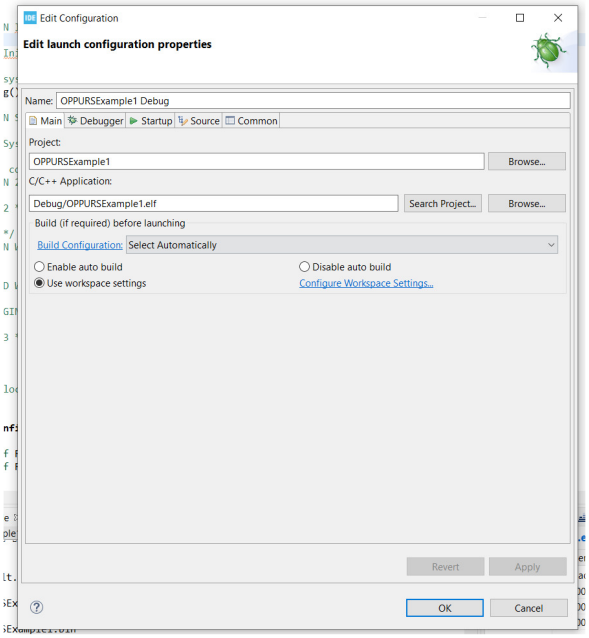
This is a minimum bare metal application skeleton with HAL support that is created by IDE, without any application-specific code yet.

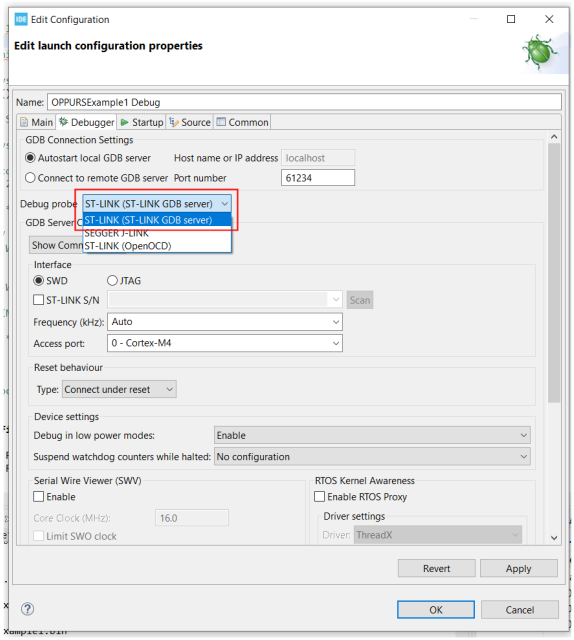
2.2 Validating the system configuration

After creation of the new project, it is recommended to check if all system components are properly configured. The goal of this step is to verify the following:

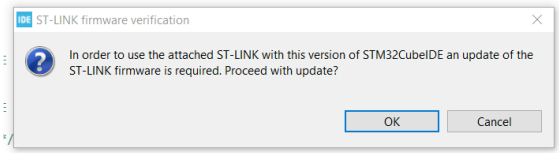
- project builds successfully (IDE and microcontroller-specific support are properly installed)
- target development kit hardware connected to the development machine can be successfully programmed with autogenerated program (verify that USB drivers for ST-Link are properly installed and that computer recognizes the target hardware)

Click on *Run-Debug Configurations* and ensure ST-LINK is selected as a debugger:

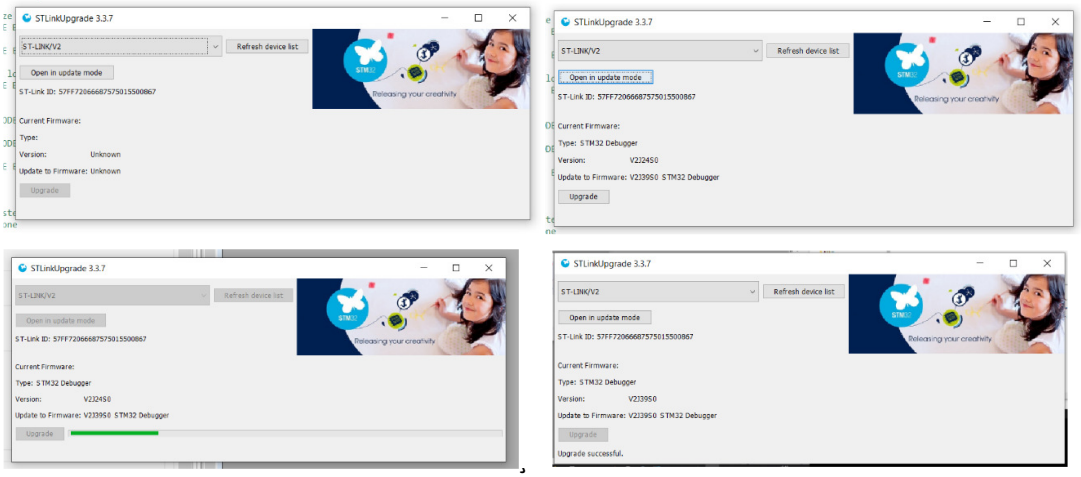




In some cases, an outdated firmware my be present on development kit:

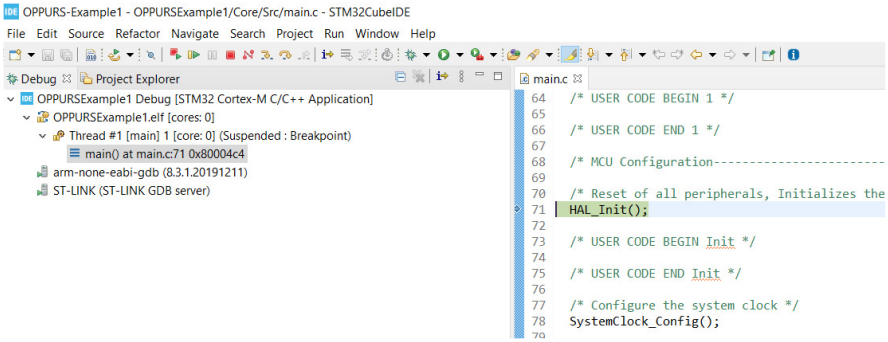


In this case it is necessary to upgrade the firmware (use *Open in update mode* and *Upgrade* buttons):



Note: this step is required only once and only if the outdated firmware is present on a board. In a process it may be required to power cycle the board to put it into update mode.

Click on *Run-Debug* (F11):



If device is programmed successfully, debugger should stop in *main()* function as shown in figure.

2.3 Setting up CubeMX IOC file

STM32CubeMX provide a convenient GUI for setting up device configuration and automatic code generation.

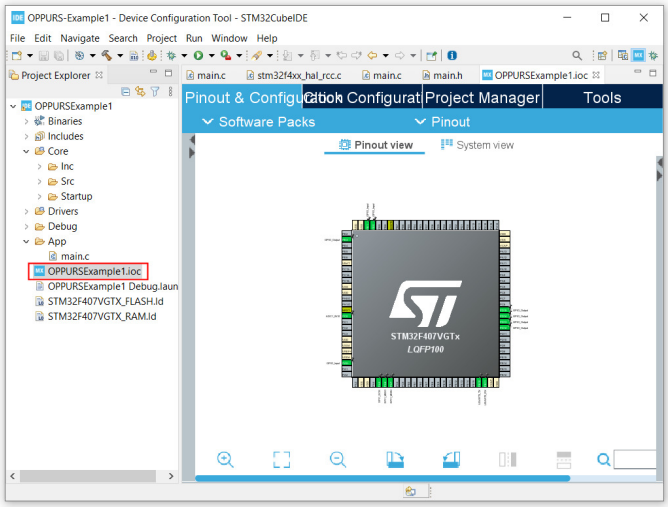
IMPORTANT: Although generated code may be readily included in application as is, this guide will not directly include autogenerated code into final application. There are some shortcomings of such an approach:

- autogenerated code is under the control of configuration GUI and any misconfiguration in user interface will introduce bugs in autogenerated code that are hard to manage
- the program structure is inflexible because it forces the programmer to use the predefined program structure, what may not be well-suited for complex and production-ready software
- all device driver related code is kept in a single large file[2], what introduces problems with tightly coupled code and harder portability
- it is difficult for programmer to do advanced code customizations when autogenerated and manually generated code are mixed, especially in terms of maintenance, upgrades etc.

This guide will make use of autogenerated code but only for a reference and to speed up a development process. The programmer is encouraged to bypass autogenerated code and use own custom HAL library that mimics the structure of autogenerated code but gives the programmer freedom how to organize and structure the code.

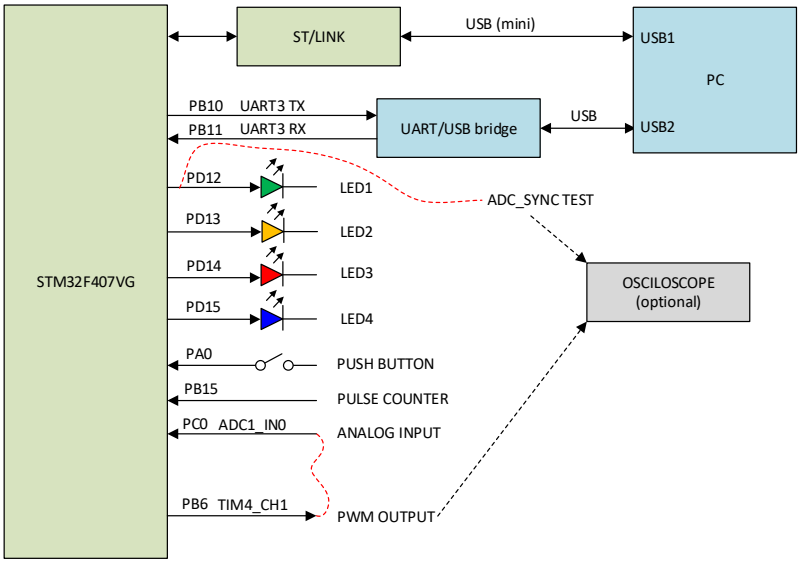
2.3.1 Setting up pin functions

Double-click on OIC file to open GUI configurator:

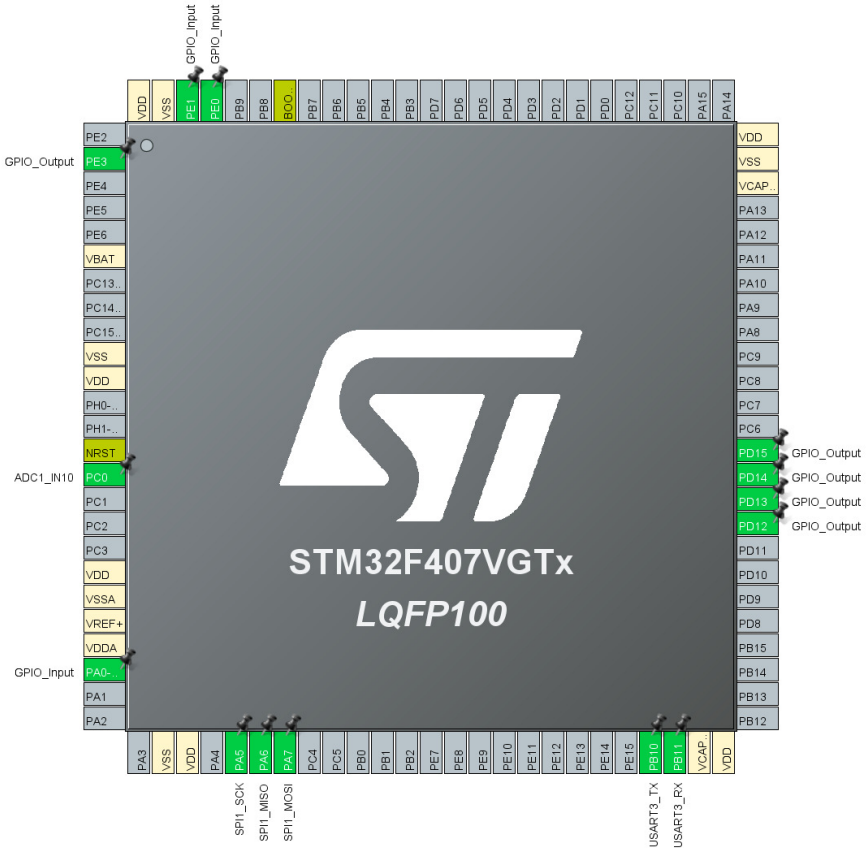


Configurator will be used to set up sample application on a development kit. The following figure shows the block diagram of hardware configuration for examples:

[2] There is an option in IDE that enables the generation of separate files per each peripheral, what makes the code organization more readable and easy to understand. However, the problem of dependancy and code coupling with auto generator still remains.



Use configurator to set up the pins assignments:



IMPORTANT: STM32CubeMX provides two hardware abstraction layer APIs (application-programming interface):

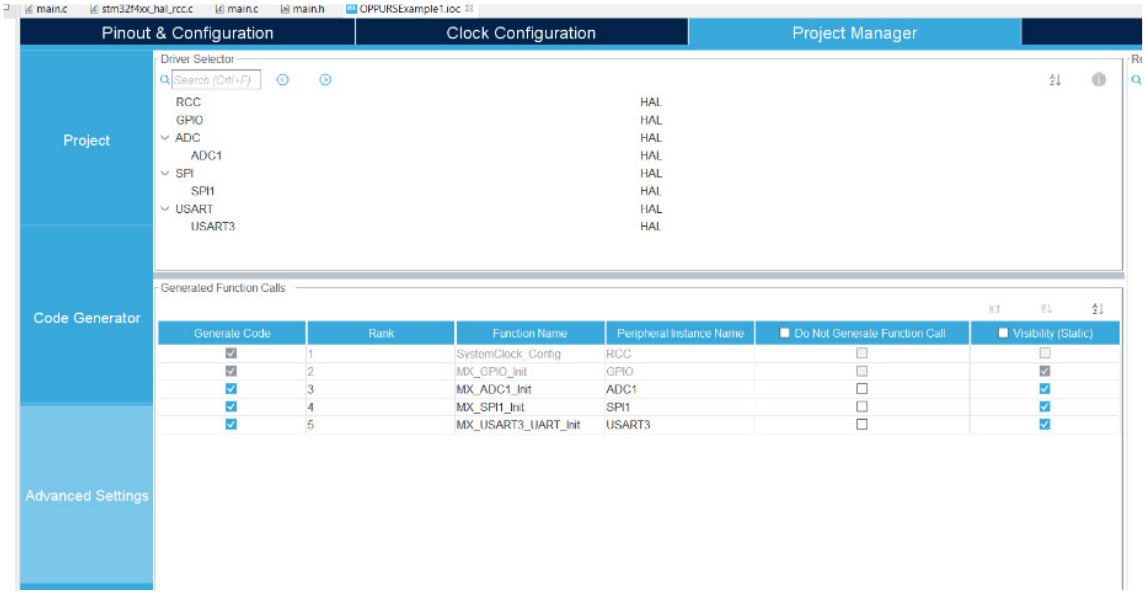
- HAL – high-level hardware abstraction API
- LL – low-level hardware abstraction API

While the HAL functions provide easier and more intuitive programming interface, with programs that may be easier to port across different microcontrollers, there are some drawbacks of using HAL API style library:

- high-level API introduces more layers of abstraction between function calls and actual register access, what makes it harder for programmer to understand what is going on behind the HAL function call; when some very specific chip feature is needed, it takes lots of effort to ensure that HAL API with the associated data structures is properly used
- additional layers of abstraction makes code bigger and slower

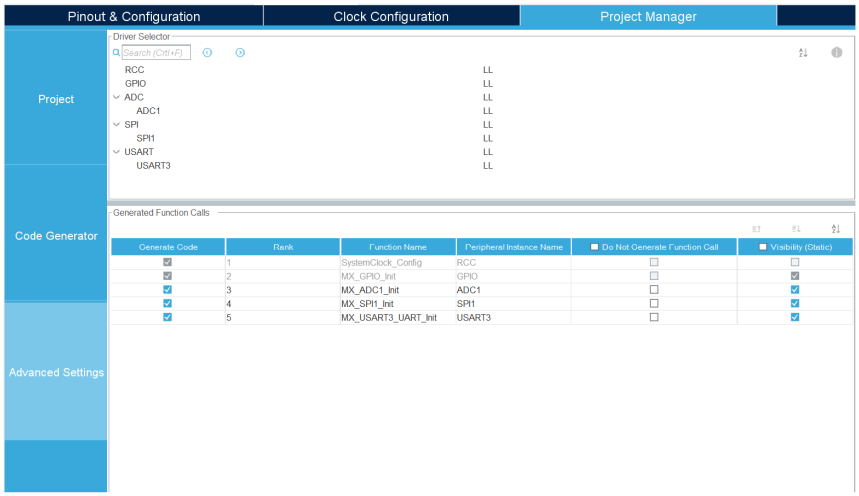
On the other hand, LL API style library is a thin wrapper around the register-oriented programming, saving the programmer lots of time and effort by providing a structured access to microcontroller features and registers in more human readable form than plain direct register oriented programming. It follows closer the guidelines from the datasheet when some specific features are concerned, in contrast to HAL API. Therefore, this guide will assume LL API as the reference one and all examples will be based on LL style API.

Note: Although STM32CubeMX provides HAL API for every peripheral, LL API is provided for most, but not for all peripherals. Peripherals that are not covered by LL support are typically more complex peripheral devices (such as CAN, USB, Ethernet etc.). Even if HAL-only peripheral is used in the project, programmer may still use LL drivers for a chosen set of peripherals while using HAL for others in the same project. However, this use case will not be covered by this guide. Code generator will automatically assume HAL API for every peripheral included via graphic configurator and create configuration code based on HAL. To force the use of LL API, one must select *Project Manager-Advanced Settings*:

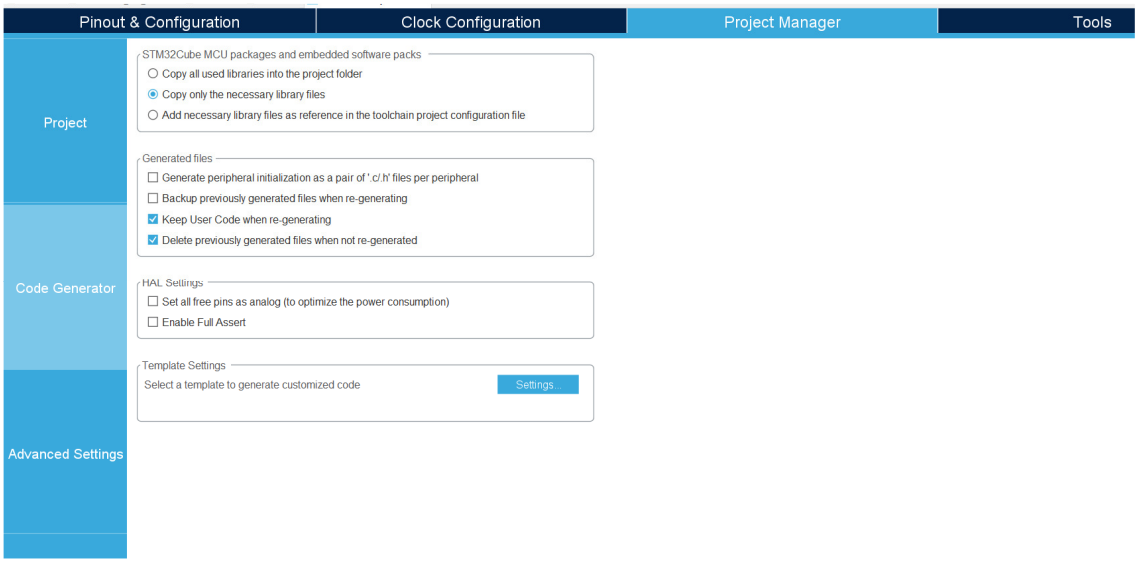


Driver selector shows all peripherals currently selected somewhere in *Configurator* user interface. Whenever a choice for some peripheral is made (e.g. GPIO, timer, ADC etc.) it will appear in this list. HAL API will be assumed by default as shown in figure.

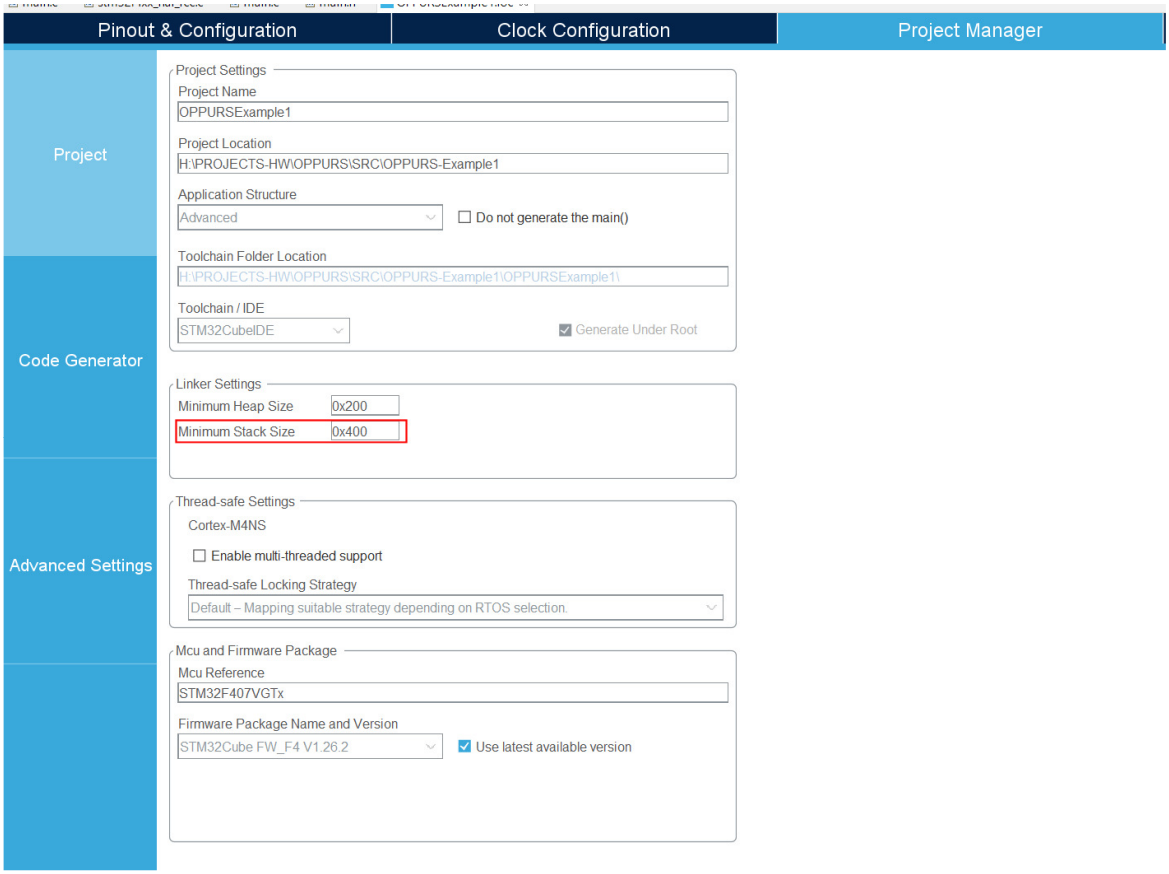
It is important to **manually select LL** for every active peripheral:



Additionally, *Code Generator* tab should look like this:



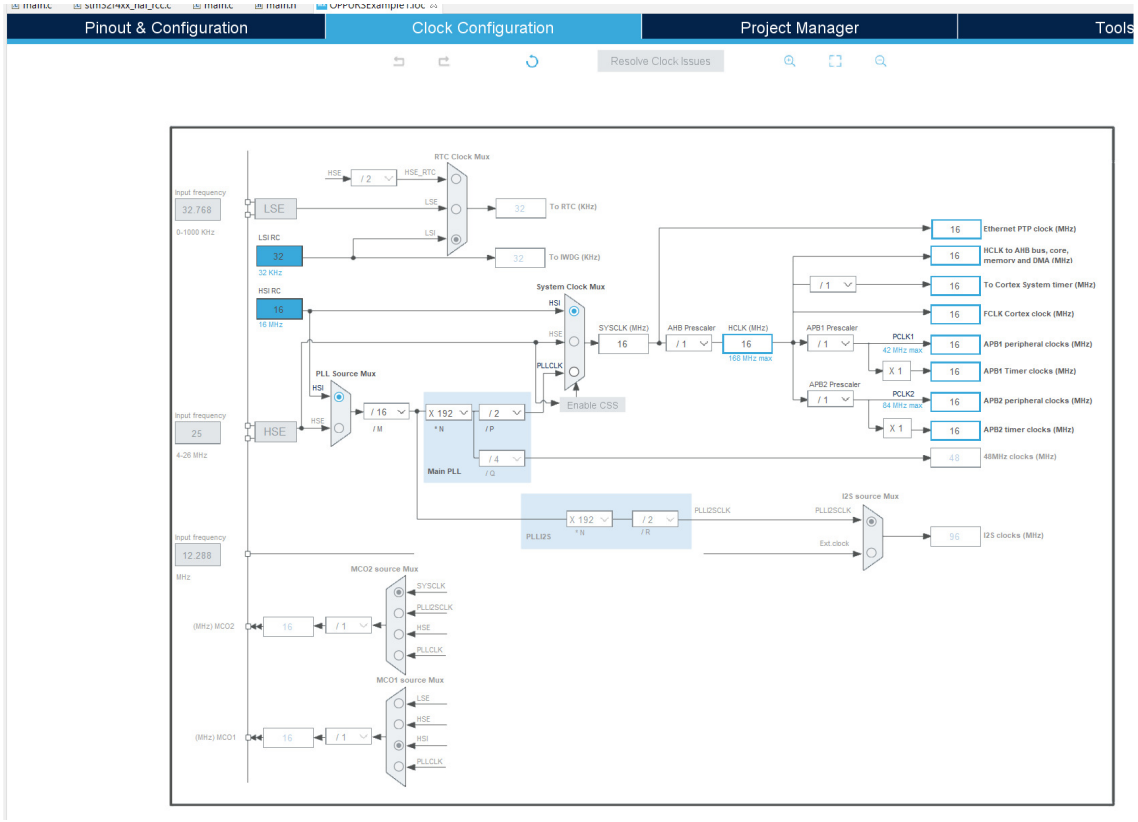
Very important part of the *Project Manager* is a *Project* tab where stack size is defined (0x400 = 1024 bytes by default):



This value will be used in linker scripts for setting up stack size.

2.3.2 Setting up system clock

Another important part of the initial chip setup is clocking scheme. Example of an initial configuration (internal RC oscillator, 16 MHz system clock) is shown in the figure:

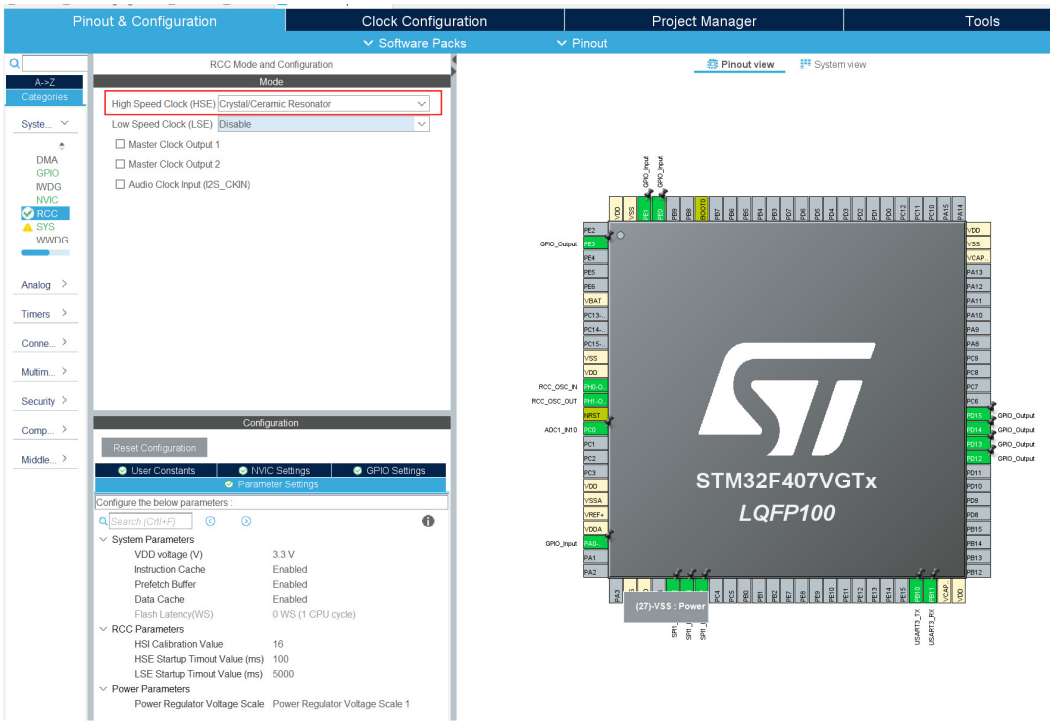


Although this may be satisfactory for some initial development, this setup is not usually desirable:

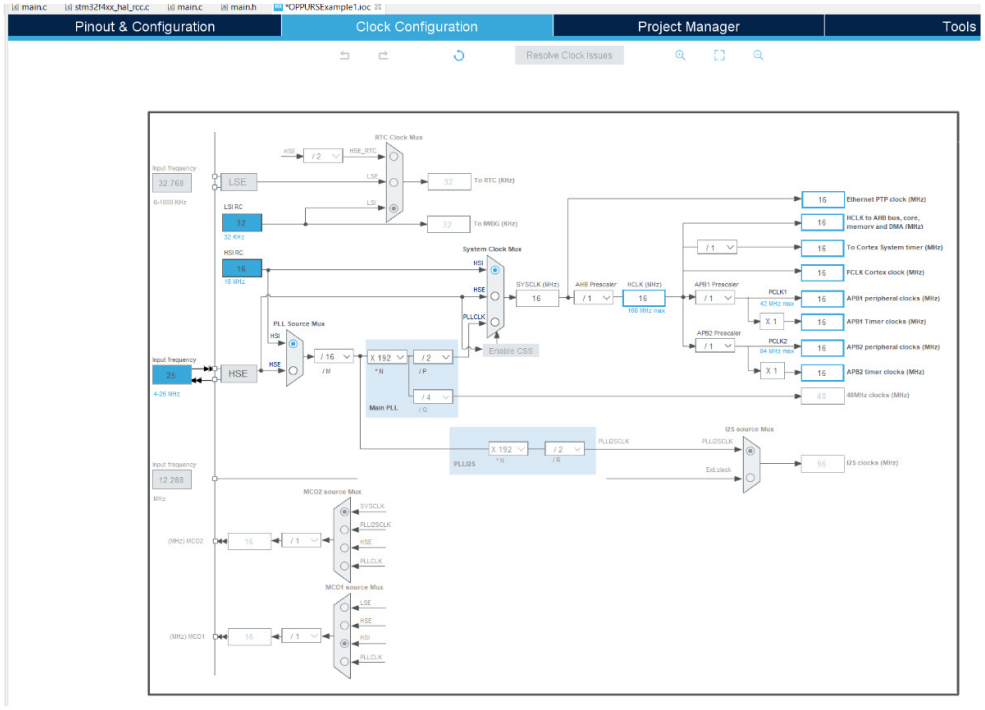
- internal RC clock is not very accurate,
- 16 MHz is slow compared to the maximum speed of the chosen microcontroller (168 MHz).

The following steps will show how to enable *external crystal oscillator* and boost system clock by means of *internal PLL circuitry*.

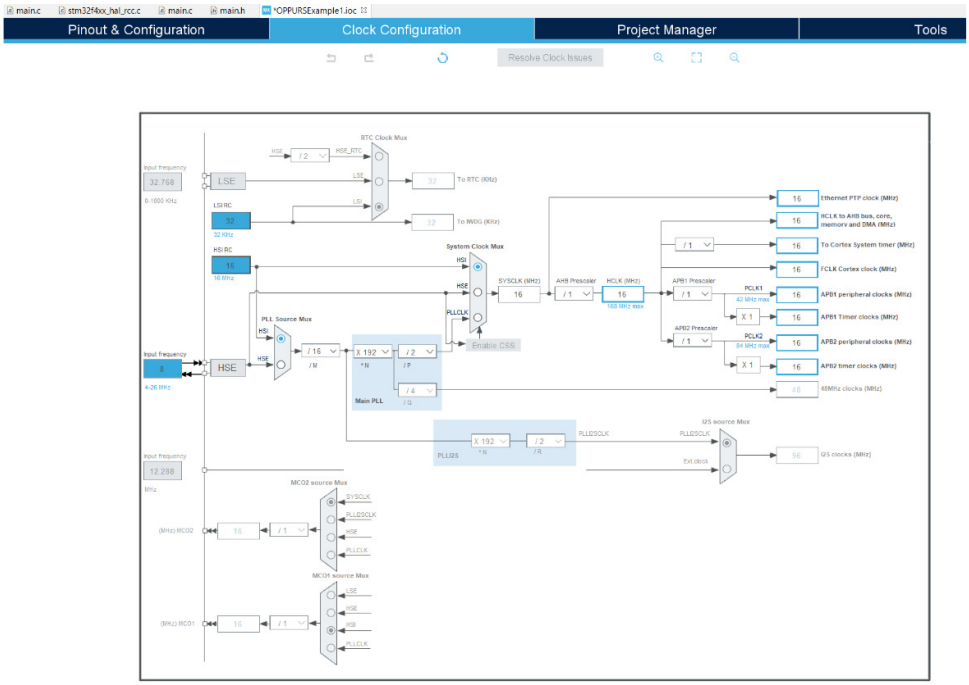
First, enable *High Speed Clock (HSE) – Crystal/Ceramic Resonator* (otherwise, HSE is not accessible in previously clock definition interface):



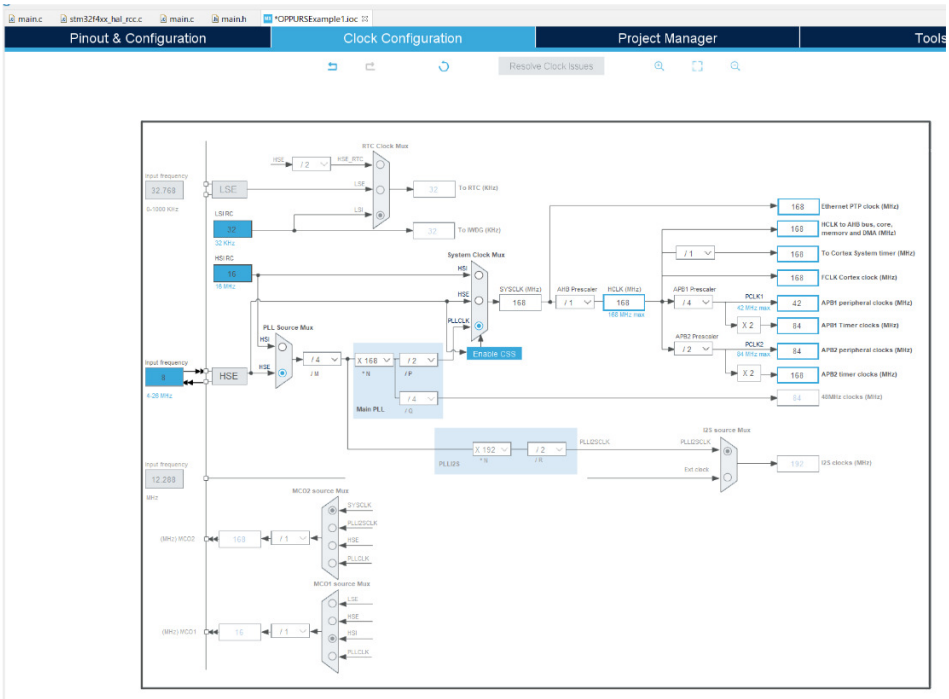
Now external clock (HSE) may be configured:



However, the initial frequency (25 MHz) differs from the on board crystal soldered on the STM43F4DISCOVERY board – it must be changed to 8 MHz:



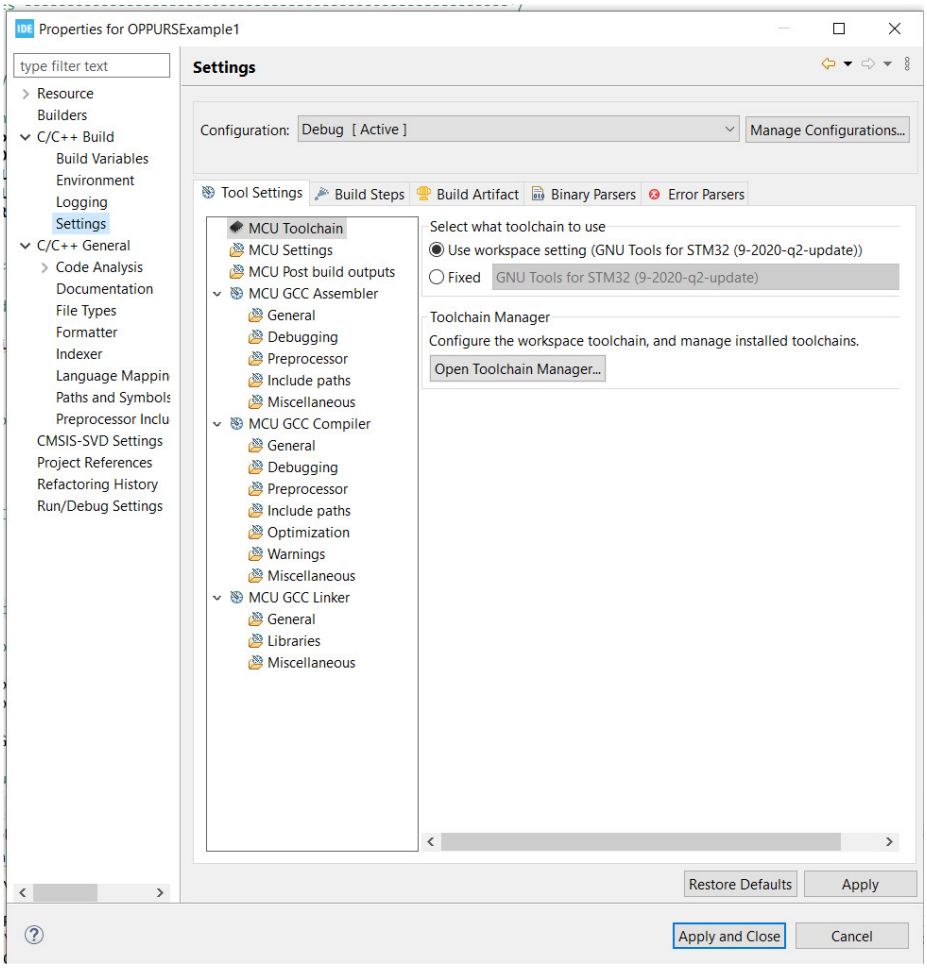
System clock is still 16 MHz. Now adjust clock dividers and PLL circuit to achieve maximum frequency for processor core (168 MHz):



Note: the highest frequency is to processor core and AHB bus. APBx devices operate on a lower maximum frequency (42 / 84 MHz). Please see the microcontroller datasheet for details.

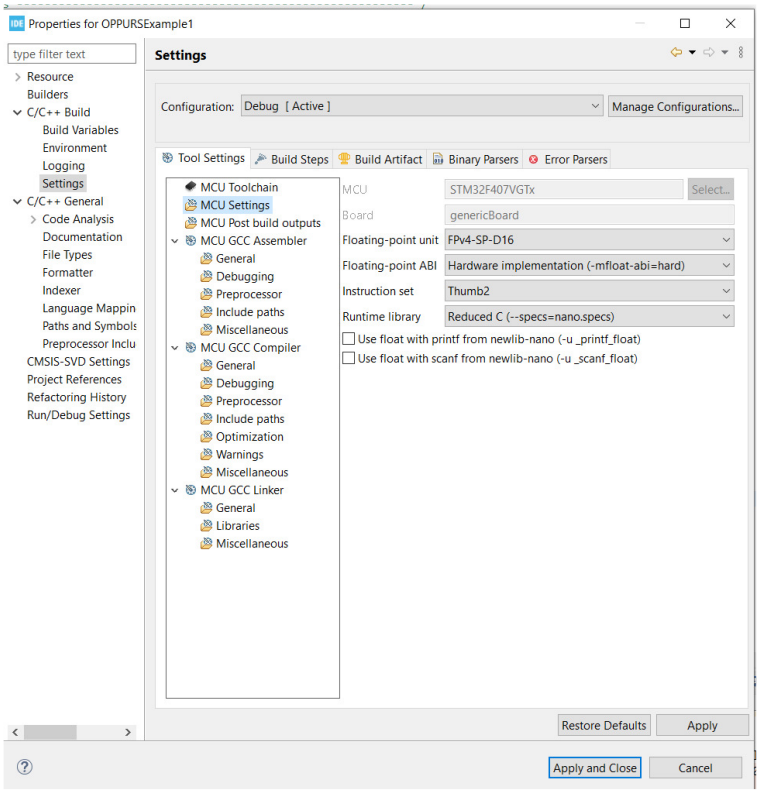
2.4 Setting up compiler options

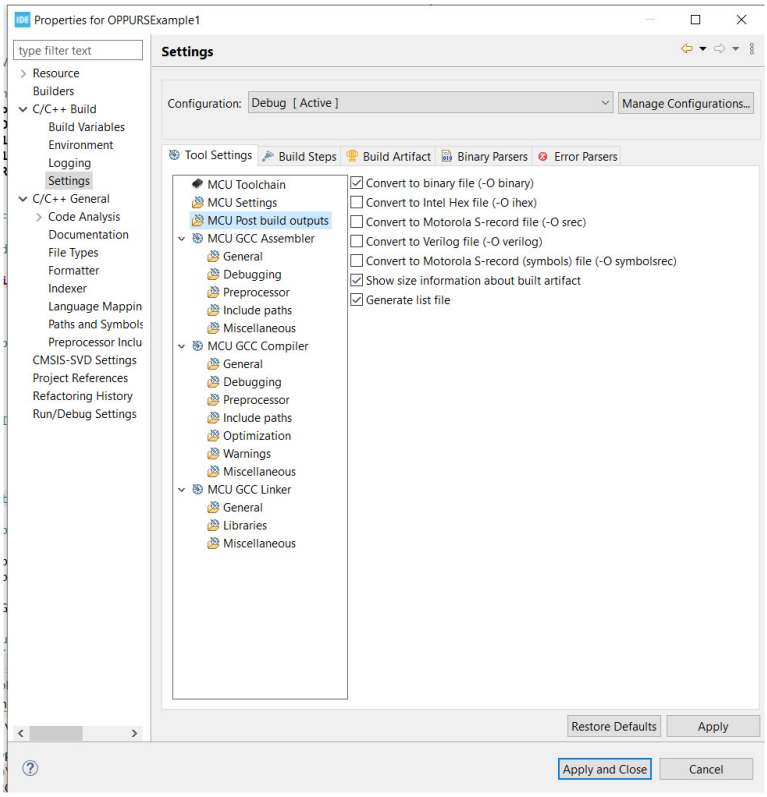
Right click on the project, choose *Properties-C/C++ Build-Settings*:

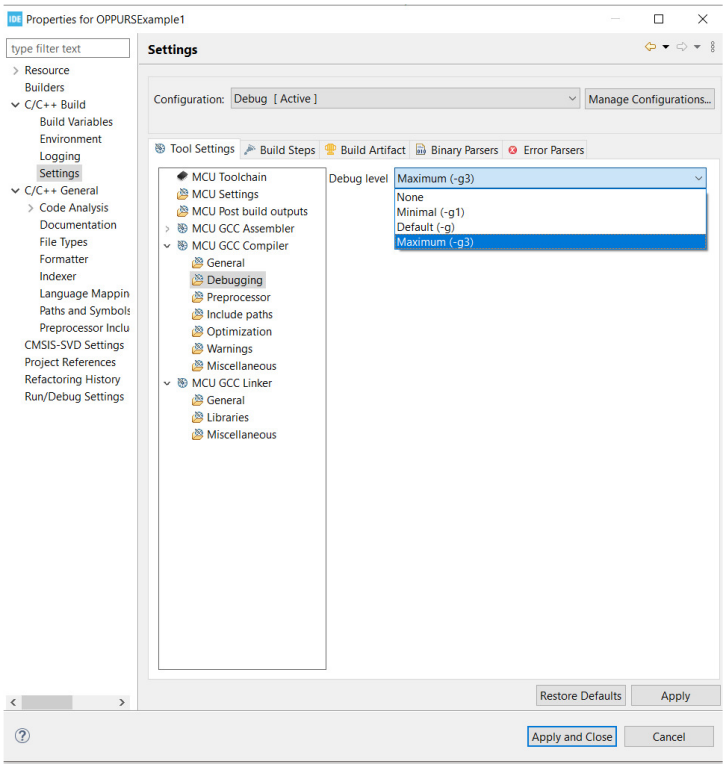


As it is shown in the figure, GNU GCC toolchain is used for building the programs.

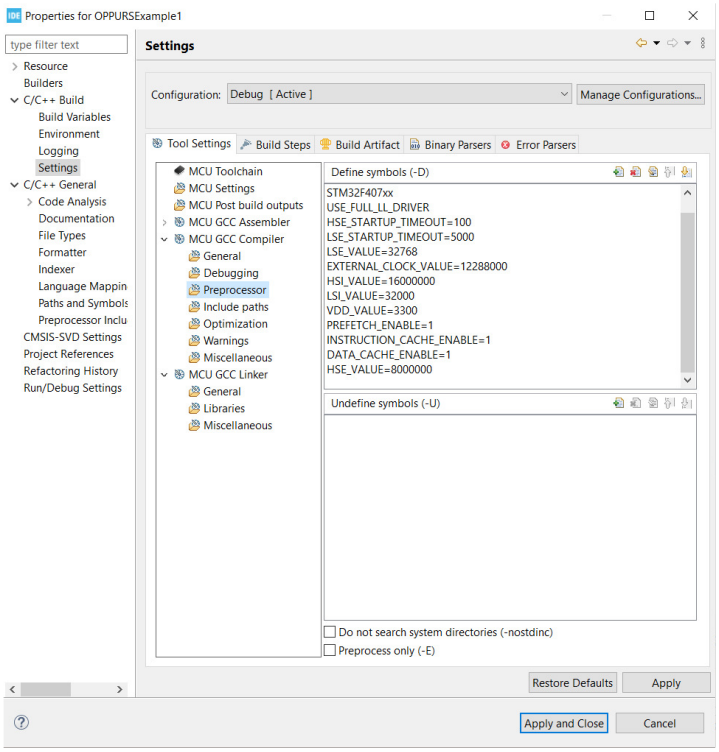
The following screenshots show recommended setup values.


Properties for OPPURSExample1
type filter text
Resource
Builders
C/C++ Build
Build Variables
Environment
Logging
Settings
C/C++ General
Code Analysis
Documentation
File Types
Formatter
Indexer
Language Mappin
Paths and Symbols
Preprocessor Inclu
CMSIS-SVD Settings
Project References
Refactoring History
Run/Debug Settings
Settings
Configuration: Debug [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
MCU Toolchain
MCU Settings
MCU Post build outputs
MCU GCC Assembler
General
Debugging
Preprocessor
Include paths
Miscellaneous
MCU GCC Compiler
General
Debugging
Preprocessor
Include paths
Optimization
Warnings
Miscellaneous
MCU GCC Linker
General
Libraries
Miscellaneous
MCU
STM32F407VGTx
Select...
Board
genericBoard
Floating-point unit
FPv4-SP-D16
Floating-point ABI
Hardware implementation (-mfloat-abi=hard)
Instruction set
Thumb2
Runtime library
Reduced C (--specs=nano.specs)
Use float with printf from newlib-nano (-u _printf_float)
Use float with scanf from newlib-nano (-u _scanf_float)
Restore Defaults
Apply
Apply and Close
Cancel

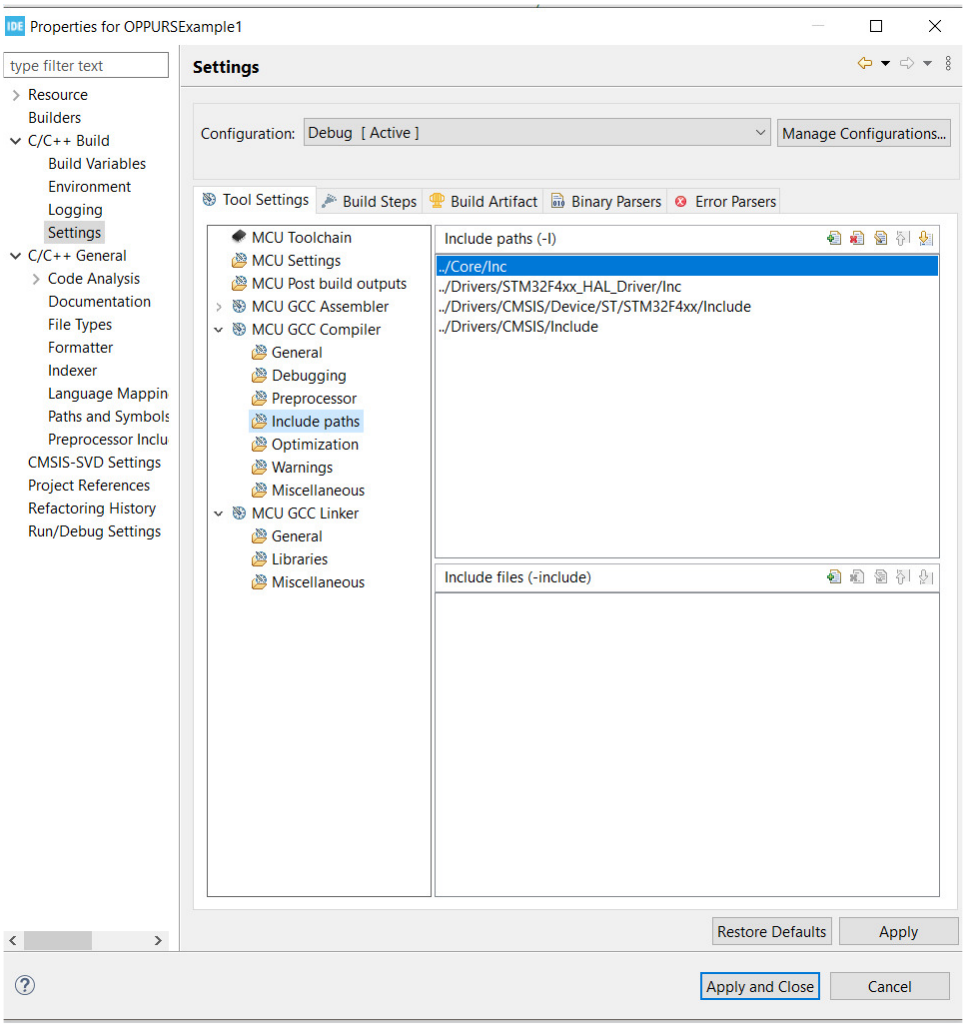

Properties for OPPURSExample1
type filter text
Resource
Builders
C/C++ Build
Build Variables
Environment
Logging
Settings
C/C++ General
Code Analysis
Documentation
File Types
Formatter
Indexer
Language Mappin
Paths and Symbols
Preprocessor Inclu
CMSIS-SVD Settings
Project References
Refactoring History
Run/Debug Settings
Settings
Configuration: Debug [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
MCU Toolchain
MCU Settings
MCU Post build outputs
MCU GCC Assembler
General
Debugging
Preprocessor
Include paths
Miscellaneous
MCU GCC Compiler
General
Debugging
Preprocessor
Include paths
Optimization
Warnings
Miscellaneous
MCU GCC Linker
General
Libraries
Miscellaneous
Convert to binary file (-O binary)
Convert to Intel Hex file (-O ihex)
Convert to Motorola S-record file (-O srec)
Convert to Verilog file (-O verilog)
Convert to Motorola S-record (symbols) file (-O symbolsrec)
Show size information about built artifact
Generate list file
Restore Defaults
Apply
Apply and Close
Cancel



Note: it is recommended during the development to use *Maximum Debugging level* setting, otherwise some important run-time information will not be available during in-circuit debugging (optimized out by compiler).



Note: preprocessor values are automatically updated from Configurator GUI (e.g. HSE_VALUE = 8000000 as we chose 8 MHz for external crystal value in GUI).



Note: Include paths are locations where compiler looks for header files. Later when we make our own additional folders to properly structure the application some additional include paths will be added.

Example source code: archive with complete code example after this step is provided in archive:

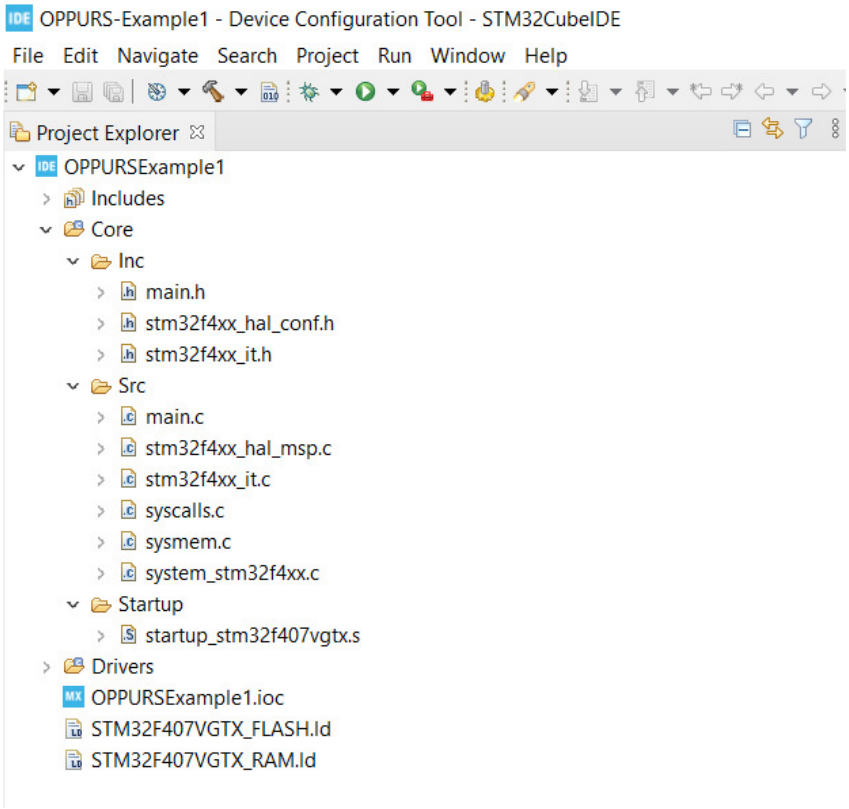
OPPURS-Example1-CreateNew.rar

2.5 Decoupling the application source tree from autogenerated structure

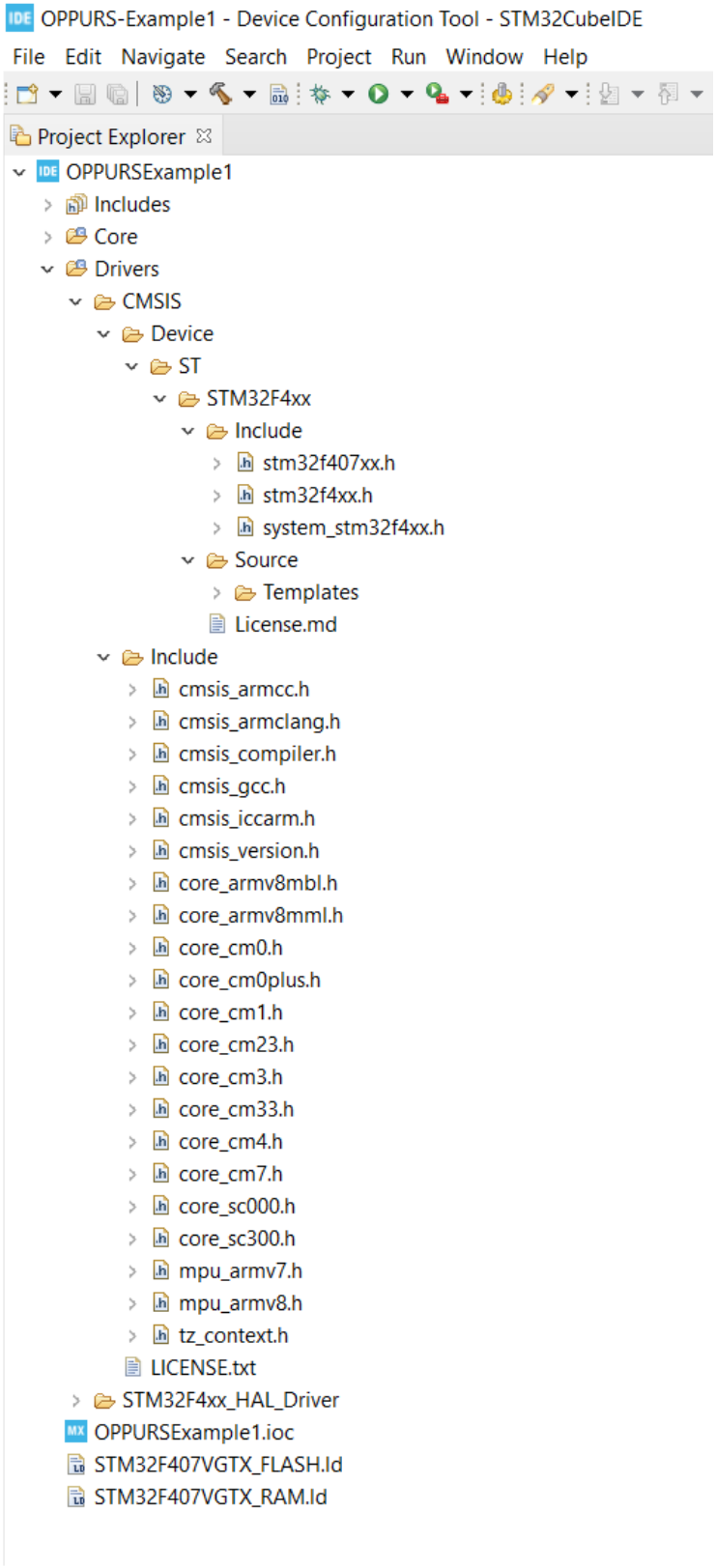
After new project creation, two main subfolders are created:

- *Core* - contains all autogenerated code and sample application structure (controlled by STM32CubeMX code generator)
- *Drivers* - contains the source code of all included HAL/LL peripheral drivers

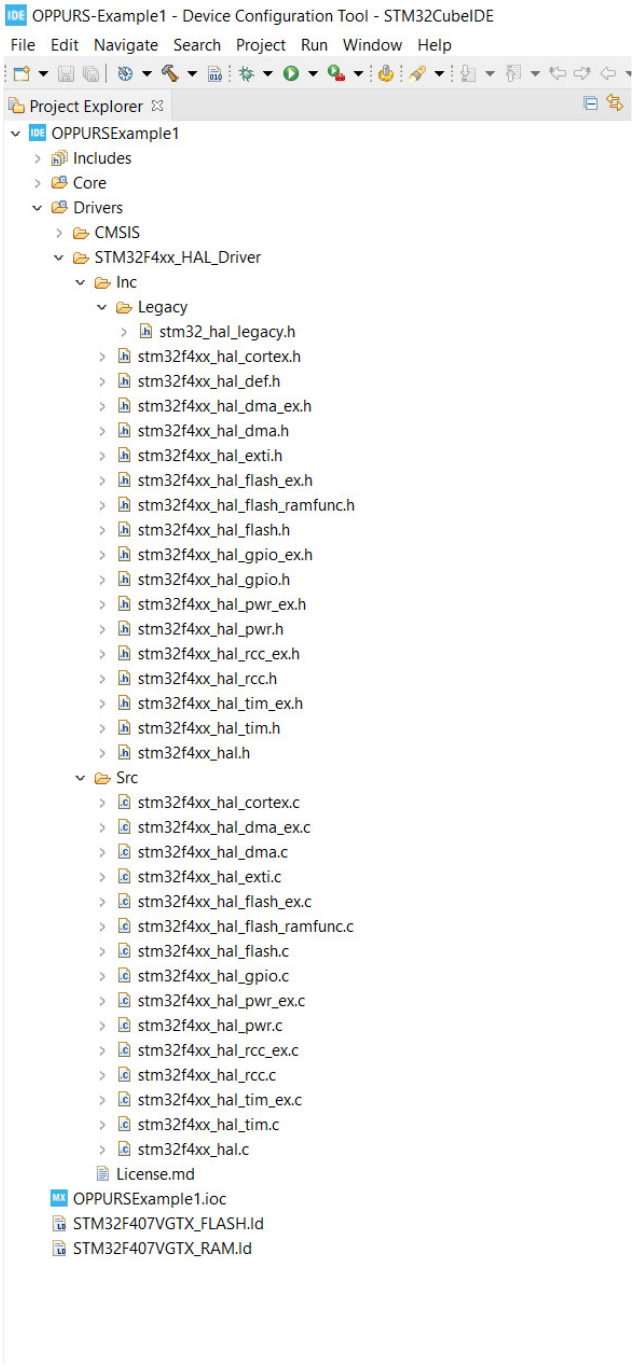
The structure of *Core* folder:



The structure of *Drivers/CMSIS* folder (contains all Cortex-M processor core generic stuff):



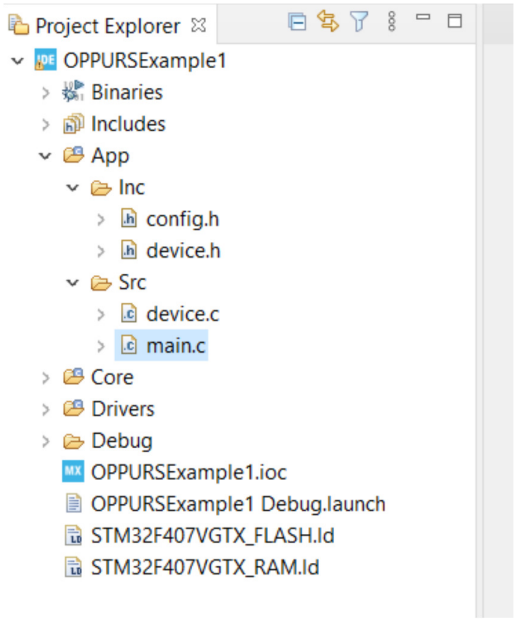
The structure of *Drivers/CMSIS* folder (contains all STM32F4x family specific stuff):



Note: In this example only HAL-style libraries are included (suffix *_hal*). When LL drivers are included the STM32CubeMX will automatically include LL drivers (suffix *_ll*).

All autogenerated files should be left as created, with only few minor changes that will be described below.

The goal is to make our application and all source files completely independent of autogenerated folders *Core* and *Drivers*. For that we need to create new folder called *App*:



In this folder we add two subfolders:

- *Inc* – all header (.h) files
- *Src* – all implementation (.c) files

Header files:

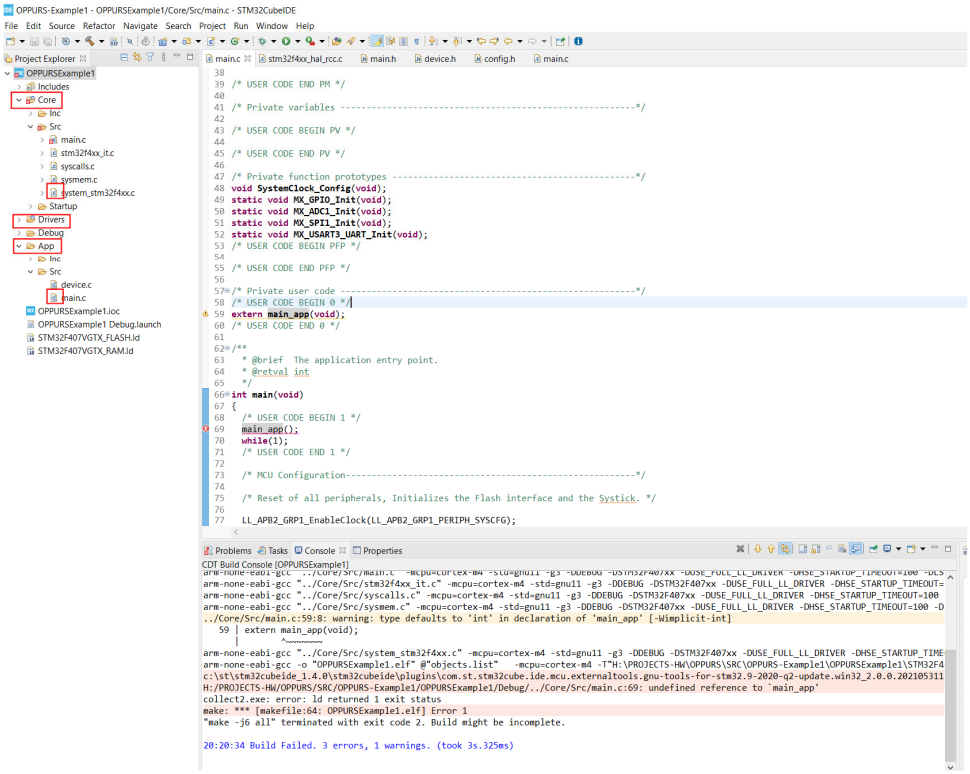
- *config.h* – single file to hold all configurable parameters to build different application versions
- *device.h* – header file to hold definitions of all custom-developed HAL layer (device drivers) to be developed (upon HAL/LL drivers provided by manufacturer).

Implementation files:

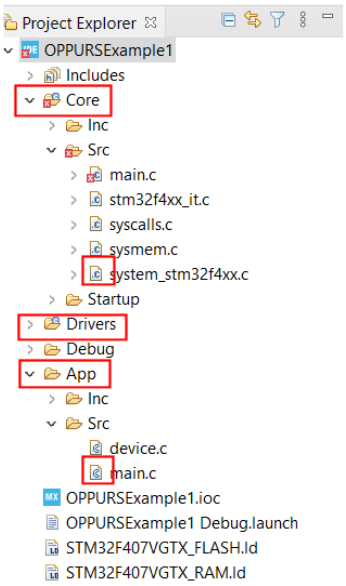
- *device.c* – implementation of all custom device drivers
- *main.c* – implementation of target application

Separation of application source tree enables code decoupling from the autogenerated structure. Realistic application may be far more complex and require many additional header and implementation files (even additional folders under *App* root) but this is the minimum recommended structure to start with.

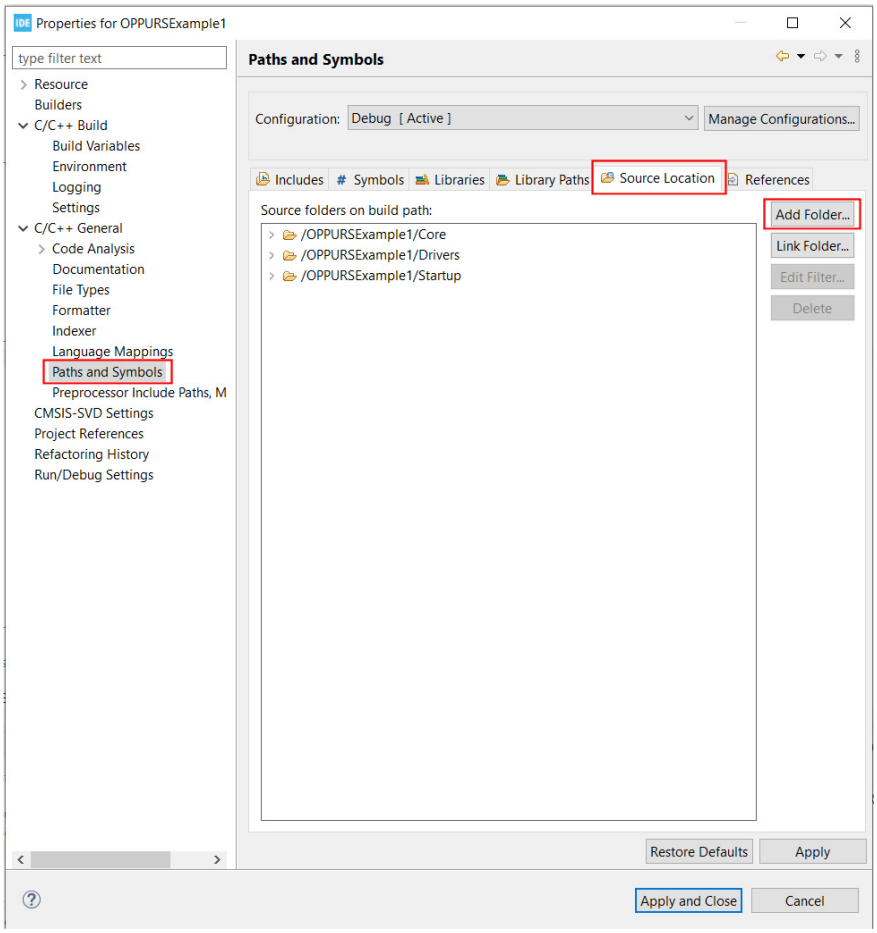
IMPORTANT: One must take care of some particularities of Eclipse environment: simply adding new folder and placing .h and .c files will not automatically include new folder in a build! If proper steps are not undertaken, the following error will arise:

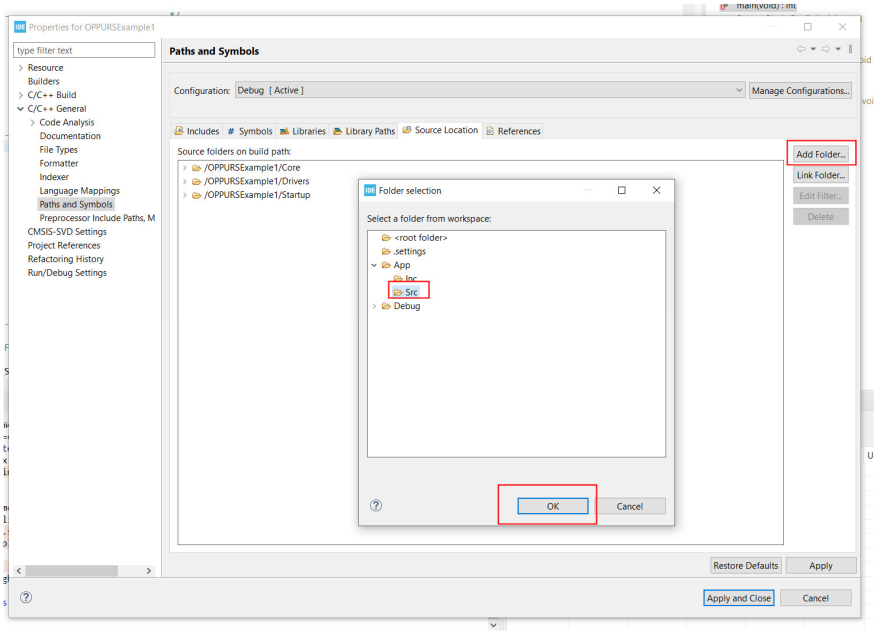


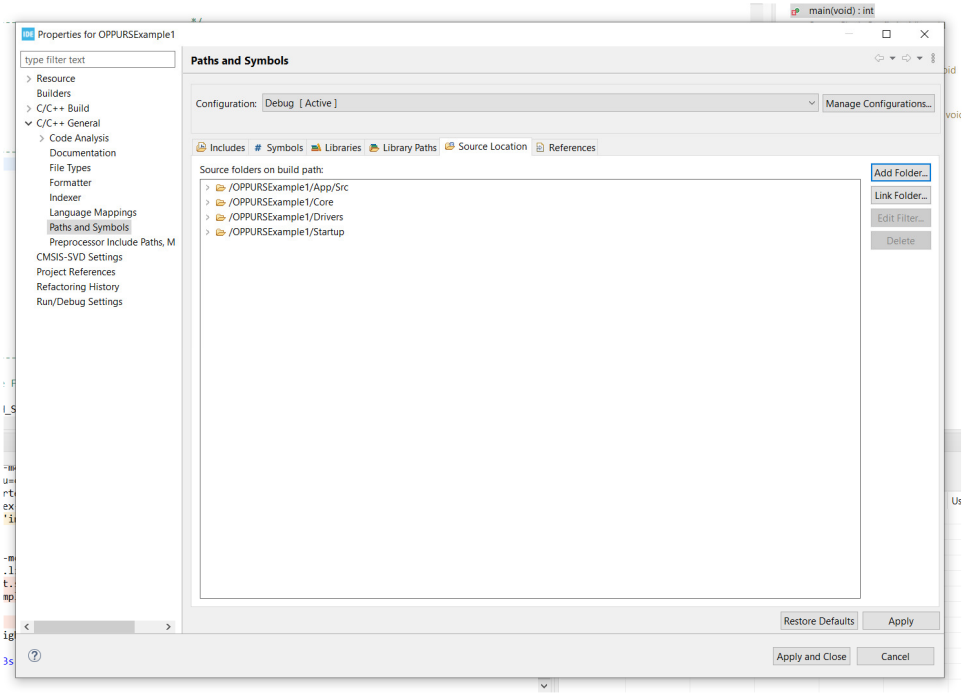
Note that *App* folder icon looks different after creation of new folder (compare with *Core* and *Drivers*):



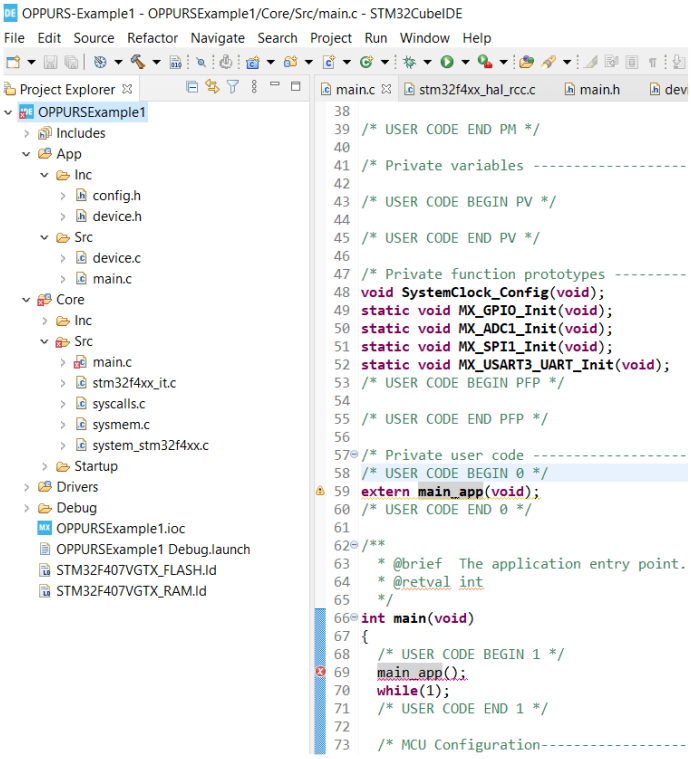
The reason for that is that *App* is not marked as a **source folder**. To add new source folder to project please follow the steps:



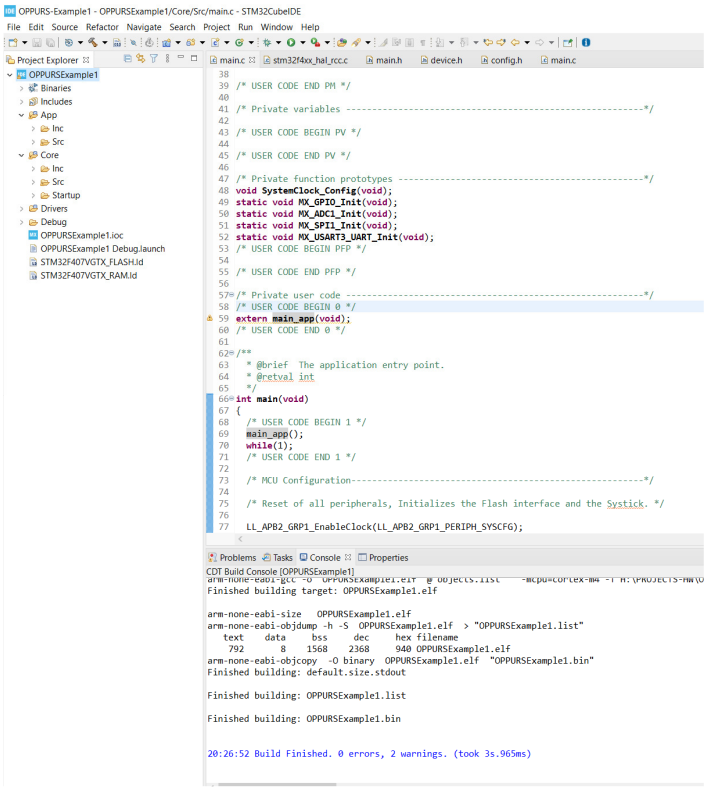




Now *App* folder is properly included in project as it is marked as a source folder:



After that, build should work properly:



How to "bypass" autogenerated code in *Core/Src/main.c*?

Program will enter *main()* function in file *Core/Src/main.c* – it is recommended in the context of this guidelines to exit code generator *main()* function as soon as possible, for example:

```
int main(void)
{
  /* USER CODE BEGIN 1 */
  main_app();
  while(1);
  /* USER CODE END 1 */
```

Function *main_app()* serves as a user-defined entry point where *main()* function jumps immediately. In this example this function is implemented in *App/Src/main.c*:

```
void main_app()
{
   while(1);
}
```

Example source code: archive with complete code example after this step is provided in archive:

OPPURS-Example1-AddingAppFolder.rar

2.6 Examining autogenerated code structure

This chapter will give short and high level overview of the most important files under *Core* folder, explain the most important features in these files and point out how to use the content of these files, either to copy some of the generated content in the custom developed device drivers or to make minimum modifications of these files to connect them with a code in a separated *App* folder.

2.6.1 Core/Src/startup_stm32f407xx.s

It is the assembler startup file with important bootstrapping routines.

Some of the most important features in this file are as follows:

2.6.1.1 Reset handler

This is the only *exception handler* coded in assembler – it runs automatically after system reset:

```
    .section  .text.Reset_Handler
  .weak  Reset_Handler
  .type  Reset_Handler, %function
Reset_Handler:
  ldr   sp, =_estack     /* set stack pointer */

...
```

Note that Reset handler finally jumps into *main()*:

```
/* Call the application's entry point.*/
  bl  main
  bx  lr
.size  Reset_Handler, .-Reset_Handler
```

2.6.1.2 Interrupt vector table (IVT)

IVT is coded in this source file:

```
/******************************************************************************
*
* The minimal vector table for a Cortex M3. Note that the proper constructs
* must be placed on this to ensure that it ends up at physical address
* 0x0000.0000.
*
******************************************************************************/
   .section  .isr_vector,"a",%progbits
  .type  g_pfnVectors, %object
  .size  g_pfnVectors, .-g_pfnVectors


g_pfnVectors:
  .word  _estack
  .word  Reset_Handler
  .word  NMI_Handler
  .word  HardFault_Handler
  .word  MemManage_Handler
  .word  BusFault_Handler
  .word  UsageFault_Handler
  .word  0
  .word  0
  .word  0
  .word  0
  .word  SVC_Handler
  .word  DebugMon_Handler
  .word  0
  .word  PendSV_Handler
  .word  SysTick_Handler

  /* External Interrupts */
  .word     WWDG_IRQHandler                   /* Window WatchDog              */
  .word     PVD_IRQHandler                    /* PVD through EXTI Line detection
*/
```

```
  .word     TAMP_STAMP_IRQHandler             /* Tamper and TimeStamps through the
EXTI line */
  .word     RTC_WKUP_IRQHandler               /* RTC Wakeup through the EXTI line
*/
  .word     FLASH_IRQHandler                  /* FLASH                        */

...
```

2.6.1.3 ISR handlers

ISR handler names are provided for all IVT entries – they are used in *stm32f4xx_it.h/c* files providing all other exception handlers (except Reset handler):

```
/*******************************************************************************
*
* Provide weak aliases for each Exception handler to the Default_Handler.
* As they are weak aliases, any function with the same name will override
* this definition.
*
*******************************************************************************/
   .weak      NMI_Handler
   .thumb_set NMI_Handler,Default_Handler

   .weak      HardFault_Handler
   .thumb_set HardFault_Handler,Default_Handler
```

2.6.2 Core/Src/stm32f4xx_it.h

It contains the headers of the interrupt handlers, for example:

```
/* Exported functions prototypes ---------------------------------------------*/
void NMI_Handler(void);
void HardFault_Handler(void);
void MemManage_Handler(void);
void BusFault_Handler(void);
void UsageFault_Handler(void);
void SVC_Handler(void);
void DebugMon_Handler(void);
void PendSV_Handler(void);
void SysTick_Handler(void);
void ADC_IRQHandler(void);
void TIM2_IRQHandler(void);
void USART3_IRQHandler(void);
void EXTI15_10_IRQHandler(void);
void DMA2_Stream0_IRQHandler(void);
/* USER CODE BEGIN EFP */
```

The first part shows all handlers automatically included (yellow). The second part shows autogenerated ISR definitions only for peripherals defined by Configurator that uses ISRs (green).

For example, if the application code sets up TIM2 interrupt (e.g. on counter overflow) the function *TIM2_IRQHandler()* will be called because it was defined in *Core/Src/startup_stm32f407xx.s*. Files *stm32f4xx_it.h/c* provide only C function implementation for this ISR symbol as defined in startup assembly file.

2.6.3 Core/Src/stm32f4xx_it.c

This file contains bodies of ISR as C functions. For example, this is the autogenerated body for TIM2 ISR:

```
/**
  * @brief This function handles TIM2 global interrupt.
  */
void TIM2_IRQHandler(void)
{
  /* USER CODE BEGIN TIM2_IRQn 0 */
   TIM2_IRQHandler_Callback();
  /* USER CODE END TIM2_IRQn 0 */
  /* USER CODE BEGIN TIM2_IRQn 1 */

  /* USER CODE END TIM2_IRQn 1 */
}
```

Since this code is autogenerated, programmer may put custom code into appropriate places marked by comments. However, this introduces unwanted code dependencies between *App* and *Core* source tree since ISR may use custom defined global data and structures only available within *App* source tree.

To maximally decouple such dependencies and need for unnecessary cross-reference between source trees, it is recommended to use autogenerated *stm32f4xx_it.h/c* files as entry point into ISRs while keeping the custom ISR logic strictly under *App* source tree. This can be easily done by providing only a single function call from autogenerated ISR function to a ISR handler (callback function) which is the integral part of *App* source tree, for example named *TIM2_IRQHandler_Callback()*.

The function *TIM2_IRQHandler_Callback()* should be placed in *App/Src/device.c* and it is a plain C void function that ISR calls. *TIM2_IRQHandler_Callback()* is able to use global variables and data structures defined for *device.h/c* under *App* source tree, without a need to introduce unwanted cross-dependencies towards *Core* source tree.

In the examples below the content of *TIM2_IRQHandler_Callback()* will be shown into more details but at this point this content is not important, only the mechanism how ISR calls the appropriate application-defined callbacks.

2.6.4 Core/Src/main.h

This header file contains all include files with chosen HAL libraries (HAL/LL type), for example:

```
/* Includes ------------------------------------------------------------------*/
#include "stm32f4xx_ll_adc.h"
#include "stm32f4xx_ll_dma.h"
#include "stm32f4xx_ll_rcc.h"
#include "stm32f4xx_ll_bus.h"
#include "stm32f4xx_ll_system.h"
#include "stm32f4xx_ll_exti.h"
#include "stm32f4xx_ll_cortex.h"
#include "stm32f4xx_ll_utils.h"
#include "stm32f4xx_ll_pwr.h"
#include "stm32f4xx_ll_spi.h"
#include "stm32f4xx_ll_tim.h"
#include "stm32f4xx_ll_usart.h"
#include "stm32f4xx_ll_gpio.h"
```

If we implement our own HAL layer in *device.h/c*, it is recommended to include (in *device.h*) the list of the same header files as shown above for autogenerated *main.h* (since */Core/Src/main.h* reflects what peripheral drivers we have chosen in *Configurator* GUI).

It is also recommended to copy defines for priority group definitions from */Core/Src/main.h* to custom HAL driver header *device.h*:

```
/* Private defines -----------------------------------------------------------*/
#ifndef NVIC_PRIORITYGROUP_0
#define NVIC_PRIORITYGROUP_0         ((uint32_t)0x00000007) /*!< 0 bit  for pre-
emption priority,
                                                                 4 bits for
subpriority */
#define NVIC_PRIORITYGROUP_1         ((uint32_t)0x00000006) /*!< 1 bit  for pre-
emption priority,
                                                                 3 bits for
subpriority */
#define NVIC_PRIORITYGROUP_2         ((uint32_t)0x00000005) /*!< 2 bits for pre-
emption priority,
                                                                 2 bits for
subpriority */
#define NVIC_PRIORITYGROUP_3         ((uint32_t)0x00000004) /*!< 3 bits for pre-
emption priority,
                                                                 1 bit  for
subpriority */
#define NVIC_PRIORITYGROUP_4         ((uint32_t)0x00000003) /*!< 4 bits for pre-
emption priority,
                                                                 0 bit  for
subpriority */
#endif
```

2.6.5 Core/Src/main.c

2.6.5.1 Redirecting main()to user-defined main_app()

This file contains the body of *main()* function, which should be exited as soon as possible to jump into custom defined entry point function (*main_app()*). However, the compiler will complain if we do not provide a definition for *main_app()* – it is sufficient just to provide a manual reference to an external function, there is no need to include any headers from the *App* source tree:

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
extern main_app(void);
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */
  main_app();
  while(1);
  /* USER CODE END 1 */
```

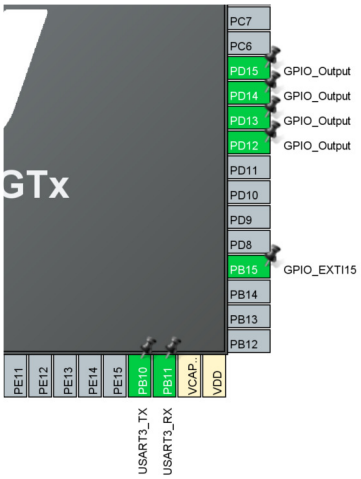
2.6.5.2 Analysis of autogenerated code in main.c

Most of the device-specific initialization code generated for configuration defined by GUI in Configurator will be placed in *main.c*.[3]

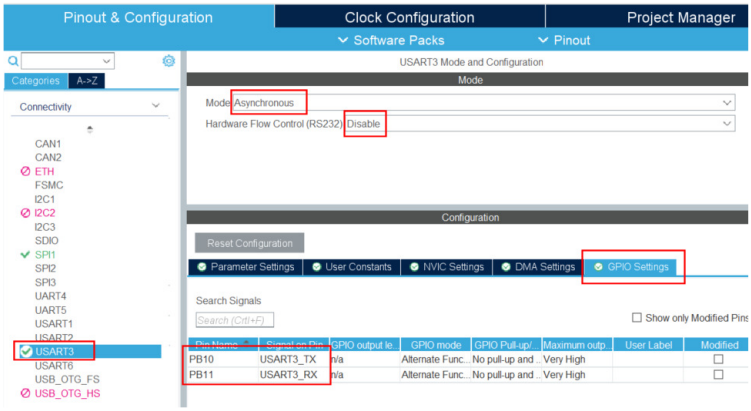
[3] Or multiple files, if the option to auto generate device driver code in file-per-peripheral fashion was chosen.

The following example will show how GPIO and USART definitions in *Configurator* GUI are translated into autogenerated code.

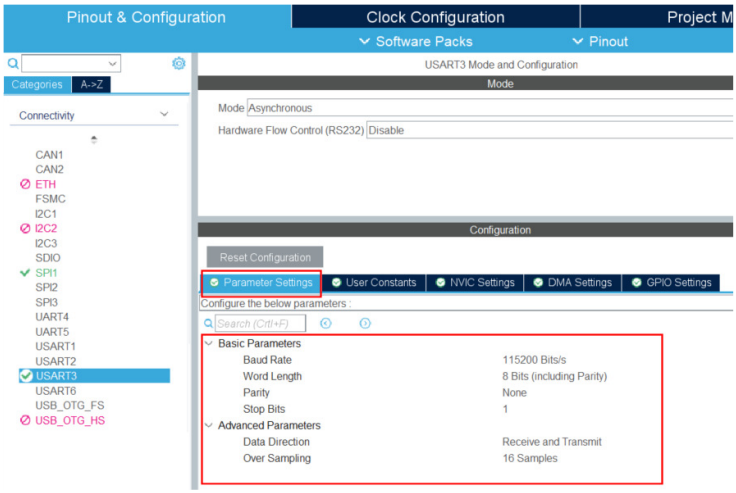
In *Configurator* we choose PD12 – PD15 as GPIO outputs and assign PB10, PB11 to USART3 (PB10 = USART3_TX, PB11 = USART3_RX):



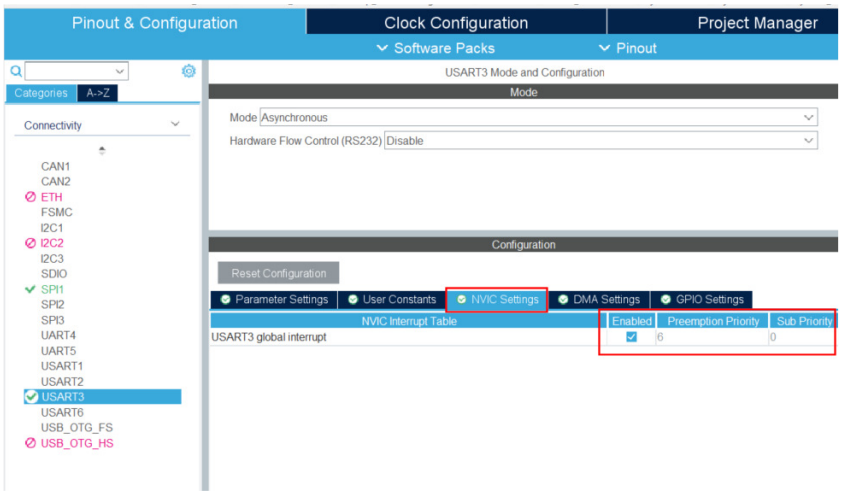
USART3 - *asynchronous communication, no handshaking*:



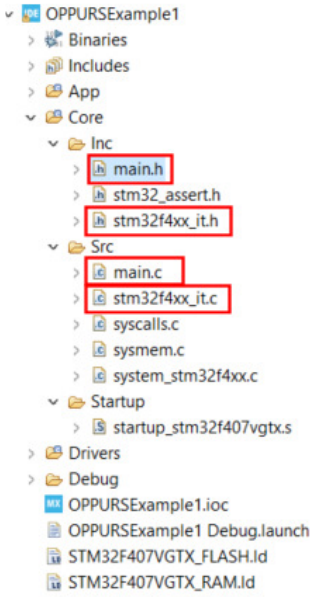
USART3 – *115200 bps, 8 bits, no parity, 1 stop bit*:



USART3 – *use USART3 interrupt (preemption priority 6, subpriority 0)*:



Code generated for two peripherals (GPIO and USART3) affects the following files[4]:



`Core/Inc/main.h`

```
/* Includes ------------------------------------------------------------------*/
#include "stm32f4xx_ll_usart.h"
#include "stm32f4xx_ll_gpio.h"
```

Note: *main.h* includes LL drivers for USART and GPIO

`Core/Inc/stm32f4xx_it.h`

```
void USART3_IRQHandler(void);
```

Note: USART3 ISR definition is places in header file (in this example we do not use GPIO interrupts).

[4] Only portions of the files relevant for autogenerated code analysis will be extracted from each file

`Core/Inc/stm32f4xx_it.c`

```
void USART3_IRQHandler(void)
{
  /* USER CODE BEGIN USART3_IRQn 0 */
  USART3_IRQHandler_Callback();
  /* USER CODE END USART3_IRQn 0 */
  /* USER CODE BEGIN USART3_IRQn 1 */

  /* USER CODE END USART3_IRQn 1 */
}
```

Note: User code must be placed between USER CODE BEGIN and USER CODE END marks in comments. Code placed between these marks will not be affected by code auto generator but anything placed outside of these guards will be deleted next time the IOC file refreshes autogenerated code. In this example we only place a single function call to the external callback handler *USART3_IRQHandler_Callback()* which is defined in *App* source tree. External definition using `extern` keyword may be needed if *linker* cannot locate function with that name in the build (although in STM32CubeMX build configuration it works without need to use `extern`).

`Core/Inc/main.c`

Here goes the most of the generated code (**note**: only parts related to the system clock, GPIO and USART3 are shown in this listing):

```
#include "main.h"

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART3_UART_Init(void);

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
extern main_app(void);
/* USER CODE END 0 */

int main(void)
{
  /* USER CODE BEGIN 1 */
  main_app();
  while(1);
  /* USER CODE END 1 */

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

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */

  LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);
  LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR);

  NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);

  /* System interrupt init*/
  /* SysTick_IRQn interrupt configuration */
  NVIC_SetPriority(SysTick_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),15, 0));

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

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART3_UART_Init();
  while (1)
  {
  }
}
```

init functions for each peripheral using LL driver

prevent entering autogenerated code – jump directly to user-defined *main_app()* entry

this code should be incorporated in *App* source tree manually in strctured way

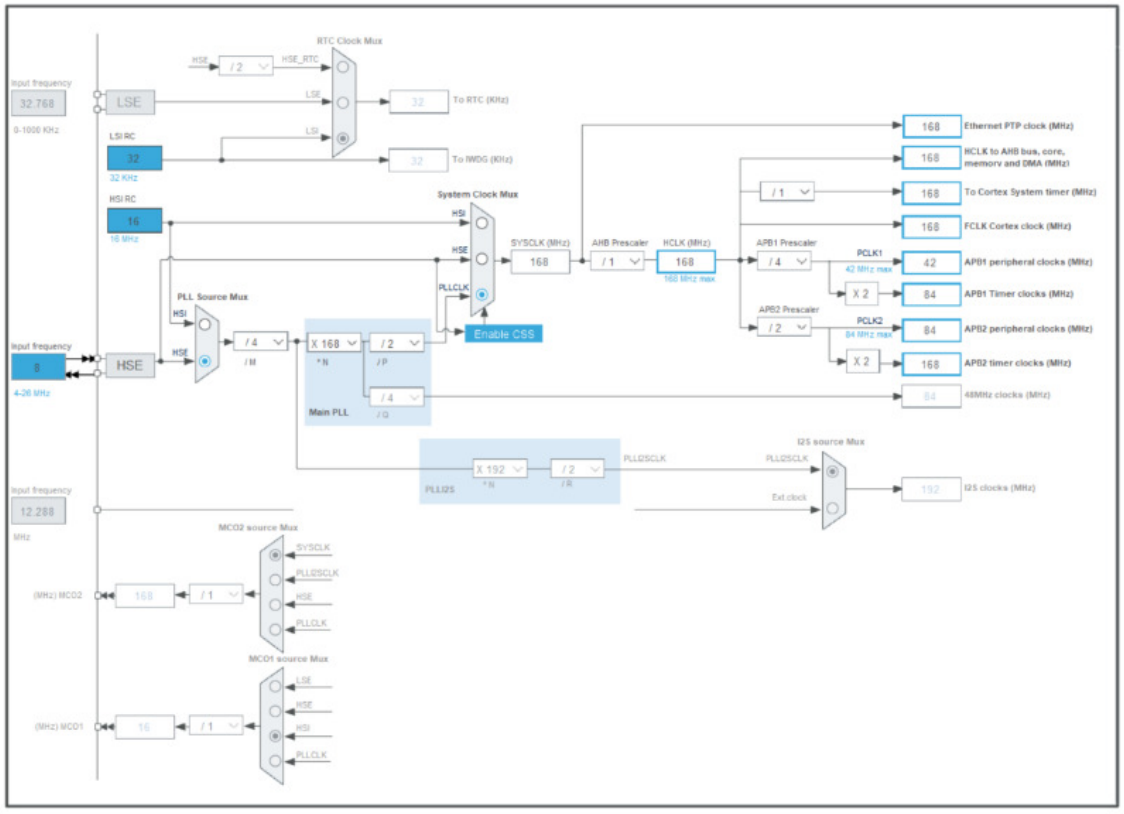
enable clock for essential system peripherals

NVIC setup: priority group setting; SYSTICK mandatory **only** if HAL is used! (not necessary for LL)

configure (1) system clock and (2) peripherals (GPIO, USART3)

this is autogenerated code for clock configuration according to settings shown in the figure below; it is recommended to copy this code in *App* source tree and update it manually each time global clock configuration is altered (usually it is defined only once and does not change during project development lifetime)

```
void SystemClock_Config(void)
{
  LL_FLASH_SetLatency(LL_FLASH_LATENCY_5);
  while(LL_FLASH_GetLatency()!= LL_FLASH_LATENCY_5)
  {
  }
  LL_PWR_SetRegulVoltageScaling(LL_PWR_REGU_VOLTAGE_SCALE1);
  LL_RCC_HSE_Enable();

   /* Wait till HSE is ready */
  while(LL_RCC_HSE_IsReady() != 1)
  {

  }
  LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_HSE, LL_RCC_PLLM_DIV_4, 168, LL_RCC_PLLP_DIV_2);
  LL_RCC_PLL_Enable();

   /* Wait till PLL is ready */
  while(LL_RCC_PLL_IsReady() != 1)
  {

  }
  LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);
  LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_4);
  LL_RCC_SetAPB2Prescaler(LL_RCC_APB2_DIV_2);
  LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);

   /* Wait till System clock is ready */
  while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL);
  LL_Init1msTick(168000000);
  LL_SetSystemCoreClock(168000000);
}
```



```
static void MX_USART3_UART_Init(void)
{

  LL_USART_InitTypeDef USART_InitStruct = {0};
  LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* Peripheral clock enable */
  LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_USART3);
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);
  /**USART3 GPIO Configuration
  PB10   ------> USART3_TX
  PB11   ------> USART3_RX
  */
  GPIO_InitStruct.Pin = LL_GPIO_PIN_10|LL_GPIO_PIN_11;
  GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
  GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
  GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
  GPIO_InitStruct.Alternate = LL_GPIO_AF_7;
  LL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* USART3 interrupt Init */
  NVIC_SetPriority(USART3_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),6, 0));
  NVIC_EnableIRQ(USART3_IRQn);

  USART_InitStruct.BaudRate = 115200;
  USART_InitStruct.DataWidth = LL_USART_DATAWIDTH_8B;
  USART_InitStruct.StopBits = LL_USART_STOPBITS_1;
  USART_InitStruct.Parity = LL_USART_PARITY_NONE;
  USART_InitStruct.TransferDirection = LL_USART_DIRECTION_TX_RX;
  USART_InitStruct.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
  USART_InitStruct.OverSampling = LL_USART_OVERSAMPLING_16;
  LL_USART_Init(USART3, &USART_InitStruct);
  LL_USART_ConfigAsyncMode(USART3);
  LL_USART_Enable(USART3);
}

static void MX_GPIO_Init(void)
{
  LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD);

  /**/
  LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_12|LL_GPIO_PIN_13|LL_GPIO_PIN_14|LL_GPIO_PIN_15);

  /**/
  GPIO_InitStruct.Pin = LL_GPIO_PIN_12|LL_GPIO_PIN_13|LL_GPIO_PIN_14|LL_GPIO_PIN_15;
  GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
  GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_LOW;
  GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
  GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
  LL_GPIO_Init(GPIOD, &GPIO_InitStruct);

}
```

enable clock to peripherals (GPIOB on AHB1 bus, USART3 on APB1 bus)

set up GPIO pins to alternate function (USART)

enable USART interrupt (**note**: this autogenerated code IS NOT COMPLETE since the conditions under which USART3 ISR is fired are not defined – e.g. received character, sent character etc.)

set USART3 communication parameters and enable USART3 peripheral

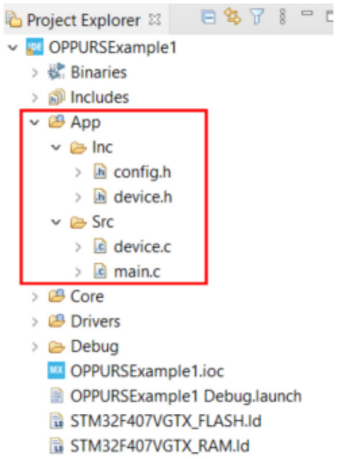
enable clock to GPIO port with LEDs

set GPIO pins as outputs

2.6.6 App source tree skeleton

As it was noted before, we need to place application code under separate folder to decouple the application and custom device driver code from the autogenerated code.

An example of minimum content of a source tree:



config.h

```
#ifndef CONFIG_H
#define CONFIG_H

#endif
```

device.h

```
#ifndef DEVICE_H
#define DEVICE_H

#include <config.h>

#endif
```

main.h

```
#include <config.h>

void main_app()
{
   while(1);
}
```

If at this point you try to build the project, it should succeed if everything is set up properly.

Example source code: archive with complete code example after this step is provided in archive:

OPPURS-Example1-AddingAppFolder.rar

However, now we need to populate the empty files under *App* source tree with useful autogenerated code, mostly from `main.c`. The next chapter will focus only on structuring the code under the *App* folder, based on autogenerated code, but structured in an extensible and scalable manner, without restrictions on format imposed by the auto generator, what is necessary to properly rebuild the code each time we make changes in the *Configurator* GUI.

3 Building the first application with custom developed HAL (app1)

Well structured application should completely separate application part from the autogenerated code. In this chapter we shall see how to make this separation from autogenerated code on a sample application, using STM32CubeIDE as development environment. Although the presented details are related to STM32F4x microcontroller family, STM32CubeIDE and STM HAL/LL libraries, the concepts can be easily adapted in general, for any microcontroller family, development environment and HAL libraries.

The reference code for the example that will be described in this chapter can be found in archive: **OPPURS-Example1-LED_USART.rar**

The chapter will be organized as follows:

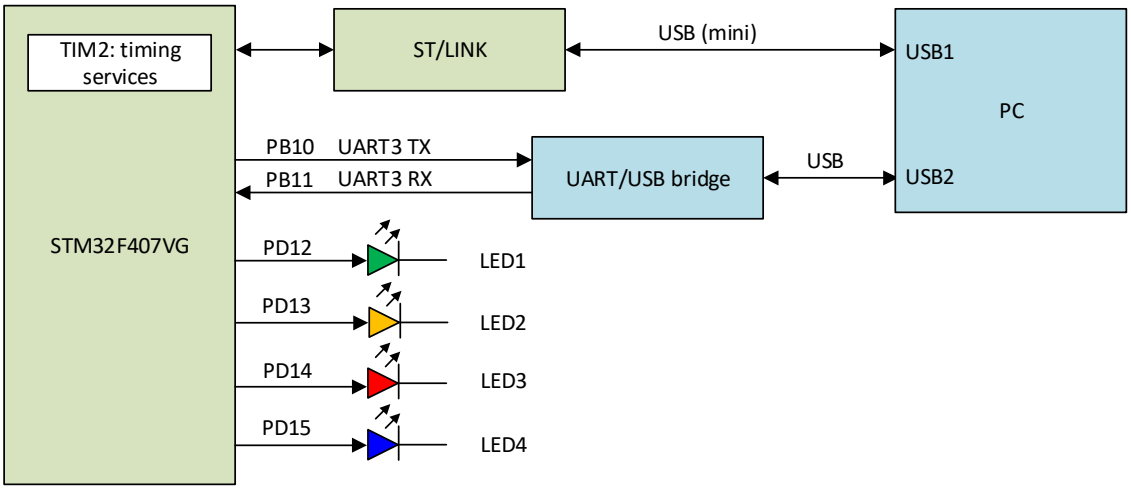
- specification of the sample application functionality and peripheral resources it uses
- description how to organize source files
- description of each custom HAL device driver (GPIO, timer, USART)
    - each device driver will be explained in details – initialization, setting up interrupts etc.
    - simple and effective ways how to implement critical sections by means of interrupt disable/enable will be presented
    - the concept of using queues for relaxing hard real-time requirements will be shown on the example of USART driver (although it will not be needed for this simple examples, full custom USART driver with TX/RX buffering support will be presented)
- description how to set up tests to verify each device driver peripheral (in isolation to other peripherals)
- description how to use custom HAL to develop final application, in accordance with the specification

3.1 Sample application functionality specification

Final application is simple and it has to provide the following functionality:

- upon reset initialize the peripherals:
    - on-board LED1-LED4 GPIOs as outputs
    - USART3 port for bidirectional serial communication (115200, 8, N, 1)
    - set up timer service with 1 ms timer resolution for generating delays, measuring elapsed time etc. (using TIM2)
- set up initial interval for toggling LEDs to 100 ms
- listen continuously the serial port for incoming characters; valid characters are *'1'-'9'* – when a character is received, change LED toggling interval to *value x 100 ms* (e.g. for character *'4'* toggling interval must be set to 400 ms)
- serial port outputs the information about the current toggling interval, after each LED toggle in a format:
    - "*LED blinking interval: %d ms*"
- upon receiving the character:
    - if command is valid (received character in interval *'1'-'9'*):
        - send answer string *"OK"*
        - change LED toggling interval according to formula *(c – '0') x 100 ms*
    - if command is not valid (received character not in interval *'1'-'9'*):
        - send answer string *"Command not recognized!"*

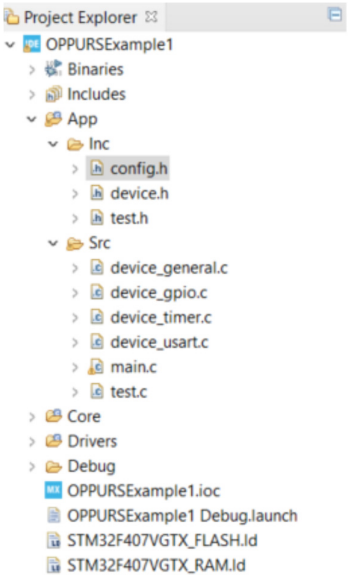
Block diagram of peripherals and system connection is shown in figure below:



3.2 Organization of source files

Organization of source files will follow the guidelines given in 2.6.6. However, instead of using a single *device.c* file for all drivers (what is the usual practice) we shall use multiple *device_<peripheral>.c* file, to separate and explain more easily the code related to the each peripheral device. Just like in the example given in 2.6.6., the whole application source code (device drivers, peripheral tests and the actual final application) will reside under *App* source tree, without dependencies to autogenerated code[5].

The figure below shows the organization of all files in this example under *App* source tree:



App folder is divided into *Inc* and *Src* subfolders, to separate headers and implementation files. This is recommended practice to keep application structure clean although one can decide to make more complex organization with multiple subfolders under *App* source tree, especially for very complex embedded applications.

[5] The only dependancies are: (1) the call from the *main()* function to *main_app()* and (2) callbacks in interrupt service handlers autogenerated by code generator.

The content of each file in the shown *App* source tree is as follows:

Include files:

- *config.h* – header file where all global configuration parameters should be maintained in a single place (i.e. all source files should include *config.h* to adjust build configuration according to the content of this file)
- *device.h* – header file with all function declarations of the custom HAL API
- *test.h* – header file with all function declarations for tests

Source files:

- *device_general.c* – device drivers related to common initialization procedures (e.g. setting up clock, global interrupt system initialization etc.); these routines contain hardware bootstrap functionality that is common to all use cases and necessary to initialize microcontroller before dealing with other peripherals
- *device_gpio.c* – device driver for GPIO (LEDs for now)
- *device_timer.c* – timer TIM2 based time-keeping services (useful in cases when RTOS with accompanying timing services is not used); timing services may provide useful functionality, such as blocking delays, non-blocking read of time stamps (e.g. for implementing non-blocking stopwatch functionality) etc.
- *device_usart.c* – high level driver for USART communication (blocking and non-blocking character send and receive, with both TX and RX buffering support, with configurable size user-defined FIFOs)
- *test.c* – implementation of all tests for testing each peripheral device driver separately to other peripherals
- *main.c* – main application file with *main_app()* custom application entry point; depending on the build configuration, either tests or user application may be executed upon entering the *main_app()* function

3.3 Device driver descriptions

3.3.1 Configuration header and main app file

Before going into description of each device driver, let us examine the content of *config.h* and *main.c*.

Config.h contains **all configuration** for the build in a source code in a **single place**. It is a good practice to:

- avoid hardcoding constants in the code – use *#defines* instead
- avoid scattering global *#defines* among many headers
- use *#defines* in a single configuration header, if possible
- use **include guards (for all header files!)**

config.h

```
#ifndef CONFIG_H
#define CONFIG_H

#define RUN_TESTS      1
#define RUN_APP        0

// FIFO buffers for USART3
#define USART3_TX_BUFFER_SIZE256
#define USART3_RX_BUFFER_SIZE256

#endif
```

include guards

build configuration selector

USART FIFO buffer sizes (could have been placed also in *device.h*)

The whole sample application functionality defined in 3.1 is shown in the listing of main.c :

main.c

```
#include <config.h>
#include <device.h>
#include <test.h>
```

include configuration, device drivers and test definitions

```
void run_test()
{
   // LED_User_Test();
   // LED_User_Test_Toggle();
   // LED_User_Test_WithTimer();
   // LED_User_Test_Toogle_WithTimer();
   // Timer2_Test();
   USART3_Test();
}
```

main test driver function from which we can call any test; in this case we just comment out the test we do not want to run in some build, although test selection could have been defined in *config.h* as well

```
void app1()
{
char c;
char msg[32];
int interval;

   interval = 500;
   while(1)
   {
      LED_User_Toggle(LED1);
      Timer2_WaitMillisec(interval);
      if (USART3_Dequeue(&c))
      {
         if (c >= '1' && c <= '9')
         {
            USART3_WriteLine("OK");
            interval = 100 * (c - '0'); // interval for LED blink 100-900 ms
         }
         else
         {
            USART3_WriteLine("Command not recognized!");
         }
      }
      sprintf(msg, "LED blinking interval: %d ms", interval);
      USART3_WriteLine(msg);
   }
}
```

This is how final application should look like. It should not directly access any registers, only HAL functions which abstracts hardware access. For maximum flexibility, it is recommended to wrap manufacturer provided HAL (in case of STM - HAL or LL drivers) into custom HAL device drivers to have maximum control how they are implemented (i.e. blocking vs non-blocking transmit and receive, fine control over interrupts (triggering conditions, ISR implementation), buffering etc.). Although manufacturer provided HAL can provide simplicity and speed when simple application needs to be designed quickly, more complex application will benefit of user designed and customized HAL.

```
void main_app()
{
#if (RUN_TESTS)
   run_test();
#endif

#if (RUN_APP)
   InitDevice();
   app1();
#endif
}
```

custom user-defined main() function; depending on configuration in config.h, we shall run either tests or main application (app1()); if we run main application, we should call custom initializator for all used hardware, which is InitDevice() residing in device.c

3.3.2 Definition of custom HAL API (device.h)

As explained in the previous section, the goal is to develop *custom HAL library* in a way that is the most suitable for a concrete application. STMicroelectronics provided HAL libraries are not an optimal choice to be directly called out-of-the-box from *main.c* due to following:

- *STMCubeMX LL libraries* – LL libraries provide very thin wrapper around register-oriented programming and do not provide high-level mechanisms which must be implemented by user; therefore, LL API calls should not be directly used in application code in *main.c* because they do not provide higher functional abstraction of peripheral access mechanisms in comparison to register-oriented programming; they just make register oriented programming more convenient for the programmer but still this approach needs to be wrapped into higher level API by a programmer, to make it suitable to be called from the application code in *main.c*
- *STMCubeMX HAL libraries* – HAL libraries provide high level of abstraction, just like a custom high level API that we are going to develop in this example; therefore, STM provided HAL library *may be a suitable choice* for calling directly from application code in *main.c*; however, using STM provided HAL library for that has two shortcomings:
    - STM HAL library hides the implementation details and makes it challenging for a programmer to change the way how HAL calls work in the background; these libraries should be used on „as-is“ principle, meaning if they satisfy the requirements of the application, they should be called as is, otherwise, it is better to develop own HAL library
    - from the educational point of view STM HAL is not a good choice since all important details are hidden in API function calls.

When designing own custom HAL, it is also a bad choice to base it on another high level API (STM HAL). Therefore, there are two options what to choose to develop own custom HAL:

- register programming directly
- STM LL-style HAL library, as a thin wrapper around direct register programming.

Although both approaches are valid, we shall use LL drivers to develop our own high level HAL library because it is more convenient and modern approach and we can benefit from the code that is autogenerated from GUI *Configurator* and placed in *Core/Src/main.c* function. Only minor benefit of using direct register programming approach may be speed but the gains are negligible compared to the effort of not taking a leverage of autogenerated code and structuring provided via STM LL libraries.

Any C module that wants to consume the custom developed API must include header files with definitions of constants, functions and structures. Header file *device.h* contains all definitions required for other modules to use HAL API. The structure of *device.h* header will be explained in details.

```
#ifndef DEVICE_H
#define DEVICE_H
```

always use include guards, in any user-defined header file

```
#include <config.h>
```

common definitions – useful for all parts of application

```
// *** copied from main.c BEGIN ***
// includes for LL framework (copied from main.h, generated by code generator)
#include "stm32f4xx_ll_adc.h"
#include "stm32f4xx_ll_rcc.h"
#include "stm32f4xx_ll_bus.h"
#include "stm32f4xx_ll_system.h"
#include "stm32f4xx_ll_exti.h"
#include "stm32f4xx_ll_cortex.h"
#include "stm32f4xx_ll_utils.h"
#include "stm32f4xx_ll_pwr.h"
#include "stm32f4xx_ll_dma.h"
#include "stm32f4xx_ll_spi.h"
#include "stm32f4xx_ll_tim.h"
#include "stm32f4xx_ll_usart.h"
#include "stm32f4xx_ll_gpio.h"
```

here we must include LL API headers for each used peripheral; note that there are more include files than just GPIO, TIM and USART – this is because hardware bootstraping code may use additional features (*note*: in this example more peripherals than shown in subsequent code was chosen in Configurator GUI so this list of headers was not particularly optimized for minimum subset needed by this example); this should be the only place in the whole application code where we include LL HAL support – all other C modules should get LL API support by including device.h header

```
#ifndef NVIC_PRIORITYGROUP_0
#define NVIC_PRIORITYGROUP_0         ((uint32_t)0x00000007) /*!< 0 bit  for pre-emption priority, 4
bits for subpriority */
#define NVIC_PRIORITYGROUP_1         ((uint32_t)0x00000006) /*!< 1 bit  for pre-emption priority, 3
bits for subpriority */
#define NVIC_PRIORITYGROUP_2         ((uint32_t)0x00000005) /*!< 2 bits for pre-emption priority, 2
bits for subpriority */
#define NVIC_PRIORITYGROUP_3         ((uint32_t)0x00000004) /*!< 3 bits for pre-emption priority, 1
bit  for subpriority */
#define NVIC_PRIORITYGROUP_4         ((uint32_t)0x00000003) /*!< 4 bits for pre-emption priority, 0
bit  for subpriority */
#endif
// *** copied from main.c END ***
```

definitions for easier choice of pre-emption priority group, needed for global interrupt system configuration

```
////////////////////////////////////
// system clock
////////////////////////////////////
void SystemClockConfig(void);
```

API functions for system clock setup – only configuration function is needed

```
////////////////////////////////////
// GPIO output (LED)
////////////////////////////////////
/*
PD12   LED4   GREEN  LED1
PD13   LED3   ORANGE LED2
PD14   LED5   RED    LED3
PD15   LED6   BLUE   LED4
*/
#define LED1    1
#define LED2    2
#define LED3    3
#define LED4    4
```

user defined constants for easier referencing of LEDs; when calling API function for LED control, we shall reference each LED by symbolic name, not e.g. by physical port pin or schematic symbol designator, we want to decouple LED naming from other sources and keep it completely within our code naming conventions

```
#define LED1_GPIO   GPIOD
#define LED1_PINLL_GPIO_PIN_12
#define LED2_GPIO   GPIOD
#define LED2_PINLL_GPIO_PIN_13
#define LED3_GPIO   GPIOD
#define LED3_PINLL_GPIO_PIN_14
#define LED4_GPIO   GPIOD
#define LED4_PINLL_GPIO_PIN_15
```

definitions relating LED connections on physical device; it is good practice to define all physical pin mappings in header file and **never** reference physical pins directly in C implementation files; this allows (1) single definition of pin mapping in header file and (2) changing the mapping (e.g. hardware revision changes) only in one single hardware related configuration file (device.h)

```
void LED_User_Init(void);
void LED_User_On(int LED_Id);
void LED_User_Off(int LED_Id);
void LED_User_Toggle(int LED_Id);
```

high level HAL API abstraction layer around LED control; this is what user in application code should call in order to control LEDs; application code should be completely decoupled from ANY hardware particularities or register access; notice that this approach is similar to STMCubeMX HAL-style approach in terms that it represents very high level abstraction of access to hardware resources, however, completely customized for specific application needs that is not usually provided by out-of-the-box HAL libraries such as STMCubeMX HAL (implementation file: *device_gpio.c*)

```
////////////////////////////////////
// TIM2 timer (manual timekeeping service)
////////////////////////////////////
void Timer2_Init(void);
void TIM2_IRQHandler_Callback(void);
void Timer2_Restart(void);
uint32_t Timer2_GetMillisec(void);
uint32_t Timer2_GetTickCount(void);
void Timer2_WaitMillisec(uint32_t ms);
uint32_t Timer_Elapsed_Millisec(uint32_t t1_ms, uint32_t t2_ms);

////////////////////////////////////
// USART3
////////////////////////////////////
void USART3_Init(void);                    // init USART3 for normal TX/RX operation
void USART3_IRQHandler_Callback(void);
void USART3_SendChar(char c);
void USART3_Flush(void);

void USART3_puts(char* msg);
void USART3_WriteLine(char* msg);

void USART3_Clear_TX_Buffer(void);
void USART3_Clear_RX_Buffer(void);
void USART3_Clear_Buffers(void);

int USART3_TX_InBufferCount(void);
int USART3_RX_InBufferCount(void);

void USART3_Enqueue_Blocking(char c);
int  USART3_Dequeue(char *c);

int USART3_ReadString(char* buf, int maxlen);

// non-blocking transmission
int USART3_Enqueue_WholeBuffer(char* buf, int length);
int USART3_Start_WholeBuffer_Transmit(void);

////////////////////////////////////
// master init
////////////////////////////////////
void InitDeviceCommon(void);
void InitDevice(void);

#endif
```

timer services via TIM2 (millisecond resolution, blocking delays, timekeeping capabilities etc.); these services make use of background processing based on custom ISR routine; detailed description of timer services will be provided in separate chapter (implementation file: *device_timer.c*)

advanced custom driver for USART communication, using interrupts for TX/RX events, FIFO buffering etc.; providing functions both for blocking and non-blocking transmission and reception of characters; this USART HAL API is very versatible, beyond requirements of specification for simple example and the same implementation will be used in more complex examples (implementation file: *device_usart.c*)

common init of most important features of microcontroller, needed to properly start and setup hardware (implementation file: *device_general.c*)

Although the whole API provided by a *device.h* could be implemented in a single *device.c* implementation file, the API design was divided into few files to make the whole API design easier to present and analyze. It is up to the programmer to decide what structure suits the best requirements of particular case.

Also note that all files (*device_general.c, device_gpio.c, device_timer.c, device_usart.c*) include only a single header *device.h*. It is a good practice to provide all necessary external includes within this single include file, to keep code cleaner and easier to maintain.

3.3.3 Common peripheral initialization (device_general.c)

device_general.c

```
#include <device.h>
```

single include directive with all necessary stuff!

```
////////////////////////////////////
// system clock
////////////////////////////////////
// *** function copied from code generator in main.c ***
// *** copied from main.c BEGIN ***
void SystemClockConfig()
{
   LL_FLASH_SetLatency(LL_FLASH_LATENCY_5);
   while(LL_FLASH_GetLatency()!= LL_FLASH_LATENCY_5)
   {
   }
   LL_PWR_SetRegulVoltageScaling(LL_PWR_REGU_VOLTAGE_SCALE1);
   LL_RCC_HSE_Enable();

   /* Wait till HSE is ready */
   while(LL_RCC_HSE_IsReady() != 1)
   {

   }
   LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_HSE, LL_RCC_PLLM_DIV_4, 168, LL_RCC_PLLP_DIV_2);
   LL_RCC_PLL_Enable();

   /* Wait till PLL is ready */
   while(LL_RCC_PLL_IsReady() != 1)
   {

   }
   LL_RCC_SetAHBPrescaler(LL_RCC_SYSCLK_DIV_1);
   LL_RCC_SetAPB1Prescaler(LL_RCC_APB1_DIV_4);
   LL_RCC_SetAPB2Prescaler(LL_RCC_APB2_DIV_2);
   LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);

   /* Wait till System clock is ready */
   while(LL_RCC_GetSysClkSource() != LL_RCC_SYS_CLKSOURCE_STATUS_PLL)
   {

   }
   LL_Init1msTick(168000000);
   LL_SetSystemCoreClock(168000000);
}
// *** copied from main.c END ***
```

initialization of system clock; copied from autogenerated code in main.c, based on selection in GUI of Configurator; usually no need for any additional manual intervention

```
////////////////////////////////////
// master init
////////////////////////////////////
void InitDeviceCommon()
{
   // copied from main() in main.c (generated by code generator):
   // *** copied from main.c BEGIN ***
   // MCU Configuration --------------------------------------------------------
   // Reset of all peripherals, Initializes the Flash interface and the Systick. */
   LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);
   LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_PWR);
   NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
   // *** copied from main.c END ***
   // init clock - copied from main.c
   SystemClockConfig();
}
```

initialization of common hardware resources (system registers, global interrupt priorities, system clock)

```
void InitDevice()
{
   // common startup init (from code generator)
   InitDeviceCommon();

   // custom device init
   LED_User_Init();
   Timer2_Init();
   USART3_Init();
}
```

master function for initialization of ALL hardware used by device; an example of *„separation of concerns“* pattern – main function must not worry how to initialize the peripherals and has to delegate this job to a single function call

initialization of common hardware resources (global interrupt system, clocl etc.); copied from autogenerated code in main for convenience; should be kept at minimum

initialization of each used peripheral; each peripheral should have its own initialization routine

3.3.4 GPIO device driver for LED control (device_gpio.c)

3.3.4.1 GPIO initializaton procedure in details

Let us first analyze the function *LED_User_Init()* in the module *device_gpio.c*:

device_gpio.c

```
#include <device.h>
```

we need to include stm32f4xx_ll_gpio.h – this is provided via device.h already

```
void LED_User_Init()
{
LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

   // Enable the GPIO LED Clock (PD => PD12, PD13, PD14, PD15! all LEDs)
   LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD);

   LL_GPIO_ResetOutputPin(LED1_GPIO, LED1_PIN);
   LL_GPIO_ResetOutputPin(LED2_GPIO, LED2_PIN);
   LL_GPIO_ResetOutputPin(LED3_GPIO, LED3_PIN);
   LL_GPIO_ResetOutputPin(LED4_GPIO, LED4_PIN);

   // LED1 GPIO output
   GPIO_InitStruct.Pin = LED1_PIN;
   GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
   GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_LOW;
   GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
   GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
   LL_GPIO_Init(LED1_GPIO, &GPIO_InitStruct);

   // LED2 GPIO output
   GPIO_InitStruct.Pin = LED2_PIN;
   LL_GPIO_Init(LED2_GPIO, &GPIO_InitStruct);

   // LED3 GPIO output
   GPIO_InitStruct.Pin = LED3_PIN;
   LL_GPIO_Init(LED3_GPIO, &GPIO_InitStruct);

   // LED4 GPIO output
   GPIO_InitStruct.Pin = LED4_PIN;
   LL_GPIO_Init(LED4_GPIO, &GPIO_InitStruct);
}
```

The function *LED_User_Init()* is the most important part of this device driver since it initializes the function of GPIO pins. All other device drivers will also have the single main initialization function that is called by the main initialization function for all peripherals. This function initializes GPIO pins connected to four on-board LEDs (push-pull outputs).

Now let us analyze how LL HAL accesses the processor registers behind the scenes. In order to decouple direct register access from high level logic, LL HAL relies on use of special *init structures*, which serves programmer to define on a logic level how to configure a peripheral, without a need to directly access registers and write values as described in datasheet, what would be the responsibility of LL layer.

In the case of GPIO pins, the init structure data type is *LL_GPIO_InitTypeDef*. If we click on *LL_GPIO_InitTypeDef* in the code of the function above (*Ctrl+Left Click* in STM32CubeIDE), we shall jump to the definition of the structure in *stm32f4xx_ll_gpio.h*:

```
typedef struct
{
  uint32_t Pin;           /*!< Specifies the GPIO pins to be configured.
                               This parameter can be any value of @ref GPIO_LL_EC_PIN */

  uint32_t Mode;          /*!< Specifies the operating mode for the selected pins.
                               This parameter can be a value of @ref GPIO_LL_EC_MODE.

                               GPIO HW configuration can be modified afterwards using unitary function
                               @ref LL_GPIO_SetPinMode().*/

  uint32_t Speed;         /*!< Specifies the speed for the selected pins.
                               This parameter can be a value of @ref GPIO_LL_EC_SPEED.

                              GPIO HW configuration can be modified afterwards using unitary function
                              @ref LL_GPIO_SetPinSpeed().*/

  uint32_t OutputType;    /*!< Specifies the operating output type for the selected pins.
                               This parameter can be a value of @ref GPIO_LL_EC_OUTPUT.

                               GPIO HW configuration can be modified afterwards using unitary function
                               @ref LL_GPIO_SetPinOutputType().*/

  uint32_t Pull;          /*!< Specifies the operating Pull-up/Pull down for the selected pins.
                               This parameter can be a value of @ref GPIO_LL_EC_PULL.

                               GPIO HW configuration can be modified afterwards using unitary function
                               @ref LL_GPIO_SetPinPull().*/

  uint32_t Alternate;     /*!< Specifies the Peripheral to be connected to the selected pins.
                               This parameter can be a value of @ref GPIO_LL_EC_AF.

                               GPIO HW configuration can be modified afterwards using unitary function
                               @ref LL_GPIO_SetAFPin_0_7() and LL_GPIO_SetAFPin_8_15().*/
} LL_GPIO_InitTypeDef;
```

Let us see how this mechanism works on an example of setting a GPIO pin as push-pull output.

First, let us examine STM32F4 family datasheet to see how to set up GPIO pin's output mode and type. On STM32F4DISCOVERY board, LED1 diode is connected to PD12 so we need to configure PD12 as a push-pull output.

Two registers are used to control GPIO output mode and type (GPIOxMODER and GPIOx_OTYPER):

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

8.4.2 GPIO port output type register (GPIOx_OTYPER) (x = A..I/J/K)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

Bits GPIOD_MODER[25..24] control *mode* for pin PD12 and can be either:

- GPIO digital input, GPIO digital output, alternate function (controlled by peripheral, not GPIO), analog (ADC input, if applicable)
- note that reset value is *digital input*

Bit GPIOD_OTYPER[12] controls *output type* for pin PD12 and can be either:

- push-pull or open-drain
- note that reset value is *push-pull*

If we want to set PD12 as a push-pull output we need to do the following:

Set GPIOD_MODER[25..24] = '01' (via bit masking, to avoid changing values of other bits):

```
GPIOD_MODER
xxxx xx01 xxxx xxxx xxxx xxxx xxxx xxxx

GPIOD_MODER &= 1111 1101 1111 1111 1111 1111 1111 1111
GPIOD_MODER &= 0xFDFF FFFF

GPIOD_MODER |= 0000 0001 0000 0000 0000 0000 0000 0000
GPIOD_MODER |= 0x01000000
```

Set GPIOD_OTYPER[12] = '0':

```
GPIOD_OTYPER
xxxx xxxx xxxx xxxx xxx0 xxxx xxxx xxxx

GPIOD_OTYPER &= 1111 1111 1111 1111 1110 1111 1111 1111
GPIOD_OTYPER &= 0xFFFF EFFF
```

If a direct register access was used instead of LL API, the operation would be coded like this:

```
GPIOD->MODER &= 0xfdffffff;
GPIOD->MODER |= 0x01000000;
GPIOD->OTYPER &= 0xffffefff;
```

This approach results in very compact and fast code. However, this code is less readable, portable and extensible compared to LL API approach.

To achive the same goal with LL API, we shall first populate desired values in *GPIO_InitStruct* structure:

```
// LED1 GPIO output
GPIO_InitStruct.Pin = LED1_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_LOW;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
LL_GPIO_Init(LED1_GPIO, &GPIO_InitStruct);
```

Note how definitions in *device.h* help to avoid hardcoding values about port number and pin - *device.h* have definitions:

```
#define LED1_GPIO   GPIOD
#define LED1_PIN    LL_GPIO_PIN_12
```

will expand in code as:

```
GPIO_InitStruct.Pin = LL_GPIO_PIN_12;
...
LL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

If we focus only on the part related to setting the port pin mode and output type:

```
// LED1 GPIO output
GPIO_InitStruct.Pin = LED1_PIN;
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_LOW;
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
LL_GPIO_Init(LED1_GPIO, &GPIO_InitStruct);
```

we see that these to GPIO pin properties are defined as fields in *LL_GPIO_InitTypeDef*, namely fields *Mode* and *OutputType*.

Comments in source code for *Mode* field say that the valid values for this fields are defined by GPIO_LL_EC_MODE – if we search for GPIO_LL_EC_MODE in *stm32f4xx_ll_gpio.h* header we can see the list of valid values:

```
/** @defgroup GPIO_LL_EC_MODE Mode
  * @{
  */
#define LL_GPIO_MODE_INPUT                (0x00000000U) /*!< Select input mode */
#define LL_GPIO_MODE_OUTPUT               GPIO_MODER_MODER0_0  /*!< Select output mode */
#define LL_GPIO_MODE_ALTERNATE            GPIO_MODER_MODER0_1  /*!< Select alternate function mode */
#define LL_GPIO_MODE_ANALOG               GPIO_MODER_MODER0    /*!< Select analog mode */
```

So we choose to set the value for the mode:

```
GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
```

Constant LL_GPIO_MODE_OUTPUT is defined as:

```
#define GPIO_MODER_MODER0_0             (0x1UL << GPIO_MODER_MODER0_Pos)       /*!< 0x00000001 */
```

This definition encodes that we need to apply '01' binary value to GPIOx_MODER, taking into account that LL API will take care to shift it left for appropriate number of positions (12 x 2 = 24 in case of pin PD12).

Comments in source code for *Output* field say that the valid values for this fields are defined by GPIO_LL_EC_OUTPUT – if we search for GPIO_LL_EC_OUTPUT in *stm32f4xx_ll_gpio.h* header we can see the list of valid values:

```
/** @defgroup GPIO_LL_EC_OUTPUT Output Type
  * @{
  */
#define LL_GPIO_OUTPUT_PUSHPULL           (0x00000000U) /*!< Select push-pull as output type */
#define LL_GPIO_OUTPUT_OPENDRAIN          GPIO_OTYPER_OT_0 /*!< Select open-drain as output type */
```

Although reset value is already '0' for push pull, we should not assume at any point that values after program started are in their reset state.

For the *output type* we choose:

```
GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
```

Now we call *LL_GPIO_Init()* function from *stm32f4xx_ll_gpio.c* which takes two parameters:

- *GPIO_TypeDef* – pointer to the structure that contains a map of peripheral registers where we can directly write to; this is the structure which can be used for direct register programming
- *LL_GPIO_InitTypeDef* – the structure that contains high level definitions how to initialize the peripheral

While *LL_GPIO_InitTypeDef* is a high level type that helps LL API to decouple logic from direct values to be written into registers, *GPIO_TypeDef* data type is much closer to peripheral register definitions:

```
typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;     /*!< GPIO port bit set/reset register,      Address offset: 0x18      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

The same structure provides memory map to directly access all registers available for some peripheral, in accordance to the description in datasheet. When we make a call

```
LL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

the first parameter is defined in *stm32f407xx.h*:

```
#define GPIOD               ((GPIO_TypeDef *) GPIOD_BASE)
```

which is the pointer to actual memory location in peripheral memory space corresponding to GPIO, Port D. We can even reveal the absolute address where typed struct pointer points to, by examining *stm32f407xx.h*:

```
#define PERIPH_BASE           0x40000000UL /*!< Peripheral base address in the alias region
#define AHB1PERIPH_BASE       (PERIPH_BASE + 0x00020000UL)
#define GPIOD_BASE            (AHB1PERIPH_BASE + 0x0C00UL)
```

Let us analyze how *LL_GPIO_Init* function uses the information provided via *LL_GPIO_InitTypeDef* structure for setting up output mode and type:

```
ErrorStatus LL_GPIO_Init(GPIO_TypeDef *GPIOx, LL_GPIO_InitTypeDef *GPIO_InitStruct)
{
  uint32_t pinpos     = 0x00000000U;
  uint32_t currentpin = 0x00000000U;

  /* Check the parameters */
  assert_param(IS_GPIO_ALL_INSTANCE(GPIOx));
  assert_param(IS_LL_GPIO_PIN(GPIO_InitStruct->Pin));
  assert_param(IS_LL_GPIO_MODE(GPIO_InitStruct->Mode));
  assert_param(IS_LL_GPIO_PULL(GPIO_InitStruct->Pull));

  /* ------------------------ Configure the port pins ---------------- */
  /* Initialize  pinpos on first pin set */
  pinpos = POSITION_VAL(GPIO_InitStruct->Pin);

  /* Configure the port pins */
```

```
  while (((GPIO_InitStruct->Pin) >> pinpos) != 0x00000000U)
  {
    /* Get current io position */
    currentpin = (GPIO_InitStruct->Pin) & (0x00000001U << pinpos);

    if (currentpin)
    {

      if ((GPIO_InitStruct->Mode == LL_GPIO_MODE_OUTPUT) ||
          (GPIO_InitStruct->Mode == LL_GPIO_MODE_ALTERNATE))
      {
        /* Check Speed mode parameters */
        assert_param(IS_LL_GPIO_SPEED(GPIO_InitStruct->Speed));

        /* Speed mode configuration */
        LL_GPIO_SetPinSpeed(GPIOx, currentpin, GPIO_InitStruct->Speed);

        /* Check Output mode parameters */
        assert_param(IS_LL_GPIO_OUTPUT_TYPE(GPIO_InitStruct->OutputType));

        /* Output mode configuration*/
        LL_GPIO_SetPinOutputType(GPIOx, currentpin, GPIO_InitStruct->OutputType);
      }

      /* Pull-up Pull down resistor configuration*/
      LL_GPIO_SetPinPull(GPIOx, currentpin, GPIO_InitStruct->Pull);

      if (GPIO_InitStruct->Mode == LL_GPIO_MODE_ALTERNATE)
      {
        /* Check Alternate parameter */
        assert_param(IS_LL_GPIO_ALTERNATE(GPIO_InitStruct->Alternate));

        /* Speed mode configuration */
        if (POSITION_VAL(currentpin) < 0x00000008U)
        {
          LL_GPIO_SetAFPin_0_7(GPIOx, currentpin, GPIO_InitStruct->Alternate);
        }
        else
        {
          LL_GPIO_SetAFPin_8_15(GPIOx, currentpin, GPIO_InitStruct->Alternate);
        }
      }

      /* Pin Mode configuration */
      LL_GPIO_SetPinMode(GPIOx, currentpin, GPIO_InitStruct->Mode);
    }
    pinpos++;
  }

  return (SUCCESS);
}
```

For example, simple register level write to GPIOD->MODER in hardcoded way shown above will be handled by LL API in parametric and configurable manner by calling the LL API function *LL_GPIO_SetPinOutputType()*:

```
__STATIC_INLINE void LL_GPIO_SetPinMode(GPIO_TypeDef *GPIOx, uint32_t Pin, uint32_t Mode)
{
  MODIFY_REG(GPIOx->MODER, (GPIO_MODER_MODER0 << (POSITION_VAL(Pin) * 2U)), (Mode <<
(POSITION_VAL(Pin) * 2U)));
}
```

effectively achieving the same result. Please take a look into LL API source files to explore other macros, structures and comments for better understanding of the whole process.

This example how LL API handles writing into a register was presented into details for case of setting the output mode and type for GPIO pin and the principles apply to any other peripheral. In the rest of these guidelines we shall not elaborate details for other peripherals and functions.

We can also see that LL API introduces some overhead, both in code size and speed, when compared to hardcoded direct register programming. However, in practice the difference between direct register programming and LL API approach would be barely noticeable, especially considering this is done only once, during the initialization on a system startup. The gain of well structured and readable code, as well as the code auto generation capabilities by IDE, justify the use of LL approach over direct register programming. However, one must understand how the register level programming works in principle and how LL API uses register level access in the background in order to properly use LL API. Moreover, in some cases, when performance is important, some portions of hardware access code may be coded in a register-level access manner, mixed with rest of the code programmed by use of LL API – both approaches may be used at the same time, if necessary, what shall be demonstrated in some examples later.

3.3.4.2 GPIO high level functionality API

The rest of the *device_gpio.c* device driver contains functions that wrap high level functionality to be called from the application code and hide low-level implementation details, while using LL API in the background to provide the desired functionality.

```
void LED_User_On(int LED_Id)                                   turn ON LED (parameter is logical LED id)
{
   switch(LED_Id)
   {
      case LED1:                                  LL API function to set GPIO to '1'
         LL_GPIO_SetOutputPin(LED1_GPIO, LED1_PIN);          we never use hardcoded values in C
         break;                                              implementation files! actual pins must be
      case LED2:                                             defined only at one place in header (config.h
         LL_GPIO_SetOutputPin(LED2_GPIO, LED2_PIN);          or device.h, better in device.h since it
         break;                                              describes particular custom hardware for
      case LED3:                                             which we develop custom HAL)
         LL_GPIO_SetOutputPin(LED3_GPIO, LED3_PIN);
         break;
      case LED4:
         LL_GPIO_SetOutputPin(LED4_GPIO, LED4_PIN);
         break;
   }
}

void LED_User_Off(int LED_Id)                                  turn OFF LED (parameter is logical LED id)
{
   switch(LED_Id)
   {
      case LED1:                                  LL API function to set GPIO to '0'
         LL_GPIO_ResetOutputPin(LED1_GPIO, LED1_PIN);
         break;
      case LED2:
         LL_GPIO_ResetOutputPin(LED2_GPIO, LED2_PIN);
         break;
      case LED3:
         LL_GPIO_ResetOutputPin(LED3_GPIO, LED3_PIN);
         break;
      case LED4:
         LL_GPIO_ResetOutputPin(LED4_GPIO, LED4_PIN);
         break;
   }
}

void LED_User_Toggle(int LED_Id)                               toggle LED state (parameter is logical LED id)
{
   switch(LED_Id)
   {
      case LED1:                                  LL API function to toggle GPIO state ('1' => '0' or '0' => '1')
         LL_GPIO_TogglePin(LED1_GPIO, LED1_PIN);
         break;
      case LED2:
         LL_GPIO_TogglePin(LED2_GPIO, LED2_PIN);
         break;
```

```
        case LED3:
            LL_GPIO_TogglePin(LED3_GPIO, LED3_PIN);
            break;
        case LED4:
            LL_GPIO_TogglePin(LED4_GPIO, LED4_PIN);
            break;
    }
}
```

3.3.4.3 GPIO HAL driver test routines

It is good programming practice to develop test suite along with the code development process in order to be able to test new functionalities or test that the rest of the codebase was not negatively affected by introducing some new code and features. Ideally, test-driven development (TDD) approach would impose a requirement to make the whole (or at least majority) of code *testable*, meaning that incremental changes in code will by followed by accompanying tests that can automatically verify that the developed code (old and new) complies with the requirements, which are programmed in tests.

Although TDD approach provides sustainable means for building robust and scalable applications, this approach is still not widely implemented in embedded development, due to some embedded development particularities (especially related to testing the code on a real hardware) what makes full TDD approach in embedded development a challenging task, unlike for purely software environments. However, tests still play important role in embedded development, even if TDD approach is not fully implemented.

Here we shall demonstrate an example of a minimal (pragmatic) test framework which serves to verify whether the device driver was properly developed. The testing approach that will be elaborated is the simplest way how to prepare and implement device driver tests although they could have been made even more detailed, structured and automated by using specialized frameworks.

In our example we have two files that implement all tests:

- *test.h* – header with definition of all tests
- *test.c* – implementation of all tests

Let us examine *test.h* with definition of all tests:

```
#ifndef DEVICE_TEST_H
#define DEVICE_TEST_H

#include <config.h>
#include <device.h>

void LED_User_Test(void);
void LED_User_Test_Toggle(void);
void Timer2_Test(void);
void LED_User_Test_WithTimer(void);
void LED_User_Test_Toogle_WithTimer(void);
void USART3_Test(void);

#endif
```

The functions above provide means to test GPIO driver (via LEDs), timer services (TIM2) and USART3 device driver.

Part of *test.c* that contains tests for GPIO (LED) only is given below:

```
#include <test.h>
#include <string.h>

void LED_User_Test()
{
int i;

    InitDeviceCommon();
    LED_User_Init();
```

test LED on and off functionality

minimum common hardware initializations

initialize ONLY GPIO (LED); we want to keep tests **simple** and **isolated** (use as less other peripherals as possible); if no else peripherals were tested then we don't want to introduce bugs from untested peripherals

```
   while(1)
   {
      for(int i = 0; i < 16000000; i++);
      LED_User_On(LED1);
      LED_User_On(LED2);
      LED_User_Off(LED3);
      LED_User_Off(LED4);
      for(int i = 0; i < 16000000; i++);
      LED_User_Off(LED1);
      LED_User_Off(LED2);
      LED_User_On(LED3);
      LED_User_On(LED4);
   }
}
```

quick and dirty way to generate pause but without precise timing control; later we shall show an example how to make a pause with precise timing, once we have timer services developed and tested

blink all LEDs with chosen pattern

```
void LED_User_Test_Toggle()
{
int i;

   InitDeviceCommon();
   LED_User_Init();

   while(1)
   {
      for(int i = 0; i < 16000000; i++);
      LED_User_Toggle(LED1);
      LED_User_Toggle(LED2);
      LED_User_Toggle(LED3);
      LED_User_Toggle(LED4);
   }
}
```

test LED toggle functionality; very similar test to the previous one but with toggling LEDs instead of explicitely turning them on and off

How tests are executed?

In *config.h* we need to set RUN_TESTS define to trigger appropriate conditional compilation in *main.c*:

```
#define RUN_TESTS      1
#define RUN_APP        0
```

In *main.c* we manually choose which test to run[6]:

```
void run_test()
{
   LED_User_Test();
   // LED_User_Test_Toggle();
   // LED_User_Test_WithTimer();
   // LED_User_Test_Toggle_WithTimer();
   // Timer2_Test();
   // USART3_Test();
}

void main_app()
{
#if (RUN_TESTS)
   run_test();
#endif

#if (RUN_APP)
   InitDevice();
   app1();
#endif
}
```

[6] Although automated TDD environments work differently, i.e. they provide means to run all tests in a test suite, this approach is simple and practical way which is still useful for less complex development scenarios

3.3.5 Timer services via TIM2 (device_timer.c)

3.3.5.1 TIM2 timer services API description

Custom HAL API timer services functions are declared in *device.h*:

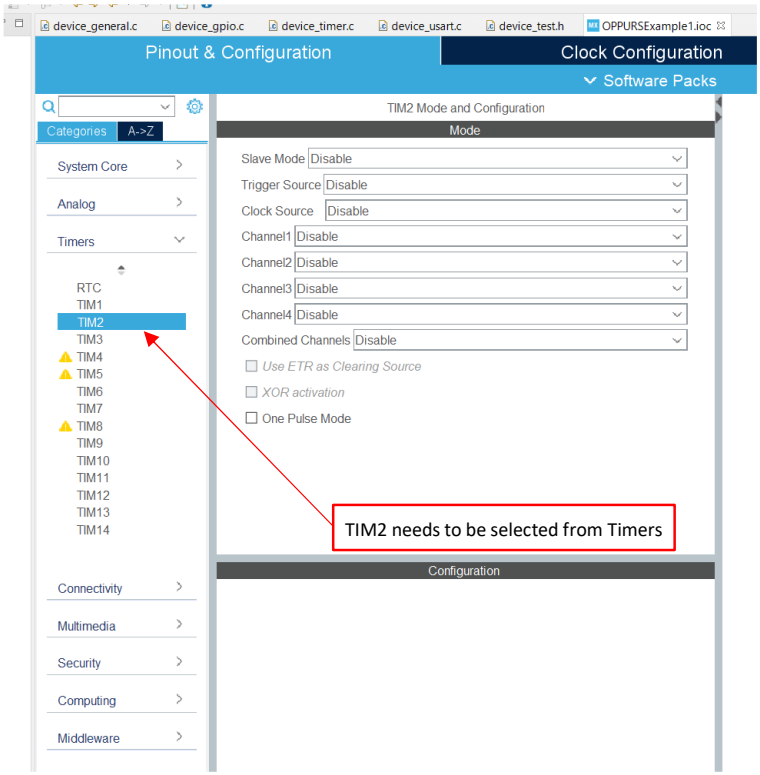
```
void Timer2_Init(void);
void TIM2_IRQHandler_Callback(void);
void Timer2_Restart(void);
uint32_t Timer2_GetMillisec(void);
uint32_t Timer2_GetTickCount(void);
void Timer2_WaitMillisec(uint32_t ms);
uint32_t Timer_Elapsed_Millisec(uint32_t t1_ms, uint32_t t2_ms);
```

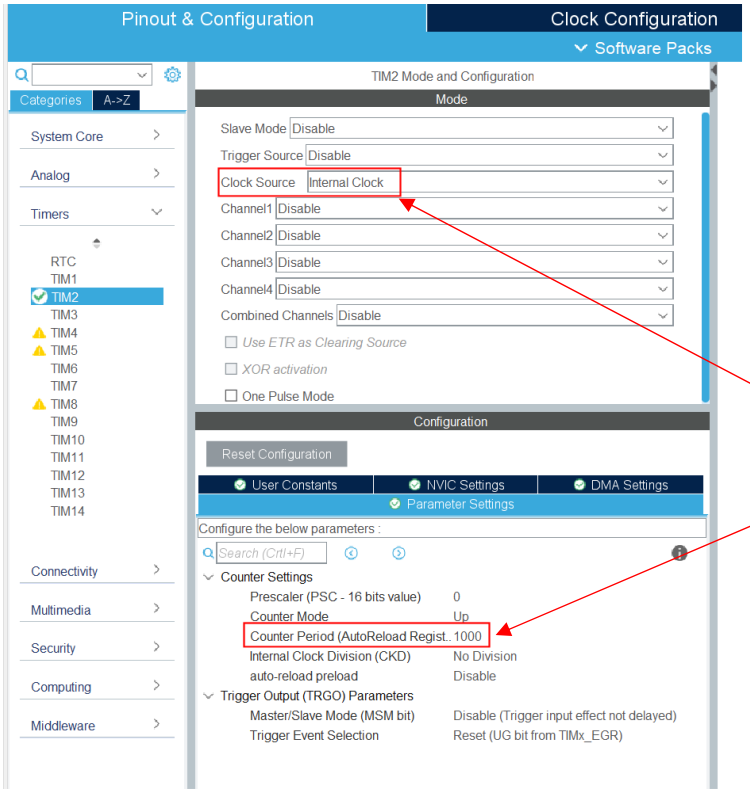
Short description what each of timer service API functions does:

- *Timer2_Init* – initializes the timer service (hardware initialization of TIM2 peripheral, interrupts, global shared variables etc.); initialize *internal counter* to zero (counter is incremented on each timer overflow in timer interrupt)
- *Timer2_Restart* – restart timer and reset internal counter
- *Timer2_GetMillisec – non-blocking* call to read number of milliseconds (*timestamp*) elapsed since the timer service (re)start (automatically calculated from internal counter, in case that chosen timer overflow period differs from 1 ms);
- *Timer2_GetTickCount – non-blocking* call to read the value of internal counter (elapsed since the timer service (re)start)
- *Timer2_WaitMillisec – blocking* call which blocks the current thread for the chosen number of *milliseconds*; main thread waits inside an idle loop, waiting for a desired pause to elapse
- *Timer_Elapsed_Millisec* – calculates time elapsed between to timestamps

3.3.5.2 Configuring TIM2 via Configurator GUI

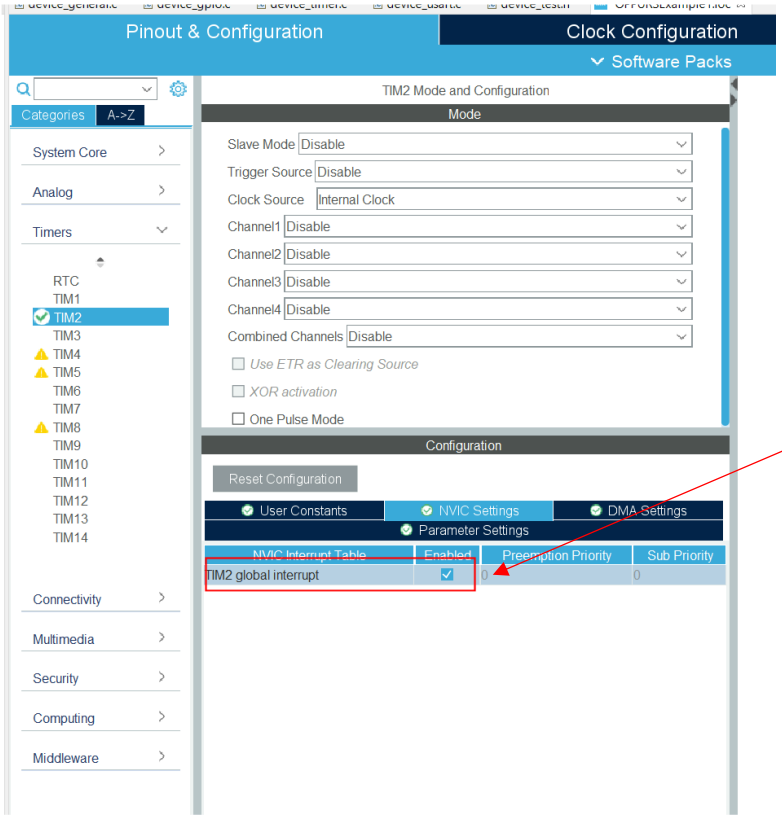
Before going into detailed description of TIM2 device driver and related timer services, let us see how to configure TIM2 in Configurator GUI:



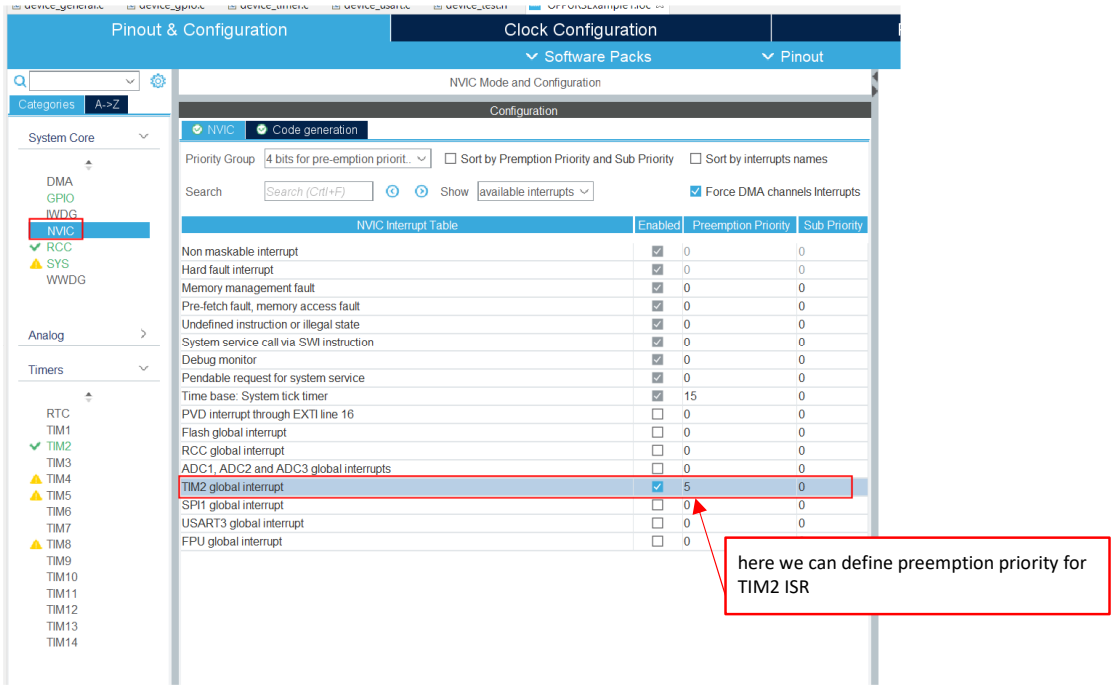


choose internal clock

period is selected here; however, this represents raw integer value which does not automatically sets overflow period, if TIM2 clock is not known; it is necessary to initialize this value manually in code, as it will be shown later in code



enable TIM2 interrupt; this will put appropriate ISR *stm32f4xx_it.c*:; ISR will call callback function that is part of our custom driver



NOTE: although the figures above show all the places where some parts of TIM2 configuration can be chosen from GUI, it is not necessary to worry about to make final correct configuration of TIM2 peripheral from *Configurator* GUI. It is recommended to define some of the most important settings (e.g. enabling TIM2, marking that we want to generate TIM2 ISR in *stm32f4xx_it.c*, mark that we use LL (instead of HAL drivers) to include them in *Core/Src* etc.). This process will help the project to automatically include all necessary files related to LL API and provide template for ISR in *Core/Src* source tree. During the process of manual coding of timer device driver in *device_timer.c* all autogenerated code from *main.c* may be used as a reference but it is still recommended to edit and adjust it manually, with necessary details to have a tailored made device driver in accordance to specific application requirements (e.g. time stamping etc.). Students are encouraged to study the content of the accompanying source code archive that contains autogenerated code for TIM2 in *main.c* (this code will not be analyzed here) and how it is used to create the final, manually developed TIM2 device driver code in *device_timer.c*.

3.3.5.3 TIM2 timer services source code description

The whole listing of TIM2 device driver and timer service coded in *device_timer.c*, along with additional comments is given below:

device_timer.c

```
// privately defined global variable, volatile access! (main thread and interrupt)
volatile uint32_t timer2_Ticks_Millisec;
```

NOTE: variable *timer2_Ticks_Millisec* is a global shared variable which is visible *only* to functions in *device_timer.c* modules – other modules cannot access it since it is not exported via *device.h*[7]. The value of this variable will be incremented for each TIM2 interrupt call in ISR routine. We shall configure TIM2 timer to generate interrupts periodically with a period of 1 ms, upon UPDATE event when timer counter overflows. Note also that the use of *volatile* keyword is **mandatory** because this variable is

[7] They *could* access it if they import it manually via *extern* C keyword but that is not intended use of this variable.

shared between thread code and interrupts and when thread or ISR access the value of this variable program must always fetch the last state from the memory and not rely on a cached value in register.

```
void Timer2_Init()
{
LL_TIM_InitTypeDef TIM_InitStruct = {0};
LL_RCC_ClocksTypeDef RCC_Clocks;
uint32_t APB1_TIMER_CLK, TimerPeriod;

   // Peripheral clock enable
   LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM2);

   // calculate TimerPeriod for 1 ms time base
   LL_RCC_GetSystemClocksFreq(&RCC_Clocks);
   APB1_TIMER_CLK = RCC_Clocks.PCLK1_Frequency * 2;
   TimerPeriod = (APB1_TIMER_CLK/ 1000 ) - 1;       // 1 kHz

   TIM_InitStruct.Prescaler = 0;
   TIM_InitStruct.CounterMode = LL_TIM_COUNTERMODE_UP;
   TIM_InitStruct.Autoreload = TimerPeriod;
   TIM_InitStruct.ClockDivision = LL_TIM_CLOCKDIVISION_DIV1;
   LL_TIM_Init(TIM2, &TIM_InitStruct);

   LL_TIM_DisableARRPreload(TIM2);
   LL_TIM_SetClockSource(TIM2, LL_TIM_CLOCKSOURCE_INTERNAL);
   LL_TIM_SetTriggerOutput(TIM2, LL_TIM_TRGO_RESET);
   LL_TIM_DisableMasterSlaveMode(TIM2);

   // run timer (direct register manipulation, no LL abstraction available)
   // TIM2->EGR:       __IO uint32_t EGR; TIM event generation register,       Address offset: 0x14
   // TIM_EGR_UG:      Update Generation
   TIM2->EGR |= TIM_EGR_UG; // may be done via LL API too!
   // TIM2->SR:        __IO uint32_t SR; TIM status register                   Address offset: 0x10
   // TIM_SR_UIF:      Update interrupt Flag
   TIM2->SR &= ~ TIM_SR_UIF; // clear flag, just in case - may be done via LL API too!
   // TIM2->DIER:      __IO uint32_t DIER; TIM DMA/interrupt enable register,   Address offset: 0x0C
   // TIM_DIER_UIE:    Update interrupt enable
   TIM2->DIER |= TIM_DIER_UIE; // Enable TIM2 interupt - may be done via LL API too!

   // init global shared timekeeping variable
   timer2_Ticks_Millisec = 0;

   // TIM2 interrupt Init
   NVIC_SetPriority(TIM2_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),5, 0));
   NVIC_EnableIRQ(TIM2_IRQn);

   // TIM2->CR1 |= TIM_CR1_CEN; // Start timer in forward counting mode - direct register programming
   LL_TIM_EnableCounter(TIM2); // the same operation like direct register programming but via LL API
}
```

TIM2 initialization function

enable clock for TIM2; note that TIM2 timer is connected to APB1 bus

very important – we need to know what is the *actual* frequency that is fed to TIM2 - this is independently configured in function *SystemClockConfig()* so we should **not make any assumptions** about TIM2 clock; see further explanations later in text

calculate autoreload value to define 1 ms overflow for TIM2

initialize TIM2 in counter up mode, autoreload, 1 ms period

In this example we use register-level programming to mix it with LL API; this settings could also have been made by means of LL API

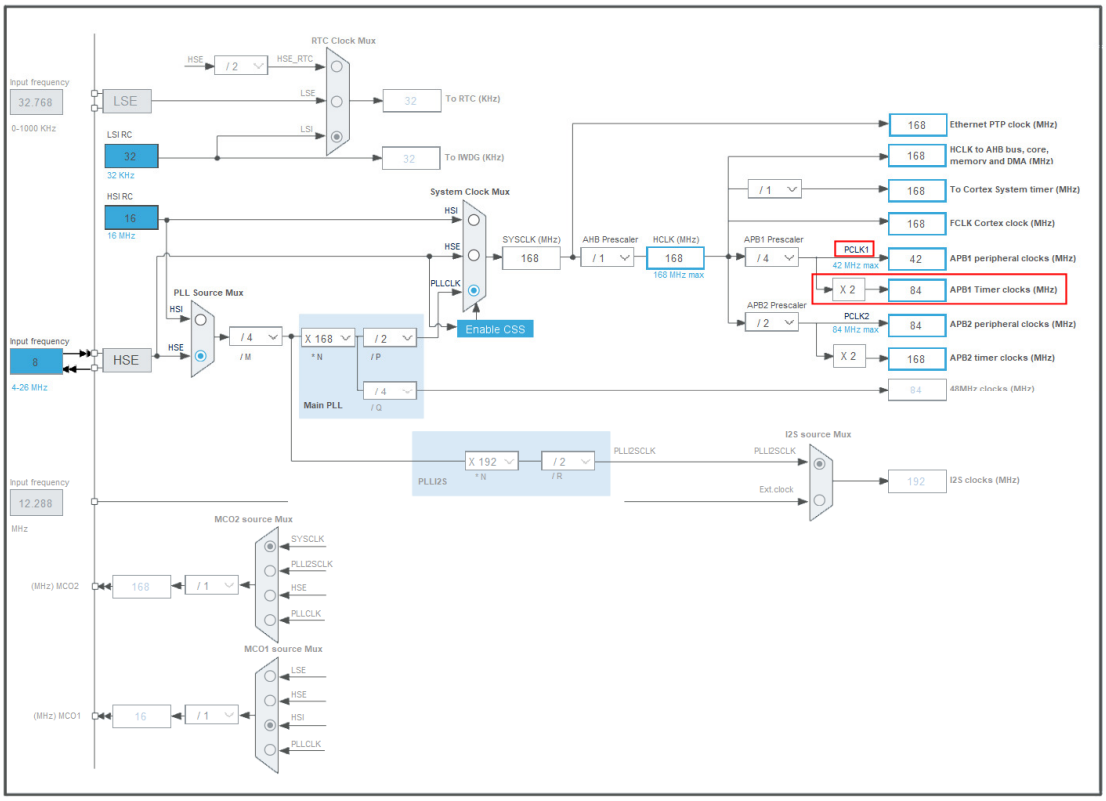
initialize internal counter

enable TIM2 interrupt in NVIC

start counter

TIM2 peripheral is much more complex that GPIO so some additional comments below will clarify some important points when configuring timer peripherals. Not all details will be explained so students are encouraged to study furthermore example source code, source code of LL API functions and structures and STM32F4x family datasheet, describing the TIM2 timer.

First important note is regarding the clock of the timer TIM2 resource. Let us take a look how system clock is distributed on a microcontroller by examining GUI *Configurator*:



We can see that timers connected to APB1 bus use *PCLK1 frequency x 2*. In our system setup processor and AHB bus is clocked with a frequency 168 MHz, however, we must take into account that APBx bus is not capable for running at the highest speed. APB1 peripherals are clocked with PCLK1 frequency (42 MHz) but timers connected to APB1 are clocked with x2 of that frequency (84 MHz). On the other hand, APB2 peripherals are clocked with PCLK2 frequency (84 MHz) but timers connected to APB2 are clocked with x2 of that frequency (168 MHz). Graphical tool only helps to visualize more easily what is provided in microcontroller datasheet.

In out case we have TIM2 connected to APB1 bus so the clock frequency of the timer TIM2 is PCLK1 x 2 = 84 MHz. To get PCLK1 frequency we can use LL API function:

```
LL_RCC_GetSystemClocksFreq(&RCC_Clocks);
```

This call will fill the structure with clocks info:

```
/**
  * @brief  RCC Clocks Frequency Structure
  */
typedef struct
{
  uint32_t SYSCLK_Frequency;        /*!< SYSCLK clock frequency */
  uint32_t HCLK_Frequency;          /*!< HCLK clock frequency */
  uint32_t PCLK1_Frequency;         /*!< PCLK1 clock frequency */
  uint32_t PCLK2_Frequency;         /*!< PCLK2 clock frequency */
} LL_RCC_ClocksTypeDef;
```

Then, we can use the obtained PCLK1 frequency to calculate the clock of timers on APB1 bus:

```
APB1_TIMER_CLK = RCC_Clocks.PCLK1_Frequency * 2;
```

Now when we obtained a clock for TIM2 in a runtime we can set properly autoreload period:

```
TimerPeriod = (APB1_TIMER_CLK/ 1000 ) - 1;      // 1 kHz
```

Since we set up our timer to count up, when TIM2 counts *TimerPeriod* number of pulses of frequency APB1_TIMER_CLK (84 MHz in our case) it will generate an UPDATE event (timer compare event), meaning that we have reached the value of *TimerPeriod*. Upon reaching the timer compare event, TIM2 automatically resets internal counter and count from the beginning. To set up TIM2 to work in autoreload mode with period defined my *TimerPeriod* we need to do the following:

```
TIM_InitStruct.Prescaler = 0;
TIM_InitStruct.CounterMode = LL_TIM_COUNTERMODE_UP;
TIM_InitStruct.Autoreload = TimerPeriod;
TIM_InitStruct.ClockDivision = LL_TIM_CLOCKDIVISION_DIV1;
LL_TIM_Init(TIM2, &TIM_InitStruct);
```

Timer *will not generate* update event until explicitly instructed to do so:

```
// TIM_EGR_UG:      Update Generation
TIM2->EGR |= TIM_EGR_UG; // may be done via LL API too!
```

Timer *will not generate* an interrupt request upon update event until explicitly instructed to do so:

```
// TIM_DIER_UIE:    Update interrupt enable
TIM2->DIER |= TIM_DIER_UIE; // Enable TIM2 interupt - may be done via LL API too!
```

Timer interrupt *will not be fired* until TIM2 interrupts are explicitly enabled in NVIC:

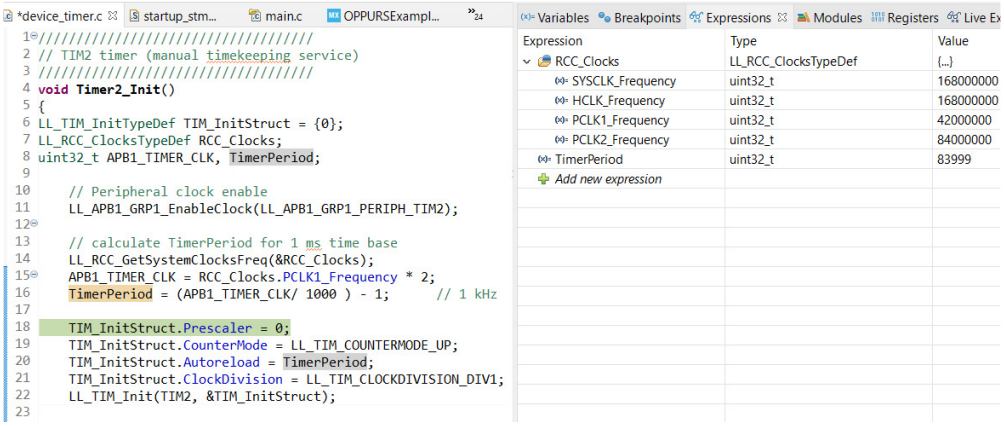
```
// TIM2 interrupt Init
NVIC_SetPriority(TIM2_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),5, 0));
NVIC_EnableIRQ(TIM2_IRQn);
```

Timer TIM2 will not be running (consequently, not counting nor generating updates and interrupts) until we explicitly enable it:

```
// TIM2->CR1 |= TIM_CR1_CEN; // Start timer in forward counting mode - direct register programming
LL_TIM_EnableCounter(TIM2); // the same operation like direct register programming but via LL API
```

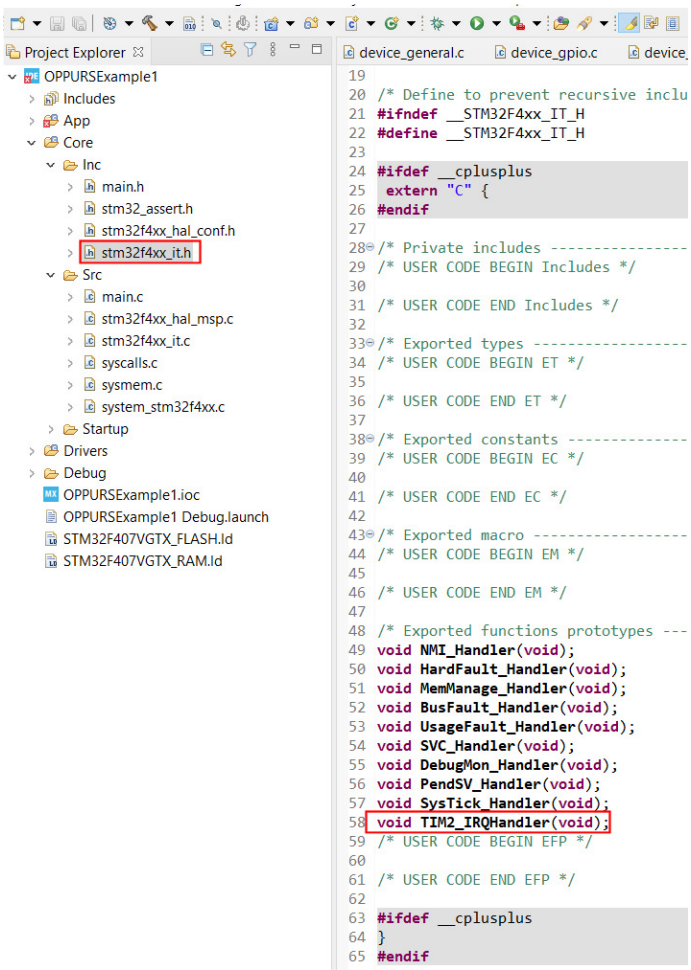
Note: in the step above (enable TIM2) we can use either direct register-access (commented code) or LL API approach to achieve the same goal.

If we run our program and put a breakpoint inside TIM2 initialization routine, we can check the values by regarding the clock information and autoreload period calculation, as shown in the figure below.



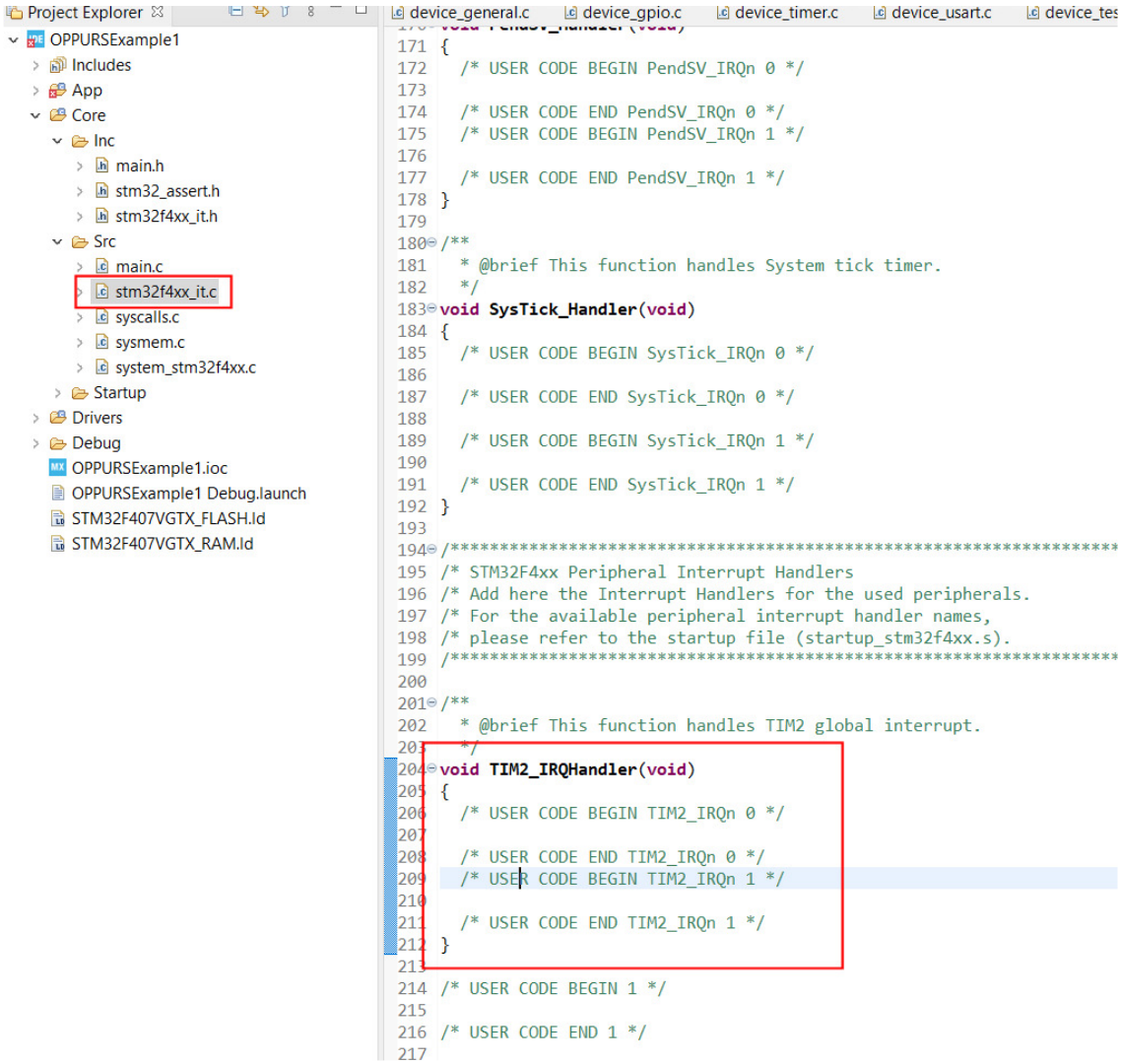
Now we have TIM2 configured, along with interrupt, but how we can connect TIM2 interrupt from autogenerated code in *main.c* with callback in *device_timer.c*?

Let us see the autogenerated code under */Core/Inc/stm32f4xx_it.h*:



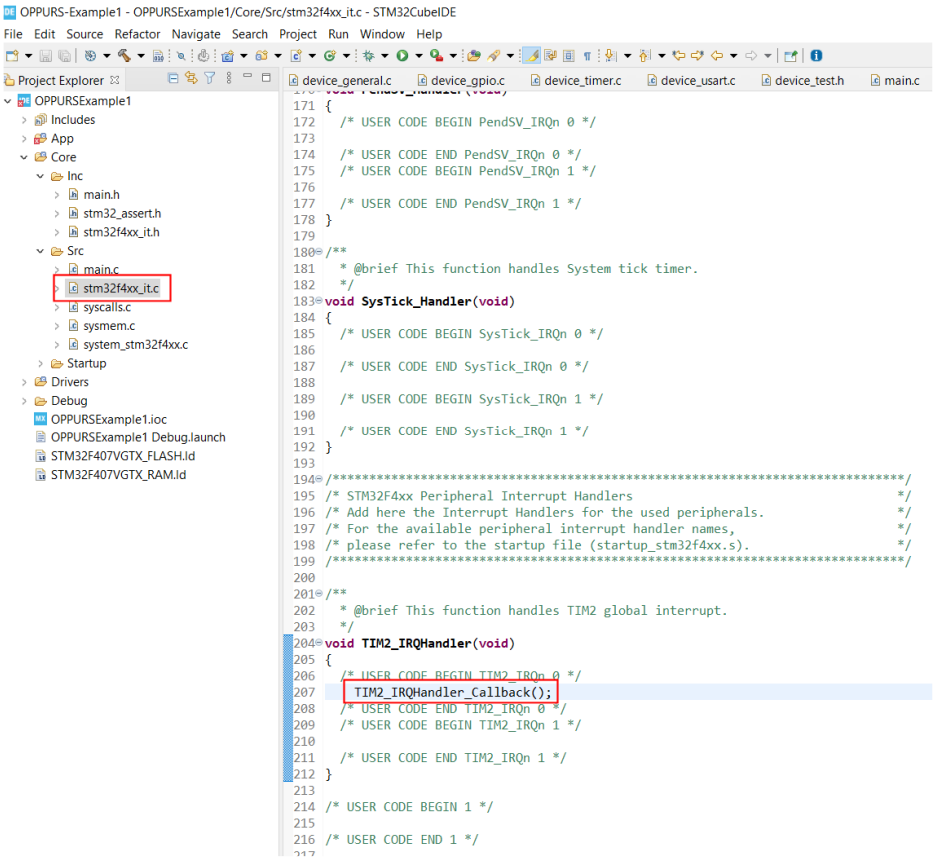
Since we configured TIM2 in *Configurator* GUI to trigger interrupts, code generator will insert definition for *TIM2_IRQHandler* in header file. Note that this name was not „randomly“ chosen, it must conform to definitions in *startup_stm32f407vgtx.s*, as previously explained (interrupt vector table definition in startup file).

Next, we examine TIM2 ISR placed inside */Core/Src/stm32f4xx_it.c*:



When TIM2 interrupt is generated, this ISR function will be called. Since we want to place our custom logic inside ISR, that is related to something the we have in *App* source tree, we want to use this ISR function body in *Core* source tree only as an entry point from where we *dispatch* ISR call to our *custom callback function,* that is under our control in *App* source tree, decoupled from any autogenerated code.

We shall call our callback function *TIM2_IRQHandler_Callback()* from autogenerated ISR like this (*TIM2_IRQHandler_Callback()* is defined in *device.h* and body of function is coded in *device_timer.c*):



Note: linker should be able to locate *TIM2_IRQHandler_Callback()* even if we do not explicitly use *extern* keyword for this function or include *device.h* header, depending on its settings. However, if the build fails, one of said mechanisms must be used to provide information to */Core/Src/stm32f4xx_it.c* where to locate callback function.

Our *TIM2_IRQHandler_Callback* function is very simple:

```
void TIM2_IRQHandler_Callback()
{
   if ((TIM2->SR & TIM_SR_UIF) == TIM_SR_UIF)
   {
      TIM2->SR &= ~ TIM_SR_UIF; // Clear interrupt flag
      timer2_Ticks_Millisec++;
   }
}
```

It checks whether the source of the interrupt was update event, clears the update event flag and increments the internal software counter *timer2_Ticks_Millisec*, which is a global variable. The only way that interrupt can exchange data with thread code is via shared global variables because thread code does not explicitly call ISRs, it is done in an asynchronous manner by NVIC. This is the reason why we need to use volatile with all global shared variables accessed by interrupts. **But we also need to take care about critical sections when we have some variable accessed both from thread code and interrupts!**

In our case, *timer2_Ticks_Millisec* is incremented every 1 ms, so it is internal millisecond counter for custom timer service.

Also note that it is preferable to use register-access over LL API in interrupts: the reason for that is that interrupts must be written in a manner that we spend as little as possible time in interrupts because only essential low-level actions in hardware should be done in interrupts, any further more complexing processing is considered coding „anti-pattern". LL would introduce some unnecessary overhead so register-level access is preferred, although LL API could be used in ISRs, if necessary, especially for lower priority interrupts handling more complex operations with hardware.

We can simply restart the timer service by calling initialization function:

```
void Timer2_Restart()
{
   Timer2_Init();
}
```

This will reset TIM2 hardware and internal data (in this case, millisecond counter).

The first example where we need to take care of critical sections is when the thread code wants to access milliseconds counter:

```
uint32_t Timer2_GetMillisec()
{
uint32_t value;

   NVIC_DisableIRQ(TIM2_IRQn);
   value = timer2_Ticks_Millisec;
   NVIC_EnableIRQ(TIM2_IRQn);
   return value;
}
```

Thread code should never directly access shared variable because if this access cannot be ensured in a single machine instruction there is a possibility that an *atomic read operation* when accessing any shared resource would be disrupted asynchronously (by interrupt) and data may be altered before full read cycle is finished. The most primitive mechanism to provide data integrity for shared variables is to *enable and disable all interrupts* that may affect the shared resource (no matter whether it is read or write operation) so to ensure that only the thread code is running while atomic operation is executed. Although in some cases all interrupts are disabled and then enabled to realize robust critical sections protecting any shared resources, we can apply this approach *only to TIM2 interrupt* if we are sure that no other interrupts access this shared resource. In this example we shall fetch the value of shared variable in a local *value* (which resides on the stack of *Timer2_GetMillisec()* function) so once we exit the critical section *value* is safe from any interrupt interference.

The function

```
uint32_t Timer2_GetTickCount()
{
uint32_t value;

   NVIC_DisableIRQ(TIM2_IRQn);
   value = timer2_Ticks_Millisec;
   NVIC_EnableIRQ(TIM2_IRQn);
   return value;
}
```

has exactly the same code as *Timer2_GetMillisec()*. However, this function was included into API for completeness: if we decide to change the time base of TIM2 (e.g. from 1 ms to 100 us or 10 ms) then the number of ticks will not be aligned with millisecond interval. In that case we want to have an API that allows us to access the raw value of *timer2_Ticks_Millisec* (in atomic manner, but it would contain

the number of ticks, not milliseconds) and milliseconds from a convenient function which would convert ticks to milliseconds.

The function

```
uint32_t Timer_Elapsed_Millisec(uint32_t t1_ms, uint32_t t2_ms)
{
uint32_t dt;

   if (t2_ms >= t1_ms) // usual situation
   {
      dt = t2_ms - t1_ms;
   }
   else
   {
      // on overflow (every 49 days...)
      dt = 0xffffffff - (t1_ms - t2_ms);
   }
   return dt;
}
```

checks two input timestamps (*t1_ms* and *t2_ms*), expressed in milliseconds, and calculates elapsed period between two timestamps. The code covers very rare (but valid) situation when *t2 < t1*.

The function:

```
void Timer2_WaitMillisec(uint32_t ms)
{
uint32_t t1, t2;

   t1 = Timer2_GetMillisec();
   while(1)
   {
      t2 = Timer2_GetMillisec();
      if (Timer_Elapsed_Millisec(t1, t2) >= ms) break;
   }
}
```

generates a *blocking delay* for *ms* milliseconds, essentially blocking the calling thread until desired time period elapses.

3.3.5.4 TIM2 timer services test routines

As explained previously, for each device driver it is recommended to develop a minimum set of test functions to verify the developed code. For TIM2 device driver a very simple test is implemented:

```
void Timer2_Test()
{
   InitDeviceCommon();
   LED_User_Init();
   Timer2_Init();

   while(1)
   {
      LED_User_Toggle(LED1);
      Timer2_WaitMillisec(1000);
   }
}
```

This test will simply blink the LED with 1 second period. This is a quick and easy way to check that the timer works well. It is easy to see whether the clock is misconfigured by the period of LED blink, even without an oscilloscope.

If more precise test of timing is required, one can use oscilloscope connected to a GPIO driving the LED and measure the pulse width to check whether it is exactly 1000 ms or to change the repetition frequency (e.g. to 10 ms) and check the resulting waveform via oscilloscope.

Two additional functions are provided in *test.c* (*LED_User_Test_WithTimer, LED_User_Test_Toggle_WithTimer*) as an extension for previously described tests for GPIO and TIM2, although everything they test is already contained in more simple tests (this functions were added for convenience of making simple blinky example with precise timing control via TIM2).

3.3.6 Advanced custom USART3 driver (device_usart.c)

3.3.6.1 USART3 device driver HAL API description

Custom USART3 device driver HAL API functions are declared in *device.h*:

```
void USART3_Init(void);                  // init USART3 for normal TX/RX operation
void USART3_IRQHandler_Callback(void);
void USART3_SendChar(char c);
void USART3_Flush(void);

void USART3_puts(char* msg);
void USART3_WriteLine(char* msg);

void USART3_Clear_TX_Buffer(void);
void USART3_Clear_RX_Buffer(void);
void USART3_Clear_Buffers(void);

int USART3_TX_InBufferCount(void);
int USART3_RX_InBufferCount(void);

void USART3_Enqueue_Blocking(char c);
int  USART3_Dequeue(char *c);

int USART3_ReadString(char* buf, int maxlen);

// non-blocking transmission
int USART3_Enqueue_WholeBuffer(char* buf, int length);
int USART3_Start_WholeBuffer_Transmit(void);
```

Short description what each of USART service API functions does:

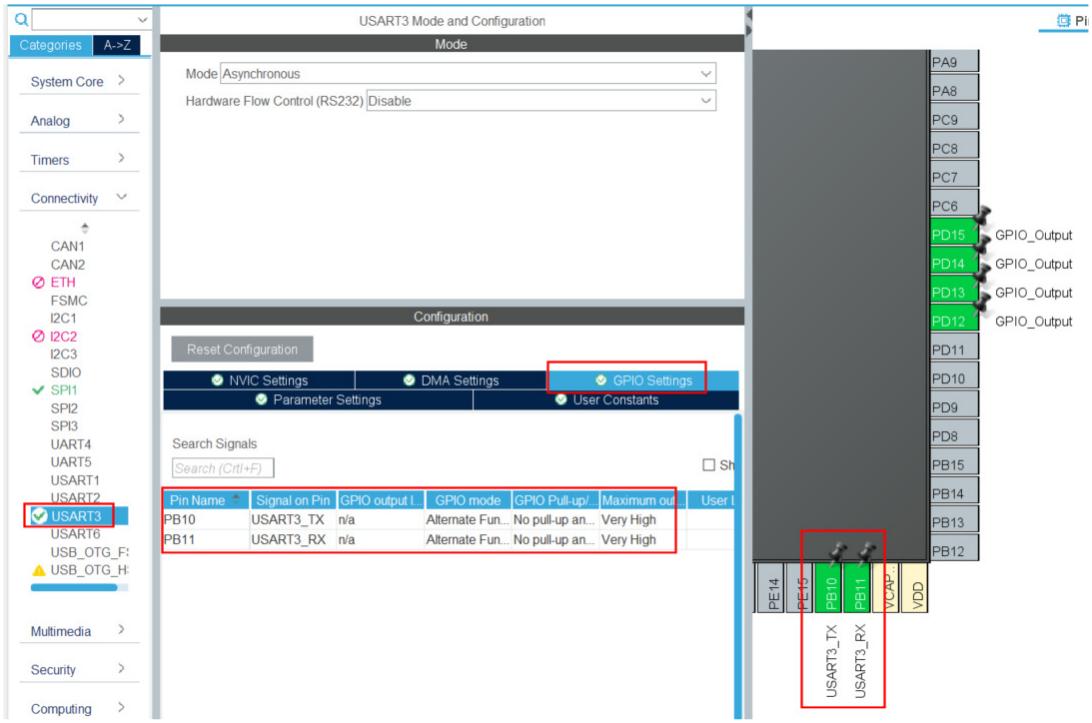
- *USART3_Init* – initializes USART3 (peripheral parameters, interrupts, FIFO buffers, flags)
- *USART3_SendChar* – *blocking* (*unbuffered*) transmission of a single character
- *USART3_Flush* – initializes *non-blocking* (*buffered*) transmission
- *USART3_puts* – *blocking* (*unbuffered*) transmission of a whole string
- *USART3_WriteLine* – *blocking* (*unbuffered*) transmission of a whole string, with newline automatically inserted
- *USART3_Clear_TX_Buffer*, *USART3_Clear_RX_Buffer, USART3_Clear_Buffers* – routines for clearing internal FIFO buffers
- *USART3_TX_InBufferCount, USART3_RX_InBufferCount* – routines for determining the number of characters currently present in TX and RX FIFO buffers
- *USART3_Enqueue_Blocking* – inserts a new character in transmit (TX) FIFO buffer; if the previous operation could not be performed due to buffer full condition, *blocks* the caller for small amount of time, until the space for next character is available; although this function is *blocking*, it should block for very small amount of time, until the FIFO buffer is ready to receive the next character (time period for sending a single character)
- *USART3_Dequeue* – *non-blocking* read of a new character (if any) from RX buffer
- *USART3_ReadString* – *non-blocking* read (dequeue) of the present content of RX buffer as a string (null terminated automatically)
- *USART3_Enqueue_WholeBuffer* – *non-blocking* transmission of the buffer (filling the TX buffer with a content to be sent in a *buffered* manner); this function only fills the buffer while *USART3_Start_WholeBuffer_Transmit* triggers buffered transmission
- *USART3_Start_WholeBuffer_Transmit* – triggers buffered transmission of the content in TX FIFO buffer

NOTE: USART3 custom device driver supports both *blocking* and *non-blocking*, and also *buffer/unbuffered* transmission and reception of characters. Driver was developed with high flexibility in mind, to be able to adapt for different usage scenarios. Device driver will be explained in details although this example will use only small functionality of developed device driver.

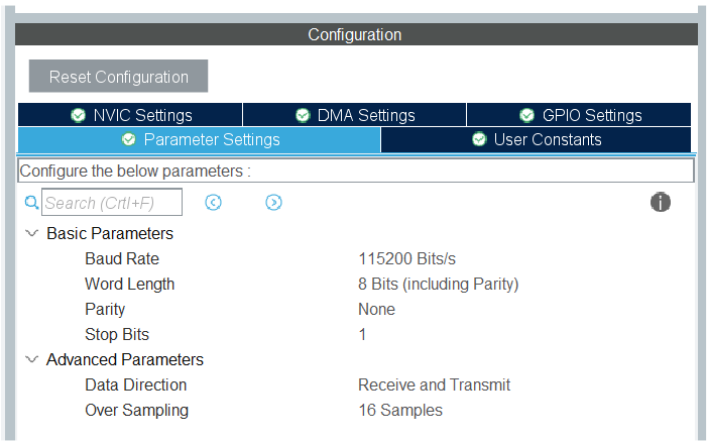
3.3.6.2 Configuring USART3 via Configurator GUI

Before analyzing the code of USART3 device driver, we shall take a look what is recommended to configure in *Configurator* GUI in order for IDE to properly include USART LL API support and to generate ISRs for USART under *Core* source tree.

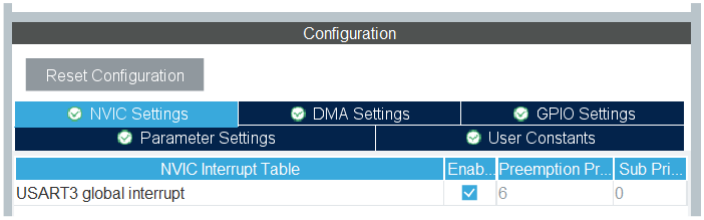
Select USART3: *asynchronous mode, no hardware flow control, GPIO pins with alternate functions (PB10 TX, PB11 RX)*:



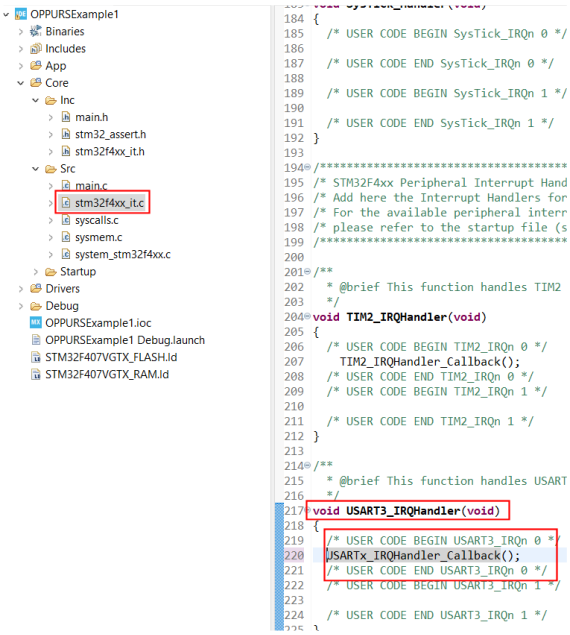
Set communication parameters (115200 bps, 8, N, 1):



Set USART3 interrupt:



Set the callback function for USART3 in *Core/Src/stm32f4xx_it.c*:



3.3.6.3 USART HAL usage scenarios

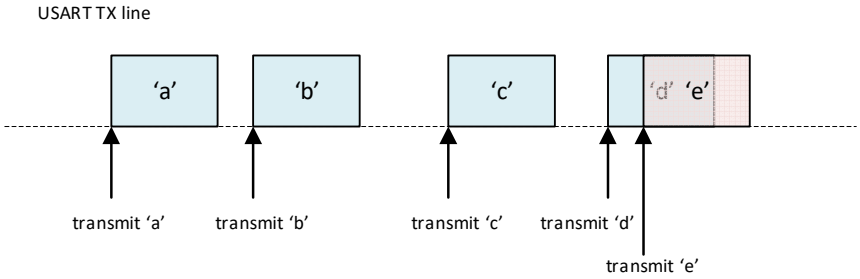
In order to better understand the code and mechanisms implemented in USART device driver, first the concepts and principles of *blocking/non-blocking* and *unbuffered/buffered* data transfer will be explained in details.

Blocking transmission, unbuffered, polling-based

One of the main goals of well designed device driver is to provide flexible and reliable mechanisms for controlling peripheral, while effectively hiding and encapsulating hardware specific implementation details from the programmer who uses the device driver. To achieve this goal, it is necessary not only to understand how to handle peripheral registers but also what is the goal we want to achieve on a conceptual and algorithmic level.

First, let us consider the common problem of sending any kind of frames via some communication media (similar principles that will be described for USART apply to many similar communication interfaces).

Let us consider the case of sending series of characters via serial line as shown in the figure below:

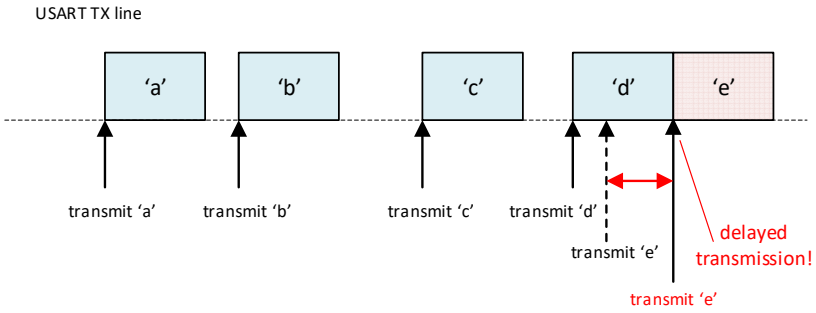


We can trigger character transmission procedure from microcontroller by placing outgoing character into hardware USART TX buffer. Microcontroller will immediately start with transmission of the written character[8]. In the figure above, we started with transmission of characters *'a', 'b', 'c'* and *'d'* after the previous character was *fully transmitted* over the serial line. But what if we instruct the microcontroller to send character *'e'* while previous character *'d'* is still being sent? Microcontroller and USART will not complain, they will just start sending *'e'* before character *'d'* transmission is finished. This will cause communication error and receiver will not correctly decode neither of these two characters.

If we try to send characters without checking the status of the previously sent character, this is what is going to happen. Therefore, we cannot have a reliable communication if we do not have some mechanism for monitoring when the previous character was fully transmitted. USART has status register and flags that enable monitoring the state of the last sent character, by either of two mechanisms:

- polling the USART status register in a thread code loop and waiting for a bit in the status register to change to notify us that the previous character was fully transmitted (not recommended approach, due to blocking of the main thread!)
- letting the ISR handle the next character transmission right at the moment when the previous character was fully transmitted

No matter which approach we choose, the final timing should look like this:



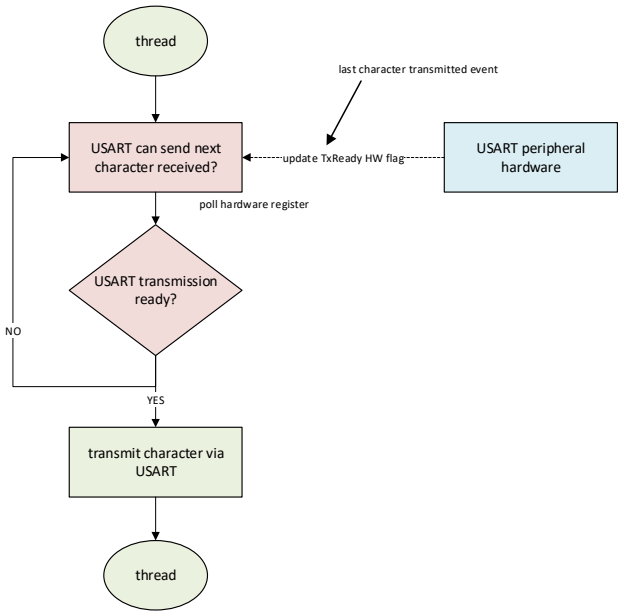
Either mechanism of the delayed transmission must ensure that there is no collision between characters 'd' and 'e'.

[8] Assuming that it does not have implemented built-in FIFO buffering capabilities, what is usually not implemented for USART interface on microcontrollers.

Let us consider how to implement this mechanism by means of:

- *blocking transmission* – blocking the thread until we are ready to send the next character
- *unbuffered* – we do not use FIFO buffering
- *polling* – we do not use interrupts, only peripheral polling within a main thread loop

The principle of this approach is shown in the flowchart below:



We want to send character from the main thread (*main()* function). We enter an infinite loop, polling the status register in each pass. We do not transmit a new character until we detect that USART status register has changed, indicating that the previous character was fully transmitted. This will happen asynchronously, under the control of peripheral USART hardware, once it detects that character has been sent. At that moment, we are ready to exit the polling loop and transmit the next character.

The main drawback of this approach is that we block the main thread during the serial transmission. We are not able to do anything else (at least on a thread level) while we send messages over the serial port. While this may be acceptable in some cases and for very short messages, this approach is rarely used in real production environment, especially if RTOS that enables multitasking is not used[9].

Blocking reception, unbuffered, polling-based

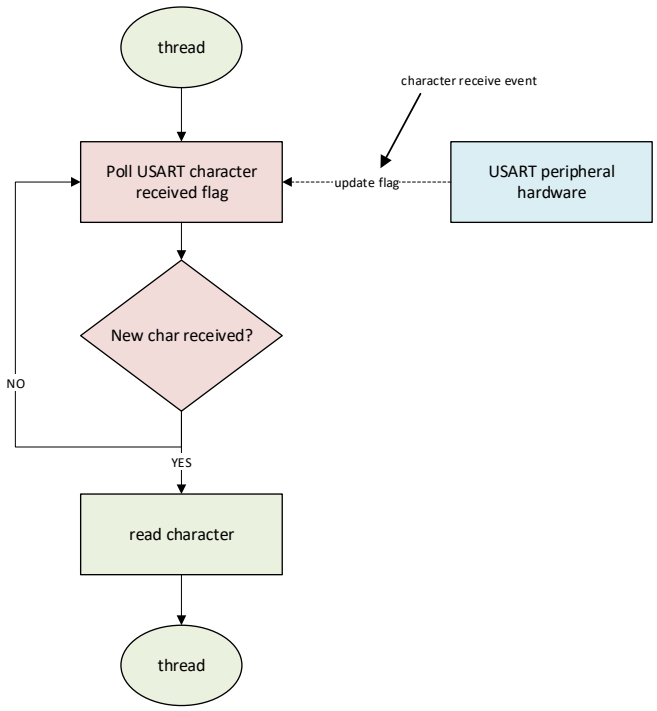
Similar considerations may be applied to the case of character reception, although we do not have a problem of possible invalidation of data on communication media due to adjacent character overrun.

The simplest USART character reception approach has the following properties:

- *blocking reception* – we block the thread until we detect a new character received by USART peripheral
- *unbuffered* – we do not use FIFO buffering
- *polling* – we do not use interrupts, only peripheral polling within a main thread loop

[9] This problem may be alleviated in multitasking systems if we block only one task (thread) while others may still do some useful jobs.

The flowchart of this approach is shown below.



Just like in the previous example, the main thread blocks for an indefinite period of time, waiting for a character to be received by the USART interface. We poll the status register in an infinite loop, until we detect a new character. The status register will be updated asynchronously, under the control of the peripheral USART hardware. At this moment, we are ready to exit the polling loop and to read the new character from USART peripheral registers.

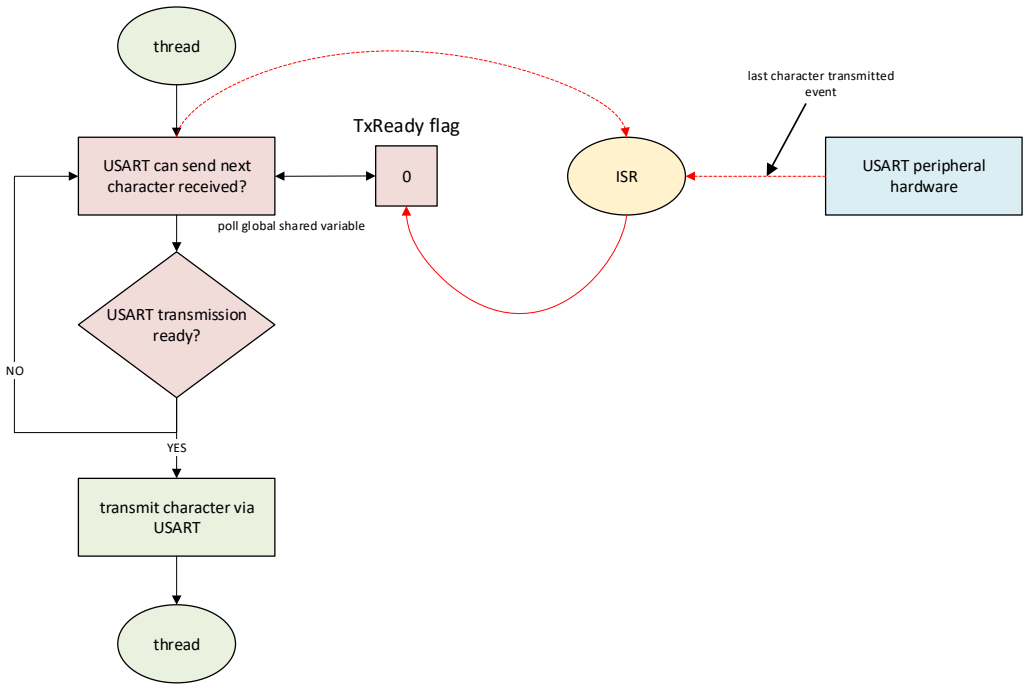
The main drawback of this approach is that we block the main thread during the reception, possibly forever. Other drawbacks of this approach are similar as discussed for serial transmission via polling approach.

Blocking transmission, unbuffered, interrupt-driven

Higher flexibility and better characteristics of the solution may be achieved if interrupts are used instead of polling. Let us consider the solution for transmitting characters which is based on:

- *blocking transmission* – we block the thread until we detect that USART is ready for the next character to be sent (same as with the blocking polling approach)
- *unbuffered* – we do not use FIFO buffering
- *interrupt* – we use interrupt to update some *global variable flag* that will signal the thread that USART transmission is ready, *instead of directly polling the peripheral* hardware register from the thread.

The flowchart for this approach is shown in the figure below:



This approach is very similar to the *blocking unbuffered polling transmission* approach, but with some important differences. Like in the polling approach, we enter an infinite loop in which we wait for USART transmission ready. The difference is that we do not poll hardware registers directly but instead we poll some global shared variable (*TxReady* flag), updated from USART ISR. Let us say that initially *TxReady = 0*. At the moment when the character sending over USART line is complete, ISR will be generated (*transmitted character* (TC) event will be the source of the interrupt). In the ISR we shall detect that the interrupt source was TC and we shall accordingly update *TxReady = 1*. Upon ISR exit, the main thread will be notified via *TxReady* flag that USART is ready for next character to be sent. We exit the loop and send the next character, without a need to „double check" USART TX availability by polling USART hardware register.

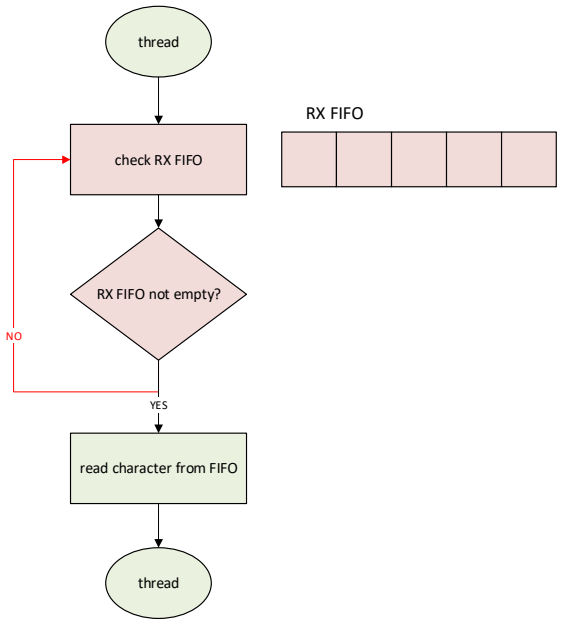
Although there is no big difference between this approach and the polling-based unbuffered character transmission from the functional standpoint, this approach is a base for much more advanced solution*, buffered interrupt transmission*. This mechanism will enable efficient implementation of *non-blocking* transmission, an approach that will be elaborated in details later. But first we shall see how to implement *buffered reception*, since it is easier case to begin with.

Blocking reception, buffered, interrupt-driven

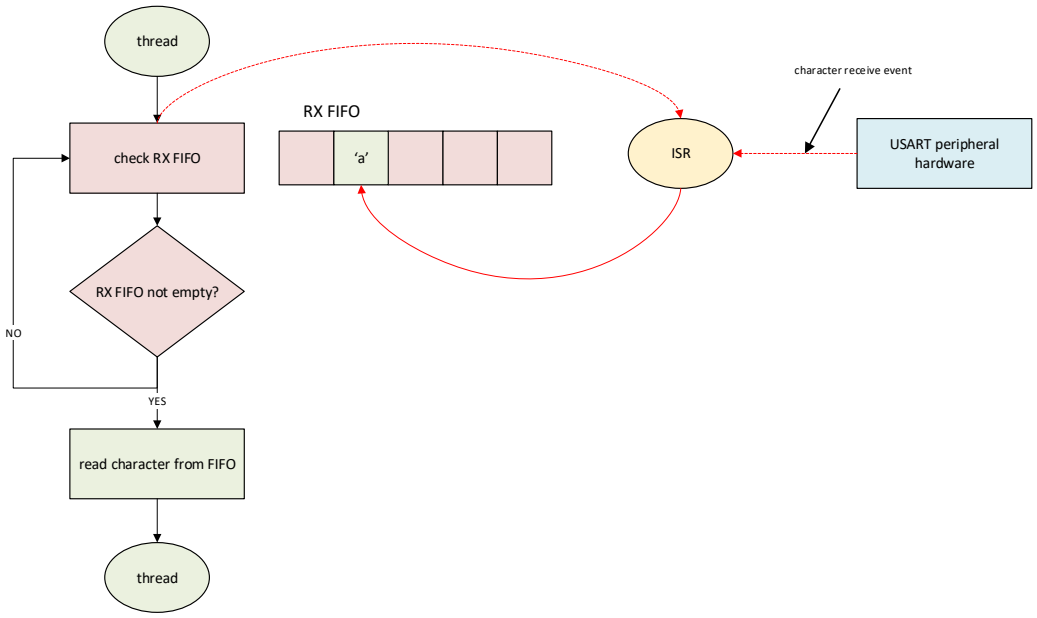
Let us see how we can improve the character *reception* by using both *interrupts* and *buffering* mechanisms. First, we shall see how buffered interrupt reception can be used to build better blocking API. Then, we shall extend this approach to *non-blocking reception*, what will enable higher flexibility in terms of minimal impact on the main thread and maximum real time flexibility, since we would be able to delay received characters processing as they are saved in a FIFO buffer, without losing characters while we do some other more important job in the main thread.

Let us analyze the flowchart below. Instead of polling directly USART registers, the thread will check are there any characters in the received characters queue (RX FIFO). RX FIFO is a queue that is initially empty and when a new character is received the USART ISR will put that character in a queue. RX FIFO must be implemented as a *global shared variable* to be accessible from both the main thread and the ISR. The figure below shows the initial case when the RX FIFO is empty and no characters have been received yet. The main thread polls a queue, not a hardware, to detect received characters. *Red arrows*

show the line of code execution – when there are no received characters in RX FIFO, we are in an infinite loop, until we receive the first character.

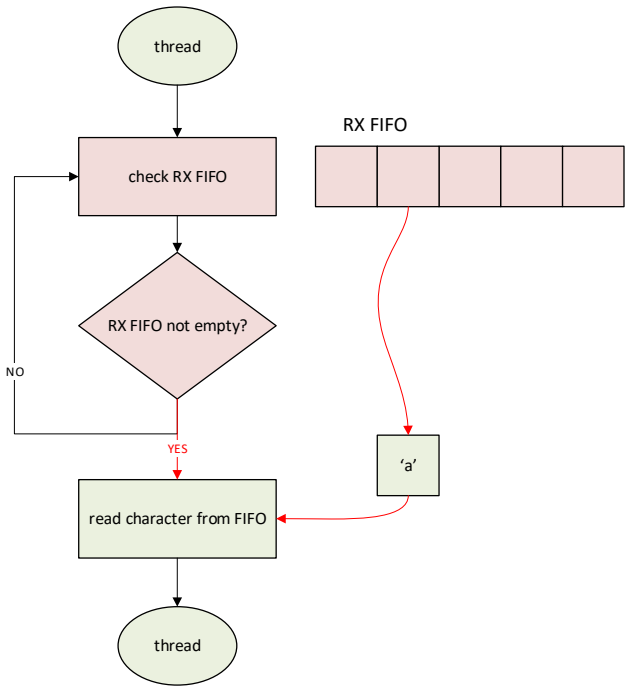


Now let us consider what happens when first character is received via USART interface:

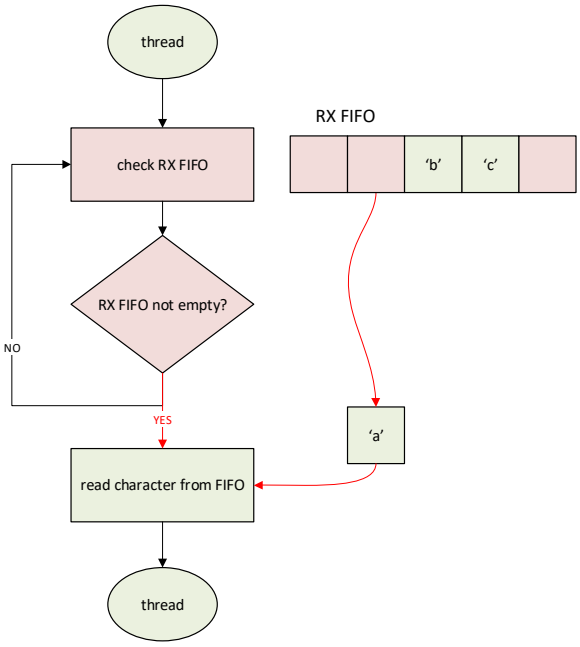


Upon the character reception, the main thread is suspended and we enter the USART ISR. ISR gets the new character from the hardware register and places it into the RX FIFO queue (character *'a'* in this example). After exiting the ISR, RX FIFO will be in a state „*not empty*".

As soon as a new character is pushed into RX FIFO (*„not empty"*), the main thread will exit the infinite loop and get the new character from the queue, as shown in the flowchart below. Getting the character from the queue is a *„destructive"* process in terms that the element is permanently removed from the queue (which is now again in the state *„empty"*, if there was only one element in the queue). Once the main thread gets newly received character, it may decide either to block again on the empty queue or to do some other job.



The principal advantage of using a *queue* for *buffered reception* is not only that we decouple the logic of accessing hardware registers from the main thread to the interrupt, but we also may process received characters saved to the temporary storage later, thus relaxing real-time response requirements in the main thread. Let us consider the flowchart below:



It shows the situation when the main thread was busy with some other more important job while three new characters arrived over the serial line. Although the main thread was busy with something else, USART ISR was called three times and all three characters (*'a', 'b', 'c'*) were saved in a RX FIFO buffer for later processing. When the main thread gets the time to process the incoming characters,

they are available from RX FIFO and no characters have been lost due to the slower main thread response.

Without buffering, very stringent timing requirements are imposed over the main thread: if the received characters are *not buffered*, then *each characters must be processed before new character arrives*, otherwise old character will be lost (microcontrollers typically can hold only a single received character, as there is no typically built-in support for hardware buffering). For example, if we set a communication speed to 115200 bps, duration of a single character (8 bits, start and stop bit) is around 87 µs. It means that the main thread must process *every character* within a deadline of 87 µs. That means that the duration of *each iteration* of the main loop must be shorter than 87 µs. In practice, it may be very hard and in most cases impossible to comply with such strict restrictions in deterministic way, even on very fast microcontrollers.

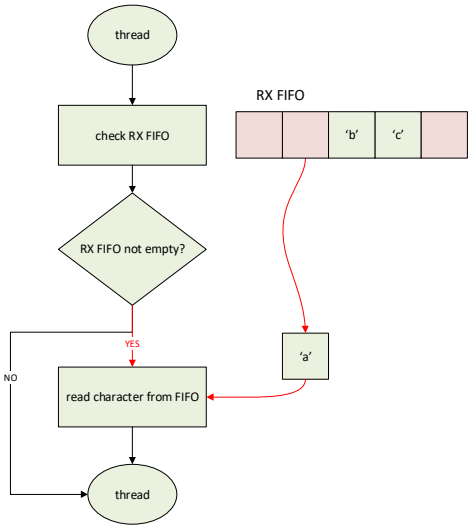
On the other hand, if we have e.g. RX FIFO buffer with a depth of 256 characters, then we need to process received character before queue gets full, what happens approx. every 2.2 ms. This is significantly less demanding deadline requirement for hard real-time system and the reason why buffering is extensively used for handling peripherals with streaming data. If even more relaxed timing restrictions are need, size of the FIFO buffer may be increased accordingly, respecting the limits of the available RAM memory.

Non-blocking reception, buffered, interrupt-driven

Now it is easy to extend the previous architecture to the more advanced and flexible one:

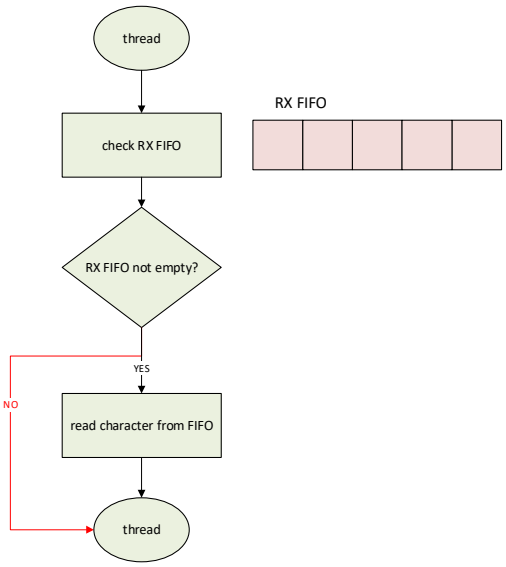
- *non-blocking reception* – we *do not have to block* the main thread on an empty RX FIFO – we can just resume the thread, until the next iteration check
- *buffered* – we use the buffer under the control of ISR
- *interrupt-driven* – we use the ISR to receive characters from USART and put them into the RX FIFO buffer

The flowchart for this case is shown in the figure below:



One may notice that this flowchart is almost the same as for *blocking buffered interrupt-driven reception*. The only difference is that now *we do not block the thread* if no characters are in the RX FIFO at the moment of check. In this architecture the main thread will be never blocked, no matter the state of the RX FIFO at the moment of check. In the case shown in the last figure, we just get one character from the RX FIFO and resume the thread (even if we have more than one character in a queue).

The case when there are no characters in RX FIFO is shown below. In this case we also do not block a thread: although we do not get any characters, we just resume the main thread operation.

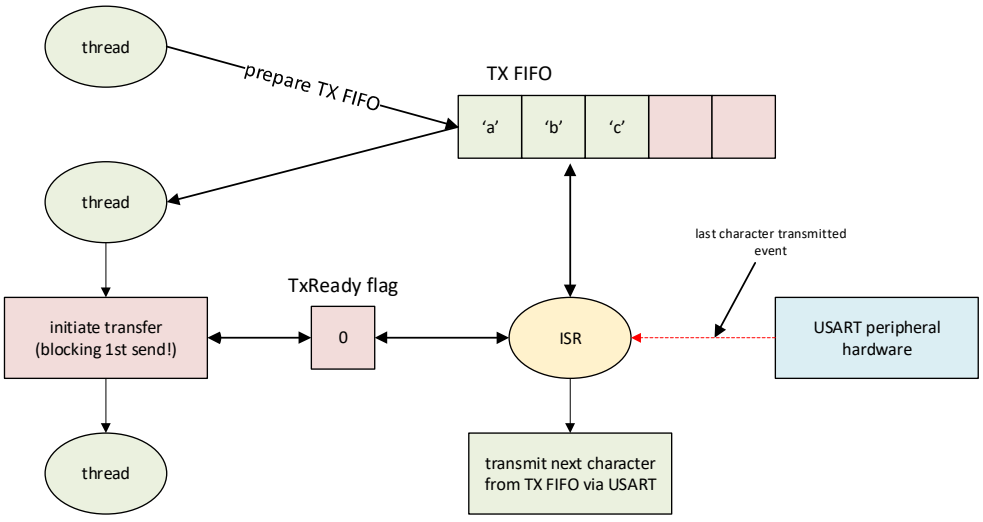


Which of two interrupt-driven reception approaches to use (unbuffered vs buffered) is up to the application developer. There are use cases when each of the approaches has some advantages. Our custom USART device driver will implement both approaches, as will be explained later in a code.

Non-blocking transmission, buffered, interrupt-driven

It is now easy to adapt a similar approach for buffered transmission. As noted before, transmission is much more sensitive to data corruption problem because we must be sure that we do not send a new character prematurely and disrupt the transmission of the previous character. Although the blocking unbuffered transmission (either polling or interrupt-driven) solves this problem, the main shortcoming of both unbuffered transmission approaches is that the main thread *must be blocked* until a transmission is over. In some cases, this may be an acceptable behavior but in general it is important to have a mechanism how to send arbitrary amount of data via serial port, without blocking the main thread at any moment and thus disrupting its timing and hard real-time constraints.

This case is more complex than previously described buffered reception and an example is shown in the flowchart below.



Let us say we want to transmit an array of three characters (*'a', 'b', 'c'*) using *non-blocking, buffered, interrupt-driven* transmission approach.

The steps and the sequence of operations is as follows:

- first, we put the message that we want to transmit into TX FIFO buffer; we put characters *'a', 'b'* and *'c'* at some point of the code execution in a main thread; since the TX FIFO buffer is just a data structure, the mere placement of the characters in a queue will not trigger their transmission over the USART – we need to explicitly initiate the transmission!
- what we need to do in the main thread is to send the first character (block „*initiate transfer...*“); in this block we do the following:
    - get one character from TX FIFO (in case when TX FIFO is empty, just resume the thread)
    - next we send this character using the same approach as described for blocking transmission (poll the hardware registers for USART transmission or wait for a *TxReady* flag to be set by ISR); once when we are ready to send the first character, write it into USART transmit hardware data register
    - once we have successfully written the first character into hardware register, resume the main thread and do not do any transmission logic in the main thread anymore - everything will be automatically handled by USART ISR!
- if the first character was sent by the procedure described above, upon the completion of character transmission over the line the USART interrupt will be triggered:
    - upon entering the USART ISR, we check the interrupt source and confirm that the interrupt was caused by TC event - USART is now ready to sent the next character
    - we take the next character from TX FIFO, if any available; we send this character via USART from ISR by writing it into USART transmit hardware data register
- this process is automatically repeated by ISR after transmission of each character
- the process is automatically ended when ISR has no more characters to get from an empty TX FIFO; the next cycle of buffered transmission must be started manually again from the main thread, as described before.

Interrupt-driven buffered transmission is a very effective way to completely lift off the burden of handling the USART transmission from the main thread.

Moreover, we do not have to wait for TX FIFO to be fully transmitted and empty before we can enqueue new characters. If we handle TX FIFO access within a critical section, it does not matter whether we enqueue new characters while some old ones are still being sent, as long as we have space to hold new characters in TX FIFO.

How to handle the situation when we enqueue elements into TX FIFO too fast, i.e. the case when we want to put a new character while TX FIFO is still full? There are two choices how to resolve TX FIFO full situation, depending on application requirements:

- *non-blocking resolution* – if we cannot put a new character in TX FIFO, we do not block and just drop the character, if dropping characters is an acceptable option; however, dropping characters is very rarely an acceptable option in communication and this will signal a situation of hard-real time constraints violation (incomplete data stream)
- *blocking resolution* – if we cannot put a new character in TX FIFO, we block the main thread until we have place for at least one character that we want to put into queue; this is the usual approach how to handle this situation since we shall block only for a very short period of time (at most until one characters is sent) and no characters will be lost on buffer full situation.

One has to take into account that the buffering is able to provide help only to handle a *peak load problem*, i.e. the situations when we have a burst of data that we want to send. If we want to send *continuously* more data on *average* than the capacity of the data link (e.g. 200 kbps of data over

115 kps) then no approach to buffer full resolution can solve this situation. TX FIFO buffer size is chosen according to the expected peak load, e.g. to be able to buffer large messages that may come in bursts, under the assumption that the amount of data over the time on average would not exceed the bandwidth of a data link.

In the following sections we shall describe how above described principles are implemented in our custom USART device driver. Some of the elaborated principles will be also used for development of some other similar device drivers (e.g. ADC samples buffering).

3.3.6.4 USART3 HAL device driver source code description

The whole listing of USART3 device driver coded in a file *device_usart.c* is given below, along with additional comments. The descriptions in this section will assume that students are familiar with all previously described particularities of device driver development already elaborated for TIM2 driver and familiarity with different USART transmission and reception approaches, described in the previous section.

device_usart.c

```
// TX FIFO
volatile char       USART3_TX_BUFFER[USART3_TX_BUFFER_SIZE];
volatile int        USART3_TX_BUFFER_HEAD;
volatile int        USART3_TX_BUFFER_TAIL;
// RX FIFO
volatile char       USART3_RX_BUFFER[USART3_RX_BUFFER_SIZE];
volatile int        USART3_RX_BUFFER_HEAD;
volatile int        USART3_RX_BUFFER_TAIL;
// TX blocking send ready flag
volatile uint8_t    Tx3Ready;
```

First, we define TX and RX FIFO buffers as global variables to be accessible both from thread code and interrupts. FIFO buffers are implemented via static C arrays with accompanying head and tail pointers[10]. All FIFO buffer variables are declared as volatile and queue sizes are configured in *config.h*. Additionally, *Tx3Ready* flag is defined to provide means to implement blocking USART transmission API, as previously described.

Note: None of these variables should be accessed directly from application code (outside of *device_usart.c*). All shared global variables are intended to be used as internals of this device driver, from thread-called API functions and USART ISR.

```
void USART3_Init()
{
LL_GPIO_InitTypeDef GPIO_InitStruct = {0};
LL_USART_InitTypeDef USART_InitStruct = {0};

  // init global variables
  Tx3Ready = 1;
  USART3_TX_BUFFER_HEAD = 0; USART3_TX_BUFFER_TAIL = 0;
  USART3_RX_BUFFER_HEAD = 0; USART3_RX_BUFFER_TAIL = 0;

  // Peripheral clock enable
  LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);   // TX, RX GPIO pins
  LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_USART3);  // USART3 peripheral

  // USART3 GPIO Configuration
  // PB10   ------> USART3_TX
  // PB11   ------> USART3_RX
  GPIO_InitStruct.Pin = LL_GPIO_PIN_10 | LL_GPIO_PIN_11;
  GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
  GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
  GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
  GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
  GPIO_InitStruct.Alternate = LL_GPIO_AF_7;
  LL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

USART3 initialization routine

initialization of all global shared variables

enable clocks both for USART3 and alternate function GPIO pins

set GPIO pins for alternate function, USART3

[10] It is assumed that students are familiar how to implement FIFO buffers in C with this approach.

```
  USART_InitStruct.BaudRate = 115200;
  USART_InitStruct.DataWidth = LL_USART_DATAWIDTH_8B;
  USART_InitStruct.StopBits = LL_USART_STOPBITS_1;
  USART_InitStruct.Parity = LL_USART_PARITY_NONE;
  USART_InitStruct.TransferDirection = LL_USART_DIRECTION_TX_RX;
  USART_InitStruct.HardwareFlowControl = LL_USART_HWCONTROL_NONE;
  USART_InitStruct.OverSampling = LL_USART_OVERSAMPLING_16;
  LL_USART_Init(USART3, &USART_InitStruct);
  LL_USART_ConfigAsyncMode(USART3);
  LL_USART_Enable(USART3);
```

set USART3 communication parameters

enable USART3 interrupt in NVIC

```
  // USART3 interrupt Init
  NVIC_SetPriority(USART3_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),6, 0));
  NVIC_EnableIRQ(USART3_IRQn);

  // enable interrupt sources
  LL_USART_EnableIT_RXNE(USART3);
  //LL_USART_EnableIT_TC(USART3);
  LL_USART_Enable(USART3);
  LL_USART_EnableIT_TC(USART3);
}
```

enable RXNE interrupt source (ISR on received character)

enable USART3 peripheral

enable TC interrupt source (ISR on transmit character finished)

After initialization, the most important part of the USART3 device driver is ISR: it implements the support for *interrupt-driven blocking and non-blocking transmission and reception* (depending on used API function):

```
void USART3_IRQHandler_Callback()
{
static int rx_char;
static int rx_head;

   // RX event
   if (LL_USART_IsActiveFlag_RXNE(USART3))
   {
      rx_char = LL_USART_ReceiveData8(USART3); // it also clears RXNE flag
      // calculate new head to see will it cause buffer overrun
      rx_head = USART3_RX_BUFFER_HEAD + 1;
      if (rx_head == USART3_RX_BUFFER_SIZE) rx_head = 0;
      if (rx_head != USART3_RX_BUFFER_TAIL)
      {
         // adding new character will not cause buffer overrun
         USART3_RX_BUFFER[USART3_RX_BUFFER_HEAD] = rx_char;
         USART3_RX_BUFFER_HEAD = rx_head;// update head
      }
   }
   // TX event
   if (LL_USART_IsEnabledIT_TC(USART3) && LL_USART_IsActiveFlag_TC(USART3))
   {
      LL_USART_ClearFlag_TC(USART3);
      // if there is anything in TX buffer, send it...
      if (USART3_TX_BUFFER_HEAD != USART3_TX_BUFFER_TAIL)
      {
         LL_USART_TransmitData8(USART3, USART3_TX_BUFFER[USART3_TX_BUFFER_TAIL]);
         USART3_TX_BUFFER_TAIL++;
         if (USART3_TX_BUFFER_TAIL == USART3_TX_BUFFER_SIZE) USART3_TX_BUFFER_TAIL = 0;
         Tx3Ready = 0;
      }
      else
      {
         Tx3Ready = 1;
      }
   }
}
```

Blocking unbuffered interrupt-driven approach for *transmission* a *single character* is implemented in API function *USART3_SendChar*:

```
void USART3_SendChar(char c)
{
   // wait until USART finishes sending the previous character
   while(!Tx3Ready);
   LL_USART_TransmitData8(USART3, c);
   Tx3Ready = 0;
}
```

When this function is called from the thread code, it will block the thread for short amount of time, until ISR sets *Tx3Ready* flag, to notify the thread that it is safe to send a next character. Character is sent by using LL API function *LL_USART_TransmitData8*.

Blocking unbuffered interrupt-driven approach for *transmission* a *string of characters* is implemented in following two API functions:

```
void USART3_puts(char* msg)
{
   while(*msg)
   {
      USART3_SendChar(*msg);
      msg++;
   }
}

void USART3_WriteLine(char* msg)
{
   USART3_puts(msg);
   USART3_SendChar(13);
   USART3_SendChar(10);
}
```

The only difference is that *USART3_WriteLine* appends automatically CR/LF (newline).

Non-blocking buffered interrupt-driven transmission is triggered by a function *USART3_Flush*:

```
void USART3_Flush()
{
char c;

   if (Tx3Ready == 1)
   {
      if (USART3_TX_InBufferCount() > 0)
      {
         NVIC_DisableIRQ(USART3_IRQn);
         c = USART3_TX_BUFFER[USART3_TX_BUFFER_TAIL];
         USART3_TX_BUFFER_TAIL++;
         if (USART3_TX_BUFFER_TAIL == USART3_TX_BUFFER_SIZE) USART3_TX_BUFFER_TAIL = 0;
         NVIC_EnableIRQ(USART3_IRQn);

         LL_USART_TransmitData8(USART3, c);
         Tx3Ready = 0;
      }
   }
}
```

CRITICAL SECTION

The use case for this function is as follows:

- in thread code first fill the USART3_TX_BUFFER (TX FIFO) with desired data to be sent
- one can trigger the transmission either by calling a function *USART3_SendChar* (next time ISR call is issued it will continue with buffered transmission) or by calling this function as more clean approach to buffered transmission (*USART3_SendChar* will send first character which is *not* in TX FIFO buffer, so such approach should be avoided)
- note that any access from the thread code to the shared global variables must be protected by critical section!

Functions for clearing the TX and RX FIFO buffers:

```
void USART3_Clear_TX_Buffer(void)
{
   NVIC_DisableIRQ(USART3_IRQn);
   USART3_TX_BUFFER_HEAD = 0; USART3_TX_BUFFER_TAIL = 0;
   NVIC_EnableIRQ(USART3_IRQn);
}

void USART3_Clear_RX_Buffer(void)
{
   NVIC_DisableIRQ(USART3_IRQn);
   USART3_RX_BUFFER_HEAD = 0; USART3_RX_BUFFER_TAIL = 0;
   NVIC_EnableIRQ(USART3_IRQn);
}

void USART3_Clear_Buffers(void)
{
   NVIC_DisableIRQ(USART3_IRQn);
   USART3_TX_BUFFER_HEAD = 0; USART3_TX_BUFFER_TAIL = 0;
   USART3_RX_BUFFER_HEAD = 0; USART3_RX_BUFFER_TAIL = 0;
   NVIC_EnableIRQ(USART3_IRQn);
}
```

Note that all functions use critical sections to protect the access to the queues.

Functions to get number of characters currently stored in TX or RX queues:

```
int USART3_TX_InBufferCount(void)
{
int head, tail;

   NVIC_DisableIRQ(USART3_IRQn);
   head = USART3_TX_BUFFER_HEAD;
   tail = USART3_TX_BUFFER_TAIL;
   NVIC_EnableIRQ(USART3_IRQn);
   if (head > tail)
   {
      return (head - tail);
   }
   else if (head < tail)
   {
      return (USART3_TX_BUFFER_SIZE - (tail - head ));
   }
   return 0;
}

int USART3_RX_InBufferCount(void)
{
int head, tail;

   NVIC_DisableIRQ(USART3_IRQn);
   head = USART3_RX_BUFFER_HEAD;
   tail = USART3_RX_BUFFER_TAIL;
   NVIC_EnableIRQ(USART3_IRQn);
   if (head > tail)
   {
      return (head - tail);
   }
   else if (head < tail)
   {
      return (USART3_RX_BUFFER_SIZE - (tail - head ));
   }
   return 0;
}
```

Note that all functions use critical sections to protect the access to queues in a way that they take a snapshot of current values of head and tail to use them for calculations. They do not take into account the fact that ISR may change the content of the queue after exiting the critical section, since it will not corrupt the shared data.

One very important API function for *non-blocking buffered interrupt-driven transmission* is *USART3_Enqueue_Blocking*:

```
void USART3_Enqueue_Blocking(char c)
{
int len;

   while(1)
   {
      USART3_Flush();
      len = USART3_TX_InBufferCount();
      if (len < (USART3_TX_BUFFER_SIZE - 3)) break;    // ensure small space redundancy
   }                                                                    CRITICAL SECTION
   NVIC_DisableIRQ(USART3_IRQn);
   USART3_TX_BUFFER[USART3_TX_BUFFER_HEAD] = c;
   USART3_TX_BUFFER_HEAD++;
   if (USART3_TX_BUFFER_HEAD == USART3_TX_BUFFER_SIZE) USART3_TX_BUFFER_HEAD = 0;
   NVIC_EnableIRQ(USART3_IRQn);
}
```

This function is used to put a new character in TX FIFO to be sent by *non-blocking buffered interrupt-driven transmission* approach. Function itself is *blocking* because we want to be sure that character *was enqueued* in TX FIFO but the transmission itself is *non-blocking*. The sequence of operations is as follows:

- first, if not already initiated, we force *non-blocking buffered interrupt-driven transmission* by issuing the API call to the function *USART3_Flush*
- since we do not want to resume the thread until we put a character in TX FIFO, the function will wait in a loop until there is a place in FIFO queue; note about the condition when to put a new character into TX FIFO queue:
    - if a C array for implementation of FIFO has *USART3_TX_BUFFER_SIZE* bytes, the maximum number of bytes that we can put in a queue is *(USART3_TX_BUFFER_SIZE-1)* (we need *one free space* to differentiate empty from the full queue, since *head=tail* means an empty queue!)
    - therefore, if we have already *(USART3_TX_BUFFER_SIZE-1)* bytes in a queue we cannot put a new element
    - if we have *(USART3_TX_BUFFER_SIZE-2)* or less, we can put a new character into queue
    - in this implementation we take one character more to make some extra redundancy by introducing the condition *(USART3_TX_BUFFER_SIZE-3)*
- once we have place for a new character (since *USART3_Flush* will force ISR to start sending characters and emptying TX FIFO queue), we shall exit the infinite loop
- once we exit the infinite loop, we put a new character into TX FIFO buffer, in atomic manner using the critical section as shown in the code above.

The only proper and recommended way how to put new characters into TX FIFO queue is by using *USART3_Enqueue_Blocking()* API function. One must be very careful if this operation is done manually without using this API function, with proper precautions taken regarding the implementation of critical sections for such an operation outside of this function.

The *non-blocking interrupt-driven buffered reception of a single character* functionality is implemented in API function *USART3_Dequeue*:

```
int USART3_Dequeue(char *c)
{
int ret;

   ret = 0;                                                          CRITICAL SECTION
   *c = 0;
   NVIC_DisableIRQ(USART3_IRQn);
   if (USART3_RX_BUFFER_HEAD != USART3_RX_BUFFER_TAIL)
   {
      *c = USART3_RX_BUFFER[USART3_RX_BUFFER_TAIL];
      USART3_RX_BUFFER_TAIL++;
      if (USART3_RX_BUFFER_TAIL == USART3_RX_BUFFER_SIZE) USART3_RX_BUFFER_TAIL = 0;
      ret = 1;
   }
   NVIC_EnableIRQ(USART3_IRQn);
   return ret;
}
```

This function does the following:

- checks whether we have anything in RX FIFO buffer
- if RX FIFO buffer is empty, immediately exit the function (without blocking the thread) and return 0, signaling to the caller that no character was pulled from the queue
- if RX FIFO buffer is not empty, get the next character and return 1, signaling to the caller that *byref* parameter (*character pointer*) contains a newly received character from the USART
- note that the critical section must be used to protect the access to the shared resources.

The *non-blocking interrupt-driven buffered reception of a string* functionality is implemented in API function *USART3_ReadString*:

```
int USART3_ReadString(char* buf, int maxlen)
{
int i, cnt;

   cnt = USART3_RX_InBufferCount();// cnt is maximum number of available characters to be read
   if (cnt == 0)
   {
      buf[0] = 0;
      return 0;
   }
   if (cnt > maxlen) cnt = maxlen;                                   CRITICAL SECTION

   NVIC_DisableIRQ(USART3_IRQn);
   for(i = 0; i < cnt; i++)
   {
      buf[i] = USART3_RX_BUFFER[USART3_RX_BUFFER_TAIL];
      USART3_RX_BUFFER_TAIL++;
      if (USART3_RX_BUFFER_TAIL == USART3_RX_BUFFER_SIZE) USART3_RX_BUFFER_TAIL = 0;
   }
   NVIC_EnableIRQ(USART3_IRQn);
   buf[cnt] = 0;

   return cnt;
}
```

Input parameters to the function are:

- *buf* – buffer where to copy read characters from RX FIFO buffer; must have at least a space for *(maxlen+1)* bytes to store (+1 is for string NULL terminator)
- *maxlen* – maximum number of characters to be read (even if more than *maxlen* characters are available in RX FIFO buffer, they will not be read)

Return value is the number of successfully read characters and placed into *buf* buffer.

This function does the following:

- checks whether we have anything to be read from RX FIFO buffer
- then the function determines how many characters to read (*cnt*)
- then it sequentially gets *cnt* characters from RX FIFO buffer and puts them into *buf* buffer
- then it places NULL character to terminate the string in *buf* buffer (since the content of *buf* buffer will be interpreted in caller as a string)
- note that the critical section must be used to protect the access to shared resources

This function is useful when we want to pull a bulk of characters from the RX FIFO in non-blocking manner, instead of getting characters one by one from RX FIFO.

Another handy API function for *non-blocking interrupt-driven buffered transmission of the whole buffer of data* is *USART3_Enqueue_WholeBuffer*:

```
// function for non-blocking transmission
// returns 0 if: Tx3Ready not 1, if current TX buffer not empty
int USART3_Enqueue_WholeBuffer(char* buf, int length)
{
int cnt, i;

   if (Tx3Ready == 0) return 0;
   cnt = USART3_TX_InBufferCount();
   if (cnt > 0) return 0;
   if (length > (USART3_TX_BUFFER_SIZE - 3)) return 0; // leave some space (min 1, here 2) in buffer

   // fill whole buffer                                          CRITICAL SECTION
   NVIC_DisableIRQ(USART3_IRQn);
   USART3_TX_BUFFER_HEAD = 0;
   USART3_TX_BUFFER_TAIL = 0;
   for (i = 0; i < length; i++)
   {
      USART3_TX_BUFFER[i] = buf[i];
      USART3_TX_BUFFER_HEAD++;     // no need to check because buf length is already limited
   }
   NVIC_EnableIRQ(USART3_IRQn);
   return 1;
}
```

Input parameters to the function:

- *buf* – buffer with data to be enqueued in TX FIFO and sent in *non-blocking buffered* manner
- *length* – number of bytes in *buf* to be sent

This function does the following:

- checks if USART is ready to receive new data for transmission; if USART is not immediately ready, it returns 0
- checks if TX FIFO is **empty**; if TX FIFO is not empty, it returns 0
- checks if TX FIFO can accommodate **all data** at once; if TX FIFO cannot do so, it returns 0
- otherwise, place **all data** at once in TX FIFO and return 1
- note that critical section must be used to protect the access to shared resources

Some notes about the function above:

- this function may be easily improved to work also for situations when TX FIFO is not empty, but can accommodate all data,
- it could be also accommodate to enqueue buffer partially and return number of bytes successfully enqueued
- this function **does not trigger buffered transmission** – it must be done separately!

The function that is used to trigger TX FIFO transmission is shown below:

```
// return 0 of operation of init transfer not successful; 1 otherwise
int USART3_Start_WholeBuffer_Transmit()
{
char c;

   if (Tx3Ready == 1)
   {
      if (USART3_TX_InBufferCount() > 0)
      {
         NVIC_DisableIRQ(USART3_IRQn);
         c = USART3_TX_BUFFER[USART3_TX_BUFFER_TAIL];
         USART3_TX_BUFFER_TAIL++;
         if (USART3_TX_BUFFER_TAIL == USART3_TX_BUFFER_SIZE) USART3_TX_BUFFER_TAIL = 0;
         NVIC_EnableIRQ(USART3_IRQn);

         LL_USART_TransmitData8(USART3, c);
         Tx3Ready = 0;
         return 1;
      }
   }
   return 0;
}
```

It is almost the same function as *USART3_Flush()*, with only a difference regarding the return value:

- function returns 0 if USART is still busy and transmission of the buffer could not be initialized
- function returns 1 if USART transmission was successfully initiated.

For most cases the function *USART3_Flush()* should be sufficient although in some cases it is useful to have an information about status of *non-blocking transmission* initiation result.

3.3.6.5 USART3 HAL device driver test routines

Although USART driver provides lots of versatility and functionality, in this example only very simple test to ensure USART is working correctly is provided:

```
void USART3_Test()
{
int i;
char msg[32];

   InitDeviceCommon();
   Timer2_Init();
   USART3_Init();

   i = 1;
   while(1)
   {
      sprintf(msg, "Hello World #%d!", i++);
      USART3_WriteLine(msg);
      Timer2_WaitMillisec(1000);
   }

}
```

Note that although we did not retarget *printf()* function to use it in conjunction with USART3 driver, we can still achieve the similar functionality with combination of *sprintf()* function and *USART3_WriteLine()*. It is up to the programmer to decide what approach is best for a concrete case and how to structure the device driver to best suit the application and development workflow needs.

3.4 Final application example (app1)

Now let us revisit the application code and analyze how some important mechanisms incorporated into device drivers are used in the final application. The whole application code is now very simple, once we implemented all low-level details properly in device drivers and decoupled them from the application code:

```
void app1()
{
char c;
char msg[32];
int interval;

   interval = 500;
   while(1)
   {
      LED_User_Toggle(LED1);
      Timer2_WaitMillisec(interval);
      if (USART3_Dequeue(&c))
      {
         if (c >= '1' && c <= '9')
         {
            USART3_WriteLine("OK");
            interval = 100 * (c - '0'); // interval for LED blink 100-900 ms
         }
         else
         {
            USART3_WriteLine("Command not recognized!");
         }
      }
      sprintf(msg, "LED blinking interval: %d ms", interval);
      USART3_WriteLine(msg);
   }
}
```

set initial blinking interval to 500 ms

we run user application in a single thread (*Round-Robbin with Interrupts* architecture)

toggle the state of logical LED1

block the thread execution for *intervanl* number of milliseconds

non-blocking receive of new characters (from RX FIFO handled by USART ISR)

blocking API USART transmission; application will block for very short time

blocking API USART transmission

emulate *printf* with *sprintf* + *USART3_WriteLine* (blocking transmission)

This is just one possibility how we can use the developed HAL to implement the application specifications. The following points are important for efficient use of HAL API:

- avoid any reference to the actual hardware details in application code (e.g. exact port and pin numbers for LEDs; use logical LED ids and let the device driver and configuration via header files figure out how the LEDs are actually connected on the board)
- avoid any register-level access (directly or via STM HAL/LL) and let the high-level custom HAL library handle all hardware related access (registers, interrupts, buffering etc.)
- decide when to use blocking and non-blocking API calls, since both approaches can be useful, depending on the context and real-time constraints

4 Building simple real-time DAQ application (app2)

In this chapter we shall see how to develop more realistic application on an example of building a simple data acquisition application (DAQ) that works in a real time. The goals are to show: how to use A/D converter (ADC) for acquiring analog samples, how to set up sampling frequency and other ADC parameters, produce sample waveform for DAQ by means of PWM modulator, and check the correct setup of PWM and ADC sampling frequencies (by means of GPIO based frequency meter, without need for using the oscilloscope). Moreover, the example will show how to achieve data transfer between peripheral and application code using *interrupt* and *DMA transfers*.

The reference code for the example that will be described in this chapter can be found in archive: **OPPURS-Example2-ADC.rar**

The chapter will be organized as follows:

- specification of a sample application functionality and peripheral resources it uses
- description of each new custom HAL device driver (GPIO pulse counter / frequency meter, GPIO input (pushbutton), ADC (with timer TIM3 for sampling rate determination), timer for PWM generator (TIM4))
    - the example will use other device drivers already developed in the previous chapter (GPIO outputs, timer TIM2 for timing services, USART3)
    - particular emphasis will be on the ADC driver and the mechanisms how to implement interrupt and DMA transfers will be presented in details
- for each device driver accompanying small tests will be developed and explained
- description how to use upgraded custom HAL to develop final application, in accordance with specification

4.1 Sample application functionality specification

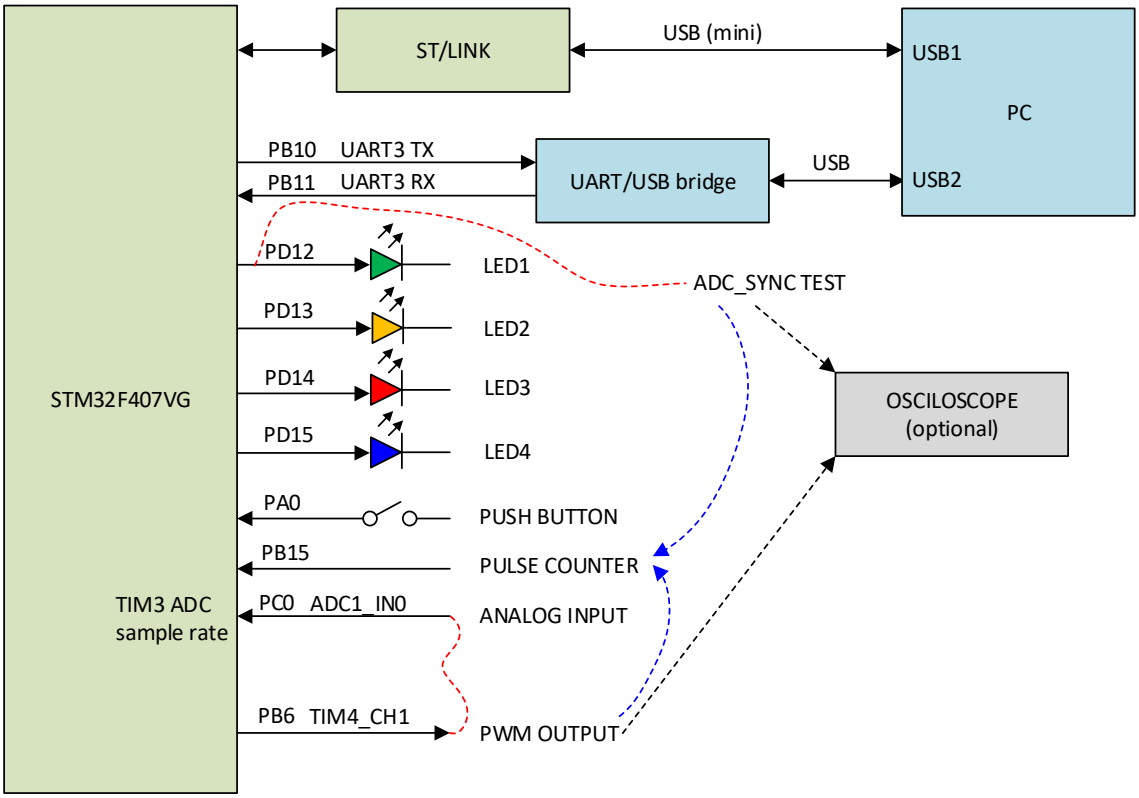
Final application has to provide the following functionality:

- depending on the configuration in *config.h*, run the following versions of application:
    - ADC data transfer in *ISR mode*
    - ADC data transfer in *DMA mode*
        - each variant has the same functionality but different mechanism of sample transfer to the RAM memory
- initialize timer services (TIM2), USART3 (115200 bps, 8, N, 1), GPIO outputs for on-board LEDs (as explained in the previous chapter)
- initialize pushbutton (PB) GPIO input on PA0; pushbutton will be used to initialize new measurement sequence
- initialize and run PWM generator (controlled by TIM4) which will generate a PWM signal on pin PB6; this signal must have PWM frequency higher than 1285 Hz (due to timer restrictions under chosen configuration, as it will be explained later), with chosen duty cycle 0-100%; this signal will show how ADC is used to capture *dynamic analog signal* and send captured samples to the computer (terminal)
- initialize GPIO pin PB15 as digital input that triggers an interrupt on rising edge; this interrupt will be used for counting pulses, what can be used to measure the frequency with microcontroller when this functionality is coupled with timing services, as it will be shown later; this functionality enables to check the validity of PWM frequency, but also ADC sampling frequency and correct DMA operation, what will be also explained later
- initialize ADC to collect samples with the chosen sampling rate (controlled by TIM3) and transferring data either via ISR or DMA approach; moreover, ADC ISR or DMA ISR, depending on the chosen transfer method, will toggle ADC_SYNC TEST pin (PD12, connected to the green LED1), to provide timing information about sampling rate or DMA buffer filling frequency

- one can use oscilloscope to check PWM and ADC_SYNC TEST frequency but it can be also checked without oscilloscope by using PB15 pulse counter as handy self-made frequency meter on the same microcontroller running the rest of the code.

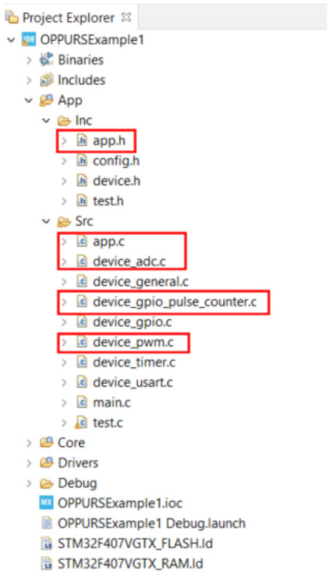
Program will wait for user to press pushbutton (PB). After PB press is detected, program will capture the sequence of 1000 samples and output them in a human readable form via USART. After all samples are sent, the program waits for the next pushbutton event.

The block diagram of peripherals and system connections on STM32F4DISCOVERY board is shown in the figure below:



4.2 Organization of source files

Organization of source files is similar to the example elaborated in the previous chapter (*app1*). Some additional files are added to the *App* source tree as shown in the figure:



Most of the files previously developed are the same as in the previous example for *app1*. In this chapter only the differences for already explained files and the details about new files will be described.

The content of each file in the *App* source tree is as follows (only new files):

Include files:

- *app.h* – header file with definitions of different application implementation functions (*app1, app2*)

Source files:

- *app.c* – implementation of different application functions (*app1, app2*)
- *device_adc.c* – device driver for ADC (with TIM3 to define sampling frequency)
- *device_gpio_pulse_counter.c* – device driver for GPIO-based interrupt driven pulse counter (frequency meter)
- *device_pwm.c* – device driver for PWM (based on TIM4 timer)

4.3 Device driver descriptions

4.3.1 Configuration header and main app file

The *config.h* header was upgraded with some additional definitions:

```
#ifndef CONFIG_H
#define CONFIG_H

// select app to run:
#define APP1        0
#define APP2_ISR    0
#define APP2_DMA    1

#define RUN_TESTS      0
#define RUN_APP        1
```

selects which application variant to build (app1 or app2; app2 can be either ADC in ISR or DMA data transfer mode; only a single choice must be set to '1' while other must be set to '0'

selects whether we want to run application or device driver tests

```
// FIFO buffers for USART3
#define USART3_TX_BUFFER_SIZE256
#define USART3_RX_BUFFER_SIZE256

// ADC samples transfer type
#define ADC_USE_INTERRUPT    1
#define ADC_USE_DMA          2
// enable toggle LED1 (PD12) for ADC sampling freq./DMA transfer freq. debugging
#define ADC_DEBUG_LED1       1

// ADC_BUFFER - buffer for collecting samples via ISR transfer
#define ADC_BUFFER_SAMPLES_SIZE          200
// ADC_DMA_BUFFER - buffer for collecting samples via DMA transfer
// - total count          ADC_DMA_BUFFER_SAMPLES_SIZE
// - half-count (DMA HT):    ADC_DMA_BUFFER_SAMPLES_SIZE / 2
#define ADC_DMA_BUFFER_SAMPLES_SIZE      100
#define ADC_DMA_BUFFER_SAMPLES_HALFSIZE      ADC_DMA_BUFFER_SAMPLES_SIZE / 2
// off-line samples processing buffer:
#define SAMPLES_BUFFER_SIZE           4096

// pulse counter
#define PULSE_COUNTER_ENABLE_DEBUG_LED2   0

#endif
```

definitions of ADC data transfer modes

enables use of the ADC_SYNC TEST pin (LED1 toggling in ISR)

size of sample queue used for ADC ISR

size of sample queue used for ADC DMA transfer

size of thread sample queue for copying samples from ISR/DMA

enables toggling LED2 each time pulse is received on PB15 pulse counter input

Also, new header *app.h* was added to organize the application variants:

```
#ifndef APP_H
#define APP_H

#include <config.h>
#include <device.h>

void app1();
void app2(int samplingFrequency, int pwmFrequency, int dutyCycle, int adcTransferMode);

#endif
```

Function *app1()* will run example1 explained in the previous chapter while *app2()* runs parametrized example for this chapter (real-time DAQ).

Also, some new tests were added in *test.h*, which will be explained later in details:

```
#ifndef DEVICE_TEST_H
#define DEVICE_TEST_H

#include <config.h>
#include <device.h>

void LED_User_Test(void);
void LED_User_Test_Toggle(void);
void Timer2_Test(void);
void LED_User_Test_WithTimer(void);
void LED_User_Test_Toggle_WithTimer();
void USART3_Test(void);
void ADC_Test_Intr(void);
void ADC_Test_DMA(void);
void GPIO_Interrupt_Test(void);
void PWM_Test(void);

#endif
```

4.3.2 Definition of custom HAL API (device.h)

Header file *device.h* contains the definition of custom HAL API. In this chapter we shall present only *new code* added to the previous version of *device.h*, which extends previously developed HAL API functionality.

Pushbutton functionality

```
////////////////////////////////////
// GPIO input (user pushbutton)
// PA0 PUSHBUTTON
////////////////////////////////////
// Note 1: there is an external pull down resistor so '0' means unpressed, '1' pressed
// Note 2: this device driver does not include debouncing support, which should be considered in
realistic applications
#define PUSH_BUTTON_PRESSED     1
#define PUSH_BUTTON_UNPRESSED   2

#define PB_GPIO    GPIOA
#define PB_PIN     LL_GPIO_PIN_0

void PushButton_Init(void);
int PushButton_GetState(void);
```

Pulse counter (which can be used as frequency meter)

```
////////////////////////////////////////////////////////
// Simple pulse counter via GPIO interrupt
// PB15
////////////////////////////////////////////////////////
void GPIO_Pulse_Counter_Init(void);
int GPIO_Pulse_Counter_Read(void);
void GPIO_Pulse_Counter_Reset(void);
```

ADC (which can be used both in ISR or DMA mode)

```
////////////////////////////////////
// ADC1
// PC0 ADC1   ADC1_IN10
// TIM3: determines sampling rate
////////////////////////////////////
void ADC_GPIO_Configuration(void);
void ADC_ADC_Configuration(int transferType);
void ADC_DMA_Configuration(void);
void ADC_TIM3_Configuration(int samplingRate);
void ADC_NVIC_Configuration(int transferType);
void ADC_SystemInit(int samplingRate, int transferType);
// high-level functions
void ADC_Buffer_Clear(void);
int ADC_Buffer_InBufferCount(void);
int ADC_Buffer_Dequeue(uint16_t* result);
int ADC_Buffer_GetOverflowFlag();
void ADC_Buffer_ClearOverflowFlag();
void ADC_Samples_Clear(void);
int ADC_Samples_InBufferCount();
int ADC_Samples_Enqueue(uint16_t result);
int ADC_Samples_Dequeue(uint16_t* result);
```

PWM generator

```
////////////////////////////////////
// TIM4: PWM
// PB6
////////////////////////////////////
void PWM_Start(int frequency, int duty_cycle);
void PWM_Stop();
```

The API definitions follow the similar guidelines as described in previous chapter for already developed peripherals. The detailed description of each newly developed peripheral device driver will be presented below. First we shall examine simpler device drivers, then more complex ones.

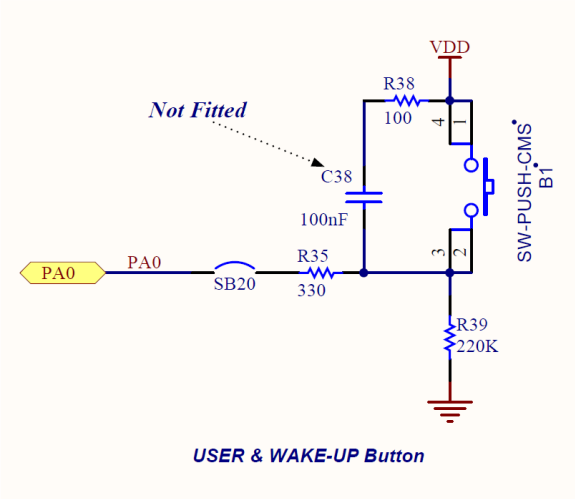
4.3.3 Pushbutton GPIO input driver (extension of device_gpio.c)

Pushbutton API is very simple. First we need to initialize GPIO pin:

```
void PushButton_Init()
{
LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

   LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);

   GPIO_InitStruct.Pin = PB_PIN;
   GPIO_InitStruct.Mode = LL_GPIO_MODE_INPUT;
   GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
   LL_GPIO_Init(PB_GPIO, &GPIO_InitStruct);
}
```

Note that only clock enable part is important, since GPIOx is not functional until we provide a clock via AHB1 bus. Second part is redundant, since every GPIO is already digital input after reset.

To determine the correct state when the pushbutton is pressed or unpressed, we need to look at the STM32F4DISCOVERY board schematic:



We can see that external pull-down resistor R39 holds logic '0' when PB is not pressed. Pressing PB will bring PA0 on VDD voltage (logical '1'):

```
int PushButton_GetState()
{
uint32_t state;

   // there is already external pull-down => '0' means unpressed, '1' pressed
   state = LL_GPIO_IsInputPinSet(PB_GPIO, PB_PIN);
   if (state == 0)
   {
      return PUSH_BUTTON_UNPRESSED;
   }
   else
   {
      return PUSH_BUTTON_PRESSED;
   }
}
```

Note: the function *PushButton_GetState* will give the current state sampled at the moment of querying the PA0 input. In realistic applications the care must be taken to provide *debouncing mechanisms* since few false inputs may be read out as an effect of mechanical vibrations of PB contacts.

4.3.4 PWM generator via TIM4 (device_pwm.c)

Setting up the timer TIM4 to generate PWM is similar to the procedure of setting up TIM2, as described in the section 2.6.6. However, we need to do some extra work because we need to determine (by examining datasheet) pins available to output TIM4 generated PWM signal and to put TIM4 in a proper PWM generation mode. We do not use interrupts with TIM4 in PWM generation mode. The code for PWM initialization is partially based on the autogenerated code and students are encouraged to study STM LL API and microcontroller datasheet (section where TIM4 and PWM generation mode is described) to better understand the code below.

```
void PWM_Start(int frequency, int duty_cycle)
{
LL_GPIO_InitTypeDef GPIO_InitStruct = {0};
LL_TIM_InitTypeDef TIM_InitStruct = {0};
LL_TIM_OC_InitTypeDef TIM_OC_InitStruct = {0};
LL_RCC_ClocksTypeDef RCC_Clocks;
uint32_t APB1_TIMER_CLK, TimerPeriod;

   // set GPIO output PWM pin
   // TIM4 GPIO Configuration
   // PB6   ------> TIM4_CH1
   LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);

   GPIO_InitStruct.Pin = LL_GPIO_PIN_6;
   GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
   GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_HIGH;
   GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
   GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
   GPIO_InitStruct.Alternate = LL_GPIO_AF_2;
   LL_GPIO_Init(GPIOB, &GPIO_InitStruct);

   // init timer
   LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM4);

   // calculate TimerPeriod that determines PWM frequency PWM_FREQ_HZ
   // TimerPeriod = (APB1_TIMER_CLK / PWM_FREQ_HZ ) - 1;     // <PWM_FREQ_HZ> output!
   LL_RCC_GetSystemClocksFreq(&RCC_Clocks);
   APB1_TIMER_CLK = RCC_Clocks.PCLK1_Frequency * 2;
   TimerPeriod = (APB1_TIMER_CLK/ frequency ) - 1;

   TIM_InitStruct.Prescaler = 0;
   TIM_InitStruct.CounterMode = LL_TIM_COUNTERMODE_UP;
   TIM_InitStruct.Autoreload = TimerPeriod;
   TIM_InitStruct.ClockDivision = LL_TIM_CLOCKDIVISION_DIV1;
   LL_TIM_Init(TIM4, &TIM_InitStruct);

   LL_TIM_DisableARRPreload(TIM4);
   LL_TIM_SetClockSource(TIM4, LL_TIM_CLOCKSOURCE_INTERNAL);
   LL_TIM_OC_EnablePreload(TIM4, LL_TIM_CHANNEL_CH1);

   TIM_OC_InitStruct.OCMode = LL_TIM_OCMODE_PWM1;
   TIM_OC_InitStruct.OCState = LL_TIM_OCSTATE_ENABLE;
   TIM_OC_InitStruct.OCPolarity = LL_TIM_OCPOLARITY_HIGH;
   TIM_OC_InitStruct.CompareValue = TimerPeriod * duty_cycle / 100; // 50% for duty cycle = 50
   LL_TIM_OC_Init(TIM4, LL_TIM_CHANNEL_CH1, &TIM_OC_InitStruct);

    LL_TIM_OC_DisableFast(TIM4, LL_TIM_CHANNEL_CH1);
    LL_TIM_SetTriggerOutput(TIM4, LL_TIM_TRGO_RESET);
    LL_TIM_DisableMasterSlaveMode(TIM4); // ***

   // TIM4->CR1 |= TIM_CR1_CEN; // Start timer in forward counting mode
   LL_TIM_EnableCounter(TIM4);
}

void PWM_Stop()
{
   LL_TIM_DisableCounter(TIM4);
}
```

TIM4 PWM may be outputed to PB6

ensure PB is clocked via AHB1

initialize PB6 in alternate function mode for generating PWM

enable clock to TIM4 via APB1

set autoreload value that determines PWM frequency

set TIM4 in counter up mode, with desired Autoreload value that determines PWM frequency; note that we use **maximum clocking frequency** with no division!

CH1 will be used for PWM generation

configure PWM on CH1; CompareValue determines duty cycle

enable TIM4 to produce PWM output on PB6

function that can be used to temporarily suspend PWM output

Input parameters to driver function are *frequency* and *duty_cycle*. Function will not do any error checking and it is a responsibility of the caller to ensure that both parameters are in a valid range.

Particular attention should be paid to the *frequency* parameter since it is obvious that *duty_cycle* must not be outside of the range from 0 to 100. How to determine the valid range for the *frequency* parameter?

We have already shown that the maximum frequency that is clocked to APB1 timers (what includes TIM4) is APB1_TIMER_CLK = 84 MHz, when system clock is set to the maximum frequency of 168 MHz. If we use the formula to calculate *Autoreload* value for e.g. PWM frequency 2000 Hz:

```
TimerPeriod = (APB1_TIMER_CLK/ frequency ) - 1;
```

we shall get a value *TimerPeriod = 41999*. If we want to generate PWM frequency 1000 Hz, then we get the value *TimerPeriod = 83999*. If we take a look how autoreload register TIM4_ARR is implemented in hardware:

18.4.12 TIMx auto-reload register (TIMx_ARR)

Address offset: 0x2C

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **ARR[15:0]**: Auto-reload value
ARR is the value to be loaded in the actual auto-reload register.
Refer to the *Section 18.3.1: Time-base unit on page 577* for more details about ARR update and behavior.
The counter is blocked while the auto-reload value is null.

it is obvious that it cannot hold a valid value for TimerPeriod = 83999 as it is implemented as 16-bit register! Therefore, the minimum PWM frequency that can be generated with the TIM4 configuration provided by our device driver is

```
min_frequency = (84 MHz / 65535 ) – 1 = 1280.76 Hz
```

If we want to generate lower frequency, we can either use prescaler different than we chose (x1):

```
TIM_InitStruct.ClockDivision = LL_TIM_CLOCKDIVISION_DIV1;
```

or change the system clock frequency to lower values.

The valid prescaler values are defined in STM LL API and described in datasheet:

```
/** @defgroup TIM_LL_EC_CLOCKDIVISION Clock Division
  * @{
  */
#define LL_TIM_CLOCKDIVISION_DIV1              0x00000000U          /*!< tDTS=tCK_INT */
#define LL_TIM_CLOCKDIVISION_DIV2              TIM_CR1_CKD_0        /*!< tDTS=2*tCK_INT */
#define LL_TIM_CLOCKDIVISION_DIV4              TIM_CR1_CKD_1        /*!< tDTS=4*tCK_INT */
```

If even the lower values are needed that it could be achieved with x4 prescaler, then we can make additional changes to the clocking system, but all possible options are beyond the scope of this example. It is important to remember that there is a limitation of the lowest frequency that timer can provide because it has limited 16-bit autoreload registers and that limits may be extended by selecting different prescalers or other changes in the clocking system. The same conclusions will be also applicable to a minimum sampling rate that may be achieved with TIM3 timer configuration, which will be used to clock ADC sampling conversion timing.

4.3.5 Pulse counter via ISR triggered by GPIO input (device_gpio_pulse_counter.c)

The goal is to provide means to count the pulses on GPIO digital input PB15. Pin will be configured to generate interrupts on a positive edge of the incoming pulse. We shall use the global variable *PulseCount* to keep a track of counted pulses. The pulse counter API provides only the functionality for counting pulses but not for measuring pulses frequency directly. This can be done in conjunction with timing services provided by TIM2, e.g. by counting the number of pulses within some time window (e.g. 1 second) and calculating the frequency from the counted pulses and the known time window duration.

The whole source code for this device driver is as follows:

```
#include <device.h>

volatile int PulseCount = 0;

void GPIO_Pulse_Counter_Init()
{
LL_EXTI_InitTypeDef EXTI_InitStruct = {0};
LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

   LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);

   LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTB, LL_SYSCFG_EXTI_LINE15);

   EXTI_InitStruct.Line_0_31 = LL_EXTI_LINE_15;
   EXTI_InitStruct.LineCommand = ENABLE;
   EXTI_InitStruct.Mode = LL_EXTI_MODE_IT;
   EXTI_InitStruct.Trigger = LL_EXTI_TRIGGER_RISING;
   LL_EXTI_Init(&EXTI_InitStruct);

   // LL_GPIO_SetPinPull(GPIOB, LL_GPIO_PIN_15, LL_GPIO_PULL_NO);
   LL_GPIO_SetPinPull(GPIOB, LL_GPIO_PIN_15, LL_GPIO_PULL_DOWN);
   LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_15, LL_GPIO_MODE_INPUT);

   // init global variables
   PulseCount = 0;

   // EXTI interrupt init
   NVIC_SetPriority(EXTI15_10_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),0, 0));
   NVIC_EnableIRQ(EXTI15_10_IRQn);
}

int GPIO_Pulse_Counter_Read()
{
int value;

   NVIC_DisableIRQ(EXTI15_10_IRQn);
   value = PulseCount;
   NVIC_EnableIRQ(EXTI15_10_IRQn);
   return value;
}

void GPIO_Pulse_Counter_Reset()
{
   NVIC_DisableIRQ(EXTI15_10_IRQn);
   PulseCount = 0;
   NVIC_EnableIRQ(EXTI15_10_IRQn);
}

void EXTI15_10_IRQHandler_Callback(void)
{
   if (LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_15) != RESET) // GPIO: PB15 is input triggered on rising
edge
   {
      LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_15);

      // signal LED
      #if (PULSE_COUNTER_ENABLE_DEBUG_LED2)
      LED_User_Toggle(LED2);
      #endif
```

shared variable to hold number of counted pulses, incremented in ISR

enable clock to PB port

set PB15 GPIO as interrupt source line

configure PB15 interrupt source line to trigger ISR on rising edge

reset pulse counter

enable interrupt

provides API call to access pulse count from application code in atomic manner

provides API call to reset pulse counter from application code in atomic manner

ISR routine called on each rising edge deteceted on PB15

reset interrupt flag

optionally, toggle LED2 to check that ISR was properly fired (e.g. via oscilloscope)

```
        PulseCount++;          increment pulse counter
    }
}
```

4.3.6 ADC device driver (device_adc.c)

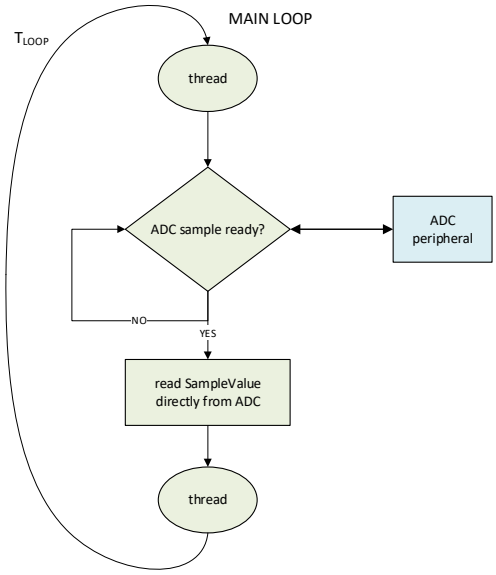
Device driver for A/D converter is the most complex one in our custom developed HAL API. Before presenting the source code, the principles behind the specific implementations of ADC data transfer modes (ISR and DMA) will be elaborated first.

4.3.6.1 ISR and DMA based approaches for transferring ADC samples

In section 3.3.6 we explained different approaches how to interact with peripherals in a real-time (polling vs interrupt, blocking vs non-blocking API, buffered vs unbuffered), applied to communication with USART peripheral. We shall revisit these mechanisms and particularities of their application to ADC, with addition how to extend ISR based data transfer to include DMA support. For sake of completeness, we shall also consider polling and unbuffered ADC samples transfer to compare it with other approaches.

Blocking unbuffered polling-based ADC data transfer

This is the simplest approach to the ADC data samples transfer. Let us assume the application architecture shown in the flowchart below:



The characteristics of this approach are the following:

- we have the *main loop* (“*superloop*”) where all application code runs to periodically serve different parallel tasks in a microcontroller application (e.g. polling sensors, calculating the state variables, controlling actuators etc., usually referred to as *jobs*)
- each loop iteration has some duration (T_{LOOP}) and we do not know in advance how long each iteration will last, especially if we have blocking calls inside a loop
- *blocking polling* approach assumes that we poll ADC hardware peripheral registers directly, until the ADC conversion is done:
    - if we set ADC sampling period e.g. to T_{SAMPLE} = 1 ms, we spend almost the whole sampling period inside an idle loop and we are unable to do anything else in a main loop during that period (for a whole millisecond!)

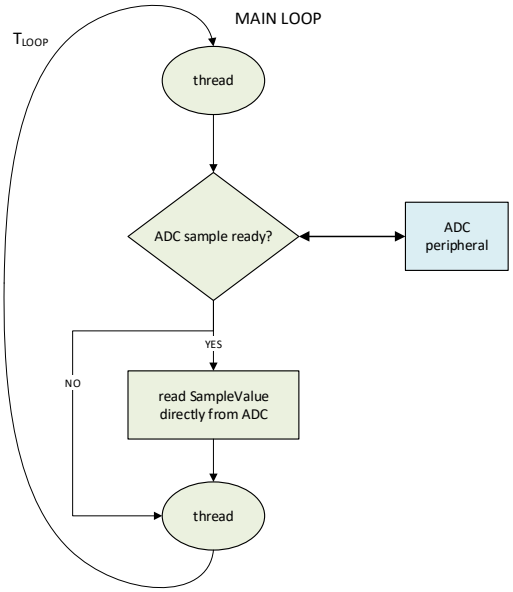
- we can only poll ADC registers and wait for ADC conversion to finish
- then we read the *SampleValue* from ADC registers directly
- we can do some useful operation only *after* the ADC conversion is finished.

The obvious drawbacks of this approach and the reasons why it is rarely used in practical applications are:

- we waste lots of processor time in idle loop (nearly 100% time!)
- we have to ensure that the loop processing time in *each pass* (i.e. for each sample) is *less than a sampling time*, to ensure accurate sample timing
- we cannot use any other peripheral blocking calls to retain well-defined sampling frequency.

Non-blocking unbuffered polling-based ADC data transfer

Slightly better approach uses a modified architecture without blocking, as shown in the figure below:



The main difference in comparison with blocking approach is that we do not block the thread if ADC sample is not ready when we poll the ADC peripheral in the iteration. If the ADC conversion is not finished yet, we just resume the loop operation and schedule ADC poll for the next iteration pass.

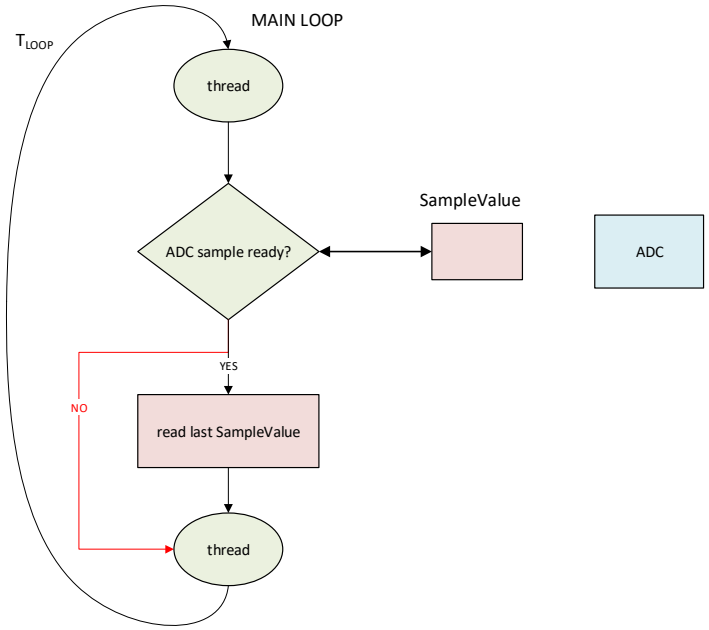
However, in this approach we must ensure that $T_{LOOP} < T_{SAMPLE}$, if we want to prevent missing samples: *worst case execution time* (WCET) for each iteration must ensure that all *jobs* are executed in a single iteration within a T_{SAMPLE} time. Even if this approach could be sometimes successfully implemented for lower sampling rates (e.g. 1 kHz), it may become an impossible task for higher sampling rates (e.g. for 1 MHz guaranteed WCET for each iteration must be less than 1 µs!).

This architecture may be applicable only for low sampling frequencies or cases when lost samples are acceptable. But in general, especially for faster signals and DSP algorithms, this approach is not a suitable one.

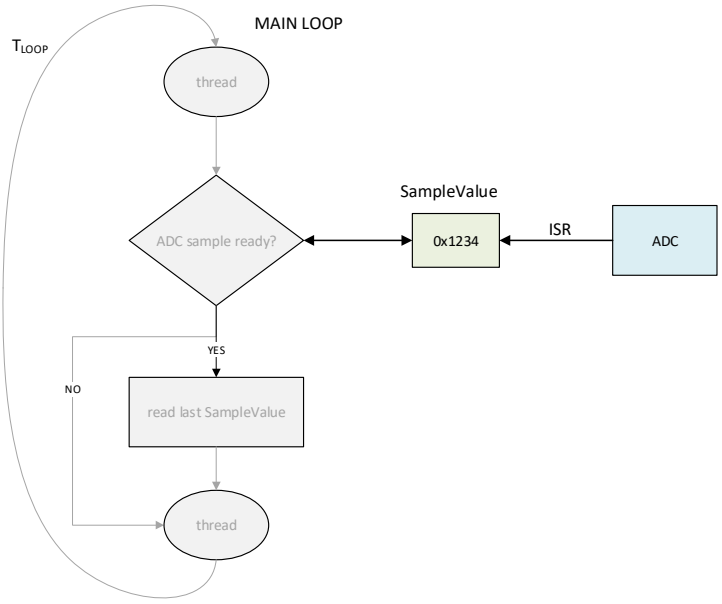
Non-blocking unbuffered interrupt-based ADC data transfer

In general, it is advisable to avoid polling-based approach for data exchange with peripherals whenever possible. Interrupt-driven transfer is preferred because it allows to decouple the main thread loop from direct hardware access and let the interrupts handle register-related operations in the most efficient and timely manner.

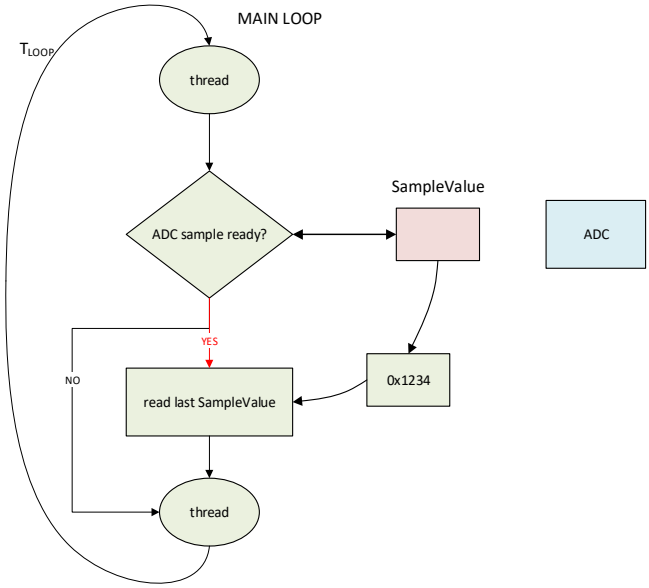
The flowchart below shows interrupt-based data exchange between the main thread and ADC ISR routine:



In this architecture we still have the main thread loop but we do not poll ADC hardware registers directly. Instead, we poll a *shared global variable*, called *SampleValue* in this example. This value is *shared* because it can be accessed both from the main thread and interrupt. We can poll from the main loop whether the result contained in *SampleValue* is valid: if it is, we can read the value, otherwise, the associated value is invalid. Initially, the *SampleValue* does not hold a valid ADC result and we should just resume the loop execution (non-blocking approach). At some point, when a new ADC conversion is finished in hardware, the main thread will be *asynchronously* interrupted and we shall jump into ADC ISR:



ADC ISR will put the ADC conversion result in a shared global variable *SampleValue*, marking it as valid for the next poll in the main thread:



Now the main thread can get a new valid ADC result. After successful read operation, the main thread must invalidate *SampleValue* to be able to detect the next valid ADC conversion after it is handled by ADC ISR.

Although this approach completely decouples the main thread from a direct hardware register access (main thread polls shared variable *SampleValue*), there is no big difference from the case of *non-blocking polling-based unbuffered ADC data transfer*: we still must finish each iteration within T_{SAMPLE} time, otherwise, the next sample will be overwritten by ADC ISR. In terms of timing constraints, there is really no significant difference between *non-blocking polling-based unbuffered* and *non-blocking interrupt-based unbuffered ADC data transfer*.

However, *interrupt-based ADC data transfer* sets a good base to extend it for much more powerful *buffered* approach.

Non-blocking buffered interrupt-based ADC data transfer

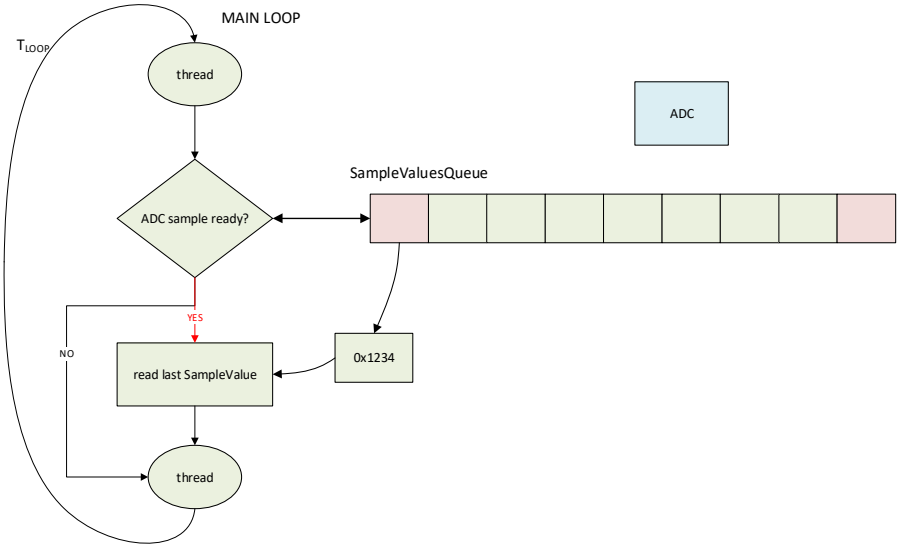
Significant difference between *non-blocking interrupt-based unbuffered and buffered ADC data transfer* is that *unbuffered* approach uses a *single (shared) variable* for storing ADC result while *buffered* uses *(shared) FIFO buffer* (queue) to store multiple results, depending on the available storage memory in RAM. What is the most important gain of the buffered architecture?

The main challenge with timing in a real-time system is to ensure *guaranteed* WCET for each thread iteration: the lower is target needed guaranteed WCET, less complex useful work can be done in each loop iteration. If loop iteration WCET must be very low, loop cannot do complex operations.

By introducing *buffering* for background hardware processes handled by ISRs we can relax thread loop iteration WCET demands significantly. Let us consider an example in the figure below, where we use a FIFO buffer with a capacity for 8 elements[11], where ADC ISR puts new samples when ADC conversion is finished. The main loop will have time of (8 x T_{SAMPLE}) to get samples from the queue before it

[11] FIFO realized by C array with size 9 can hold 8 elements (in this example we depicted elements as C array).

becomes full. This means that now iteration WCET can be 8 times longer before we start losing samples, in comparison with unbuffered approach.

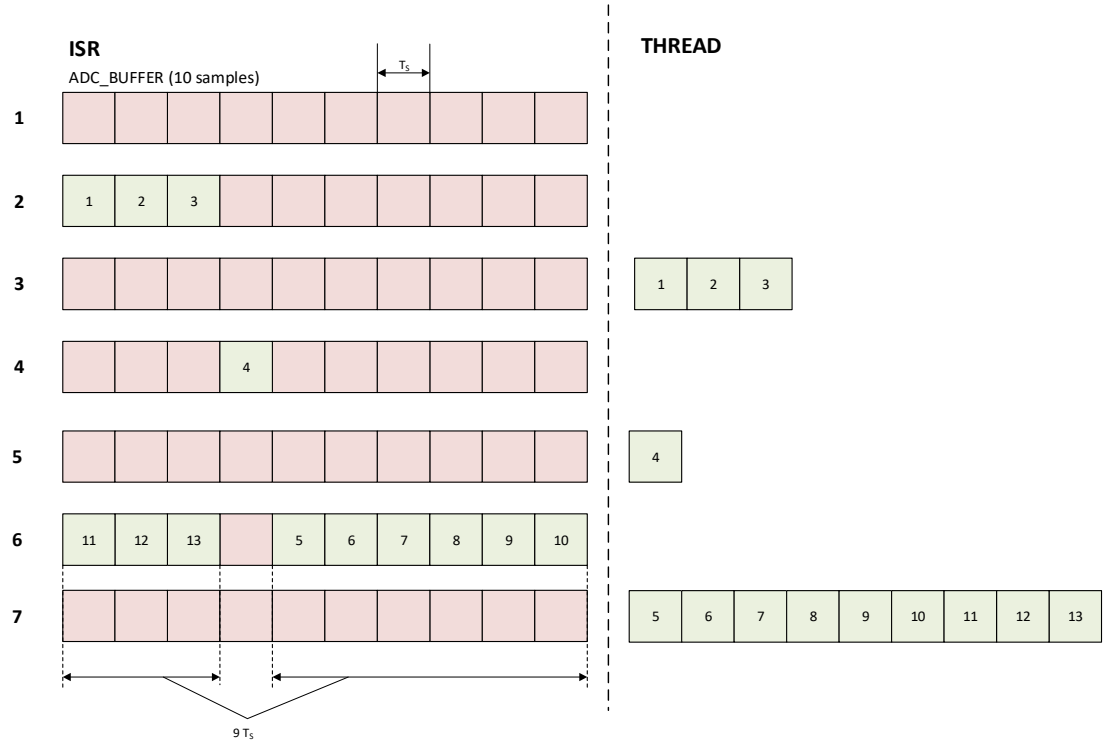


Let us assume that we have a queue with space for 1024 elements. If the sampling period is 1 ms, the loop iteration WCET is around one second! If we have a high sampling frequency e.g. 1 MHz, we have around one millisecond to finish all loop jobs.

It is important to emphasize that in practice time needed to empty the whole queue is negligible compared to the target thread loop WCET, even for larger queues. For example, if we have a processor running at 50 MHz, assuming 10 cycles to dequeue one element, for a queue with 1000 elements the total transfer time for the whole queue would be around 200 µs, what is typically much shorter then target loop iteration WCET (WCET should be chosen approximately in range of milliseconds or tens of milliseconds). Hard-real time constraints for loop WCET can be approximately expressed as $WCET = N * T_{SAMPLE}$, meaning that WCET is approximately linearly relaxed as we increase the queue size.

Practical timing example is shown in the figure below. Let us consider FIFO buffer for samples (ADC_BUFFER) that can hold up to 10 samples and consider what happens in each step:

1. ADC_BUFFER is empty; thread code will just resume with loop execution at this point
2. Main thread is busy with other jobs for duration for 3 T_{SAMPLE} (queue was not read for a period of three samples); ISR is called on each ADC conversion in the background and samples are saved in the queue
3. Now thread reaches the point where it gets all the samples from the queue at once; after that operation queue is empty
4. ISR puts a new sample
5. The next loop iteration finishes in less time; the main thread gets one sample in this pass
6. Thread has been busy handling some more demanding process; ISR enqueued 9 samples for later processing but buffer overrun still did not happen (we still have one empty space in the queue)
7. Main thread manages to get all samples at once before queue gets full; time needed to get all samples is typically significantly lower than the loop iteration WCET.



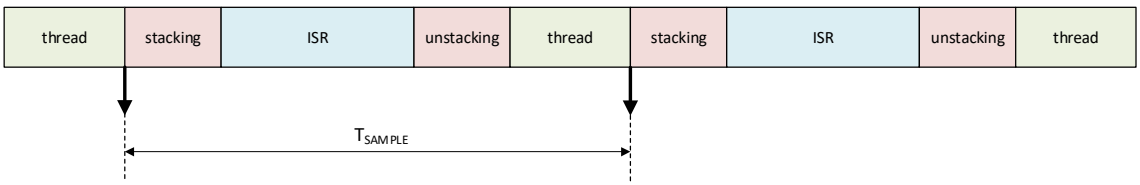
If we want to ensure that the queue in this example does not run into buffer overrun condition, we need to dequeue the samples at least every WCET = (9 x T_{SAMPLE}). If we do not clear the queue in time, we need to discard the samples that we cannot enqueue. However, this will be a violation of hard-real time constraint and the system is not considered to work properly.

Practical tip: WCET of the main loop is very important parameter for ensuring hard real-time embedded application constraints. WCET can be estimated by a technique of "*operation counting*" (i.e. machine code instructions of C compiled code) but in practice it is more convenient to measure the execution speed of some portion of the code is by toggling some GPIO pin at the beginning and at the end of tested code section and observe the response time by oscilloscope.

Non-blocking buffered DMA-based ADC data transfer

The main benefit of using *non-blocking interrupt-based buffered ADC data transfer* is to ease loop iteration WCET requirements *N*-fold, simply by replacing a single *SampleResult* with an *N*-element queue. Although this is a great step forward from polling or interrupt-driven unbuffered approach, one must take into account the limitations when the sampling frequency is very high.

Let us consider the timing diagram what is happening for ISR-based data transfer every time when new ADC result is available:



ADC sampling frequency is determined by a timer that triggers ADC to initiate new conversion. For example, if timer overflow period is set to 1 ms, it will trigger ADC with sampling rate 1 kHz. This happens in hardware without any software intervention so very precise timing may be achieved by this approach.

In interrupt-driven data transfer we need to enter ISR routine every time new ADC conversion result is available. It means that for each new sample ADC ISR will be triggered. Let us consider the following system parameters:

- processor clock frequency: 50 MHz
- interrupt stacking time: 12 cycles (0.24 µs for the chosen clock frequency)
- interrupt service routine processing time: 100 cycles (2 µs)
- interrupt unstacking time: 12 cycles (0.24 µs)
- total time spent on interrupt processing: approx. 2.5 µs

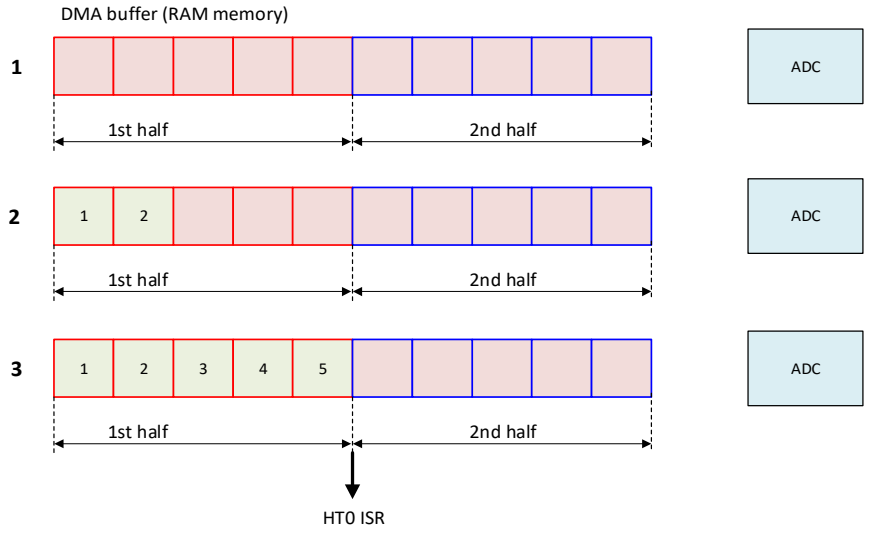
Assuming the sampling frequency 1 kHz, we would spend 2.5 µs each 1 ms to service ADC ISR and the percentage of the time spent inside ISR would be 0.25%, what is negligible processor utilization.

Now let us consider what happens if we have sampling frequency 50 kHz: now we spend 2.5 µs each 20 µs and processor utilization of ISR will rise to 12.5%, what is not negligible load any more.

If we want to use ADC ISR data transfer as explained before for sampling frequency 500 kHz, this is not feasible due to the interrupt processing time larger than sampling time of 2 µs. This means that with ADC ISR data transfer approach it is impossible to utilize full hardware potential of e.g. STM32F407VG microcontroller, which is capable to provide maximum ADC sampling rate 6 MS/s. We obviously need some other, more efficient mechanism when processor utilization time spent on ISR servicing becomes prohibitively high.

For high sampling rate, ADC ISR data transfer is also vulnerable to critical section implementation (enabling and disabling interrupts): if the critical section (time during which ADC ISR interrupt is masked) lasts longer then a sampling period T_{SAMPLE}, some samples will be lost.

DMA data transfer solves efficiently both addressed problems (high processor utilization time and critical sections) by relieving ISR from doing all peripheral-to-memory transfers for each occurrence of the target event (ADC sample conversion completion, in case of the ADC peripheral). The diagram below shows the principle how the DMA transfer of samples from the ADC to the memory buffer in RAM works in principle:



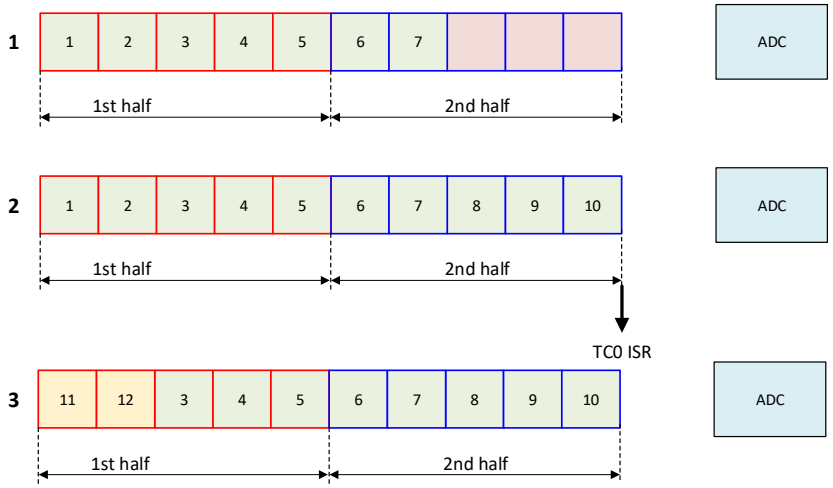
1. First we initialize ADC and DMA controller; if we set ADC to work in DMA mode, ADC *will not trigger ADC ISR* after every completed ADC conversion; ADC will instead transfer each result to the some buffer in RAM memory, which is configured in DMA controller; we just need to tell DMA controller *the starting address* and *length* of the RAM buffer and let DMA do the transfer without intervention of the processor (in this example DMA buffer can hold up to 10 samples)

2. Upon the first ADC conversion, DMA will transfer the result (sample 1) to the start of the buffer, without issuing ADC ISR; DMA controller will automatically increment the pointer to receive the next result (sample 2) after next ADC conversion; at this point two ADC results were transferred without any intervention of processor, i.e. utilization of processor for transferring ADC results was 0 % up to this point
3. The first event of interest happens after DMA buffer is *half-filled*: since the DMA controller knows the length of DMA buffer (10 samples) it triggers *DMA ISR* following the *HT0 event* (i.e. *half-transfer finished event*); this event notifies whatever part of code is in charge to transfer samples 1 - 5 to some *application buffer* (i.e. "normal" buffer in the RAM memory that is not part of the DMA buffer under control of DMA peripheral).

Up to this point we can see what benefits DMA data transfer brings:

- no ISR is triggered after *each* ADC conversion – there is no impact on processor utilization time while DMA transfer is in progress
- ISR is triggered only when half-transfer is done – signaling that we should get the content of the first half of DMA buffer as soon as possible
- for high sampling rate the frequency of entering ISR is drastically lowered – the larger the DMA buffer size is, lower is the DMA ISR triggering frequency.

Why DMA ISR issues an interrupt on half-transfer and does not wait for a full transfer to be finished? Let us consider what happens next:



1. After HT0 triggered DMA ISR was issued, ADC and DMA in the background keep filling the second half of DMA buffer; it means that if we do not respond immediately to HT0 triggered DMA ISR, we still have enough time to process this interrupt request, but no later then the time instant when the second half of DMA buffer becomes full; in this example we see that new samples 6 and 7 are placed on the beginning of the second half of DMA buffer
2. When second half of the DMA buffer becomes full, TC0 event (*transfer complete event*) triggers new DMA ISR; this is the signal to whatever part of the code is responsible that second half of DMA buffer is now full and that the data from it should be pulled out to some "normal" application buffer for further processing (samples 6 - 10)
3. DMA and ADC will just keep filling the DMA buffer in a *circular manner*; it means that after the second half of the DMA buffer is filled, new samples will be fed from the beginning of the first half of DMA buffer again. However, circular DMA transfer assumes that we have already transferred the first half of DMA buffer and that no data will be overwritten before they have been consumed by a main thread loop.

The technique when DMA buffer is divided into two halves is called *double buffering*. This approach is very useful since it lets the part of the program in charge for processing half-buffers enough time to react, before other half of the buffer is filled. In practice, it means that:

- frequency of issued ISR requests in DMA transfer, compared to the ISR transfer without DMA, is (*N/2*) times lower, where *N* is the number of elements in DMA buffer; that can lower the processor utilization due to ISR overhead to the negligible percentage, even for very high sampling frequencies
- main thread iteration WCET for processing the one half of the DMA buffer is approximately *(N/2) * T*, meaning that we achieve similar benefits for buffered processing as for ISR-only data transfer.

Use cases when DMA transfer is mandatory are related to high frequency of incoming ISR requests (for ADC, very high sampling frequency in ISR-only mode). Let us consider again the example with the sampling frequency 50 kHz, but with DMA buffer configured to receive 1000 samples:

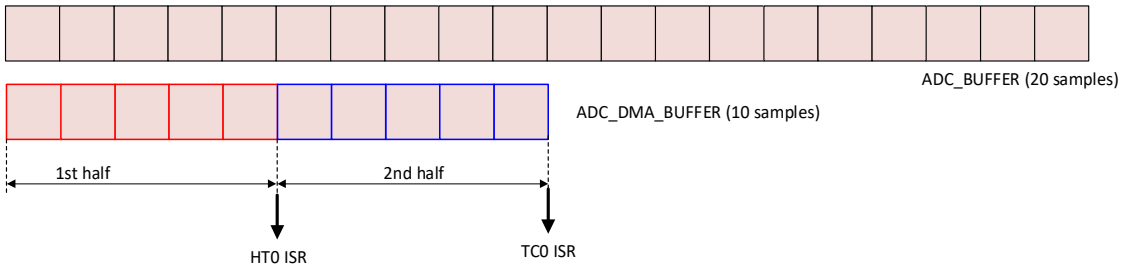
- ISR is issued every 500 samples (size of the half-buffer, HT0/TC0 interrupt source), i.e. every 10 ms
- let us assume that we now work more to transfer all the samples in DMA ISR at once (i.e. the whole half-buffer), 500 x 10 = 5000 operations for a processor clocked on 50 MHz (10 operations per sample, although this figure is probably lower in practice) would give half-buffer transfer time of around 100 µs (stacking and unstacking of around 0.5 µs into DMA ISR could be neglected)
- DMA ISR processor utilization time would be around 1%, what is significantly less than 12.5% in the case of ISR transfer without DMA for sampling frequency 50 kHz.

Under the same assumptions, for sampling rate as high as 500 kHz we would have only 10% of processor utilization time in ISR, while it was impossible to transfer the samples for this sampling rate with ISR transfer without DMA, as we discussed earlier[12].

Implementation of non-blocking DMA-based buffered ADC data transfer

This section will present in details how DMA-based transfer principles described earlier were implemented in our device driver in the example code.

First, we need to define two global arrays to hold samples:



- *ADC_DMA_BUFFER* – buffer in RAM which is filled by ADC samples via DMA transfer, as described earlier; when *ADC_DMA_BUFFER* is half-full, HT0 DMA ISR will be issued and TC0 DMA ISR is triggered when *ADC_DMA_BUFFER* is full; we assume *double buffering* approach, as described before
- *ADC_BUFFER* – buffer to which half-buffers from DMA buffer are copied on each issued HT0/TC0 interrupt; it is recommended to make ADC_BUFFER greater than the size of the

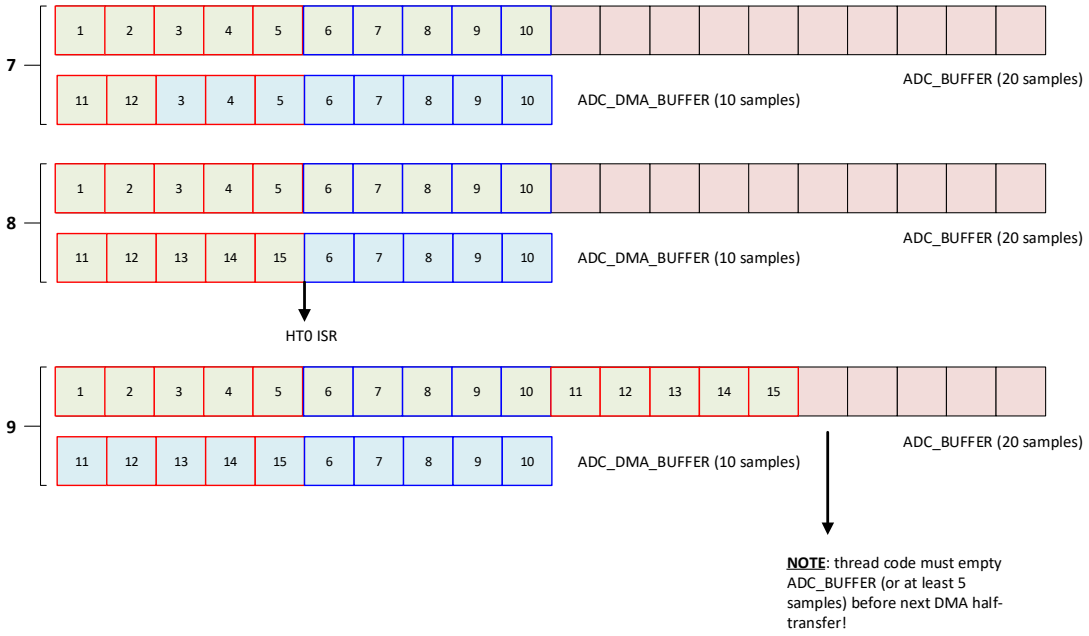
[12] The figures for estimation of processor utilization time for ISR and DMA ADC samples transfer are approximate and not based on any particular processor data; it is recommended to estimate realistic figures either by instruction counting or other practical approach, such as measuring the exact time of execution of some portions of the code using oscilloscope or logic analyzer

whole DMA buffer, to have less strict timing restrictions for emptying the *ADC_BUFFER*; in this example we chose this buffer to have twice the size of *ADC_DMA_BUFFER*

Ultimately, all samples must end up in the main thread where they are being processed. **Main thread should never directly access *ADC_DMA_BUFFER*** because (1) it is under control of DMA controller and (2) it is the responsibility of DMA ISR to transfer the content of *ADC_DMA_BUFFER* to *ADC_BUFFER*. The main thread pulls data from *ADC_BUFFER* queue in the same manner as if this buffer was filled sample by sample from ADC ISR, but without using DMA. It is only important that the main thread continuously pulls data out of the *ADC_BUFFER* in order to prevent buffer overrun when making transfers from *ADC_DMA_BUFFER*.

Let us see an example how samples are transferred between *ADC_DMA_BUFFER* and *ADC_BUFFER*:

1. In this step, some samples are transferred from ADC to the first half of *ADC_DMA_BUFFER* (samples 1 and 2).
2. When the first half of *ADC_DMA_BUFFER* is filled, HT0 DMA ISR is issued.
3. In HT0 DMA ISR we transfer the first half of *ADC_DMA_BUFFER* to *ADC_BUFFER.*
4. Meanwhile, new samples are being fed to the second half of *ADC_DMA_BUFFER* (some samples may have even arrived while HT0 DMA ISR is still active and transferring the first half of *ADC_DMA_BUFFER*)
5. When the second half of *ADC_DMA_BUFFER* is filled, TC0 DMA ISR is issued.
6. In TC0 DMA ISR we transfer the second half of *ADC_DMA_BUFFER* to *ADC_ BUFFER.*



7. Meanwhile, new samples are being fed to the first half of *ADC_DMA_BUFFER* (some samples may have even arrived while TC0 DMA ISR is still active)
8. When the first half of *ADC_DMA_BUFFER* is filled again, HT0 DMA ISR is issued.
9. In HT0 DMA ISR we transfer the first half of *ADC_DMA_BUFFER* to *ADC_ BUFFER*, but we reserved enough space to keep filling *ADC_BUFFER*, although no data have been pulled from it in a meantime.

The main thread is responsible for getting the data from *ADC_BUFFER* in time, in order to prevent buffer overrun by DMA ISR. In device driver implementation, the main thread will pull the samples from the *shared ADC_BUFFER* into a *thread private* (*not shared*) *SAMPLES_BUFFER*. The idea is to make any thread-related application specific processing on a buffer in device driver API to be completely thread-safe to use, without interference with interrupts, critical sections etc.

Above explained principles of non-blocking ISR and DMA-based buffered ADC data transfer approaches were implemented in our custom device driver code. The code description will assume the familiarity with explanations from this chapter.

4.3.6.2 Global variables

```
// buffer for raw samples from ADC (via ISR transfer)
volatile uint16_t  ADC_BUFFER[ADC_BUFFER_SAMPLES_SIZE];
volatile int       ADC_BUFFER_HEAD;
volatile int       ADC_BUFFER_TAIL;
volatile int       ADC_BUFFER_OVERFLOW_FLAG;
// buffer for raw samples from ADC (via DMA transfer)
volatile uint16_t  ADC_DMA_BUFFER[ADC_DMA_BUFFER_SAMPLES_SIZE];
// samples for processing
volatile uint16_t  SAMPLES_BUFFER[SAMPLES_BUFFER_SIZE];
volatile int       SAMPLES_BUFFER_HEAD;
volatile int       SAMPLES_BUFFER_TAIL;
// set only once, in init function, do not change later
int            ADCTransferType;
```

ADC_BUFFER that is used both for ISR and DMA ADC samples transfers

indicator that ADC_BUFFER overrun was detected

ADC_DMA_BUFFER that is filled directly from DMA (active only in DMA mode)

placeholder for application-level samples FIFO (meaning that it should not be shared between application and ISRs)

4.3.6.3 Initialization of GPIO and ADC parameters

```
void ADC_GPIO_Configuration()
{
LL_GPIO_InitTypeDef GPIO_InitStruct = {0};

   // Peripheral clock enable
   LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_ADC1);
   LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOC);

   // ADC1 GPIO Configuration
   // PC0   ------> ADC1_IN10
   GPIO_InitStruct.Pin = LL_GPIO_PIN_0;
   GPIO_InitStruct.Mode = LL_GPIO_MODE_ANALOG;
   GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
   LL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}
void ADC_ADC_Configuration(int transferType)
{
LL_ADC_CommonInitTypeDef ADC_CommonInitStruct = {0};
LL_ADC_InitTypeDef ADC_InitStruct = {0};
LL_ADC_REG_InitTypeDef ADC_REG_InitStruct = {0};

   ADC_CommonInitStruct.Multimode = LL_ADC_MULTI_INDEPENDENT;
   ADC_CommonInitStruct.CommonClock = LL_ADC_CLOCK_SYNC_PCLK_DIV4;
   ADC_CommonInitStruct.MultiTwoSamplingDelay = LL_ADC_MULTI_TWOSMP_DELAY_5CYCLES; // dont care
   LL_ADC_CommonInit(__LL_ADC_COMMON_INSTANCE(ADC1), &ADC_CommonInitStruct);

   ADC_InitStruct.Resolution = LL_ADC_RESOLUTION_12B;
   ADC_InitStruct.DataAlignment = LL_ADC_DATA_ALIGN_RIGHT;
   ADC_InitStruct.SequencersScanMode = LL_ADC_SEQ_SCAN_DISABLE;
   LL_ADC_Init(ADC1, &ADC_InitStruct);

   ADC_REG_InitStruct.TriggerSource = LL_ADC_REG_TRIG_EXT_TIM3_TRGO;
   ADC_REG_InitStruct.SequencerLength = LL_ADC_REG_SEQ_SCAN_DISABLE;
   ADC_REG_InitStruct.SequencerDiscont = LL_ADC_REG_SEQ_DISCONT_DISABLE;
   ADC_REG_InitStruct.ContinuousMode = LL_ADC_REG_CONV_SINGLE; // one conversion per trigger
   if (transferType == ADC_USE_INTERRUPT)
   {
      ADC_REG_InitStruct.DMATransfer = LL_ADC_REG_DMA_TRANSFER_NONE;
   }
   else if (transferType == ADC_USE_DMA)
   {
      ADC_REG_InitStruct.DMATransfer = LL_ADC_REG_DMA_TRANSFER_UNLIMITED; // ring buffering
   }
   LL_ADC_REG_Init(ADC1, &ADC_REG_InitStruct);

   // LL_ADC_REG_SetFlagEndOfConversion(ADC1, LL_ADC_REG_FLAG_EOC_UNITARY_CONV); NOT THIS ONE - ADC
   using sequence!
   LL_ADC_REG_SetFlagEndOfConversion(ADC1, LL_ADC_REG_FLAG_EOC_SEQUENCE_CONV); // otherwise ADC will
   not trigger!

   // Configure Regular Channel
   LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1, LL_ADC_CHANNEL_10);
   LL_ADC_SetChannelSamplingTime(ADC1, LL_ADC_CHANNEL_10, LL_ADC_SAMPLINGTIME_15CYCLES);

   LL_ADC_REG_StartConversionExtTrig(ADC1, LL_ADC_REG_TRIG_EXT_RISING); // set trigger on rising edge

   if (transferType == ADC_USE_INTERRUPT) // if we use INTR instead of DMA
   {
      // enable interrupt
      LL_ADC_EnableIT_EOCS(ADC1);
   }
   else if (transferType == ADC_USE_DMA) // if we use DMA instead of INTR
   {
      // do nothing, enabled with ADC_REG_InitStruct.DMATransfer = LL_ADC_REG_DMA_TRANSFER_LIMITED
   }

   // run ADC
   LL_ADC_Enable(ADC1);
}
```

enable clocks to GPIO port used by ADC1 (pin PC0) and ADC1 itself

configure PC0 as analog input function

sets some basic ADC parameters

sets resolution (12 bits), data alignments

sets that timer TIM3 overflow determines sampling frequency

not using DMA transfer in ISR data transfer mode

continuous data transfer (ring buffer) in in DMA data transfer mode

care must be taken to select right type of EOC trigger!

channel config

enable interrupt request on EOC (end-of-conversion) event in ISR data transfer mode; this still does not enable interrupts!

DMA interrupt request sources are configured in a dedicated DMA configuration function

enables ADC but does not automatically start conversions!

4.3.6.4 Initialization of timer TIM3 for providing sampling frequency

```
void ADC_TIM3_Configuration(int samplingRate)
{
LL_TIM_InitTypeDef TIM_InitStruct = {0};
LL_RCC_ClocksTypeDef RCC_Clocks;
uint32_t APB1_TIMER_CLK, TimerPeriod;

   // Peripheral clock enable
   LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM3);
   LL_TIM_DisableIT_UPDATE(TIM3);

   // calculate TimerPeriod for samplingRate time base
   LL_RCC_GetSystemClocksFreq(&RCC_Clocks);
   APB1_TIMER_CLK = RCC_Clocks.PCLK1_Frequency * 2;
   TimerPeriod = (APB1_TIMER_CLK/ samplingRate ) - 1;

   TIM_InitStruct.Prescaler = 0;
   TIM_InitStruct.CounterMode = LL_TIM_COUNTERMODE_UP;
   TIM_InitStruct.Autoreload = TimerPeriod;
   TIM_InitStruct.ClockDivision = LL_TIM_CLOCKDIVISION_DIV1;
   TIM_InitStruct.RepetitionCounter = 0;
   LL_TIM_Init(TIM3, &TIM_InitStruct);

   LL_TIM_SetClockSource(TIM3, LL_TIM_CLOCKSOURCE_INTERNAL);
   LL_TIM_SetTriggerOutput(TIM3, LL_TIM_TRGO_UPDATE);

   LL_TIM_ClearFlag_UPDATE(TIM3);
   LL_TIM_EnableIT_UPDATE(TIM3);

   // TIM3->CR1 |= TIM_CR1_CEN; // Start timer in forward counting mode
   LL_TIM_EnableCounter(TIM3);
}
```

enable TIM3 clock on APB1

calculate autoreload value (TimerPeriod) to enable timer overflow at samplingRate frequency; the same notes about minimum sampling frequency (1280.76 Hz) apply as described for TIM4 (PWMgenerator)

enable trigger output on update event (triggers A/D conversion)

enables update flag but do not generate interrupt

enables and starts timer

4.3.6.5 ADC and NVIC configuration to enable interrupts

```
void ADC_NVIC_Configuration(int transferType)
{
   if (transferType == ADC_USE_INTERRUPT)
   {
      // ADC1 interrupt Init
      NVIC_SetPriority(ADC_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),0, 5));
      NVIC_EnableIRQ(ADC_IRQn);
   }
   else if (transferType == ADC_USE_DMA)
   {
      // DMA interrupt Init
      NVIC_SetPriority(DMA2_Stream0_IRQn, NVIC_EncodePriority(NVIC_GetPriorityGrouping(),0, 5));
      NVIC_EnableIRQ(DMA2_Stream0_IRQn);
   }
}
```

enable ADC interrupt in ISR data transfer mode; DMA ISR not used

enable DMA interrupt in DMA data transfer mode; ADC ISR not used

4.3.6.6 DMA configuration for case of DMA transfer

This function is called as a part of initialization procedure only if DMA transfer mode is selected in *config.h*:

```
void ADC_DMA_Configuration(void)
{
LL_DMA_InitTypeDef DMA_InitStructure;

   LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_DMA2);

   // ADC1 => DMA2.Channel0!
   DMA_InitStructure.Channel = LL_DMA_CHANNEL_0;
   DMA_InitStructure.MemoryOrM2MDstAddress = (uint32_t)&ADC_DMA_BUFFER;
   DMA_InitStructure.PeriphOrM2MSrcAddress = (uint32_t)&ADC1->DR;
   DMA_InitStructure.Direction = LL_DMA_DIRECTION_PERIPH_TO_MEMORY;
   DMA_InitStructure.NbData = ADC_DMA_BUFFER_SAMPLES_SIZE;   // Count of 16-bit words
   DMA_InitStructure.PeriphOrM2MSrcIncMode = LL_DMA_PERIPH_NOINCREMENT;
   DMA_InitStructure.MemoryOrM2MDstIncMode = LL_DMA_MEMORY_INCREMENT;
   DMA_InitStructure.PeriphOrM2MSrcDataSize = LL_DMA_PDATAALIGN_HALFWORD;
   DMA_InitStructure.MemoryOrM2MDstDataSize = LL_DMA_MDATAALIGN_HALFWORD;
```

```
    DMA_InitStructure.Mode = LL_DMA_MODE_CIRCULAR;
    DMA_InitStructure.Priority = LL_DMA_PRIORITY_HIGH;
    DMA_InitStructure.FIFOMode = LL_DMA_FIFOMODE_ENABLE;
    DMA_InitStructure.FIFOThreshold = LL_DMA_FIFOTHRESHOLD_1_2;
    DMA_InitStructure.MemBurst = LL_DMA_MBURST_SINGLE;
    DMA_InitStructure.PeriphBurst = LL_DMA_PBURST_SINGLE;
    LL_DMA_Init(DMA2, LL_DMA_STREAM_0, &DMA_InitStructure);

    // Enable DMA Stream Half / Transfer Complete interrupt
    LL_DMA_EnableIT_TC(DMA2, LL_DMA_STREAM_0);
    LL_DMA_EnableIT_HT(DMA2, LL_DMA_STREAM_0);

    // DMA2_Stream0 enable
    LL_DMA_EnableStream(DMA2, LL_DMA_STREAM_0);
}
```

Some notes about DMA initialization:

- *Channel = LL_DMA_CHANNEL_0*
  - use DMA channel 0 to transfer data
- *MemoryOrM2MDstAddress = (uint32_t)&ADC_DMA_BUFFER;*
  - address of the first memory location of the buffer in RAM where DMA must transfer data from peripheral
- *PeriphOrM2MSrcAddress = (uint32_t)&ADC1->DR*
  - peripheral address that sources data to DMA (data register of ADC holding each conversion result)
- *Direction = LL_DMA_DIRECTION_PERIPH_TO_MEMORY*
  - we transfer data from peripheral to memory
- *NbData = ADC_DMA_BUFFER_SAMPLES_SIZE*
  - number of *data elements* (not bytes!) to be stored in DMA buffer
- *PeriphOrM2MSrcIncMode = LL_DMA_PERIPH_NOINCREMENT*
  - do note increment peripheral address after each DMA transfer (i.e. we always read data from the same data register in ADC)
- *MemoryOrM2MDstIncMode = LL_DMA_MEMORY_INCREMENT*
  - after each DMA transfer we increment memory location in DMA buffer in RAM
- *PeriphOrM2MSrcDataSize = LL_DMA_PDATAALIGN_HALFWORD*
- *MemoryOrM2MDstDataSize = LL_DMA_MDATAALIGN_HALFWORD*
  - data size of object we transfer from peripheral (used in conjunction with *.NbData* to know the actual size of DMA buffer in bytes)
- *Mode = LL_DMA_MODE_CIRCULAR*
  - use DMA buffer like circular buffer
- *FIFOMode = LL_DMA_FIFOMODE_ENABLE*
  - DMA in FIFO mode
- *FIFOThreshold = LL_DMA_FIFOTHRESHOLD_1_2*
  - we use half-transfer indicator (double buffering)
- *MemBurst = LL_DMA_MBURST_SINGLE*
- *PeriphBurst = LL_DMA_PBURST_SINGLE*
  - we transfer data one object at the time

The DMA configuration is set up to conform with elaborated description how we use the DMA in our device driver, although DMA may be used in some different configurations, if needed.

4.3.6.7 ADC initialization function

All previously described functions for partial configuration of ADC are called by the main ADC configuration function:

```
void ADC_SystemInit(int samplingRate, int transferType)
{
   #if (ADC_DEBUG_LED1)
   LED_User_Init(); // just in case, if LED GPIO output was not enabled somewhere else
   #endif

   // init global variables
   ADCTransferType = transferType;
   ADC_BUFFER_HEAD = 0;
   ADC_BUFFER_TAIL = 0;
   ADC_BUFFER_OVERFLOW_FLAG = 0;
   SAMPLES_BUFFER_HEAD = 0;
   SAMPLES_BUFFER_TAIL = 0;

   ADC_GPIO_Configuration();
   ADC_NVIC_Configuration(transferType);
   if (transferType == ADC_USE_DMA)
   {
      // only if DMA transfer is selected!
      ADC_DMA_Configuration();
   }
   ADC_TIM3_Configuration(samplingRate);
   ADC_ADC_Configuration(transferType);

   // start first conversion manually; otherwise, ADC will not start at all!
   LL_ADC_REG_StartConversionSWStart(ADC1);
}
```

initialize all global variables

first ADC conversion must be explicitely started!

ADC configuration function *ADC_SystemInit* takes two parameters:

- *samplingRate* – sampling frequency provided by TIM3; one must take care that under the current device driver configuration this frequency must be higher than 1280.76 Hz
- *transferType* – valid constants for this parameter are defined in *config.h* (ADC_USE_INTERRUPT, ADC_USE_DMA); the function will automatically call the appropriate configuration functions to set up either ISR or DMA data transfer mode

4.3.6.8 ADC working in ISR transfer mode

ADC_SystemInit function will automatically set up ADC ISR if ADC_USE_INTERRUPT parameter is provided. ADC ISR is triggered on EOC (*end of conversion*) event. ADC ISR callback implements *non-blocking interrupt-based buffered ADC data transfer*, as it was described earlier:

```
void ADC_IRQHandler_Callback()
{
uint16_t result;
static int adc_buffer_head;

   #if (ADC_DEBUG_LED1)
   LED_User_Toggle(LED1);
   #endif

   // Clear ADC1 EOC pending interrupt bit
   LL_ADC_ClearFlag_EOCS(ADC1);

   // get ADC value
   result = LL_ADC_REG_ReadConversionData12(ADC1);
```

this pin provides toggling output on each finished ADC conversion; it may be observed by oscilloscope or by internal frequency meter (via GPIO counter) to check whether ADC sampling frequency is valid

reset ISR flag

read ADC conversion result and store it into local variable

```
   adc_buffer_head = ADC_BUFFER_HEAD + 1;
   if (adc_buffer_head == ADC_BUFFER_SAMPLES_SIZE) adc_buffer_head = 0;
   if (adc_buffer_head != ADC_BUFFER_TAIL)
   {
      // adding new sample will not cause buffer overrun
      ADC_BUFFER[ADC_BUFFER_HEAD] = result;
      ADC_BUFFER_HEAD = adc_buffer_head;  // update head
   }
   else
   {
      ADC_BUFFER_OVERFLOW_FLAG = 1;// notify thread that buffer overflow event occured
   }
}
```

enqueue new sample into ADC_BUFFER FIFO but with checking will it result in overrun; do not put sample in FIFO in this case!

4.3.6.9 ADC working in DMA transfer mode

ADC_SystemInit function will automatically set up DMA ISR if ADC_USE_DMA parameter is provided. DMA ISR is triggered either on HT0 (*half transfer*) or TC0 (*transfer complete*) event. DMA ISR callback implements *non-blocking DMA-based buffered ADC data transfer*, as it was described earlier:

```
void DMA2_Stream0_IRQHandler_Callback ()
{
uint16_t result;
int ADC_BUFFER_FreeSpace;
int i;

   // DMA Stream Half Transfer interrupt
   if (LL_DMA_IsActiveFlag_HT0(DMA2))
   {
      // Clear DMA Stream Half Transfer interrupt pending bit
      LL_DMA_ClearFlag_HT0(DMA2);
      // transfer data from DMA buffer (1st half) to ADC_BUFFER FIFO
      // (1) check space
      ADC_BUFFER_FreeSpace = ADC_BUFFER_SAMPLES_SIZE - ADC_Buffer_InBufferCount() - 3; // 3 for some
                                                                                       redundancy
      // (2) enqueue 1st half of DMA buffer
      if (ADC_BUFFER_FreeSpace >= ADC_DMA_BUFFER_SAMPLES_HALFSIZE)
      {
         for(i = 0; i < ADC_DMA_BUFFER_SAMPLES_HALFSIZE; i++)
         {
            ADC_BUFFER[ADC_BUFFER_HEAD] = ADC_DMA_BUFFER[i];
            ADC_BUFFER_HEAD++;
            if (ADC_BUFFER_HEAD == ADC_BUFFER_SAMPLES_SIZE) ADC_BUFFER_HEAD = 0;
         }
      }
      else
      {
         ADC_BUFFER_OVERFLOW_FLAG = 1;
      }
   }

   // DMA Stream Transfer Complete interrupt
    if(LL_DMA_IsActiveFlag_TC0(DMA2))
   {
      #if (ADC_DEBUG_LED1)
      LED_User_Toggle(LED1);
      #endif
      // Clear DMA Stream Transfer Complete interrupt pending bit
      LL_DMA_ClearFlag_TC0(DMA2);
      // transfer data from DMA buffer (2nd half) to ADC_BUFFER FIFO
      // (1) check space
      ADC_BUFFER_FreeSpace = ADC_BUFFER_SAMPLES_SIZE - ADC_Buffer_InBufferCount() - 3; // 3 for some
                                                                                       redundancy
      // (2) enqueue 1st half of DMA buffer
      if (ADC_BUFFER_FreeSpace >= ADC_DMA_BUFFER_SAMPLES_HALFSIZE)
      {
         for(i = 0; i < ADC_DMA_BUFFER_SAMPLES_HALFSIZE; i++)
         {
            ADC_BUFFER[ADC_BUFFER_HEAD] = ADC_DMA_BUFFER[i + ADC_DMA_BUFFER_SAMPLES_HALFSIZE];
            ADC_BUFFER_HEAD++;
            if (ADC_BUFFER_HEAD == ADC_BUFFER_SAMPLES_SIZE) ADC_BUFFER_HEAD = 0;
         }
      }
}
```

half-transfer event triggered DMA ISR

transfer 1st half of DMA buffer into ADC_BUFFER, with overrun check

signal ADC_BUFFER overrun to main thread, if detected

transfer-complete event triggered DMA ISR

this pin provides toggling output on each **full DMA buffer transmission**; it may be observed by oscilloscope or by internal frequency meter (via GPIO counter) to check whether ADC sampling frequency and DMA transfers are valid (observed frequency must be N times smaller than sampling frequency, where N is DMA buffer size)

transfer 2nd half of DMA buffer into ADC_BUFFER, with overrun check

```
        else
        {
            ADC_BUFFER_OVERFLOW_FLAG = 1;   ——— signal ADC_BUFFER overrun to main thread, if detected
        }
    }
}
```

4.3.6.10 ADC_BUFFER access API functions

ADC_BUFFER is a FIFO buffer implemented as a plain C array, with head and tail pointers handled manually. It is accessible from both thread and ISR code. However, the application code must not access this structure directly, since it must be handled in atomic way.

Thread-safe API is provided to be called from application thread code without worrying about handling the critical sections and mutual exclusion mechanisms. First, two helper functions for implementation of critical sections accessing *ADC_BUFFER* are provided:

```
void ADC_ENTER_CRITICAL()
{
    if (ADCTransferType == ADC_USE_INTERRUPT)
    {
        NVIC_DisableIRQ(ADC_IRQn);
    }
    else if (ADCTransferType == ADC_USE_DMA)
    {
        NVIC_DisableIRQ(DMA2_Stream0_IRQn);
    }
}

void ADC_EXIT_CRITICAL()
{
    if (ADCTransferType == ADC_USE_INTERRUPT)
    {
        NVIC_EnableIRQ(ADC_IRQn);
    }
    else if (ADCTransferType == ADC_USE_DMA)
    {
        NVIC_EnableIRQ(DMA2_Stream0_IRQn);
    }
}
```

These critical functions will disable only the interrupts that may access the shared resource *ADC_BUFFER*, taking into account which transfer type is selected. Application code may safely call functions to interact with *ADC_BUFFER*:

```
void ADC_Buffer_Clear()
{
    ADC_ENTER_CRITICAL();
    ADC_BUFFER_HEAD = 0;
    ADC_BUFFER_TAIL = 0;
    ADC_BUFFER_OVERFLOW_FLAG = 0;
    ADC_EXIT_CRITICAL();
}
```

clear the buffer, in atomic way within critical section

```
int ADC_Buffer_InBufferCount()
{
int head, tail;

    ADC_ENTER_CRITICAL();
    head = ADC_BUFFER_HEAD;
    tail = ADC_BUFFER_TAIL;
    ADC_EXIT_CRITICAL();
    if (head > tail)
    {
        return (head - tail);
    }
    else if (head < tail)
    {
        return (ADC_BUFFER_SAMPLES_SIZE - (tail - head ));
    }
    return 0;
}
```

query for number of elements currently contained in ADC_BUFFER, in atomic way

```
int ADC_Buffer_Dequeue(uint16_t* result)
{
int ret;

   ret = 0;
   *result = 0;
   ADC_ENTER_CRITICAL();
   if (ADC_BUFFER_HEAD != ADC_BUFFER_TAIL)
   {
      *result = ADC_BUFFER[ADC_BUFFER_TAIL];
      ADC_BUFFER_TAIL++;
      if (ADC_BUFFER_TAIL == ADC_BUFFER_SAMPLES_SIZE) ADC_BUFFER_TAIL = 0;
      ret = 1;
   }
   ADC_EXIT_CRITICAL();
   return ret;
}
```

dequeue single sample from ADC_BUFFER, in atomic way; returns 1 if element was popped, 0 otherwise (element is returned via byref parameter)

```
int ADC_Buffer_GetOverflowFlag()
{
int value;

   ADC_ENTER_CRITICAL();
   value = ADC_BUFFER_OVERFLOW_FLAG;
   ADC_EXIT_CRITICAL();
   return value;
}
```

polls for overflag flag, in atomic way

```
void ADC_Buffer_ClearOverflowFlag()
{
   ADC_ENTER_CRITICAL();
   ADC_BUFFER_OVERFLOW_FLAG = 0;
   ADC_EXIT_CRITICAL();
}
```

clears overflag flag, in atomic way

Device driver also provides functions for handling *SAMPLES_BUFFER*, a storage that may be used safely by a thread application code for manipulation of samples once they are pulled from *ADC_BUFFER*:

```
int ADC_Samples_InBufferCount();
int ADC_Samples_Enqueue(uint16_t result);
int ADC_Samples_Dequeue(uint16_t* result);
```

Implementation of these functions is very similar to API function for handling *ADC_BUFFER*, but without critical sections, since they are not needed as we do not access *SAMPLES_BUFFER* from any interrupt service routine, only thread code.

4.3.7 Testing the upgraded device drivers

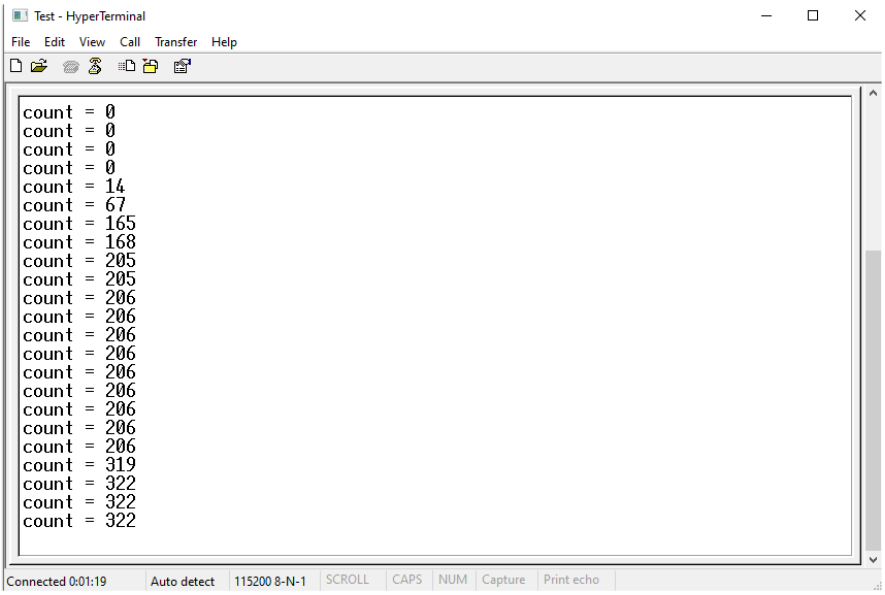
Testing GPIO pulse counter functionality

```
void GPIO_Interrupt_Test()
{
int count;
char msg[32];

   InitDeviceCommon();
   LED_User_Init();
   Timer2_Init();
   USART3_Init();
   GPIO_Pulse_Counter_Init();

   LED_User_Off(LED1);
   while(1)
   {
      Timer2_WaitMillisec(1000);
      count = GPIO_Pulse_Counter_Read();
      sprintf(msg, "count = %d", count);
      USART3_WriteLine(msg);
   }
}
```

initialize all peripherals used in this test; although it is a good practice to make isolated tests, it is handy here for a better feedback to user via serial port terminal to include the functionality of already tested peripherals, such as USART3 and TIM2

GPIO interrupt will count positive edge triggers on PB15 (for example, we can touch 3.3 V (VDD) on STM32F4DISCOVERY board via wire to make periodic contact with PB15); each second the total number of counted pulses will be sent to serial port; also, if PULSE_COUNTER_ENABLE_DEBUG_LED2 flag in *config.h* is enabled, we can see LED1 toggling on each positive edge on PB15

It is also interested to examine serial output:



We can see that on each touch we get high number of counted pulses on PB15 – this is because of contact mechanical bounces in millisecond time scale and not debouncing GPIO input in this simple implementation of the device driver.

Testing ADC ISR and validating the sampling frequency

We can run a simple test to verify that ADC ISR is triggered on the end of each ADC conversion and that samples are collected with defined sampling frequency:

```
void ADC_Test_Intr()
{
   InitDeviceCommon();
   ADC_SystemInit(4000, ADC_USE_INTERRUPT);
   while(1);
}
```

This test will initialize ADC in ISR transfer mode and run ADC with 4 kS/s sampling rate. We can set a breakpoint inside ISR to check whether our code during the execution enters ADC ISR:

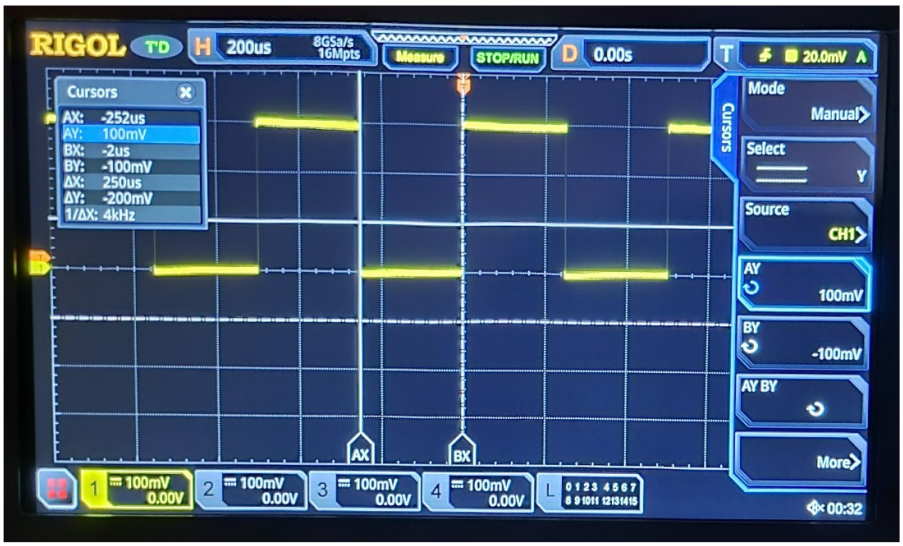
```
178 void ADC_IRQHandler_Callback()
179 {
180 uint16_t result;
181 static int adc_buffer_head;
182
183     #if (ADC_DEBUG_LED1)
184     LED_User_Toggle(LED1);
185     #endif
186
187     // Clear ADC1 EOC pending interrupt bit
188     LL_ADC_ClearFlag_EOCS(ADC1);
189
190     // get ADC value
191     result = LL_ADC_REG_ReadConversionData12(ADC1);
192
193     adc_buffer_head = ADC_BUFFER_HEAD + 1;
194     if (adc_buffer_head == ADC_BUFFER_SAMPLES_SIZE) adc_buffer_head = 0;
195     if (adc_buffer_head != ADC_BUFFER_TAIL)
196     {
197         // adding new sample will not cause buffer overrun
198         ADC_BUFFER[ADC_BUFFER_HEAD] = result;
199         ADC_BUFFER_HEAD = adc_buffer_head;  // update head
200     }
201     else
202     {
203         ADC_BUFFER_OVERFLOW_FLAG = 1;   // notify thread that buffer overflow event occured
204     }
205 }
```

If everything is set up correctly, our code should be stopped by hitting the breakpoint in ADC ISR.

Moreover, we also want to ensure that the sampling frequency is correctly set up. We can check this by enabling ADC_SYNC TEST (PD12) GPIO pin toggling in ADC ISR in *config.h*

```
// enable toggle LED1 (PD12) for ADC sampling freq./DMA transfer freq. debugging
#define ADC_DEBUG_LED1       1
```

and observing PD12 pin by oscilloscope:



We can see that pin toggling frequency measured on pin PD12 is 4 kHz.

Testing ADC DMA data transfer

To test ADC operation in DMA data transfer mode we need to use other test function:

```
void ADC_Test_DMA()
{

   InitDeviceCommon();
   ADC_SystemInit(2000, ADC_USE_DMA);
   while(1);
}
```

This function will set the sampling frequency 2 kS/s and assume that the size of DMA buffer is 100, as defined in *config.h*:

```
#define ADC_DMA_BUFFER_SAMPLES_SIZE       100
```

First check that DMA transfer is working is to set a breakpoint in DMA ISR like it was described for ADC ISR. However, we also want to check that timing is correct when we use DMA transfer. In order to do so, we can toggle the pin when a single complete DMA transfer is done (TC0 event in DMA ISR):

```
void DMA2_Stream0_IRQHandler_Callback()
{
uint16_t result;
int ADC_BUFFER_FreeSpace;
int i;

   // DMA Stream Half Transfer interrupt
   if (LL_DMA_IsActiveFlag_HT0(DMA2))
   {
      // Clear DMA Stream Half Transfer interrupt pending bit
      LL_DMA_ClearFlag_HT0(DMA2);
      ...
   }

   // DMA Stream Transfer Complete interrupt
    if(LL_DMA_IsActiveFlag_TC0(DMA2))
   {
```

```
        #if (ADC_DEBUG_LED1)
        LED_User_Toggle(LED1);
        #endif
        // Clear DMA Stream Transfer Complete interrupt pending bit
        LL_DMA_ClearFlag_TC0(DMA2);
        ...
    }
}
```

If the sampling frequency is 2 kHz and we have DMA buffer size 100, then we should see pin PD12 toggled every 50 ms, what we can check with oscilloscope:

Testing the PWM modulator and measuring the PWM frequency with GPIO pulse counter

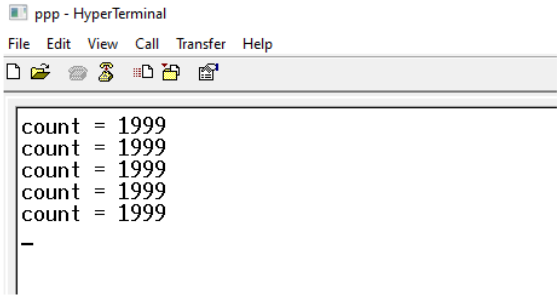
The code for testing PWM modulator is implemented in *PWM_Test* function:

```
// note: this function may be used for measuring PWM frequency, but also for ADC ISR/DMA frequency via
PD12 ADC SYNC pin!
void PWM_Test(void)
{
int count;
char msg[32];

    InitDeviceCommon();
    LED_User_Init();
    LED_User_Off(LED2);
    Timer2_Init();
    USART3_Init();
    GPIO_Pulse_Counter_Init();

    PWM_Start(2000, 50);
    while(1)
    {
        GPIO_Pulse_Counter_Reset();
        Timer2_WaitMillisec(1000);
        count = GPIO_Pulse_Counter_Read();
        sprintf(msg, "count = %d", count);
        USART3_WriteLine(msg);
        LED_User_Toggle(LED2);
    }
}
```

This test function provides more advanced functionality then simple test functions described before. Although we could only initialize PWM (TIM4) and measure PWM signal directly by the oscilloscope, this test function also include the use of TIM2 timer services (for measuring time intervals), USART3 (for outputting the results via serial port) and GPIO pulse counter (for measuring the PWM frequency with microcontroller, without a need for oscilloscope).

If we measure the output on the on pin PB4 (TIM4 PWM output), we should see PWM signal with frequency 2 kHz and duty cycle 50% on oscilloscope:

However, if the oscilloscope is not available, we can connect PB4 pin (TIM4 PWM output) to PB15 (GPIO input for counting pulses) with wire and observe the serial output. In the main loop we reset pulse count and let the GPIO interrupt measure the number of pulses during 1000 ms (while we block the main thread). Number of pulses equals the frequency measured in the time interval of one second and the measured frequency is sent to the serial output:



This test function may be also used to measure the sampling frequency in ADC ISR data transfer mode and correct timing in ADC DMA data transfer mode by connecting ADC_SYNC TEST pin (PD12) to the GPIO input for counting pulses (PB15), if oscilloscope is not available.

4.4 Final application example (app2)

The code for final application is presented below. Depending on the configuration parameter in *config.h*:

```
#define APP2_ISR1
#define APP2_DMA0
```

the appropriate version of demo application is built (either based on ADC ISR or DMA data transfer):

```
void main_app()
{
#if (RUN_TESTS)
   run_test();
#endif

#if (RUN_APP)
   #if (APP1)
   app1();
   #endif
   #if (APP2_ISR)
   app2(20000, 2000, 50, ADC_USE_INTERRUPT);
   #endif
```

```
   #if (APP2_DMA)
   app2(20000, 2000, 50, ADC_USE_DMA);
   #endif
#endif

   while(1);
}
```

Driver function for sample application

```
void app2(int samplingFrequency, int pwmFrequency, int dutyCycle, int adcTransferMode)
```

takes the following parameters:

- *samplingFrequency* – sampling frequency, in Hz (20 kHz for both examples)
- *pwmFrequency* – PWM output frequency, in Hz (2 kHz for both examples)
- *dutyCycle* – PWM output duty cycle (0-100, 50% for both examples)
- *adcTransferMode* – data transfer mode (ADC ISR or DMA, as defined by a parameter in *config.h*)

Upon reset, the application will initialize peripherals (push button, GPIO pulse counter, PWM, USART3 and ADC). It will also output the selected parameters on a serial terminal (information about sampling frequency, PWM frequency, duty cycle and selected ADC transfer mode). Program will also check whether sampling frequency is within correct range. During the ADC initialization, ISR or DMA data transfer mode will be set. Program then waits for user input via pushbutton and after pushbutton is pressed it collects 1000 samples. After all samples are collected, they are written in a human readable form via serial output, prompting the user for next pushbutton press to initiate a new sampling cycle:

```
// ADC example with ISR/DMA transfer and samples buffering (both in ISR/DMA and on application code
side)
void app2(int samplingFrequency, int pwmFrequency, int dutyCycle, int adcTransferMode)
{
int i, samplesCountToCollect, samplesCollected, samplingInProgress;
uint16_t sample;
char msg[64];

   ////////////////////////////////////////////////////////////////
   // hardware initialization
   ////////////////////////////////////////////////////////////////
   InitDevice();   // common - serial port at least for error messages
   // additional hardware initialization - application specific
   PushButton_Init();
   GPIO_Pulse_Counter_Init();            // pulse counter for testing the sampling freq.
   PWM_Start(pwmFrequency, dutyCycle);   // reference signal to be sampled (Freq, DutyCycle)

   ////////////////////////////////////////////////////////////////
   // welcome message and info
   ////////////////////////////////////////////////////////////////
   USART3_WriteLine("*** ADC sample ***");
   sprintf(msg, "Sampling frequency = %d Hz", samplingFrequency);   USART3_WriteLine(msg);
   sprintf(msg, "PWM frequency = %d Hz", pwmFrequency); USART3_WriteLine(msg);
   sprintf(msg, "Duty cycle = %d %%", dutyCycle); USART3_WriteLine(msg);
   sprintf(msg, "ADC transfer mode = %d", adcTransferMode); USART3_WriteLine(msg);
   USART3_WriteLine("");
   USART3_WriteLine("Press pushbutton to start new sampling sequence...");
   USART3_WriteLine("");

   ////////////////////////////////////////////////////////////////
   // error checking
   ////////////////////////////////////////////////////////////////
   if (samplingFrequency < 1285)// minimum freq 1280.76 Hz due to 16-bit autoreaload register in TIM3
   {
      sprintf("Error: samplingFrequency = %d < 1285 Hz", samplingFrequency);
      ErrorMessage(msg);
   }
   if (pwmFrequency < 1285)     // minimum freq 1280.76 Hz due to 16-bit autoreaload register in TIM4
   {
      sprintf("Error: pwmFrequency = %d < 1285 Hz", pwmFrequency);
```

initialize all peripherals used in this app, except ADC

output information about example configuration

```
        ErrorMessage(msg);
    }
    if ((dutyCycle < 0) || (dutyCycle > 100))
    {
        sprintf("Error: invalid dutyCycle = %d", dutyCycle);
        ErrorMessage(msg);
    }

    ////////////////////////////////////////////////////////////////
    // set up application
    ////////////////////////////////////////////////////////////////
    samplesCountToCollect = 1000;   // how many samples to collect before stopping the measurement
    samplingInProgress = 0;
    // start ADC
    switch(adcTransferMode)
    {
        case ADC_USE_INTERRUPT:
            ADC_SystemInit(samplingFrequency, ADC_USE_INTERRUPT);
            break;
        case ADC_USE_DMA:
            ADC_SystemInit(samplingFrequency, ADC_USE_DMA);
            break;
        default:
            sprintf("Error: invalid ADC transfer mode = %d", adcTransferMode);
            ErrorMessage(msg);
            break;
    }

    // start main processing infinite loop
    while(1)
    {
        // check for pushbutton pressed
        if (!samplingInProgress)
        {
            if (PushButton_GetState() == PUSH_BUTTON_PRESSED)
            {
                samplingInProgress = 1;
            }
        }
        if (samplingInProgress)
        {
            // initiate samples acquisition
            ADC_Samples_Clear();
            samplesCollected = 0;
            while(samplesCollected < samplesCountToCollect)
            {
                if (ADC_Buffer_InBufferCount() > 0)
                {
                    samplesCollected++;
                    ADC_Buffer_Dequeue(&sample);
                    ADC_Samples_Enqueue(sample);
                }
            }
            // write collected samples on serial port
            USART3_WriteLine("*** START ***");
            for(i = 0; i < samplesCountToCollect; i++)
            {
                ADC_Samples_Dequeue(&sample);
                sprintf(msg, "%d", sample);
                USART3_WriteLine(msg);
            }
            USART3_WriteLine("*** END ***");
            // prepare for next cycle
            samplingInProgress = 0;
            USART3_WriteLine("Press pushbutton to start new sampling sequence...");
        }
        // don't forget to clear ADC FIFO, otherwise it may overrun
        ADC_Buffer_Clear();
    }
}
```

choose an appropriate ADC data transfer mode

main loop

non-blocking poll of push button

initialize DAQ procedure to collect 1000 samples

collect samples within a loop, blocking the main thread until all samples are collected

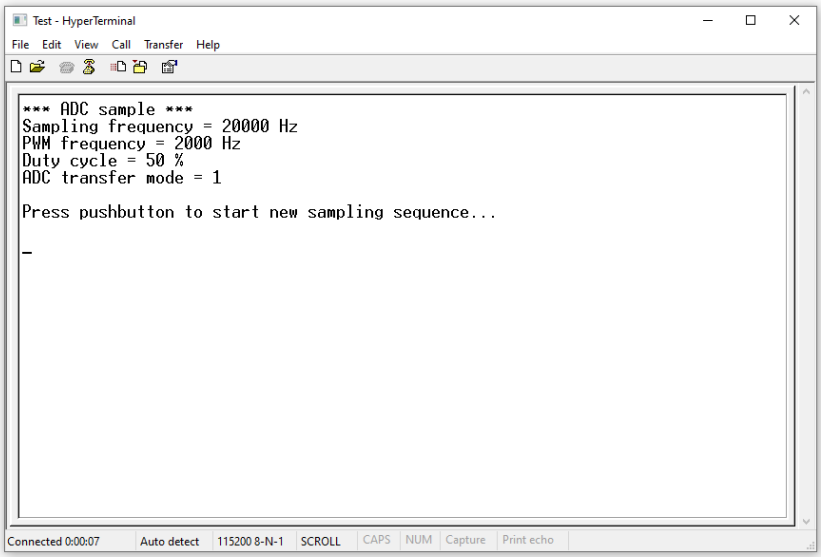
output all sample values via serial port (blocking serial output)

prepare for the next cycle

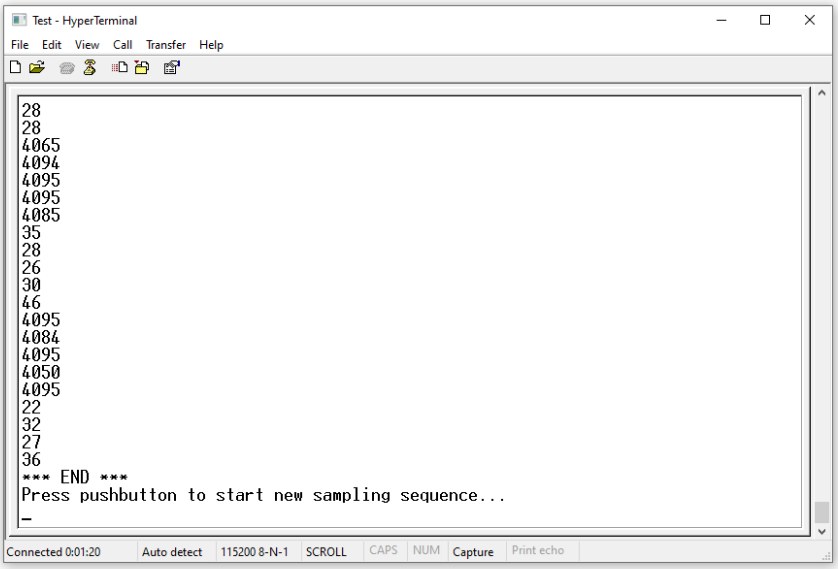
reset ADC buffer – overrun occured because ADC buffer was not timely dequeued during blocking serial output

An example of program output is shown below:

1. Waiting for user to press pushbutton



2. Samples output



One can see how the sampled sequence looks like in a terminal window:

- since we sample PWM signal, it is either HI (VDD) or LO (0 V)
- maximum value for 12-bit ADC conversion is 4095 and samples during HI period od PWM signal are close to the maximum ADC value
- during LO phase samples are close to 0
- we have sampling frequency 10x PWM frequency and we can notice that ADC is working correctly because we have 5 samples for HI phase and 5 samples for LO phase, for a PWM signal with duty cycle 50%
- the resulting output must be the same for both ISR and DMA data transfer approach.

5 Building a simple real-time DAQ application with FreeRTOS (app3)

This chapter will show how to develop a simple real-time DAQ application that uses benefits of multitasking environment provided by the RTOS to enable easier embedded development of applications with hard-real time constraints. The example will use FreeRTOS and illustrate the steps how to incorporate the operating system into the existing bare-metal project.

The example is based on the *app2* sample and shows how to upgrade the existing application to the multitasking environment running two tasks, each having different responsibility: one collecting the samples and the other doing signal processing and managing the serial console output.

The reference code for the example that will be described in this chapter can be found in archive: **OPPURS-Example3-FreeRTOS.rar.**

5.1 Application functionality specification

Final application has to provide the following functionality:

- depending on the configuration in *config.h*, ADC can operate in one of data transfer modes:
  - ADC data transfer in *ISR mode*
  - ADC data transfer in *DMA mode*
- initialize USART3 (115200 bps, 8, N, 1) and GPIO outputs for on-board LEDs
- initialize and run PWM generator (controlled by TIM4) which generates a PWM signal on a pin PB6 (as explained for *app2* example)
- initialize the ADC to collect samples with chosen sampling rate (controlled by TIM3) and transfer data either via ADC or DMA ISR

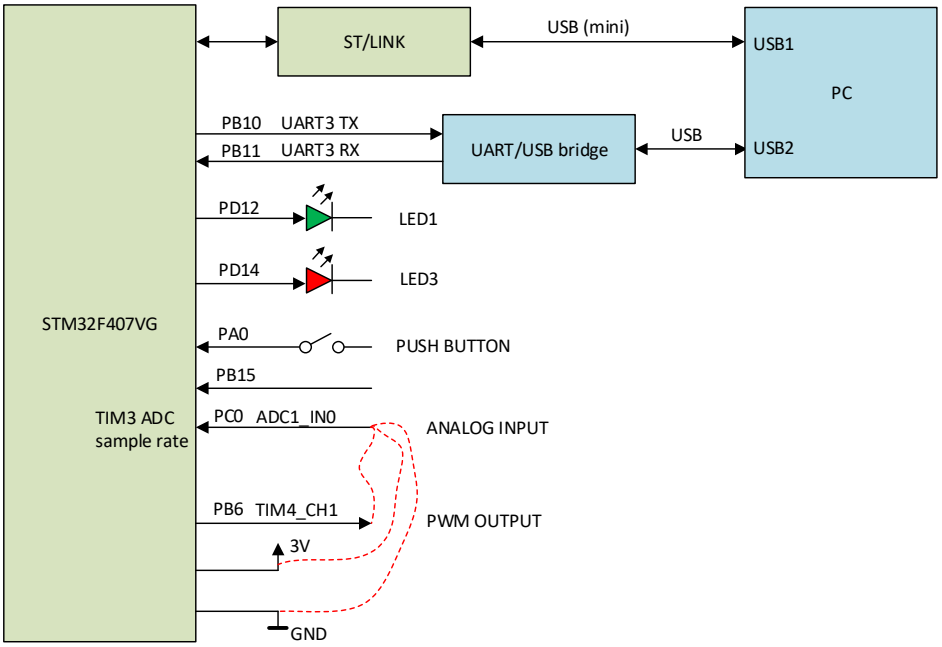
Program is divided into two tasks:

- *task_Sampling* – the purpose of this task is to periodically empty internal ADC buffer; all samples are pulled out from the ADC buffer and sent via FreeRTOS message queue to the signal processing task running in the background
- *task_SerialOutput* – this task has two purposes:
  - processing received samples in a real-time (averaging)
  - sending the output to the serial console

task_Sampling receives samples *continuously*, after the program is started (in contrast to *app2* example, where user had to push the button to start the sampling process and collect finite number of samples (1000)). Unlike *app2*, *no samples* are discarded or lost in a data acquisition process (i.e. in *app2* sampling process is stopped after 1000 samples). *All* samples from the ADC are forwarded to the *task_SerialOutput* for further processing.

task_SerialOutput does very simple signal processing task: averaging samples within a time window of 1 second and outputting the last averaged value to the serial console.

The block diagram of peripherals and the system connections on STM32F4DISCOVERY board are shown in the figure below:

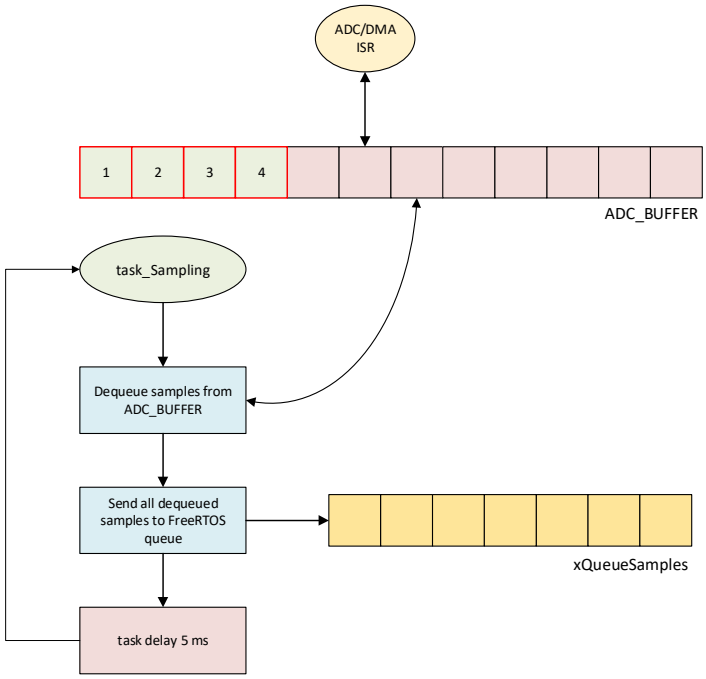


The sampling rate is set to 20 kHz and PWM output to 2 kHz, with a duty cycle 50 %.

5.2 Functionality implementation via multitasking

5.2.1 task_Sampling

The flowchart below describes the functionality of *task_Sampling* task:



Depending on the startup configuration, ADC operates either in ISR or DMA data transfer mode. Samples are acquired in the same manner via ADC or DMA ISR, as described in *app2* example. In both modes samples are stored in an internal ADC_BUFFER (FIFO), which is implemented using plain C array (not FreeRTOS queue). ADC_BUFFER is continuously enqueued with new samples via ADC or DMA ISR. Considering that the size of ADC_BUFFER is 200, for sampling rate of 20 kHz ADC_BUFFER FIFO will overflow if not cleared within a period of 10 ms.

task_Sampling periodically dequeues the samples from ADC_BUFFER FIFO. The period in which the task is emptying the FIFO is approximately 5 ms, meaning that buffer should be around at 50% of its capacity for each dequeue event. ADC_BUFFER FIFO is a global resource that can be accessed from:

- *task_Sampling* task code
- ADC/DMA ISR code

Therefore, access to the ADC_BUFFER FIFO must be implemented in atomic manner (i.e. ISR temporarily disabled when task code is accessing ADC_BUFFER FIFO[13]).

Dequeue block gets *all* samples waiting in ADC_BUFFER FIFO, thus ensuring that FIFO clearing period is less than 10 ms (approximately 5 ms). For parameters of this example we consider 10 ms as a ***hard real-time deadline*** for sampling task.

Every sample is immediately enqueued to the *xQueueSamples* FreeRTOS queue. The purpose of the *xQueueSamples* queue is to stream samples to the another signal processing task, running in the parallel as a background lower priority processing task. By separating the sampling acquisition and signal processing tasks (where we also incorporate blocking serial output) we do not need to worry that signal processing and serial output will interfere with sampling process, regarding ***hard real-time deadlines***. *task_Sampling* is assigned a ***higher priority*** than *task_SerialOutput* task because the primary requirement for sampling process is not to lose *any* sample. If we have put e.g. blocking serial output in the same loop as the sample acquisition we would risk that ADC_BUFFER FIFO overflows if the operation of serial output within one loop iteration lasts more than 10 ms, what is likely to happen if we need to send significant chunks of text via serial console. Although we could still have sampling and serial output implemented in a single loop without use of multitasking, the solution would be much more complex to implement (e.g. non-blocking buffered serial output).

5.2.2 task_SerialOutput

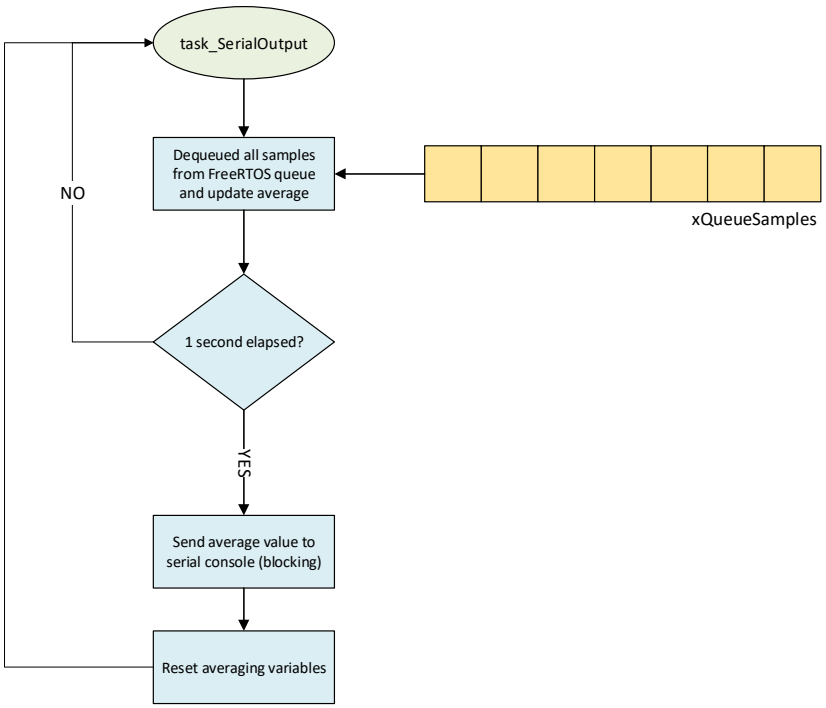
The flowchart below describes the functionality of *task_SerialOutput* task.

In each pass the task first checks if there are any samples in *xQueueSamples* queue. If queue is not empty, task dequeues all awaiting samples and updates the internal averaging variables (sum and count). Then, the task checks whether 1 second has elapsed since the last average update. If not, it returns to checking the content of *xQueueSamples* queue and dequeues new samples for the averaging process. Otherwise, the task outputs the average value to the serial console and resets the averaging variables.

Note that the serial output is *blocking* what means that the operation of a new task iteration will not start until the message is *fully sent* over the serial port. We usually use blocking serial output (*printf* function or equivalent) as it is much easier and more intuitive than using the non-blocking serial output. However, we need to be aware that usual approach of outputting messages through the serial console is blocking the thread, what can have a huge impact when we need to satisfy hard real-time requirements. For example, if we have a serial interface at speed 115200 bps, outputting the message of 100 characters will take 8.68 ms. If we need to serve some periodic task in the same loop with hard real-time requirements (e.g. clearing the ADC_BUFFER every 5 ms) then we would fail to comply to

[13] We could also use FreeRTOS queue to receive samples from ISR to task, instead of using hand-coded non-reentrant FIFO like in the example.

these constraints. By separating tasks with different hard real-time requirements we can implement solution in more elegant way, easier to understand and maintain.



It is worth noting that size of *xQueueSamples* is 500 (constant SAMPLES_AVERAGING_BUFFER_SIZE in *config.h*). It means that **hard real-time** constraint for *task_SerialOutput* (before *xQueueSamples* gets full) is 25 ms, for a sampling rate of 20 kHz. If *xQueueSamples* is not cleared within 25 ms, *task_Sampling* will not be able to enqueue new samples and they will be discarded. If this happens, in this example a *task_Sampling* will signal such event by turning on red LED and stopping the program in the case of losing even a single sample because it would mean that the corresponding hard real-time constraints has been violated.

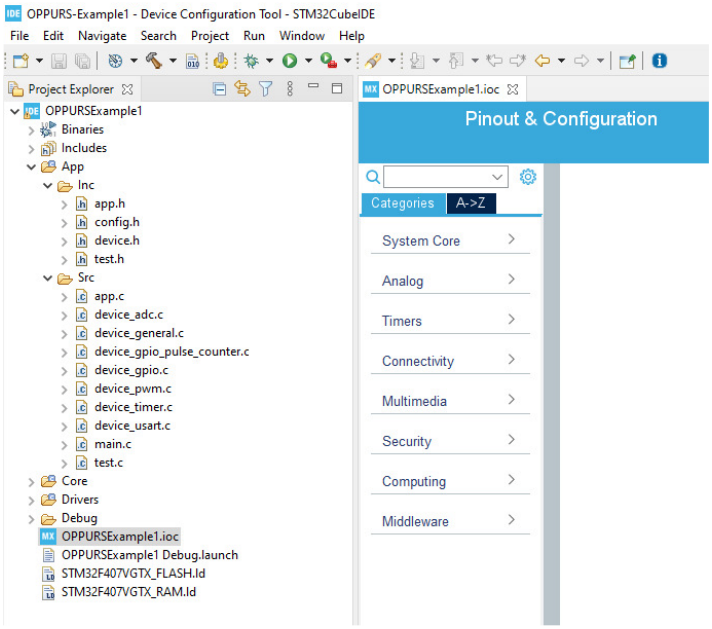
This example also illustrates the importance of queues as OS-provided inter-task communication mechanism. Firstly, using OS-provided queues *does not require* any additional mechanisms to ensure atomicity of operations (e.g. disabling interrupts, using mutexes etc.) since queues are already thread-safe and under the control of OS kernel. Secondly, the size of the queues (which depend on the available RAM memory) can be used to relax hard real-time constraints on tasks.

The following sections will provide detailed descriptions of source files for *app3* project example.

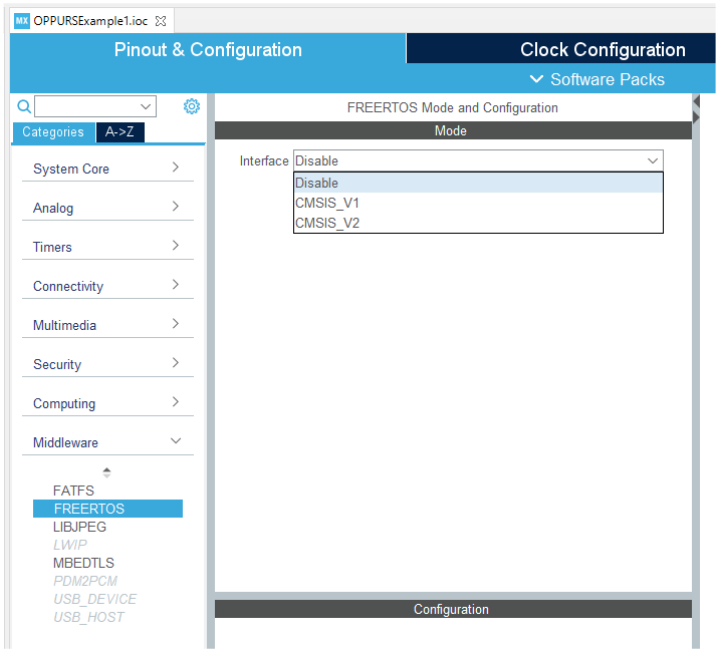
5.3 Including FreeRTOS in the project

The *app3* example is based on the upgrade of the previously described *app2* example. Although it is possible to include FreeRTOS into project manually, it is easier and more convenient to include it via STM32CubeIDE as a *MiddleWare* component.

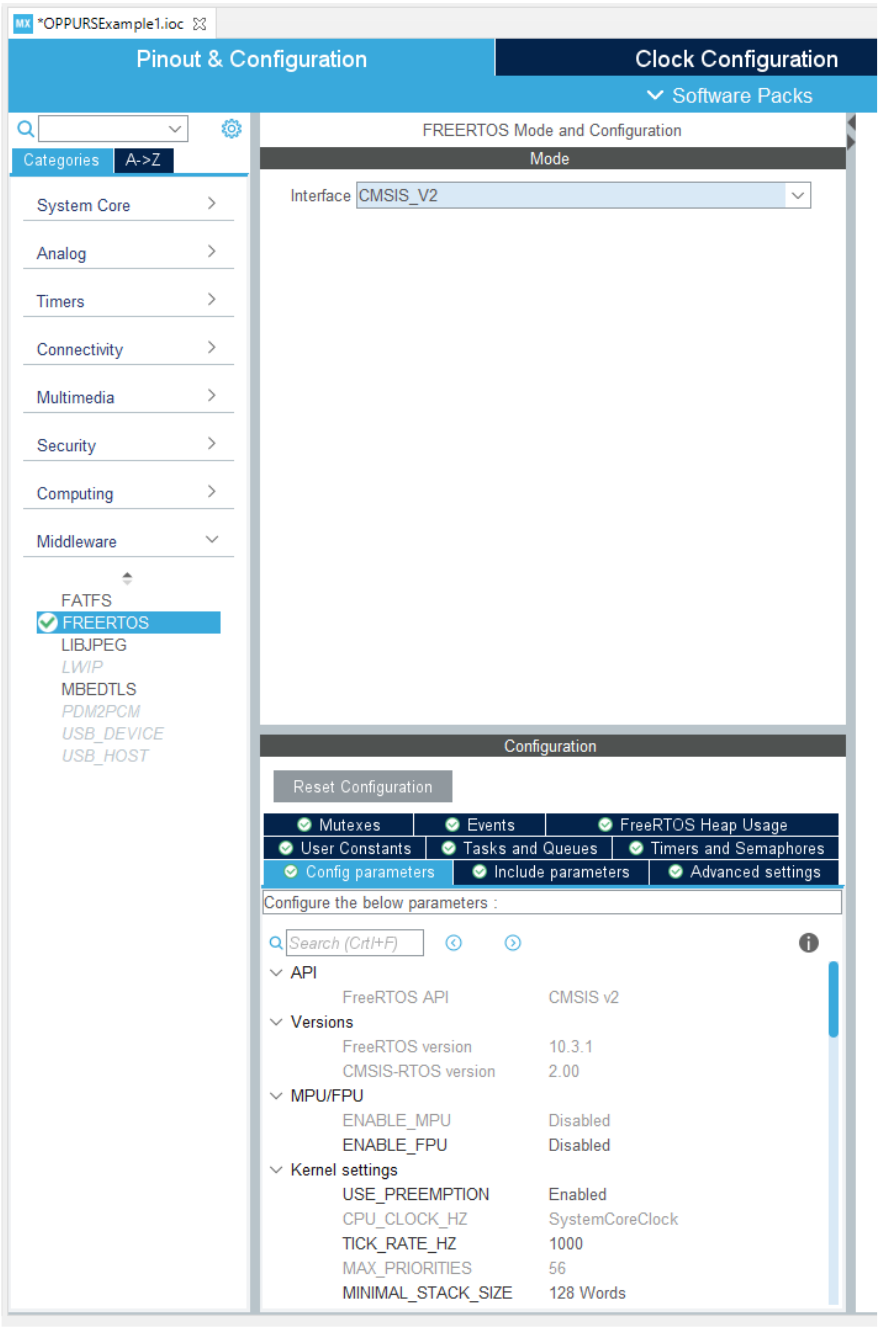
First step is to open IOC file:



Then, under *MiddleWare* tab select FreeRTOS component:



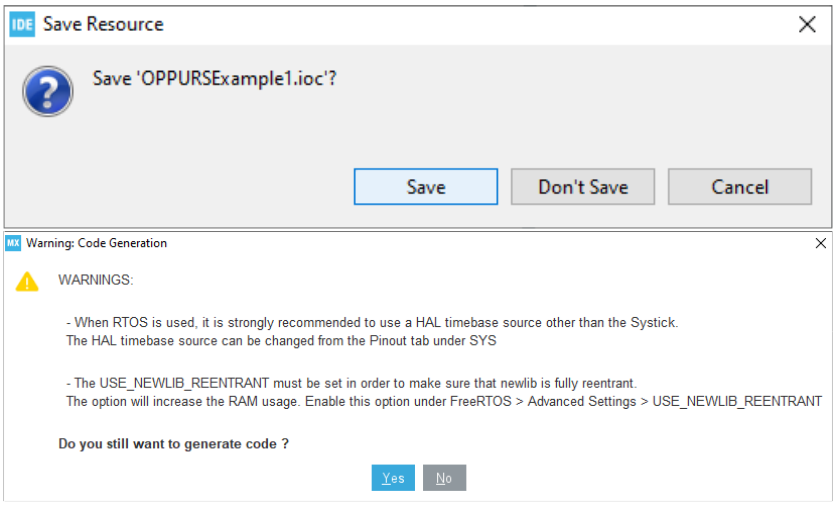
To include FreeRTOS into project we need to select CMSIS interface[14] (e.g. CMSIS_V2):



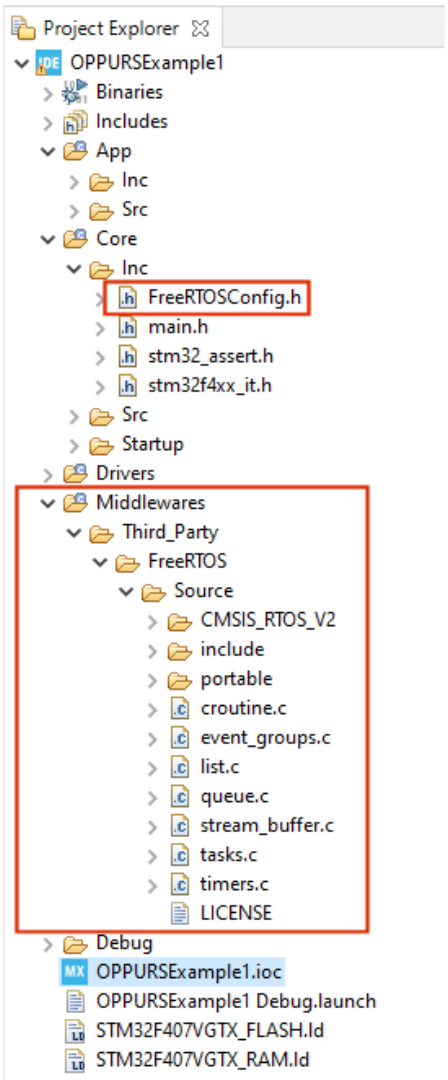
In *Configuration* part of the window we can change various configuration parameters of FreeRTOS. We can leave the default parameters. It is also possible to edit these parameters directly in code (*FreeRTOSConfig.h*).

[14] CMSIS interface is a standardized thin wrapper around RTOS which exposes the same API, regardless of the underlaying RTOS. Such approach promotes code portability, standardization and independence on the choice of RTOS. Although we need to choose CMSIS interface in this case, we shall not use CMSIS API in our examples and we shall access FreeRTOS API directly.

Let STM32CubeIDE update sources upon exiting IOC file (we can safely ignore warnings):

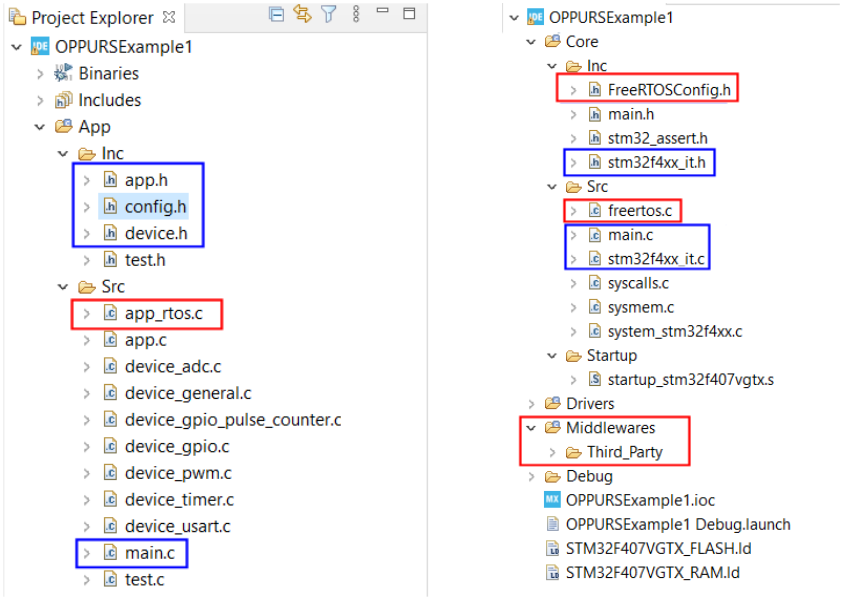


The updated source tree is shown in figure below:



5.4 Organization of source files

Organization of source files is similar to *app2* example. Additional and changed files in *app3* example are shown in the figure below (*red* – new files, *blue* – changed files from *app2* example):



The content of each file in the source tree is as follows (only new files):

Include files:

- *FreeRTOSConfig.h* – FreeRTOS header file with definitions of constants

Source files:

- *freertos.c* – IDE generated C file for FreeRTOS internals (no particular functionalities implemented, only a placeholder file)
- *Middlewares folder* – FreeRTOS sources (included by IDE)

The following sections will describe the most important details in changed or new files.

5.5 Source files descriptions

5.5.1 App/Inc/app.h

A new function *app3_rtos* is added:

```
void app1();
void app2(int samplingFrequency, int pwmFrequency, int dutyCycle, int adcTransferMode);
void app3_rtos(int samplingFrequency, int pwmFrequency, int dutyCycle, int adcTransferMode);
```

This function initializes hardware resources and FreeRTOS kernel (creates tasks, queues and launches FreeRTOS task scheduler).

5.5.2 App/Inc/config.h

New application mode to launch FreeRTOS-based *app3* is added:

```
// select app to run:
#define APP1          0
#define APP2_ISR      0
#define APP2_DMA      0
#define APP3_FREERTOS  1
```

Definition constant for FreeRTOS *xQueueSamples* size is added:

```
// FreeRTOS averaging sample buffer
#define SAMPLES_AVERAGING_BUFFER_SIZE     500
```

5.5.3 App/Inc/device.h

Small upgrade of *device.h*, for easier identification of LEDs by color:

```
#define LED_GREEN      LED1
#define LED_ORANGE     LED2
#define LED_RED        LED3
#define LED_BLUE       LED4
```

5.5.4 Core/Inc/stm32f4xx_it.h

Two interrupt handlers are removed because FreeRTOS needs them for kernel:

```
void NMI_Handler(void);
void HardFault_Handler(void);
void MemManage_Handler(void);
void BusFault_Handler(void);
void UsageFault_Handler(void);
void SVC_Handler(void);
void DebugMon_Handler(void);
void PendSV_Handler(void);
void SysTick_Handler(void);
void ADC_IRQHandler(void);
void TIM2_IRQHandler(void);
void USART3_IRQHandler(void);
void EXTI15_10_IRQHandler(void);
void DMA2_Stream0_IRQHandler(void);
```

5.5.5 Core/Src/main.c / stm32f4xx_it.c

IDE automatically inserts some headers and implementation files needed for the proper RTOS operation (default task, NVIC configuration, resolved ISRs used by task scheduler etc.).

5.5.6 App/Src/main.c

Additional case for *app3* launch was added to *main_app*() function:

```
void main_app() {

#if (RUN_APP)
   #if (APP1)
   app1();
   #endif
   #if (APP2_ISR)
   app2(20000, 2000, 50, ADC_USE_INTERRUPT);
   #endif
   #if (APP2_DMA)
   app2(20000, 2000, 50, ADC_USE_DMA);
   #endif
   #if (APP3_FREERTOS)
   app3_rtos(20000, 2000, 50, ADC_USE_INTERRUPT);
   #endif
#endif

   while(1);
}
```

5.5.7 App/Src/app_rtos.c

New C module which implements *app3* functionality.

First, we need to include some header files with FreeRTOS definitions:

```
#include <app.h>
#include <FreeRTOS.h>
#include <task.h>
#include <queue.h>
```

Then, we create a special structure to provide startup data as a parameter for each task via *pvParameters*:

```
typedef struct Task_Params
{
   int samplingFrequency;
   int pwmFrequency;
   int dutyCycle;
   int adcTransferMode;

} Task_Params;
```

We also need to define FreeRTOS queue *xQueueSamples* as a global variable, since tasks are exchanging data using message queue:

```
// queue for inter-task communication (exchange of samples)
xQueueHandle xQueueSamples;
```

The application *app3* is launched via function *app3_rtos,* which takes the same parameters like previously described for function *app2()* in *app2* case:

```
void app3_rtos(int samplingFrequency, int pwmFrequency, int dutyCycle, int adcTransferMode)
{
char msg[64];
int err_flag;
BaseType_t ret;

   // set up task param struct
   taskParameters.samplingFrequency = samplingFrequency;
   taskParameters.pwmFrequency = pwmFrequency;
   taskParameters.dutyCycle = dutyCycle;
   taskParameters.adcTransferMode = adcTransferMode;

   ////////////////////////////////////////////////////////////////
   // hardware initialization
   ////////////////////////////////////////////////////////////////
   InitDevice();   // common - serial port at least for error messages
   // additional hardware initialization - application specific
   PWM_Start(pwmFrequency, dutyCycle);    // reference signal to be sampled (Freq, DutyCycle)

   ////////////////////////////////////////////////////////////////
   // error checking
   ////////////////////////////////////////////////////////////////
   if (samplingFrequency < 1285)// minimum freq 1280.76 Hz due to 16-bit autoreaload register in TIM3
   {
      sprintf("Error: samplingFrequency = %d < 1285 Hz", samplingFrequency);
      ErrorMessage(msg);
   }
   if (pwmFrequency < 1285)     // minimum freq 1280.76 Hz due to 16-bit autoreaload register in TIM4
   {
      sprintf("Error: pwmFrequency = %d < 1285 Hz", pwmFrequency);
      ErrorMessage(msg);
   }
   if ((dutyCycle < 0) || (dutyCycle > 100))
   {
      sprintf("Error: invalid dutyCycle = %d", dutyCycle);
      ErrorMessage(msg);
   }
```

initialize the structure for sending parameters to tasks

```
    ////////////////////////////////////////////////////////////////
    // launch tasks
    ////////////////////////////////////////////////////////////////
    err_flag = 0;
    xQueueSamples = xQueueCreate(SAMPLES_AVERAGING_BUFFER_SIZE, sizeof(uint16_t));
    if (xQueueSamples == NULL) err_flag++;
    ret = xTaskCreate(task_Sampling, "", 1024, &taskParameters, 2, NULL);
    if (ret != pdPASS ) err_flag++;
    ret = xTaskCreate(task_SerialOutput, "", 1024, &taskParameters, 1, NULL);
    if (ret != pdPASS ) err_flag++;
    if (err_flag > 0)
    {
        ErrorMessage("Cannot launch FreeRTOS!");
        while(1);
    }

    // launch RTOS
    vTaskStartScheduler();
    while(1); // WARNING: taskParams not valid beyond this point!
}
```

error flag != 0 if any of the objects could not be created on internal FreeRTOS heap

at this point both tasks become active!

The code for *task_Sampling* task is explained below:

```
// task1: ADC sampling
void task_Sampling(void* pvParameters)
{
Task_Params* taskParams;
char msg[64];
int i, samplesCountToCollect;
uint16_t sample;
BaseType_t res;

    taskParams = (Task_Params*)pvParameters;
    ////////////////////////*////////////////////////////////////
    // set up sampling process
    ////////////////////////////////////////////////////////////////
    switch(taskParams->adcTransferMode)
    {
        case ADC_USE_INTERRUPT:
            ADC_SystemInit(taskParams->samplingFrequency, ADC_USE_INTERRUPT);
            break;
        case ADC_USE_DMA:
            ADC_SystemInit(taskParams->samplingFrequency, ADC_USE_DMA);
            break;
        default:
            sprintf("Error: invalid ADC transfer mode = %d", taskParams->adcTransferMode);
            ErrorMessage(msg);
            break;
    }

    // signal normal state (GREEN LED)
    LED_User_On(LED_GREEN);
    LED_User_Off(LED_RED);
    while(1)
    {
        // sleep for 5 ms
        vTaskDelay(5 / portTICK_PERIOD_MS);
        samplesCountToCollect = ADC_Buffer_InBufferCount();
        for(i = 0; i < samplesCountToCollect; i++)
        {
            ADC_Buffer_Dequeue(&sample);
            res = xQueueSendToBack(xQueueSamples, &sample, 0);
            if (res == errQUEUE_FULL)
            {
                // signal error state (RED LED)
                LED_User_Off(LED_GREEN);
                LED_User_On(LED_RED);
                while(1);
            }
        }
    }
}
```

cast void* pointer to correct data type (struct)

green LED signals correct system operation

atomic due to the implementation in *drivers_adc.c*

non-blocking enqueue of the sample in xQueueSamples; due to earlier explained timings, no queue overrun should occur

if queue overrun is detected, stop the program and turn on red LED

The code for *task_SerialOutput* task is explained below:

```
// task2: serial output
void task_SerialOutput(void* pvParameters)
{
Task_Params* taskParams;
char msg[64];
int sum, count;
uint16_t sample;
TickType_t timeStamp1_ms, timeStamp2_ms;
int i, samplesCountToCollect;
float average;

   taskParams = (Task_Params*)pvParameters;
   ////////////////////////////////////////////////////////////////
   // welcome message and info
   ////////////////////////////////////////////////////////////////
   USART3_WriteLine("*** ADC sample ***");
   sprintf(msg, "Sampling frequency = %d Hz", taskParams->samplingFrequency);USART3_WriteLine(msg);
   sprintf(msg, "PWM frequency = %d Hz", taskParams->pwmFrequency); USART3_WriteLine(msg);
   sprintf(msg, "Duty cycle = %d %%", taskParams->dutyCycle); USART3_WriteLine(msg);
   sprintf(msg, "ADC transfer mode = %d", taskParams->adcTransferMode); USART3_WriteLine(msg);
   USART3_WriteLine("");

   // run serial output task
   sum = 0; count = 0;
   timeStamp1_ms = xTaskGetTickCount() * portTICK_PERIOD_MS;
   timeStamp2_ms = timeStamp1_ms;
   while(1)
   {
      // dequeue samples
      samplesCountToCollect = (int)uxQueueMessagesWaiting(xQueueSamples);
      for(i = 0; i < samplesCountToCollect; i++)
      {
         xQueueReceive(xQueueSamples, &sample, 0);
         sum += (int)sample;
         count++;
      }
      // averaging period: 1 s
      timeStamp2_ms = xTaskGetTickCount() * portTICK_PERIOD_MS;
      if ((timeStamp2_ms - timeStamp1_ms) > 1000)
      {
         if (count > 0)
         {
            average = (float)sum / (float)count;
            sum = 0; count = 0;
            timeStamp1_ms = timeStamp2_ms;
            sprintf(msg, "average = %d", (int)average);
            USART3_WriteLine(msg);
         }
      }
   }
}
```

dequeue all available samples from the xQueueSamples, while updating avergaing variables

send averaged result every second and reset internal averaging variables

5.6 Running app3 example

After successfully compiling and flashing *app3* example, observe the serial console while connecting PC0 (ADC1_IN0) to three different analog sources (via external wire connection):

- PB6 (TIM4_CH1) PWM output: averaged result should be approximately half of 12-bit ADC full range (around 2048 raw result)
- GND: averaged result should be approximately 0
- VCC (**3V!**): averaged result should be approximately at full range (4095)

Example serial output:

```
*** ADC sample ***
Sampling frequency = 20000 Hz
PWM frequency = 2000 Hz
Duty cycle = 50 %
ADC transfer mode = 1

average = 2050
average = 2049
average = 2050
average = 2049
average = 2050
average = 2050
average = 2049
average = 2050
average = 2050
average = 2050
average = 2104
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 1993
average = 1
average = 1
average = 1
average = 1
average = 2250
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 4094
average = 3939
average = 2307
average = 2050
average = 2050
average = 2050
average = 2050
average = 2050
```

ADC input connected to PWM (50% duty cycle)

ADC input connected to 3V

ADC input connected to GND