STM32

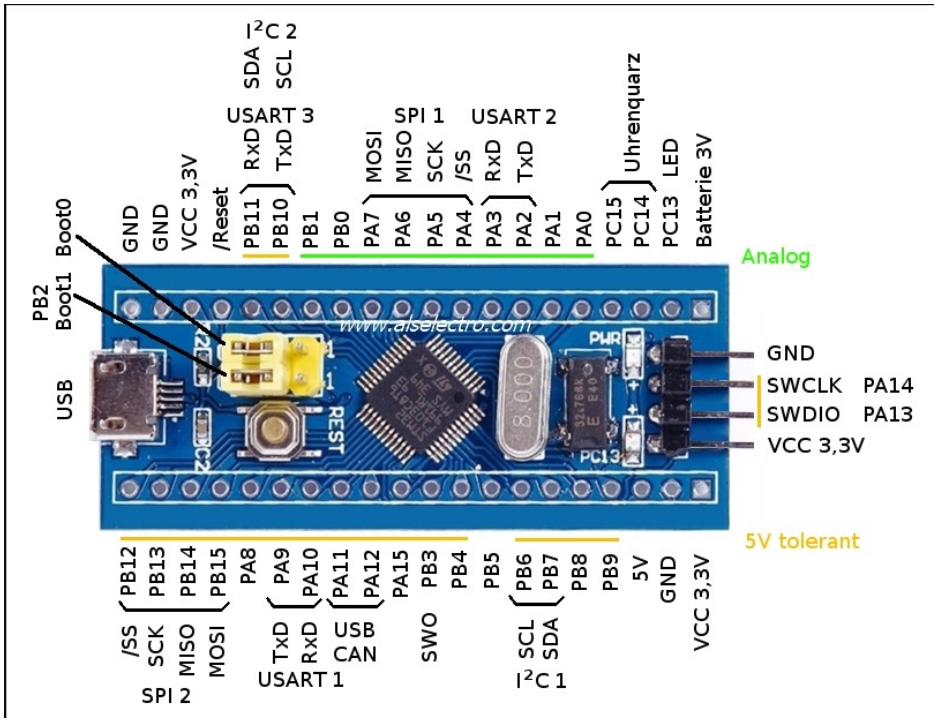
yinchheanyun21

December 2022

I Getting started with STM32

The STM32 series of microcontrollers is one of the most popular ones among the 32-Bit microcontrollers. STMicroelectronics provides multiple of product lines for the STM32 parts. There is a low-power, mainstream, and high-performance product lines. And a more application-specific wide variety of parts that enables you to pick the right part for your project.

1. **Blue Pill STM32F103**

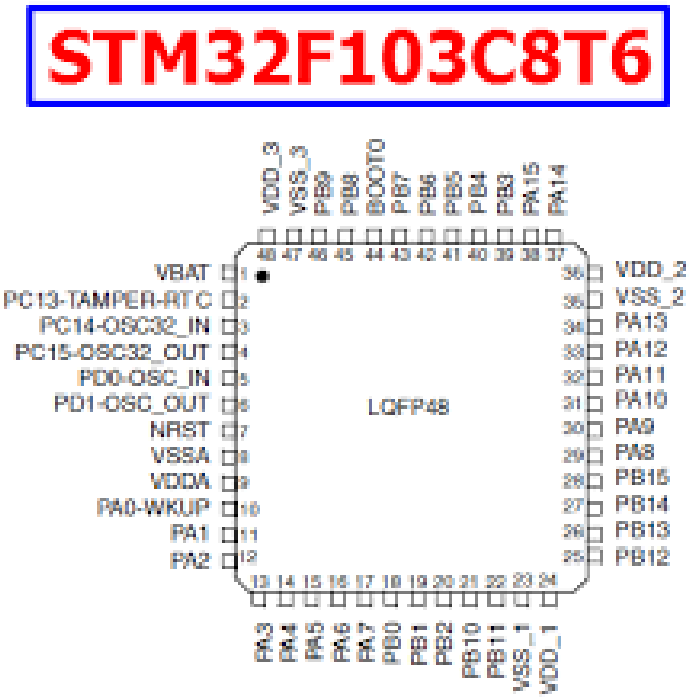


- ARM®32-bit Cortex®-M3 CPU Core
  - 72 MHz maximum frequency,1.25 DMIPS/MHz (Dhrystone 2.1) performance at 0 wait state memory access
  - Single-cycle multiplication and hardware division
- Memories
  - 64 or 128 Kbytes of Flash memory
  - 20 Kbytes of SRAM
- Clock, reset and supply management
  - 2.0 to 3.6 V application supply and I/Os

  - POR, PDR, and programmable voltage detector (PVD)
  - 4-to-16 MHz crystal oscillator
  - Internal 8 MHz factory-trimmed RC
  - Internal 40 kHz RC
  - PLL for CPU clock
  - 32 kHz oscillator for RTC with calibration
- Low-power
  - Sleep, Stop and Standby modes
  - VBAT supply for RTC and backup registers
- 2 x 12bit, 1 us A/D converters (up to 16 channels)
  - Conversion range: 0 to 3.6 V
  - Dual-sample and hold capability
  - Temperature sensor
- DMA
  - 7-channel DMA controller
  - Peripherals supported: timers, ADC, SPIs, I2Cs and USARTs
- Up to 80 fast I/O ports
  - 26/37/51/80 I/Os, all mappable on 16 external interrupt vectors and almost all 5 V-tolerant
- Debug mode
  - Serial wire debug (SWD) and JTAG interfaces
- 7 timers
  - Three 16-bit timers, each with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input
  - 16-bit, motor control PWM timer with dead-time generation and emergency stop
  - 2 watchdog timers (Independent and Window)
  - SysTick timer 24-bit downcounter
- Up to 9 communication interfaces
  - Up to 2 x I2C interfaces (SMBus/PMBus)
  - Up to 3 USARTs (ISO 7816 interface, LIN, IrDA capability, modem control)
  - Up to 2 SPIs (18 Mbit/s)
  - CAN interface (2.0B Active)
  - USB 2.0 full-speed interface

II STM32 HAL Library

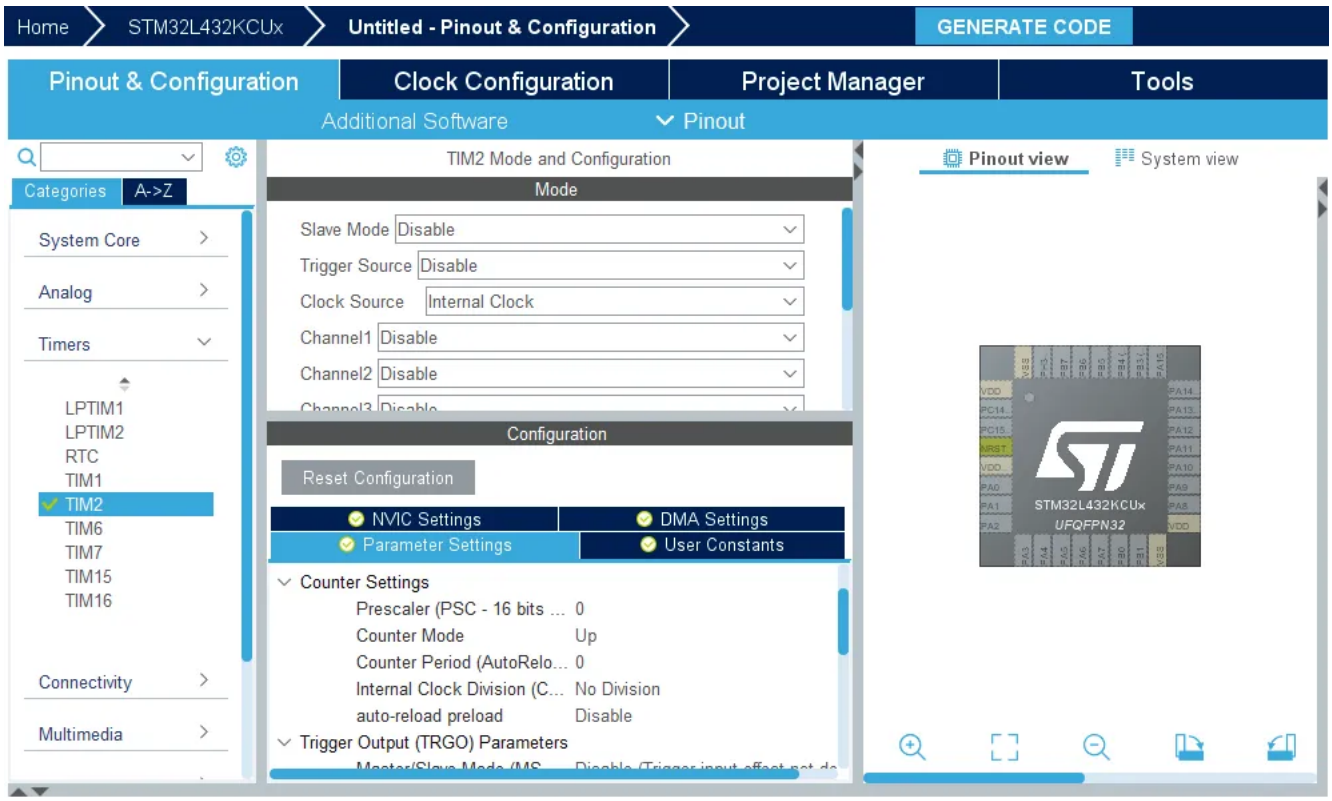
- **HAL GPIO APIs**
  - Initialization and de-initialization function



  - * HAL_GPIO_Init()
  - * HAL_GPIO_Delnit()
- – IO operation functions
  - * HAL_GPIO_ReadPin()
  - * HAL_GPIO_WritePin()
  - * HAL_GPIO_TogglePin()
  - * HAL_GPIO_LockPin()
  - * HAL_GPIO_EXTI_IRQHandler()

- HAL Detailed Function Description
  - – HAL_GPIO_init
    - * Function name : voidHAL_GPIO_Init(GPIO_TypeDef*GPIOx,GPIO_InitTypeDef*GPIO_i
    - * Function description : Initialize the GPIOx peripheral according to the specified parameters in the GPIO_Init.
    - * Parameters : **1. GPIOx :** Where x can be (A..H)to select the GPIO peripheral for STM32L4 family.
      2. GPIO_Init: pointer to a GPIO_initTypeDef structure that constains the configuration information for the specifield GPIO peripherai.
    - * Return values **None**

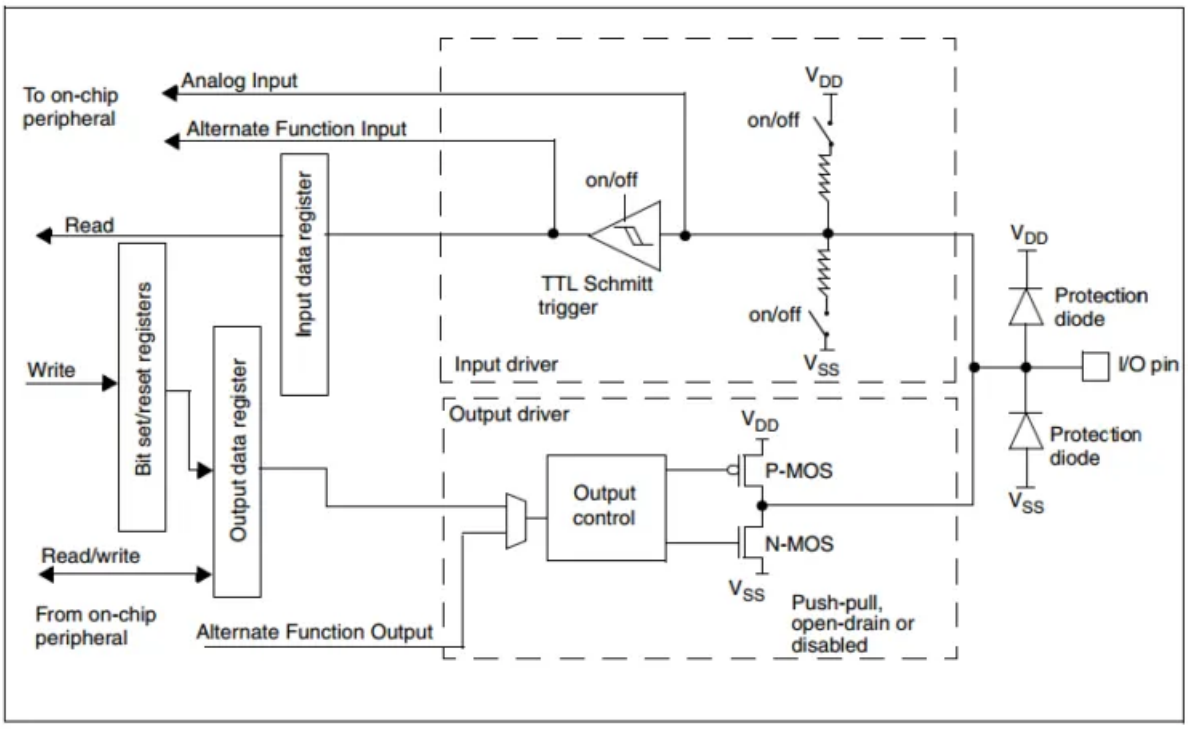
- **Hardware Timer Module configuration Within cubeMX**



III STM32 GPIO Tutorials

In this tutorial, we'll discuss the STM32 GPIO hardware. How it works and what sort of functionalities there are, so you can configure it in the best way to meet your applications' needs. We'll get into the GPIO speed, alternative functions, locking mechanism, and different possible configurations. So let's get started!

1. **STM32 GPIO Ports** Here is a digital diagram for the internal structure of a typical GPIO pin. It shows the diode protection, internal pull-up or down enable/disable, and also the push-pull output driver, output enable/disable for switching between input/output pin modes, Schmitt-triggered digital input, analog input.



Symbol	Ratings	Min	Max	Unit
V_{DD} –V_{SS}	External main supply voltage (including V_{DDA} and V_{DD})(1)	–0.3	4.0	V
V_{IN}(2)	Input voltage on five volt tolerant pin	V_{SS} –0.3	V_{DD} +4.0	
	Input voltage on any other pin	V_{SS} –0.3	4.0	

Symbol	Ratings	Max.	Unit
I_{VDD}	Total current into V_{DD}/V_{DDA} power lines (source)(1)	150	mA
I_{VSS}	Total current out of V_{SS} ground lines (sink)(1)	150	
I_{IO}	Output current sunk by any I/O and control pin	25	
	Output current source by any I/Os and control pin	–25	

2. **STM32 GPIO Speed**

   For Input Mode

   When a GPIO pin is set to input mode, the data present on the I/O pin is sampled into the Input Data Register every APB2 clock cycle. This means the APB2 bus speed determines the input sampling speed for the GPIO pins.

   For Output Mode

   When a GPIO pin is set to output mode, you'll have the option to configure the pin speed mode by programming the respective bits in the configuration registers. Down below is a table for the different modes available in the datasheet for the STM32F103C8 microcontroller.

Table 20. Port bit configuration table

Configuration mode		CNF1	CNF0	MODE1	MODE0	PxODR register
General purpose output	Push-pull	0	0	01 10 11 see *Table 21*		0 or 1
	Open-drain		1			0 or 1
Alternate Function output	Push-pull	1	0			Don't care
	Open-drain		1			Don't care
Input	Analog	0	0	00		Don't care
	Input floating		1			Don't care
	Input pull-down	1	0			0
	Input pull-up					1

Table 21. Output MODE bits

MODE[1:0]	Meaning
00	Reserved
01	Maximum output speed 10 MHz
10	Maximum output speed 2 MHz
11	Maximum output speed 50 MHz

Bits 31:0 **OSPEED[15:0][1:0]**: Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O output speed.
00: Low speed
01: Medium speed
10: High speed
11: Very high speed

Note: Typical values depend on various parameters of which the VDDio line voltage, the total pin capacitance.

- Low Speed $->$ p to 10MHz
- Medium Speed $>$ Up to 50MHz
- High Speed $->$ Up to 100MHz
- Very High Speed $->$ Up to 180MHz

3. **GPIO Bit Atomic Operations**
   There is no need for the software to disable interrupts when programming the GPIOx_ODR at bit level: it is possible to modify only one or several bits in a single atomic APB2 write access. This is achieved by programming to '1' the Bit Set/Reset Register (GPIOx_BSRR, or for reset only GPIOx_BRR) to select the bits to modify. The unselected bits will not be modified.

4. **STM GPIO External Interrupts**
   All ports have external interrupt capability. To use external interrupt lines, the port must be configured in input mode. We'll discuss this in a more detailed manner when we get into the EXTI (external interrupt/event controller) topic.

5. **Peripheral Pin Select (PPS**
   To optimize the number of peripheral I/O functions for different device packages, it is possible to remap some alternate functions to some other pins. This is achieved by software, by programming the corresponding registers.
   This option can help you remap the peripherals io pins, so you don't have to change so much in the PCB layout when you change the target microcontroller on the board. This can be extremely advantageous and ease the routing process. And help you move the high-speed signals away so as to reduce the noise level at certain parts.

6. **STM32 GPIO Locking Mechanism**
   The locking mechanism allows the IO configuration to be frozen. When the LOCK

sequence has been applied on a port bit, it is no longer possible to modify the value of the port bit until the next reset.

7. **STM32 GPIO Configurations** Subject to the specific hardware characteristics of each I/O port listed in the datasheet, each port bit of the General Purpose IO (GPIO) Ports, can be individually configured by software in several modes:
   Output Configuration

   - Output Open-Drain
   - Output Push-Pull

   - Input Floating($Hi - Z$)
   - Input Pull-Up
   - Input Pull-Down

   Alternate Function Configuration

   - Alternate Function Push-Pull
   - Alternate Function Open−Drain

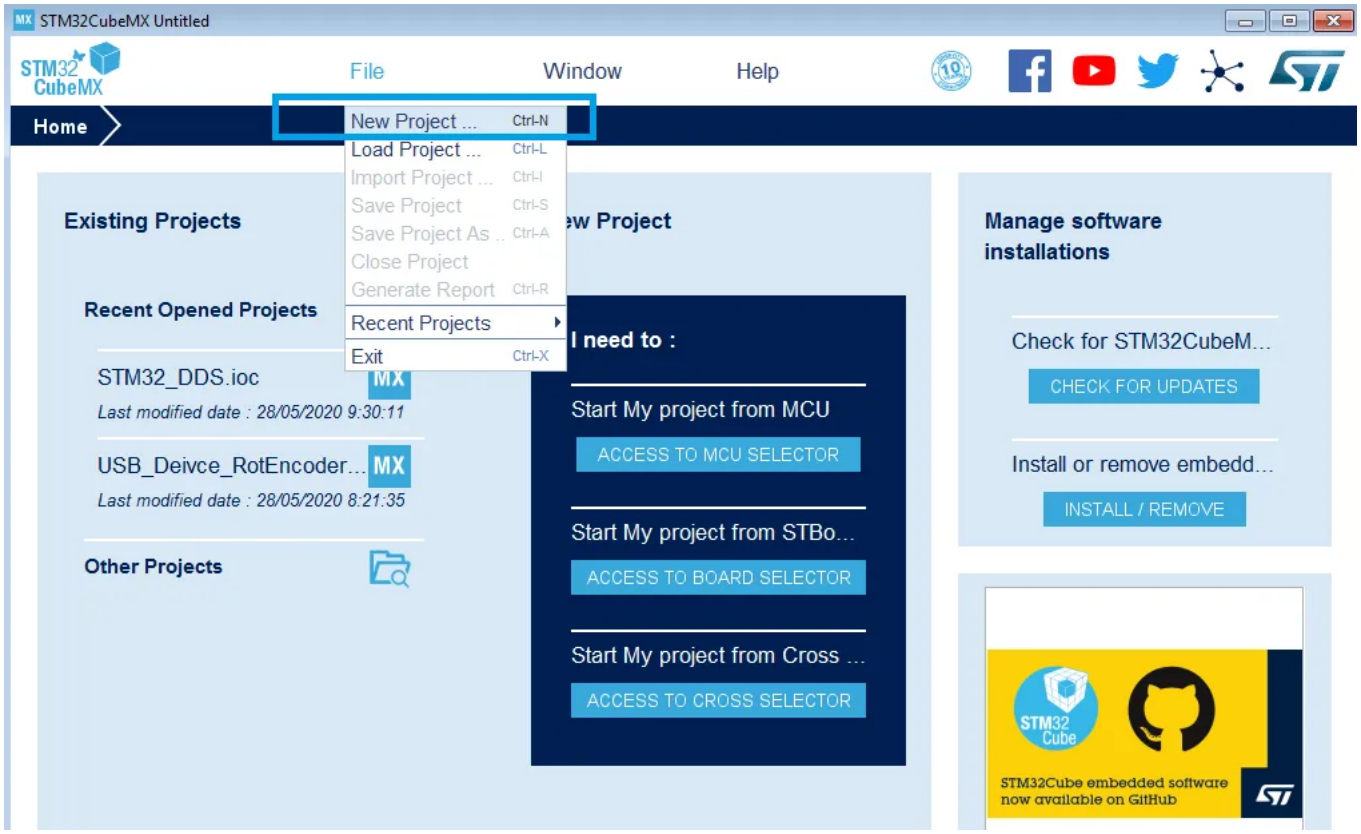
IV STM32 GPIO Write Pin Digital Output LAB

1. **Required Components For LABs**

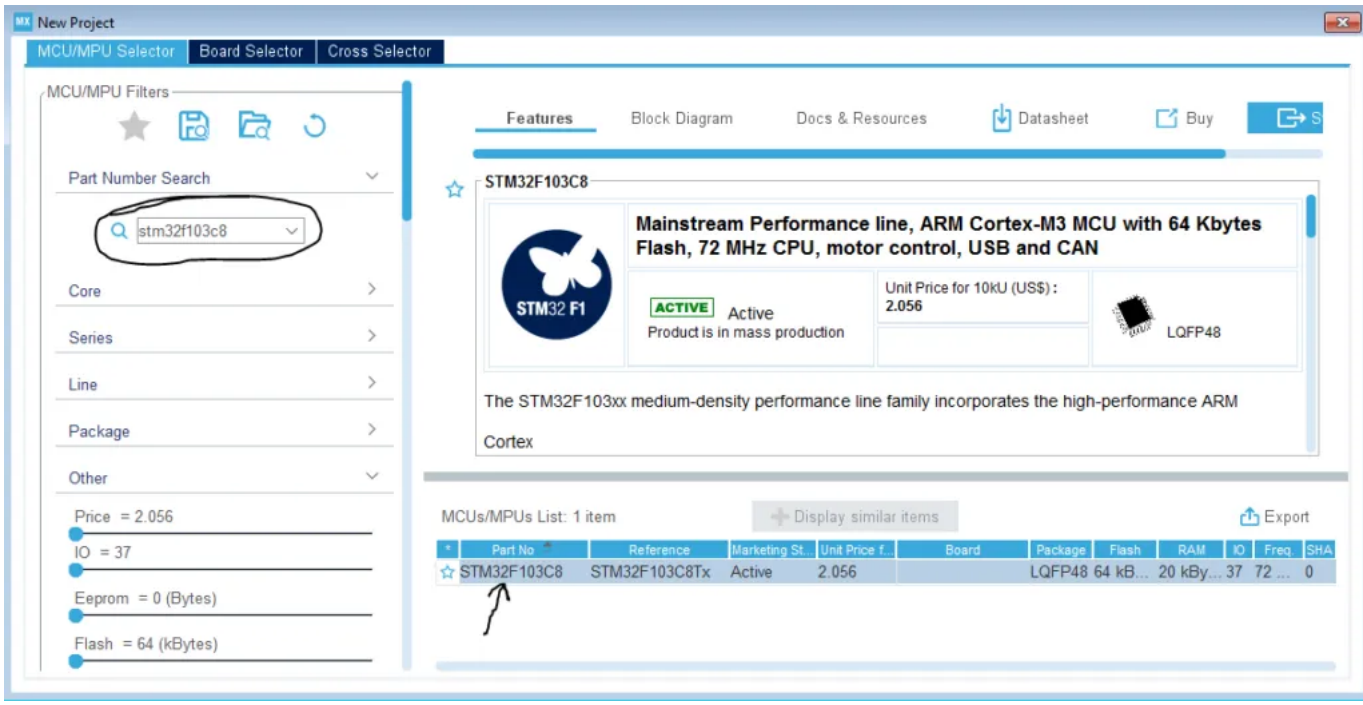
   - BreadBoard
   - Resistor
   - Capacitor
   - LED
   - Battery 9v or Dc powersuply
   - Micro USB cable
   - push botton

2. **STM32 CubeMX Configurations**

Step1: Open CubeMX & Create New Project

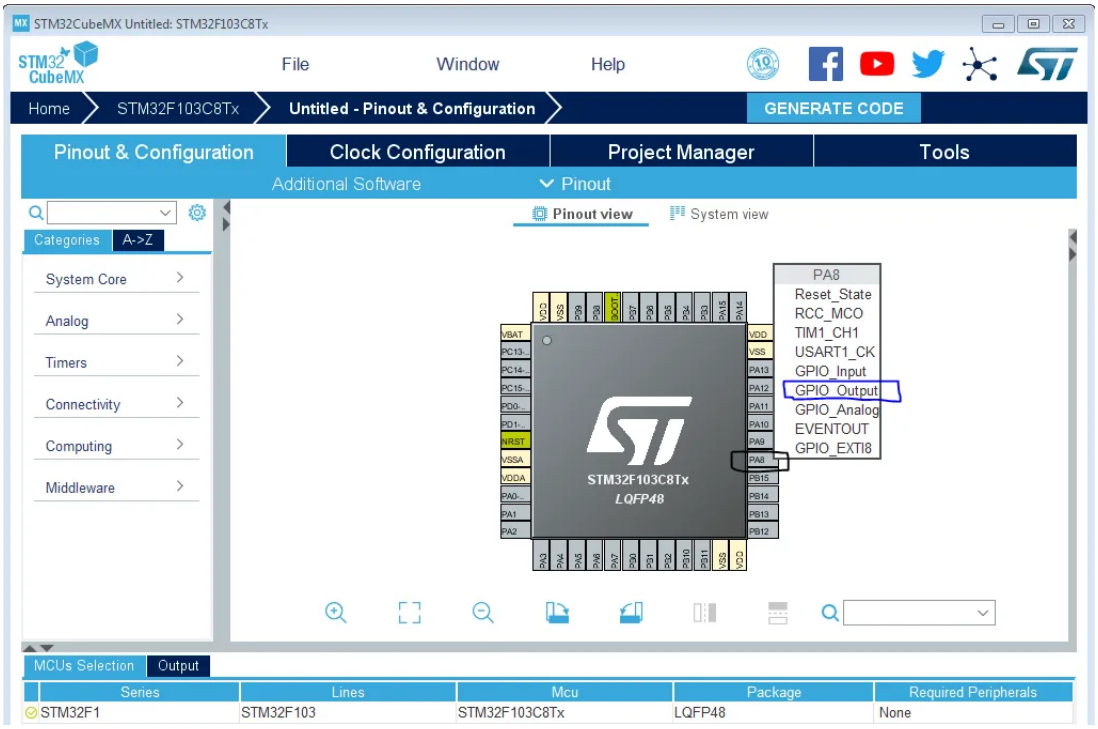


Step2: Choose The Target MCU & Double-Click Its Name

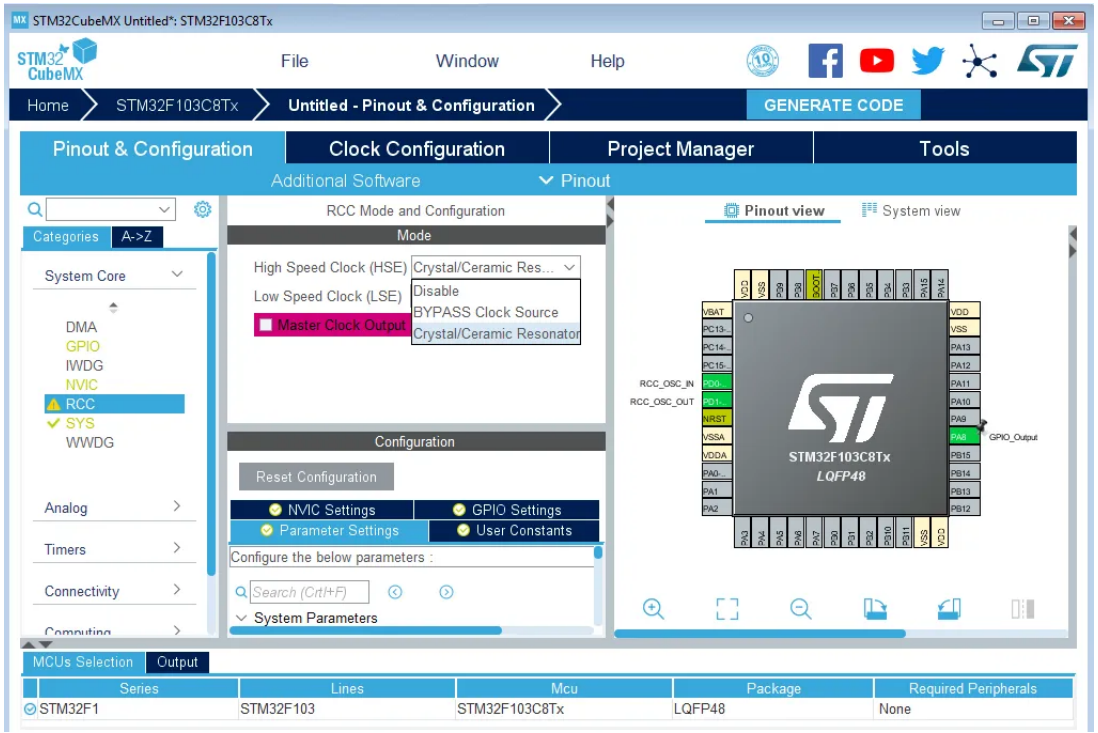


Step3: Click On The Pin You Want To Configure As An Output & Select

Output Option

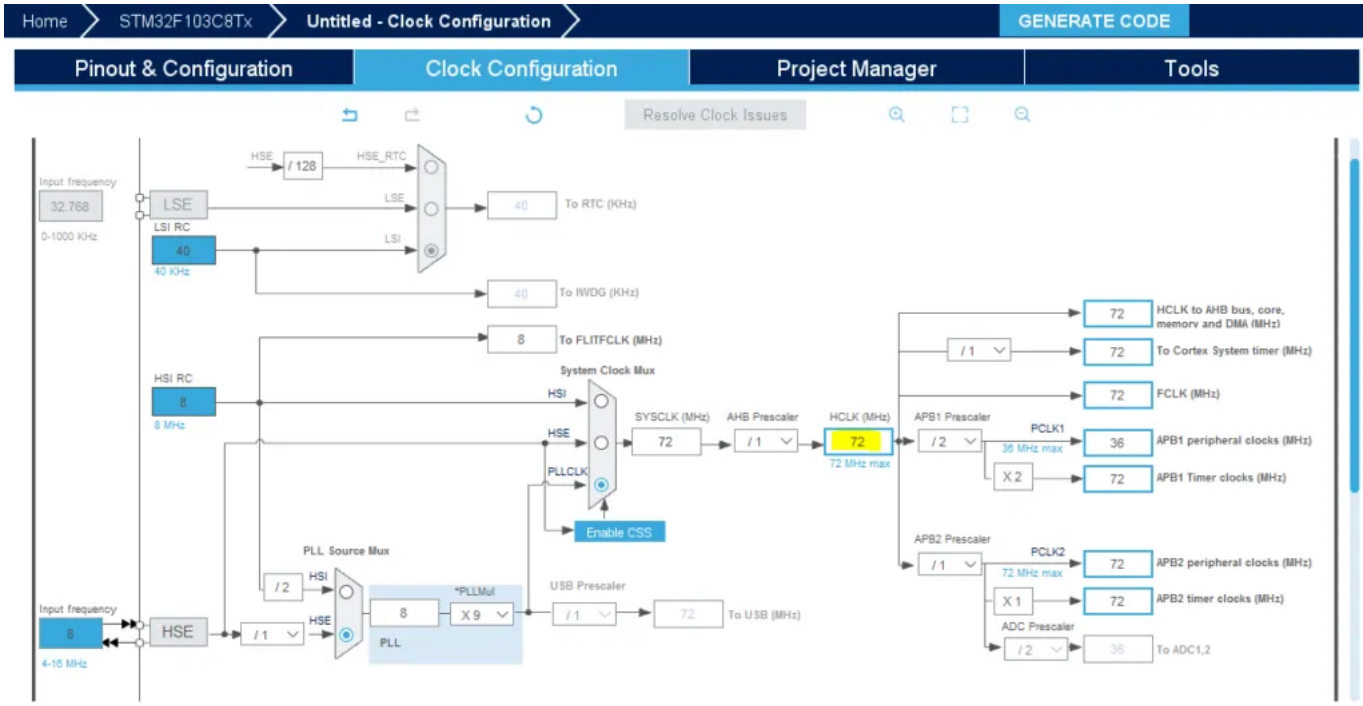


Step4: Set The RCC External Clock Source

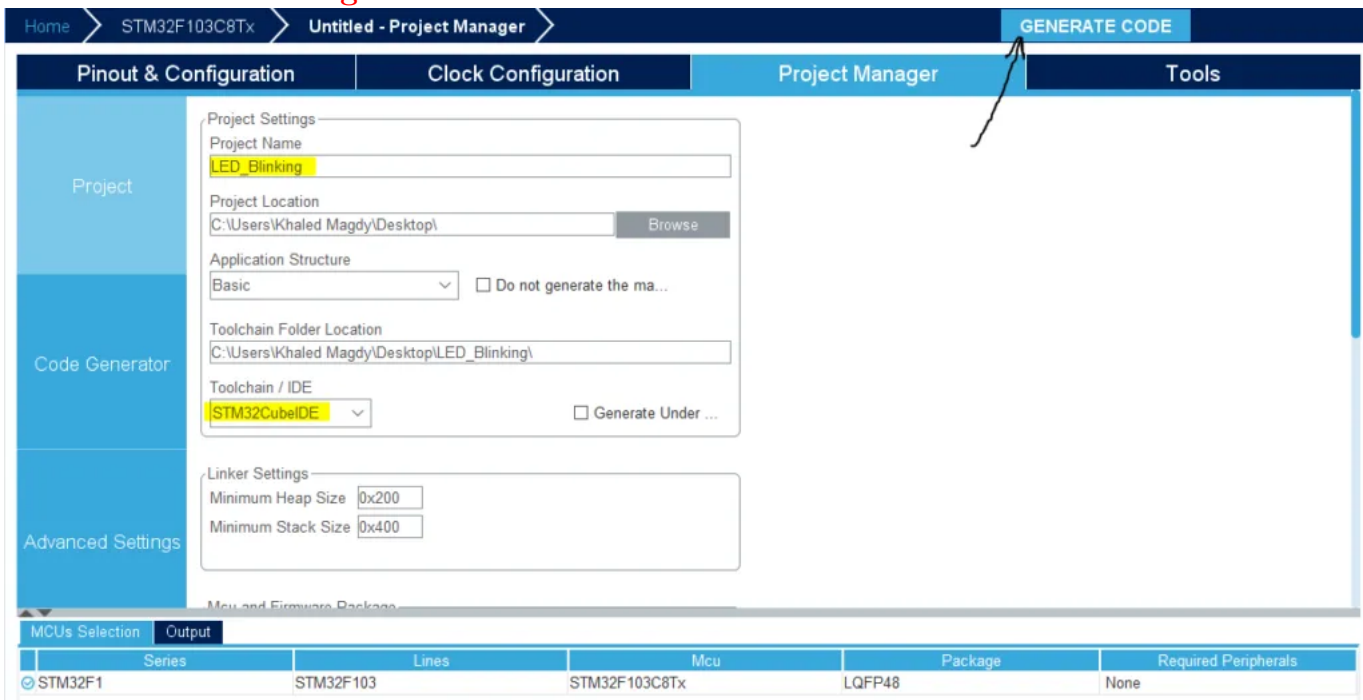


Note:The RCC controller integrated inside STM32 products manages system and peripheral clocks. STM32F7 devices embed two internal oscillators, 2 oscillators for an external crystal or resonator, and three phase-locked loops (PLL). Many peripherals have their own clock, independent of the system clock.

Step5: Go To The Clock Configuration
Step6: Set The System Clock To Be 72MHz Or Whatever You Want



Step7: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using



3. **The Application Code In CubeIDE**

```c
int main(void)
{

  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();

  while (1)
  {
      // LED ON
      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_SET);
      HAL_Delay(100);
      // LED OFF
      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_RESET);
      HAL_Delay(100);
  }

}
```

V STM32 GPIO PIN READ Digital INPUT

1. **STM32 CubeMX Configurations**

   - **Step1:Open CubeMX & Create New Project**
   - **Step2:Choose The Target MCU & DoubleClick Its Name**
   - **Step3:Click On The Pin You Want To Configure As An Output & Select Output Option**
   - **Step4:Click On The Pin You Want To Configure As An Input & Select Input Option**
   - **Step5:Set The RCC External Clock Source**
   - **Step6:Go To The Clock Configuration**
   - **Step7:Set The System Clock To Be 72MHz Or Whatever You Want**
   - **Step8:Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

2. **The Application Code In CubeIDE**

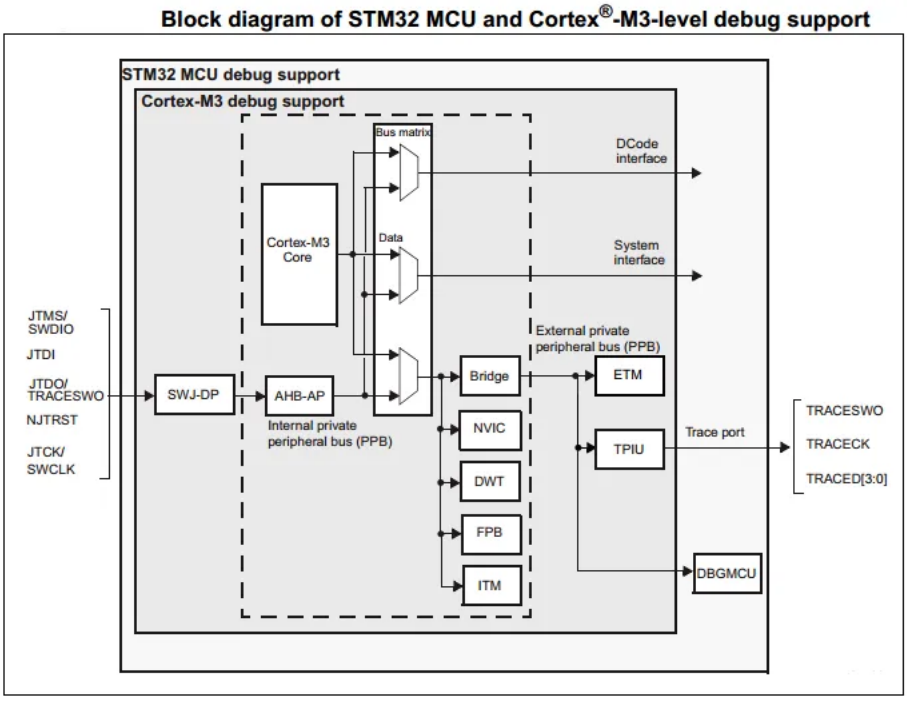
```c
int main(void)
{

  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();

  while (1)
  {
      // IF Button Is Pressed
      if(HAL_GPIO_ReadPin (GPIOA, GPIO_PIN_9))
      {
          // Set The LED ON!
          HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_SET);
      }
      else
      {
          // Else .. Turn LED OFF!
          HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, GPIO_PIN_RESET);
      }
  }

}
```

VI STM32 Debugging With ST-Link v2 SWD — Serial Wire Viewer

1. **STM32 ARM Cortex-M3 Debug Support**
   Two interfaces for debugging are available:

(a) Serial Wire Debug (SWD)

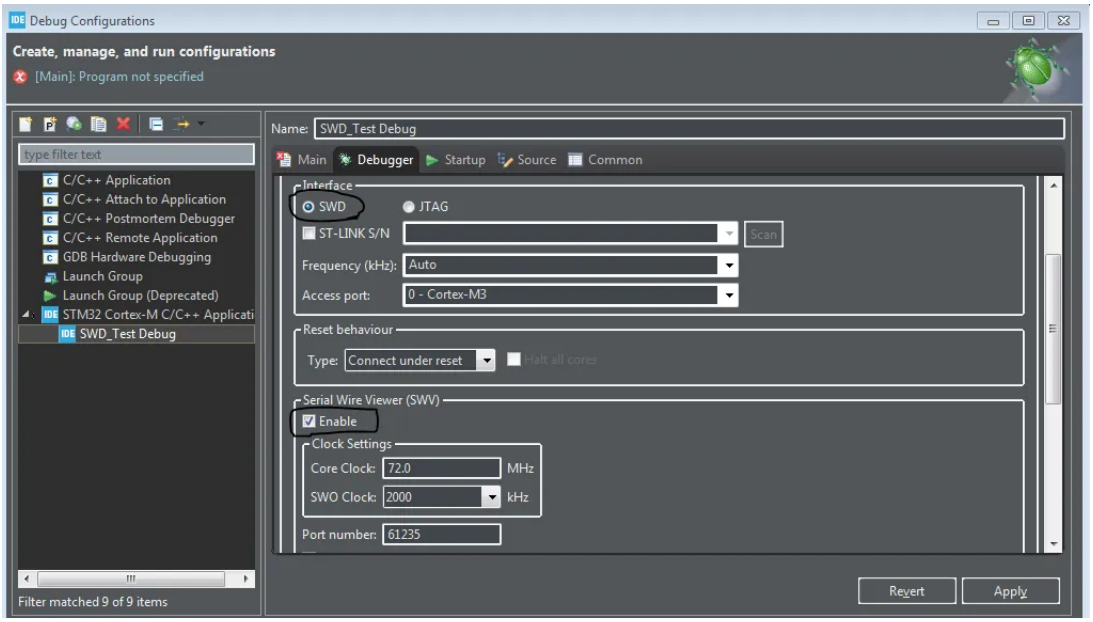
(b) JTAG Debug Port



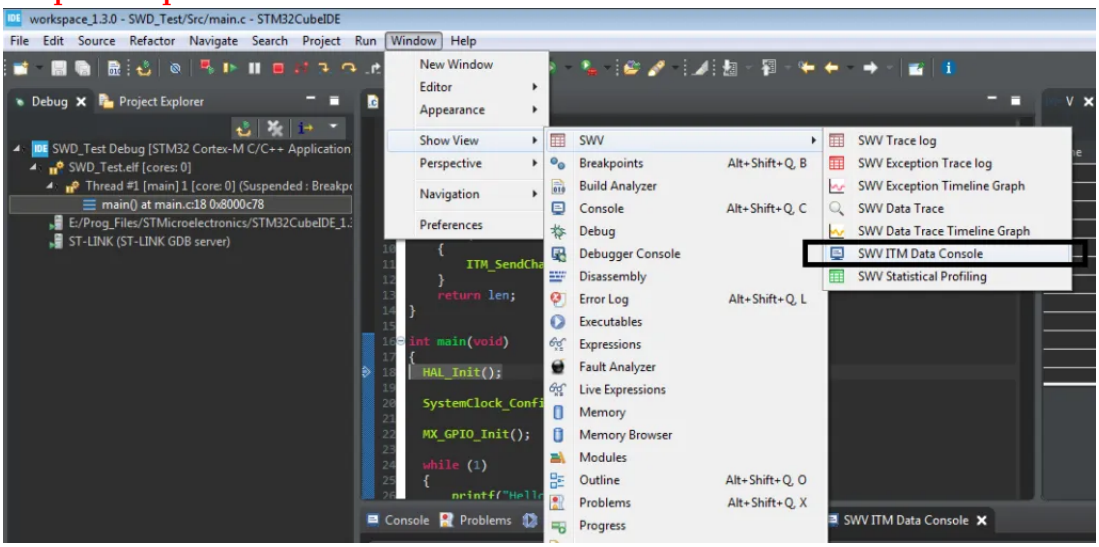
The Arm® Cortex®-M3 core provides integrated on-chip debug support. It is comprised of:

- SWJ-DP: Serial wire / JTAG debug port
- AHP-AP: AHB access port
- ITM: Instrumentation trace macrocell
- FPB: Flash patch breakpoint
- DWT: Data watchpoint trigger
- TPUI: Trace port unit interface (available on larger packages, where the corresponding pins are mapped)
- ETM: Embedded Trace Macrocell (available on larger packages, where the corresponding pins are mapped)

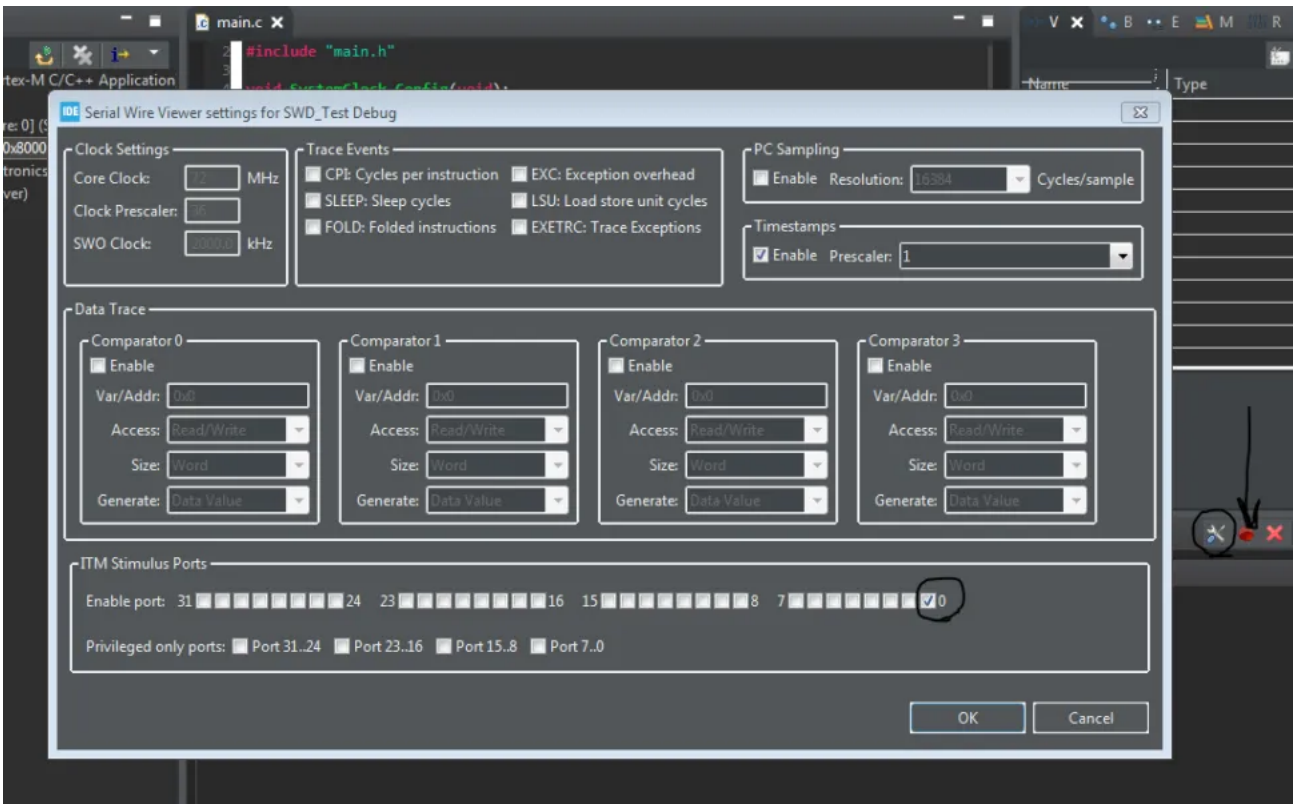
2. **ST-Link v2 (Hardware Debugger)**
3. **Create Demo Project & SWV Debug**
   - **Step9: Open The Debug Configurations & Enable SWV STM32 SWV Debug Configurations**



- **Step10: Build The Project & Start Debugging Session**
- **Step11: Open The SWV ITM Data Console Screen**



- **Step12: Click The Configure Trace Button & Enable Port0 Then Click OK .. And Don't forget to click the red button pointed to with arrow! To start the trace .. And resume the debug so the microcontroller continues printing the data**



4. **Live Expressions & Variables**

```c
int main(void)
{
  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();

  int Counter = 0;
  char MSG[] = "What's UP ?!";

  while (1)
  {
      Counter++;
      printf("Hello World!!\n");
      HAL_Delay(500);
  }

}
```

VII STM32 Interrupts Tutorial — NVIC & EXTI

1. ARM® v7 Cortex™ Exceptions / Interrupts

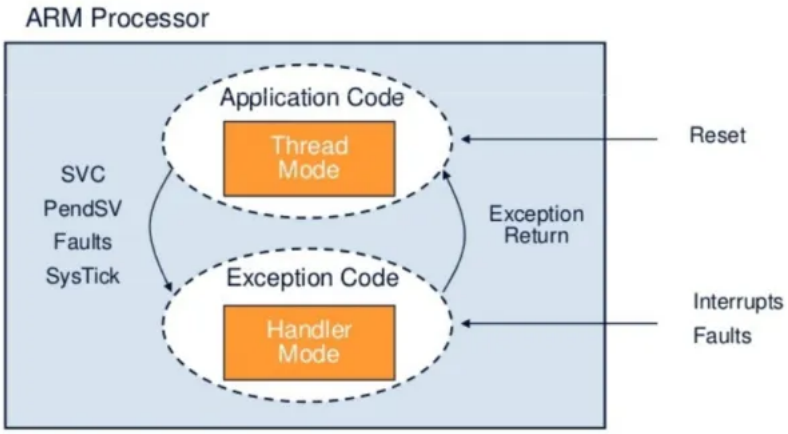
   1.1 Exceptions Overview

   ARM v7 Core supports multiple great features for handling exceptions and interrupts. Which includes the Nested Vectored Interrupt Controller (NVIC).

   1.2 Processor Mode

   The processor mode can change when exceptions occur. And it can be in one of the following modes:

   - Thread Mode: Which is entered on reset.
   - Handler Mode: Which is entered on all other exceptions.



1.3 Micro-Coded Interrupts

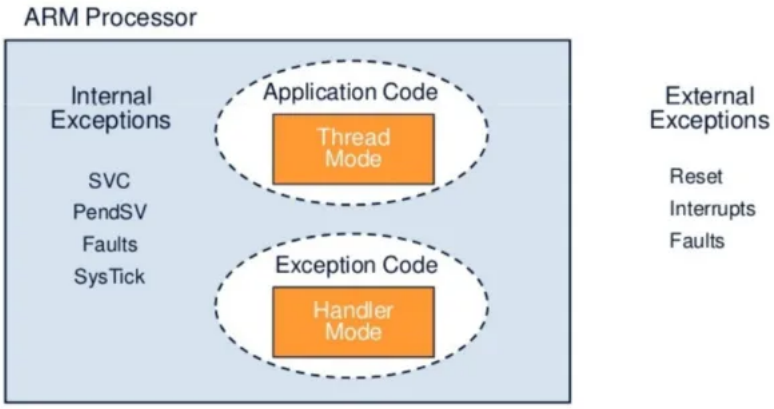
The interrupt entry and exit are hardware implemented in order to reduce the latency and speed up the response. The hardware

- Automatically saves and restores processor context
- Allows late determination of highest priority pending interrupt
- Allows another pending interrupt to be serviced without a full restore/save for processor context (this feature is called tail-chaining)

1.4 Exception Types

Exceptions can be fired by various events including:

- Internal Exceptions
- External Exceptions



1.5 Internal Exceptions

Exceptions that get fired by an internal source to the system and not by any external hardware or peripherals. And this includes:

- Reset: This is the handler routine that gets executed when the processor gets out of a reset state whatever the source is.
- System Service Call (SVC): Similar to SVC instruction on other ARM cores. it allows non-privileged software to make system calls. This provides protection for critical system functionalities.
- Pended System Call (PendSV): Operates with SVC to ease RTOS development as it's intended to be an interrupt for RTOS use.
- Non-Maskable Interrupt (NMI): As the name suggests, this interrupt cannot be disabled. If errors happen in other exception handlers, an NMI will be triggered. Aside from the reset exception, it has the highest priority of all exceptions.

1.6 Fault Exceptions

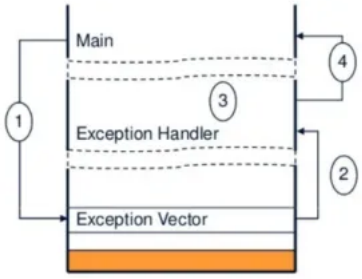
Some of the system exceptions are used to signal and handle specific faults. There are several categories for fault exceptions which include:

- Bus Faults
- Usage Faults
- Memory Management Faults
- Hard Faults

1.7 Exception Servicing Model

The processor begins executing the code instructions with a base execution priority lower than the lowest programmable priority level, so any enabled interrupt can pre-empt the processor.

1.8 Exception Behavior



i. When an exception occurs, the current instruction stream is stopped and the processor accesses the exceptions vector table.
ii. The vector address of that exception is loaded from the vector table.
iii. The exception handler starts to be executed in handler mode.
iv. The exception handler returns back to main (assuming no further nesting).

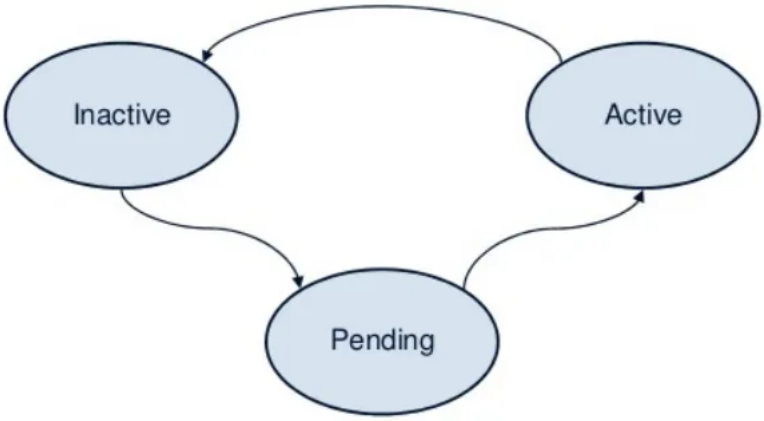
1.9 Reset Behavior

i. When a reset occurs (Reset input is asserted).
ii. The MSP (main stack pointer) register loads the initial value from the address 0x00.
iii. The reset handler address is loaded from address 0x04.
iv. The reset handler gets executed in thread mode.
v. The reset handler branches to the main program.

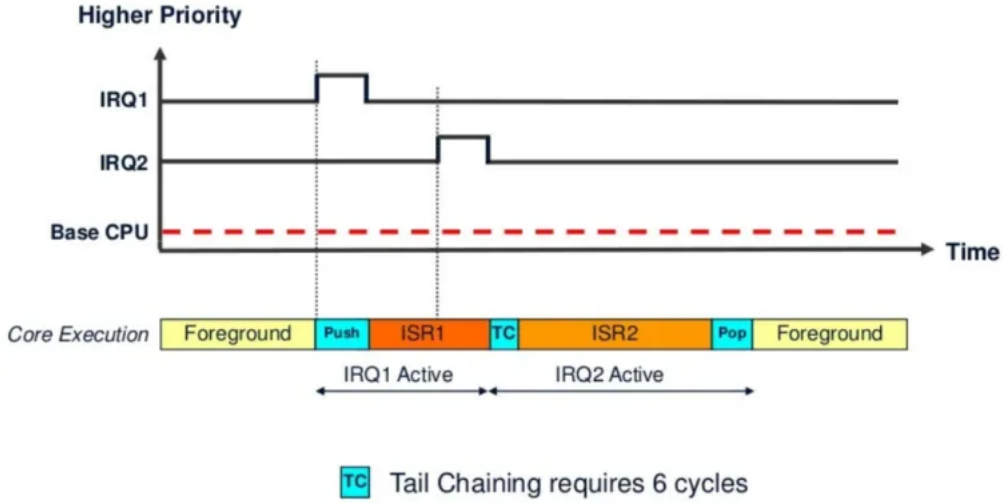
1.10 Exception States

Each exception can be in one of the following states:

- Inactive: Not pending nor active.
- Pending: Exception event has been fired but the handler is not executed yet.
- Active: The exception handler has started execution but it's not over yet. Interrupt nesting allows an exception to interrupt the execution of another exception's handler. In this case, both exceptions are in the active state.
- Active and Pending: The exception is being serviced by the processor and there is a pending exception from the same source.

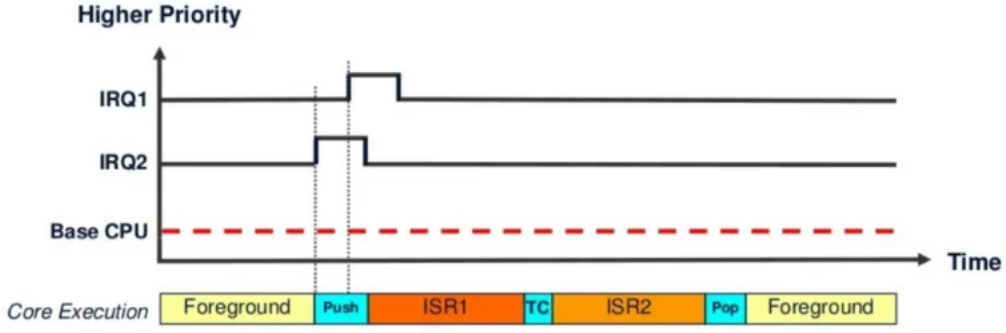


2. **Interrupts Tail-Chaining SpeedUp**



NOte:The STM32 ARM microcontroller interrupts are generated in the following manner: The system runs the ISR and then goes back to the main program. The NVIC and EXTI are configured. The Interrupt Service Routine (ISR) also known as the interrupt service routine handler is defined to enable the external interrupts.

3. **Interrupt Late Arrival SpeedUp**



A pending higher-priority exception is handled before an already pending lower-priority exception even after the exception entry sequence has started. The lower-priority exception is handled after the higher-priority exception.

4. **Pre-Emption**
The pre-emption happens when a task is abandoned (gets interrupted) in order to handle an exception. The currently running instruction stream is said to be pre-empted. When multiple exceptions with the same priority levels are pending, the one with the lowest exception number gets serviced first. And once an exception is active and being serviced

by the processor, only exceptions with a higher priority level can pre-empt it.

5. **Interrupt Vector Table**
   The first entry in the table (lowest address) contains the initial MSP. All other addresses contain the vectors (addresses) to the start of exception handlers (ISRs), each address is 4-Byte wide. The table has up to 496 external interrupts which is implementation-dependent on each specific target.

Address		Exception #
0x40 + 4*N	External N	16 + N
...	...	...
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

The interrupt vector table may be relocated in the memory easily by changing the value of the vector table offset register. The interrupt/exception vector table is usually located in the startup code file. And it looks something like this down below.

```
; Vector Table Mapped to Address 0 at Reset

            AREA     RESET, DATA, READONLY
            EXPORT   __Vectors
            EXPORT   __Vectors_End
            EXPORT   __Vectors_Size

__Vectors   DCD      __initial_sp                ;     Top of Stack
            DCD      Reset_Handler               ;     Reset Handler
            DCD      NMI_Handler                 ; -14 NMI Handler
            DCD      HardFault_Handler           ; -13 Hard Fault Handler
            DCD      MemManage_Handler           ; -12 MPU Fault Handler
            DCD      BusFault_Handler            ; -11 Bus Fault Handler
            DCD      UsageFault_Handler          ; -10 Usage Fault Handler
            DCD      0                           ;     Reserved
            DCD      0                           ;     Reserved
            DCD      0                           ;     Reserved
            DCD      0                           ;     Reserved
            DCD      SVC_Handler                 ;  -5 SVCall Handler
            DCD      DebugMon_Handler            ;  -4 Debug Monitor Handler
            DCD      0                           ;     Reserved
            DCD      PendSV_Handler              ;  -2 PendSV Handler
            DCD      SysTick_Handler             ;  -1 SysTick Handler

            ; Interrupts
; ToDo:  Add here the vectors for the device specific external interrupts handler
            DCD      Interrupt0_Handler          ;   0 Interrupt 0
            DCD      Interrupt1_Handler          ;   1 Interrupt 1
            DCD      Interrupt2_Handler          ;   2 Interrupt 2
            DCD      Interrupt3_Handler          ;   3 Interrupt 3
            DCD      Interrupt4_Handler          ;   4 Interrupt 4
            DCD      Interrupt5_Handler          ;   5 Interrupt 5
            DCD      Interrupt6_Handler          ;   6 Interrupt 6
            DCD      Interrupt7_Handler          ;   7 Interrupt 7
            DCD      Interrupt8_Handler          ;   8 Interrupt 8
            DCD      Interrupt9_Handler          ;   9 Interrupt 9

            SPACE    (214 * 4)                   ; Interrupts 10 .. 224 are left out
__Vectors_End
__Vectors_Size  EQU      __Vectors_End - __Vectors
```

Vector table for other STM32F10xxx devices

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000_0000
-	-3	fixed	Reset	Reset	0x0000_0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
-	-1	fixed	HardFault	All class of fault	0x0000_000C
-	0	settable	MemManage	Memory management	0x0000_0010
-	1	settable	BusFault	Prefetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
-	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV	Pendable request for system service	0x0000_0038
-	6	settable	SysTick	System tick timer	0x0000_003C
0	7	settable	WWDG	Window watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper interrupt	0x0000_0048
3	10	settable	RTC	RTC global interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000_0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000_0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000_0068
11	18	settable	DMA1_Channel1	DMA1 Channel1 global interrupt	0x0000_006C
12	19	settable	DMA1_Channel2	DMA1 Channel2 global interrupt	0x0000_0070
13	20	settable	DMA1_Channel3	DMA1 Channel3 global interrupt	0x0000_0074

6. Interrupt Stacking, ISR, Return

6.1 Interrupt Stacking (Context Saving)

The main program stack is used to store the program state before an interrupt is received. The stack pointer (SP) is automatically decremented on push operation, and it always points to a non-empty value. The stack is 8-Byte aligned and padding may be inserted if it's required.

6.2 Interrupt Service Routine (ISR) Handling

The ISR C-Code should be written in a clear way, that's easily readable and easily executed by the processor. Given that certain exceptions/interrupts are to be serviced hundreds or thousands of times per second. So it must run so quickly and no delays are permitted within ISR handlers unless it's a few microseconds and there is a strong reasoning and justification behind it.

6.3 Return From ISR (Context Restoration)

When an exception (ISR) handler is completely executed and no other interrupts are pending, the CPU restores the context of the main (foreground) application code. The EXC_RETURN instruction is fetched and gets executed to restore the PC and pop the CPU registers.

7. Exceptions Priorities Overview

Name	Exception Number	Exception Priority No.
Interrupts #0 - #495 (N interrupts)	16 to 16 + N	0-255 (programmable)
SysTick	15	0-255 (programmable)
PendSV	14	0-255 (programmable)
SVCall	11	0-255 (programmable)
Usage Fault	6	0-255 (programmable)
Bus Fault	5	0-255 (programmable)
Memory Management Fault	4	0-255 (programmable)
Hard Fault	3	-1
Non Maskable Interrupt (NMI)	2	-2
Reset	1	-3

Lowest

Highest

8. **Interrupts Priority & Priority Grouping**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Priority		Sub-priority		Reserved			

9. **EXTI (External Interrupts) Controller**

EXTI Controller Overview

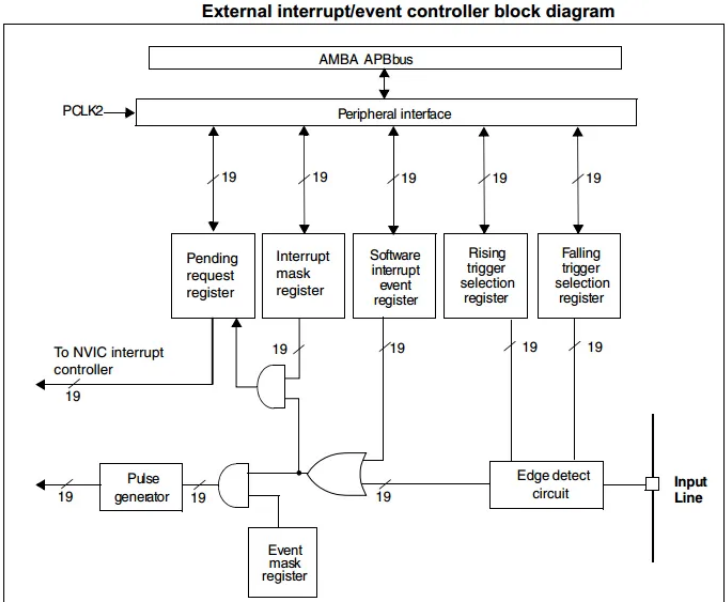
The external interrupt/event controller consists of up to 20 edge detectors in connectivity line devices, or 19 edge detectors in other devices for generating event/interrupt requests. Each input line can be independently configured to select the type (event or interrupt) and the corresponding trigger event (rising or falling or both). Each line can also masked independently. A pending register maintains the status line of the interrupt requests.

EXTI Controller Features

The EXTI controller main features are the following:

- Independent trigger and mask on each interrupt/event line
- Dedicated status bit for each interrupt line
- Generation of up to 20 software event/interrupt requests
- Detection of external signal with pulse width lower than the APB2 clock period.

EXTI Block Diagram



Functional Description

To generate the interrupt, the interrupt line should be configured and enabled. This is done by programming the two trigger registers with the desired edge detection and by enabling the interrupt request by writing a '1' to the corresponding bit in the interrupt

mask register. When the selected edge occurs on the external interrupt line, an interrupt request is generated. The pending bit corresponding to the interrupt line is also set. This request is reset by writing a '1' in the pending register.

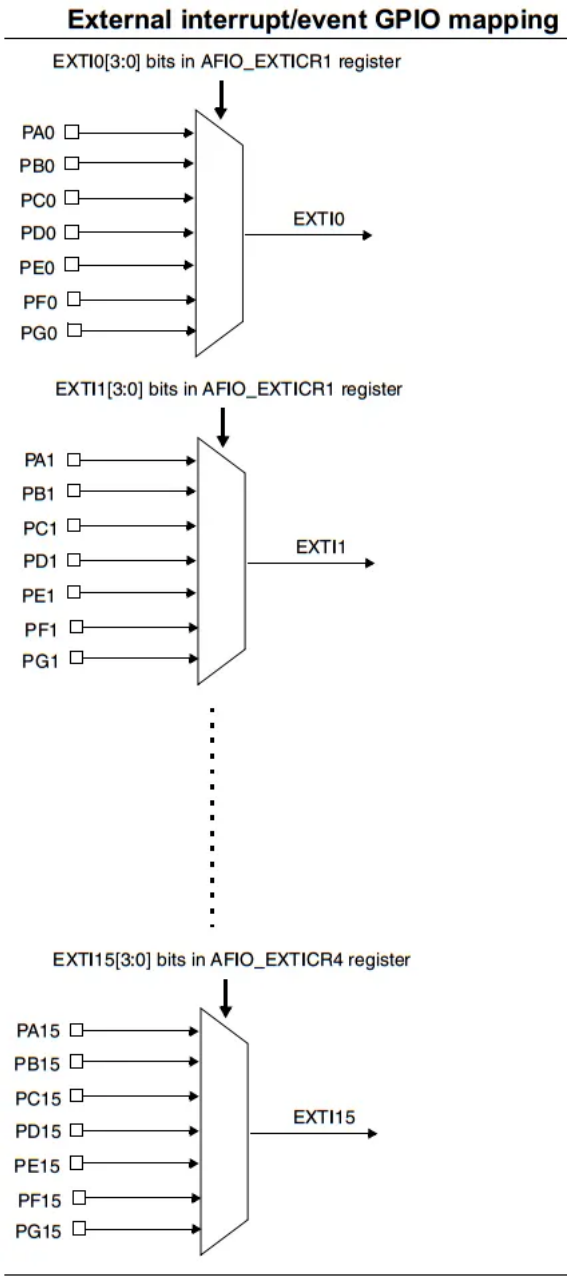
Hardware interrupt selection

To configure the 20 lines as interrupt sources, use the following procedure:

- Configure the mask bits of the 20 Interrupt lines (EXTI_IMR)
- Configure the Trigger Selection bits of the Interrupt lines (EXTI_RTSR and EXTI_FTSR)
- Configure the enable and mask bits that control the NVIC IRQ channel mapped to the External Interrupt Controller (EXTI) so that an interrupt coming from one of the 20 lines can be correctly acknowledged.

EXTI External Interrupts GPIO Mapping

GPIOs are connected to the 16 external interrupt/event lines in the following manner:



The four other EXTI lines are connected as follows:

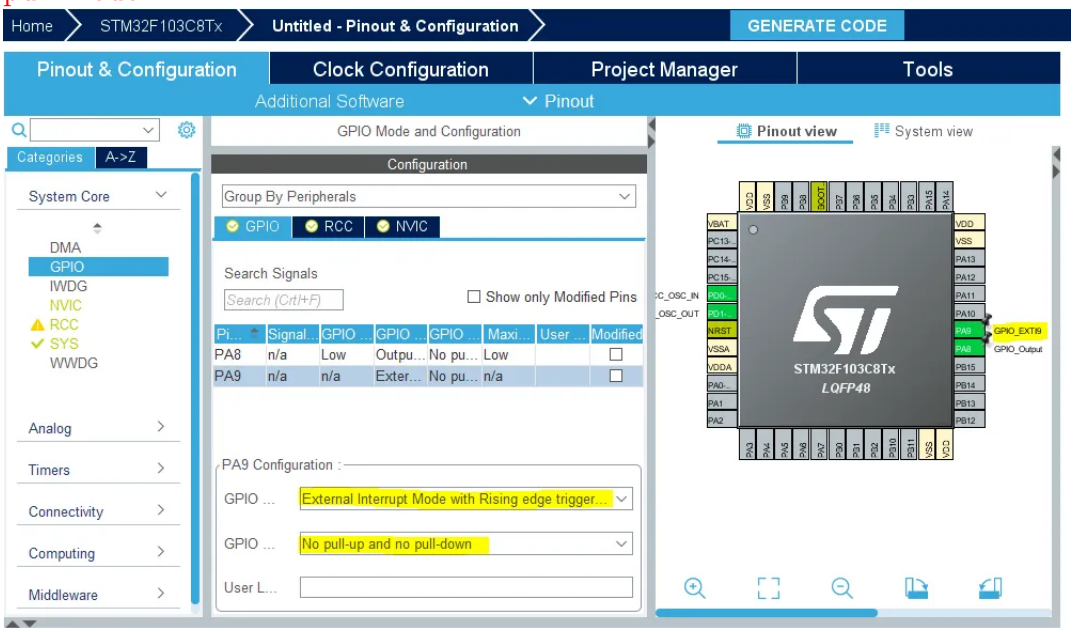
- EXTI line 16 is connected to the PVD output

- EXTI line 17 is connected to the RTC Alarm event
- EXTI line 18 is connected to the USB Wakeup event
- EXTI line 19 is connected to the Ethernet Wakeup event

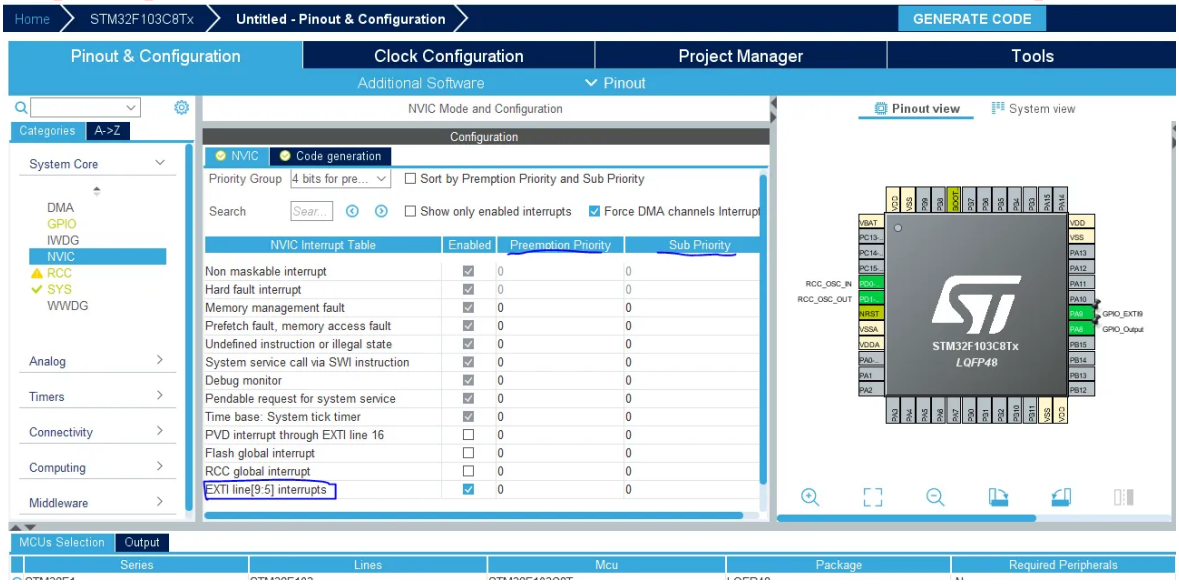
VIII External Interrupt PIN

1. STM32 CubeMX Configurations

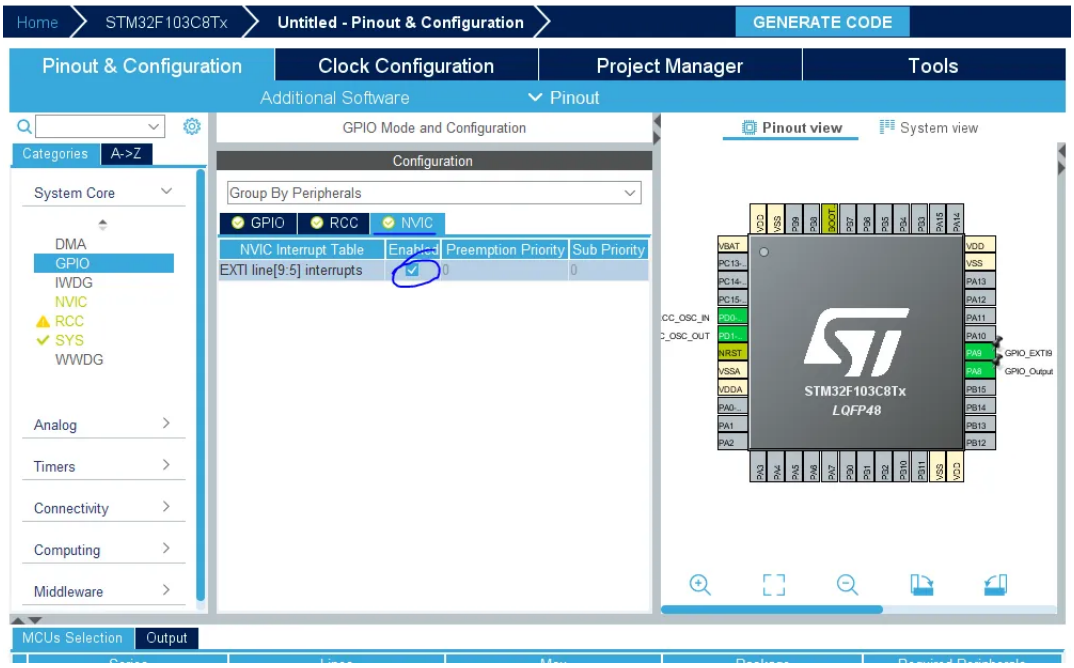
- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Click On The Pin You Want To Configure As An Output & Select Output Option**
- Step4: Click On The Pin You Want To Configure As An External Interrupt Input
- Step5: Go To GPIO Config Tab, And Select The A9 Pin EXTI interrupt edge and pull mode



- **Step6: Open The NVIC Tab And Enable The EXTI line9 Interrupt**



- **Step7: Open The NVIC Config Tab And Check if you want to change the priority of the EXTI line9 external interrupt or not (Optional Step)**



- **Step8: Set The RCC External Clock Source**
- **Step10: Set The System Clock To Be 72MHz Or Whatever You Want**

2. **The Application Code In CubeIDE**

```c
int main(void)
{
  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();

  while (1)
  {
     // Stay IDLE .. Everything is done in the ISR Handler
  }
}

// EXTI Line9 External Interrupt ISR Handler CallBackFun
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_9) // If The INT Source Is EXTI Line9 (A9 Pin)
    {
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_8); // Toggle The Output (LED) Pin
    }
}
```

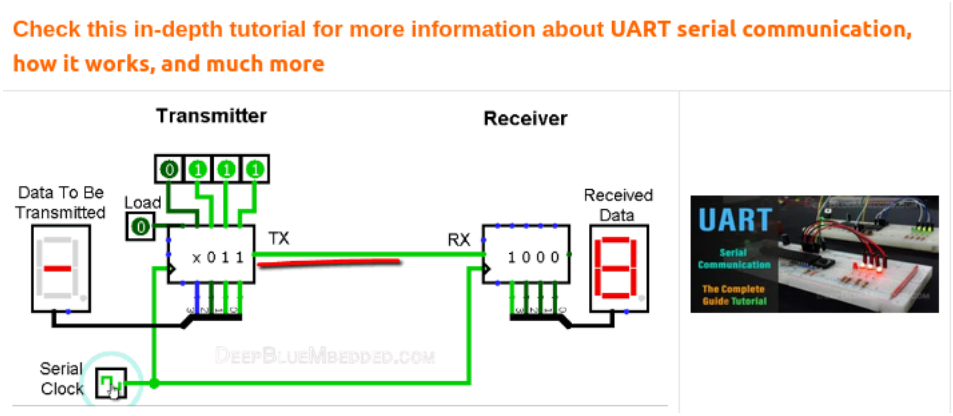
The video Test:https://youtu.be/gE0cGQ8jiYU
This includes the interrupt latency, context switching overhead, and ISR callback procedure as well as the HAL_TogglePin function call. All in all, it's great response time. You can improve this by directly writing to the GPIO output pin port without using the HAL_GPIO function. This will be a topic for a future tutorial!

IX STM32 USART / UART Tutorial

The STM32F103C8 microcontrollers' pins are not all 5v tolerant. Hence, you must be careful when receiving input signals from the USB-TTL converter. You can send a 3.3v signal from the MCU TX pin to the USB-TTL RX pin and still get the data identified absolutely fine. However, it won't work the other way around without shifting the signal's level. The TX from the USB-TTL can over-drive the MCU's RX input pin. By checking the diagram below, you'll notice that the pins for UART1 UART3 are 5v tolerant while UART2 is not.

1. Introduction To UART
   There are actually two forms of UART hardware as follows:

   - **UART** -Universal Asynchronous Receiver/Transmitter.
   - **USART** –Universal Synchronous/Asynchronous Receiver/Transmitter.
     The **Synchronous** type of transmitters generates the data clock and sends it to the receiver which works accordingly in a synchronized manner.
     The **Asynchronous** type of transmitter generates the data clock internally.



2. USART / UART Hardware In STM32

   2.1 STM32 USART Highlights
   It supports synchronous one-way communication and half-duplex single-wire communication. It also supports the LIN (local interconnection network), Smartcard Protocol and IrDA (infrared data association) SIR ENDEC specifications, and modem operations (CTS/RTS). It allows multiprocessor communication. High-speed data communication is possible by using the DMA for multi-buffer configuration.

   2.2 STM32 USART Main Features

3. STM32 USART Hardware Functionalities
   Any USART bidirectional communication requires a minimum of two pins: Receive Data In (RX) and Transmit Data Out (TX). Through these pins, serial data is transmitted and received in normal USART mode. The CK pin is required to interface in synchronous mode. The CTS & RTS pins are required in Hardware flow control mode.

   3.1 USART Block Diagram

   3.2 USART Data (Character) Packet

   - An Idle character is interpreted as an entire frame of “1”s followed by the start bit of the next frame which contains data (The number of “1” ‘s will include the number of stop bits).
   - A Break character is interpreted on receiving “0”s for a frame period. At the end of the Break frame, the transmitter inserts either 1 or 2 stop bits (logic “1” bit) to acknowledge the start bit.

   3.3 USART Transmitter
   NOTE

   - The TE bit should not be reset during the transmission of data. Resetting the TE bit during the transmission will corrupt the data on the TX pin as the baud rate counters will get frozen. The current data being transmitted will be lost.
   - An idle frame will be sent after the TE bit is enabled.

USART Data Transmission Steps (Procedure)

i. Enable the USART by writing the UE bit in USART_CR1 register to 1.
ii. Program the M bit in USART_CR1 to define the word length.
iii. Program the number of stop bits in USART_CR2.
iv. Select DMA to enable (DMAT) in USART_CR3 if Multi buffer Communication is to take place. Configure the DMA register as explained in multi-buffer communication.
v. Select the desired baud rate using the USART_BRR register.
vi. Set the TE bit in USART_CR1 to send an idle frame as the first transmission.
vii. Write the data to send in the USART_DR register (this clears the TXE bit). Repeat this for each data to be transmitted in case of a single buffer.
viii. After writing the last data into the USART_DR register, wait until TC=1. This indicates that the transmission of the last frame is complete. This is required for instance when the USART is disabled or enters the Halt mode to avoid corrupting the last transmission.

The TC bit is cleared by the following software sequence:
1.A read from the USART_SR register
2.A write to the USART_DR register
The TC bit can also be cleared by writing a '0' to it.

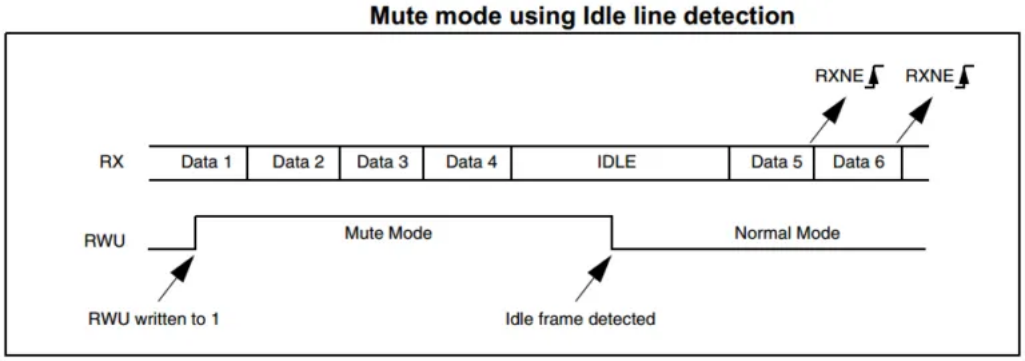
3.4 USART Receiver
NOTE: The RE bit should not be reset while receiving data. If the RE bit is disabled during the reception, the reception of the current byte will be aborted.

3.5 Fractional Buad Rate Generator (BRG)
The baud rate for the receiver and transmitter (RX and TX) are both set to the same value as programmed in the Mantissa and Fraction values of USARTDIV.
USARTDIV is an unsigned fixed-point number that is coded on the USART_BRR register. FCK is the input clock to the USART peripheral.
NOTE:

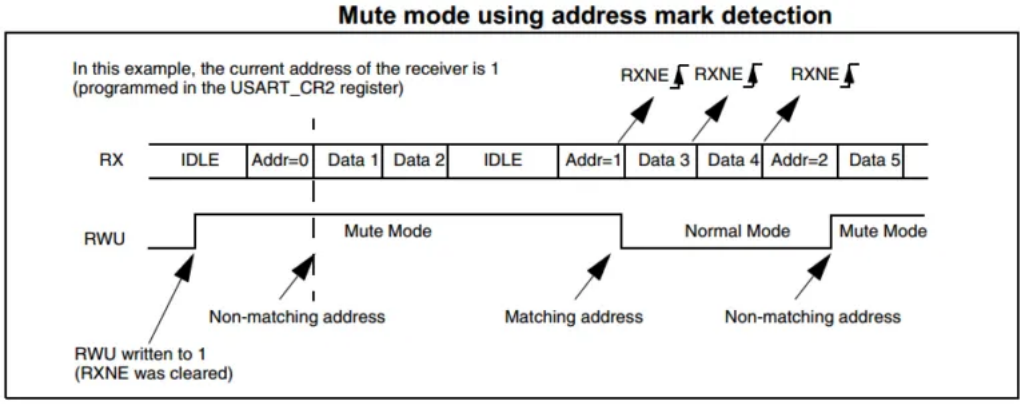
- The lower the CPU clock the lower will be accuracy for a particular Baud rate. The upper limit of the achievable baud rate can be fixed with this data.
- Only USART1 is clocked with PCLK2 (72 MHz max). Other USARTs are clocked with PCLK1 (36 MHz max).

3.6 USART Parity controll EVEN Parity: the parity bit is calculated to obtain an even number of "1s" inside the frame made of the 7 or 8 LSB bits (depending on whether M is equal to 0 or 1) and the parity bit. ODD Parity: the parity bit is calculated to obtain an odd number of "1s" inside the frame made of the 7 or 8 LSB bits (depending on whether M is equal to 0 or 1) and the parity bit.

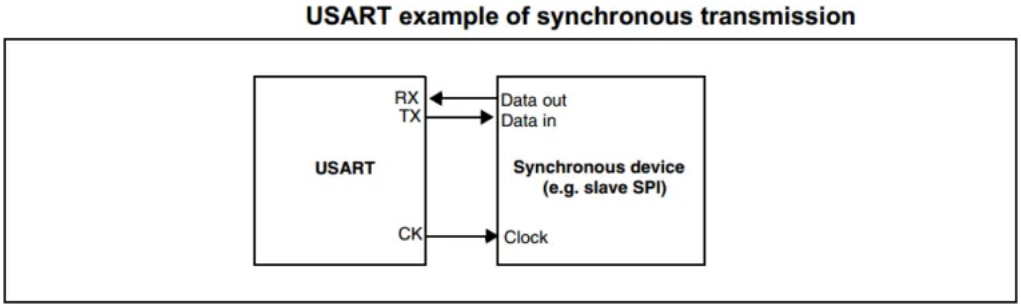
3.7 USART Multi-Processor Communication
IDLE Line Detection (WAKE=0)

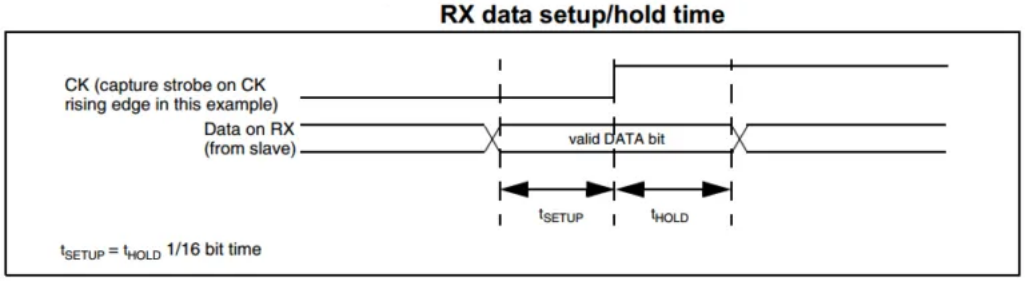


Address Mark Detection (WAKE=1)



3.8 USART Synchronous Communication





3.9 USART Single-Wire (Half-Duplex) Communication

3.10 USART ERROR

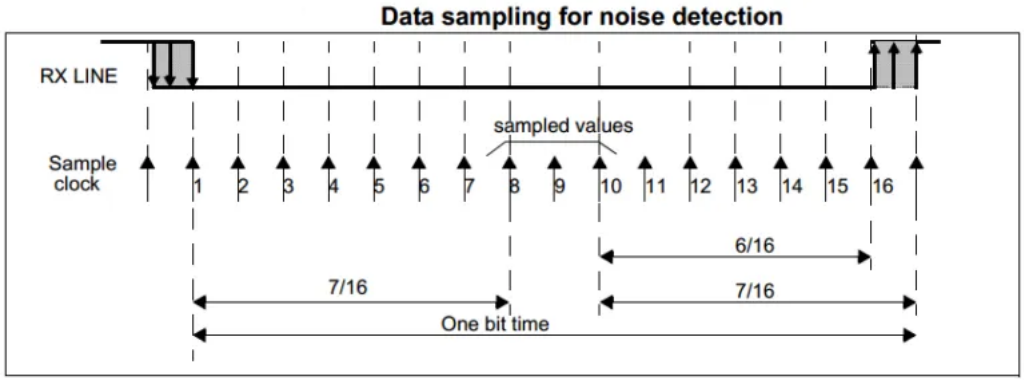
The USART hardware in STM32 microcontrollers is capable of detecting 4 errors in operations. These error signals are as follows

3.10.1 Overrun Error

When an overrun error occurs:

- The ORE bit is set.
- The RDE content will be not lost.The previous data is available when a read to USART_DR is performed.
- The shift register will be overwritten. After that point, any data received during the overrun is lost.

- An interrupt is generated if either the RXNEIE bit is set or both the EIE and DMAR bits are set.
- The ORE bit is reset by a read to the USART_SR register followed by a USART_DR register read operation.
- The ORE bit, when set, indicates that at least 1 data has been lost. There are two possibilities:
  -if RXNE=1, then the last valid data is stored in the receive register RDR and can be read.
  -if RXNE=0, then it means that the last valid data has already been read and thus there is nothing to be read in the RDR.



3.10.2 Noise Error

When the framing error is detected:

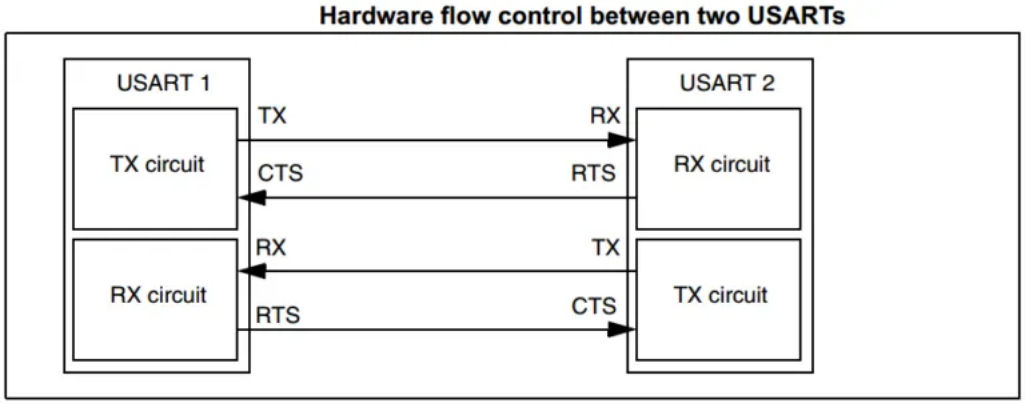
- The FE bit is set by hardware
- The invalid data is transferred from the Shift register to the USART_DR register.
- No interrupt is generated in the case of single-byte communication.

3.10.3 Parity Check Error

When a parity error is detected in the received data frame, the PE bit is set and it'll fire an interrupt if it's enabled. It can be set to even or odd parity depending on the application and whether it's implemented in the communication or not. We won't be using the parity check in all the tutorials and LABs dealing with USART.

3.11 USART Hardware Data Flow Control

It is possible to control the serial data flow between two devices by using the CTS input and the RTS output. RTS and CTS flow control can be enabled independently by writing respectively RTSE and CTSE bits to 1 (in the USART_CR3 register). The diagram below shows how to connect two devices in this mode:



3.11.1 RTS Flow Control

If the RTS flow control is enabled (RTSE=1), then RTS is asserted (tied low)

as long as the USART receiver is ready to receive new data. When the receive register is full, RTS is de-asserted, indicating that the transmission is expected to stop at the end of the current frame.

3.11.2 CTS Flow Control

If the CTS flow control is enabled (CTSE=1), then the transmitter checks the CTS input before transmitting the next frame. If CTS is asserted (tied low), then the next data is transmitted (assuming that data is to be transmitted, in other words, if TXE=0), else the transmission does not occur.

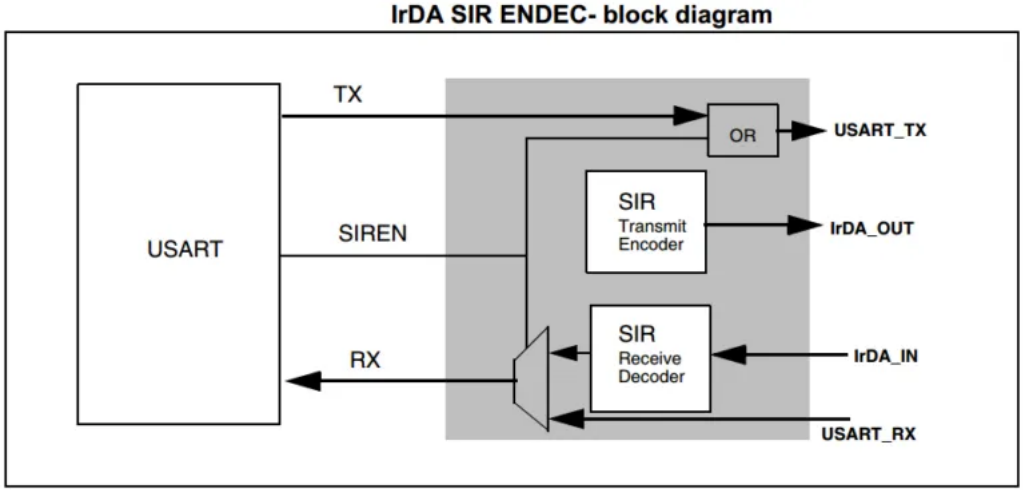
3.12 LIN MODe

The LIN (Local Interconnection Network) Mode is selected by setting the LINEN bit in the USART_CR2 register.

In LIN mode, the following bits must be kept clear (0):

- STOP[1:0], CLKEN in the USART_CR2 register.
- SCEN, HDSEL and IREN in the USART_CR3 register.

3.13 IrDA Mode



4. STM32 USART Mode Configuration

-For the STM32F103C8 microcontroller in the blue pill board we're using, there are only 3 USARTs.

-For the STM32L432KC microcontroller in the Nucleo32-L432KC board we're using, there are only 3 USARTs.

USART mode configuration

USART modes	USART1	USART2	USART3	UART4	UART5
Asynchronous mode	X	X	X	X	X
Hardware Flow Control	X	X	X	NA	NA
Multibuffer Communication (DMA)	X	X	X	X	NA
Multiprocessor Communication	X	X	X	X	X
Synchronous	X	X	X	NA	NA
Smartcard	X	X	X	NA	NA
Half-Duplex (Single-Wire mode)	X	X	X	X	X
IrDA	X	X	X	X	X
LIN	X	X	X	X	X

X = supported; NA = not applicable.

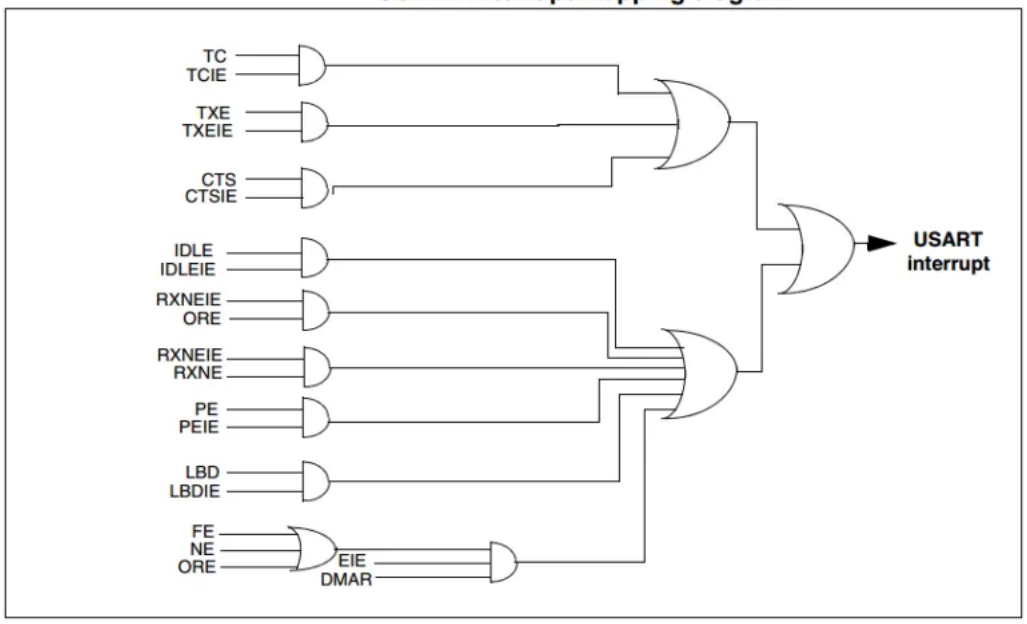
5. USART Interrupts

USART interrupt requests

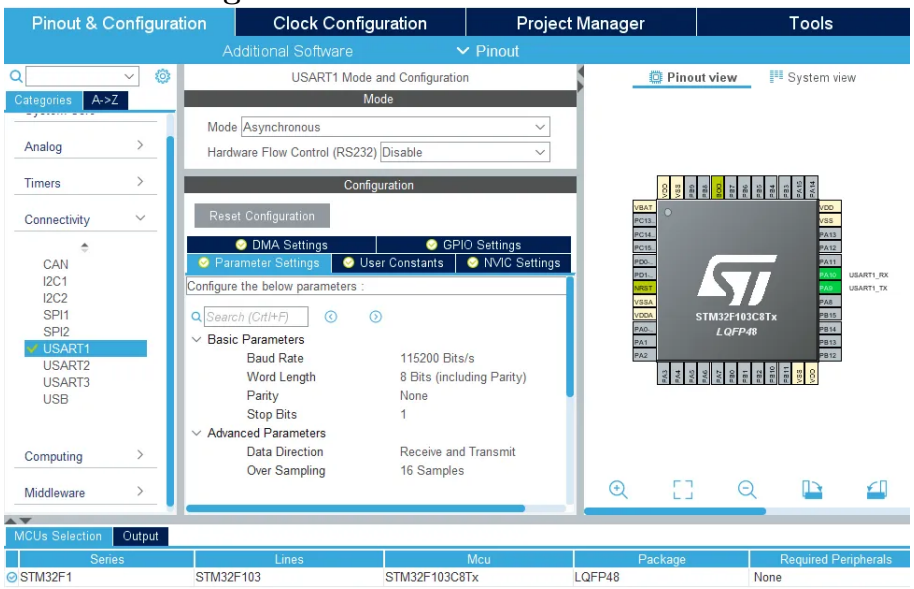
Interrupt event	Event flag	Enable Control bit
Transmit data register empty	TXE	TXEIE
CTS flag	CTS	CTSIE
Transmission complete	TC	TCIE
Received data ready to be read	RXNE	RXNEIE
Overrun error detected	ORE	
Idle line detected	IDLE	IDLEIE
Parity error	PE	PEIE
Break flag	LBD	LBDIE
Noise flag, Overrun error and Framing error in multibuffer communication	NE or ORE or FE	EIE(1)

1. This bit is used only when data reception is performed by DMA.

USART interrupt mapping diagram



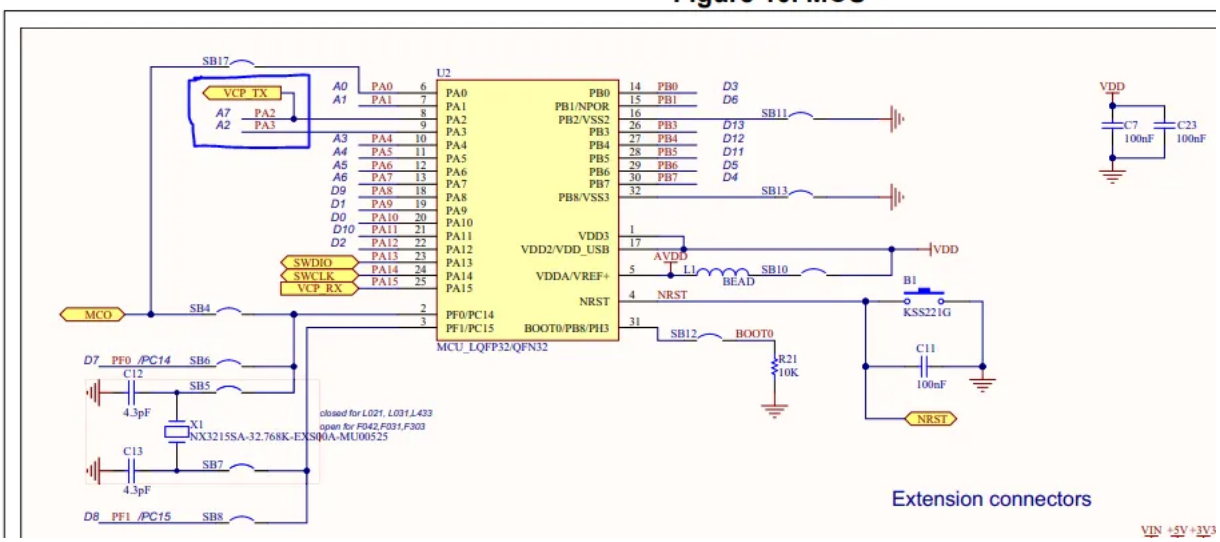
6. USART Configuration In CubeMX



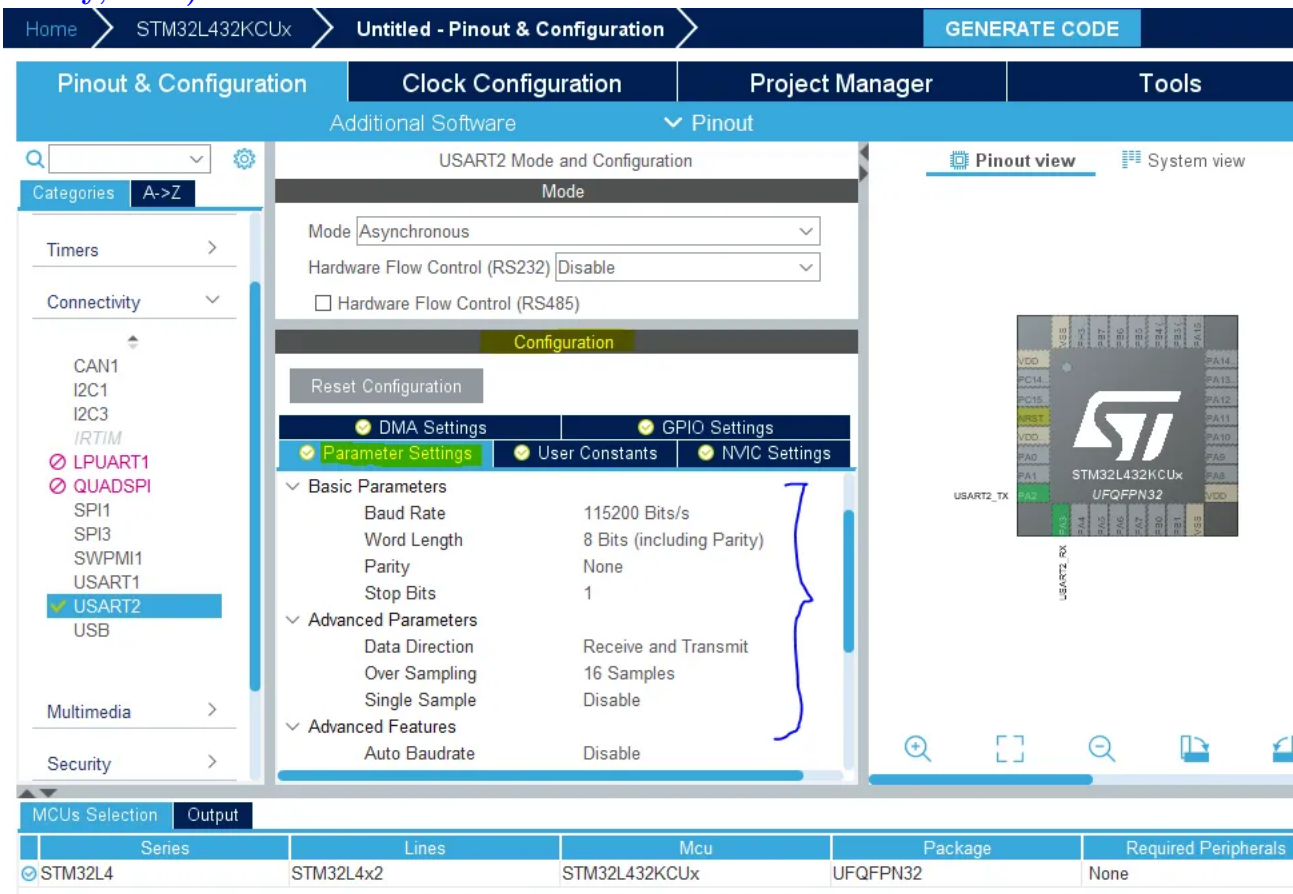
X STM32 Debugging With UART Serial Print

1. **STM32 Nucleo32-L432KC Serial Port**

Figure 10. MCU



- **Step1: Open CubeMX & Create New Project**
- **Step2:Choose The Target MCU & Double-Click Its Name**
- **Step3:Enable USART2 Module (Asynchronous Mode)**
- **Step4: Choose The Desired Settings For UART (Baud Rate, Stop Bits, Parity, etc..)**

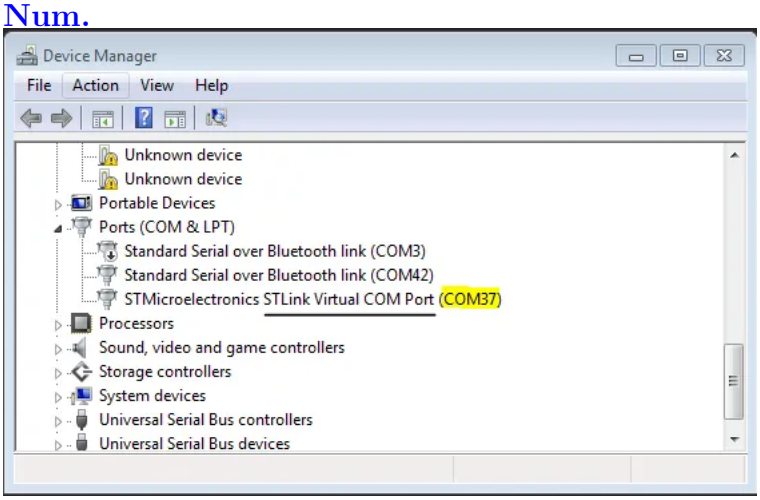


- **Step5: Go To The Clock Configuration & Set The System Clock To 80MHZ STM32 Nucleo32 Serial Port UART**

- **Step6: Generate The Initialization Code & Open The Project In CubeIDE**
- **Step7: Write The Code For Your Project & Use HAL_UART_Transmit() To Print**

```c
#include "main.h"

UART_HandleTypeDef huart2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);

int main(void)
{
  uint8_t MSG[35] = {'\0'};
  uint8_t X = 0;

  HAL_Init();

  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART2_UART_Init();

  while (1)
  {
     sprintf(MSG, "Hello Dudes! Tracing X = %d\r\n", X);
     HAL_UART_Transmit(&huart2, MSG, sizeof(MSG), 100);
     HAL_Delay(500);
     X++;
  }
}
```

- **Step8: Build & Debug To Flash The Code**
- **Step9: Go To The Device Manager & Check The ST-Link COM Port Num.**



- **Step10: Open The Terminal From CubeIDE**
  Window > Show View > Console
  In Console:
  click on the NEW icon on its menu bar > Command Shell console > Connection type: Serial port > set Baud Rate & Connection Name > Encoding: UTF-8 > And Click OK!

XI STM32 Serial Port – UART With USB-TTL Converter — PC Interfacing

1. **Preface**
   The STM32F103C8 microcontrollers' pins are not all 5v tolerant. Hence, you must be careful when receiving input signals from the USB-TTL converter. You can send a 3.3v signal from the MCU TX pin to the USB-TTL RX pin and still get the data identified absolutely fine. However, it won't work the other way around without shifting the signal's

level. The TX from the USB-TTL can over-drive the MCU's RX input pin. By checking the diagram below, you'll notice that the pins for UART1 & UART3 are 5v tolerant while UART2 is not.

2. **LAB Project Code (Step-By-Step)**

- **Step1:** Open CubeMX Create New Project.
- **Step2:** Choose The Target MCU Double-Click Its Name.
- **Step3:** Enable USART1 Module (Asynchronous Mode)
- **Step4:** Choose The Desired Settings For UART (Baud Rate, Stop Bits, Parity, etc..).
  _Set the baud rate to 9600 bps
  _Enable UART global interrupts in NVIC tab
- **Step5:** Step5: Configure The Required GPIO Pins For This Project.
  _PB12, PB13: Output Pins (For LEDs)
  _PB14: Input Pin (For The Push Button)
- **Step6:** Generate The Initialization Code Open The Project In CubeIDE.
- **Step7:** Write The Application Layer Code For This LAB:

```c
#include "main.h"

UART_HandleTypeDef huart1;

uint8_t RX1_Char = 0x00;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);

//---------[ UART Data Reception Completion CallBackFunc. ]---------
void HAL_USART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Receive_IT(&huart1, &RX1_Char, 1);
}

int main(void)
{
  uint8_t MSG1[] = "Button State: Released\r\n";
  uint8_t MSG2[] = "Button State: Pressed\r\n";

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  HAL_UART_Receive_IT(&huart1, &RX1_Char, 1);
```

```
  while (1)
  {
    //--------[ Read The Button State & Send It Via UART ]---------
    if(HAL_GPIO_ReadPin (GPIOB, GPIO_PIN_14))
    {
      HAL_UART_Transmit(&huart1, MSG2, sizeof(MSG2), 100);
    }
    else
    {
      HAL_UART_Transmit(&huart1, MSG1, sizeof(MSG1), 100);
    }
    //--------[ Read The Received Character & Toggle LEDs Accordingly ]--------
    if(RX1_Char == '1')
    {
         HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_12);
      HAL_UART_Receive_IT(&huart1, &RX1_Char, 1);
      RX1_Char = 0x00;
    }
    if(RX1_Char == '2')
    {
      HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_13);
      HAL_UART_Receive_IT(&huart1, &RX1_Char, 1);
      RX1_Char = 0x00;
    }
    HAL_Delay(100);
  }
}
```

- **Step8:** Open The Terminal From CubeIDE or Any Other Terminal:
  _Window ¿ Show View ¿ Console
  _**In Console:** click on the NEW icon on its menu bar ¿ Command Shell console ¿ Connection type: Serial port ¿ set Baud Rate Connection Name ¿ Encoding: UTF-8 ¿ And Click OK!
  $_{A}lternatively : YouCanUseAnyTerminalOnYourPC(LikeTeraTerm, ArduinoSerialMonitor, etc..)$
  $ThisisVideofortesting :$

XII How To Receive UART Serial Data With STM32 – DMA / Interrupt / Polling

1. **The Polling Method**

   - **Definition:** The polling method is essentially a blocking function being called from the main routine and it does block the CPU so it can't proceed in the main task execution until a certain amount of UART data bytes are received. After receiving the required amount of data, the function ends and the CPU continues the main code execution. Otherwise, and if the UART peripheral for some reason didn't receive the expected amount of data, the function will keep blocking the CPU for a certain amount of time "time-out" after which it gets terminated and gives back the control to the CPU to resume the main code.
   - **This function for example:** HAL_UART_Receive (&huart1, UART1_rxBuffer, 12, 5000);
     Receives 12 bytes to the buffer. if it does that in 100uSec, it's ok and the CPU will resume main code execution. If it doesn't receive that amount of data, the CPU will be kept blocked waiting for 5sec until this function returns to the main context. And obviously, it's not an efficient way to receive serial data despite the fact that it does actually work.
   - **STM32 UART Receive Polling Example:**

- Step1:Start New CubeMX Project Setup The Clock.
- Step2:Setup The UART1 Peripheral in Async Mode @ 9600bps.
- Step3:Generate Code Open CubeIDE or Any Other IDE You're Using
- Step4:Write This Application Code (main.c)

```c
#include "main.h"

uint8_t UART1_rxBuffer[12] = {0};

UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive (&huart1, UART1_rxBuffer, 12, 5000);
    HAL_UART_Transmit(&huart1, UART1_rxBuffer, 12, 100);

    while (1)
    {

    }

}
```

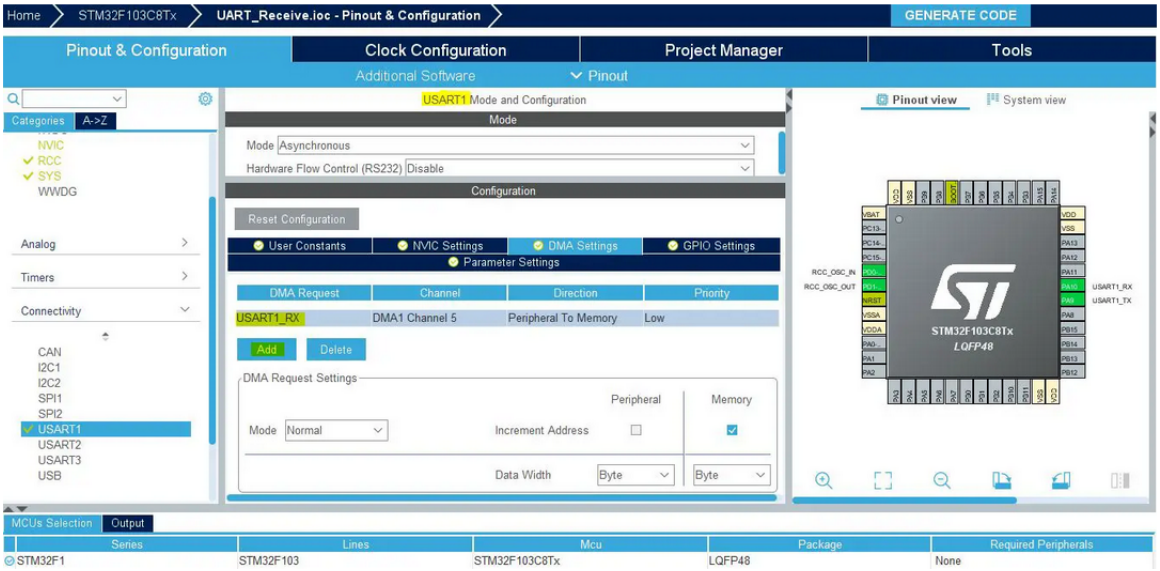
2. **The Interrupt Method**

- **Definition:** Using interrupt signals is a convenient way to achieve serial UART data reception. The CPU initializes the UART receive hardware so that it fires an interrupt signal whenever a new byte of data is received. And in the ISR code, we save the received data in a buffer for further processing.
- **This is function example:** HAL_UART_Receive_IT(&huart1, UART1_rxBuffer, 12);
- **STM32 UART Receive Interrupt Example:**
  - Step1:Start New CubeMX Project & Setup The Clock.
  - Step2:Setup The UART1 Peripheral in Async Mode @ 9600bps
  - Step3:Enable The UART1 Global Interrupt From NVIC Controller Tab
  - Step4:Generate Code & Open CubeIDE or Any Other IDE You're Using
  - Step5:Write This Application Code (main.c)

```c
#include "main.h"

uint8_t UART1_rxBuffer[12] = {0};

UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive_IT (&huart1, UART1_rxBuffer, 12);

    while (1)
    {

    }

}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Transmit(&huart1, UART1_rxBuffer, 12, 100);
    HAL_UART_Receive_IT(&huart1, UART1_rxBuffer, 12);
}
```

3. **The DMA Method**

- **Definition:** Using the DMA unit in order to direct the received serial UART data from the UART peripheral directly to the memory is considered to be the most efficient way to do such a task. It requires no CPU intervention at all, you'll have only to set it up and go execute the main application code. The DMA will notify back the CPU upon reception completion and that received data buffer will be in the pre-programmed location.
  Note: The DMA can prioritize the channels, decide on the data width, and also the amount of data to be transferred up to 65536 bytes. Which is amazing in fact, and all you've got to do is to initialize it like this.
- **This is function Example:** HAL_UART_Receive_DMA (&huart1, UART1_rxBuffer, 12);
- **STM32 UART Receive DMA Example:**
  - Step1:Start New CubeMX Project & Setup The Clock
  - Step2:Setup The UART1 Peripheral in Async Mode 9600bps
  - Step3:Add A DMA Channel For UART RX From The DMA Tab



– Step4:Generate Code & Open CubeIDE or Any Other IDE You're Using

– Step5:Write This Application Code (main.c)

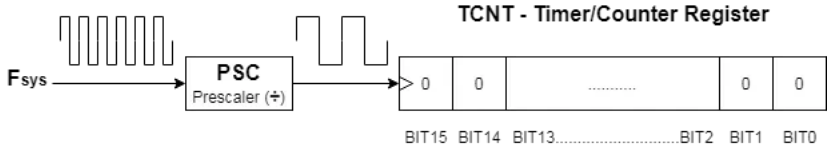
```c
#include "main.h"

uint8_t UART1_rxBuffer[12] = {0};

UART_HandleTypeDef huart1;
DMA_HandleTypeDef hdma_usart1_rx;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive_DMA (&huart1, UART1_rxBuffer, 12);

    while (1)
    {

    }

}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Transmit(&huart1, UART1_rxBuffer, 12, 100);
    HAL_UART_Receive_DMA(&huart1, UART1_rxBuffer, 12);
}
```

XIII STM32 Timers Tutorial — Hardware Timers Explained

1. **Introduction to timer Module**

   A Timer Module in its most basic form is a digital logic circuit that counts up every clock

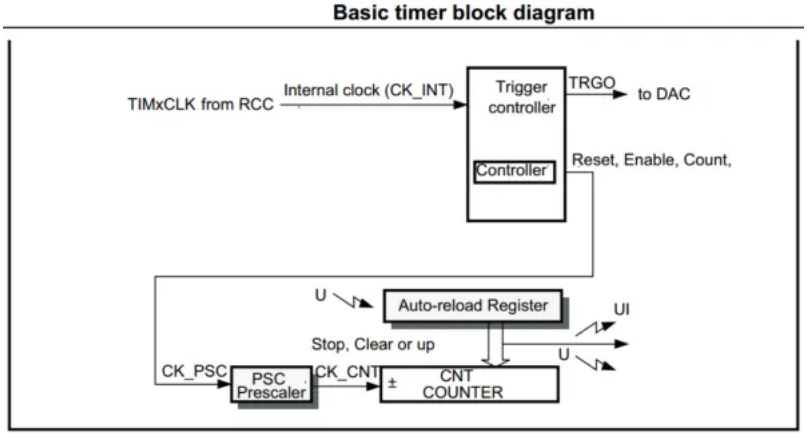
cycle.
Let's consider a basic 16-Bit timer like the one shown below. As a 16-Bit time, it can count from 0 up to 65535.



it will give you an interrupt signal once every 0.839 Second.

2. STM32 Timers Hardware

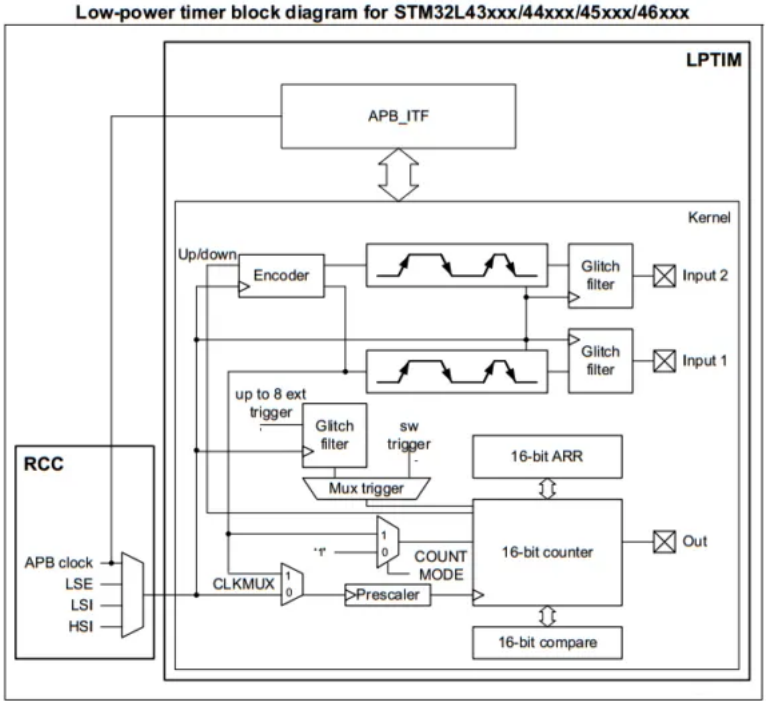
2.1 Basic Timers Modules



The time-base unit includes:

- Counter Register (TIMx_CNT)
- Prescaler Register (TIMx_PSC)
- Auto-Reload Register (TIMx_ARR)

2.2 Low-Power Timers Modules

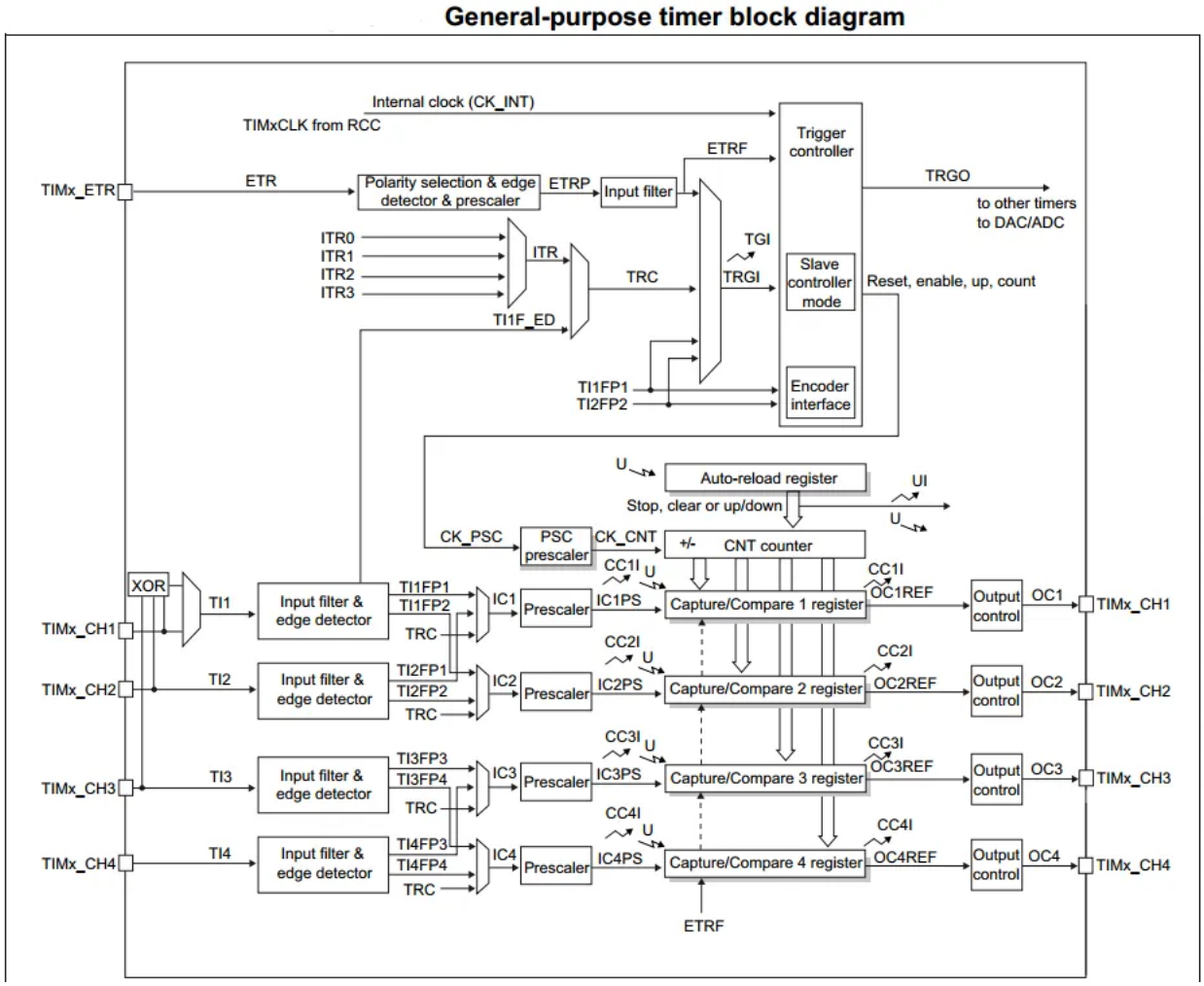


The Low-Power Timers (LPTIM) Main Features:

- 16-bit up-counter
- 3-bit Prescaler with 8 possible dividing factors (1,2,4,8,16,32,64,128)

- Selectable clock
  – Internal clock sources: LSE, LSI, HSI16 or APB clock
  – External clock source over LPTIM input (working with no LP oscillator running, used by Pulse Counter application)
- 16 bit ARR auto-reload register
- 16 bit compare register
- Continuous/One-shot mode
- Selectable software/hardware input trigger
- Programmable Digital Glitch filter
- Configurable output: Pulse, PWM
- Configurable I/O polarity
- Encoder mode
- Repetition counter

2.3 General-Purpose Timers Modules



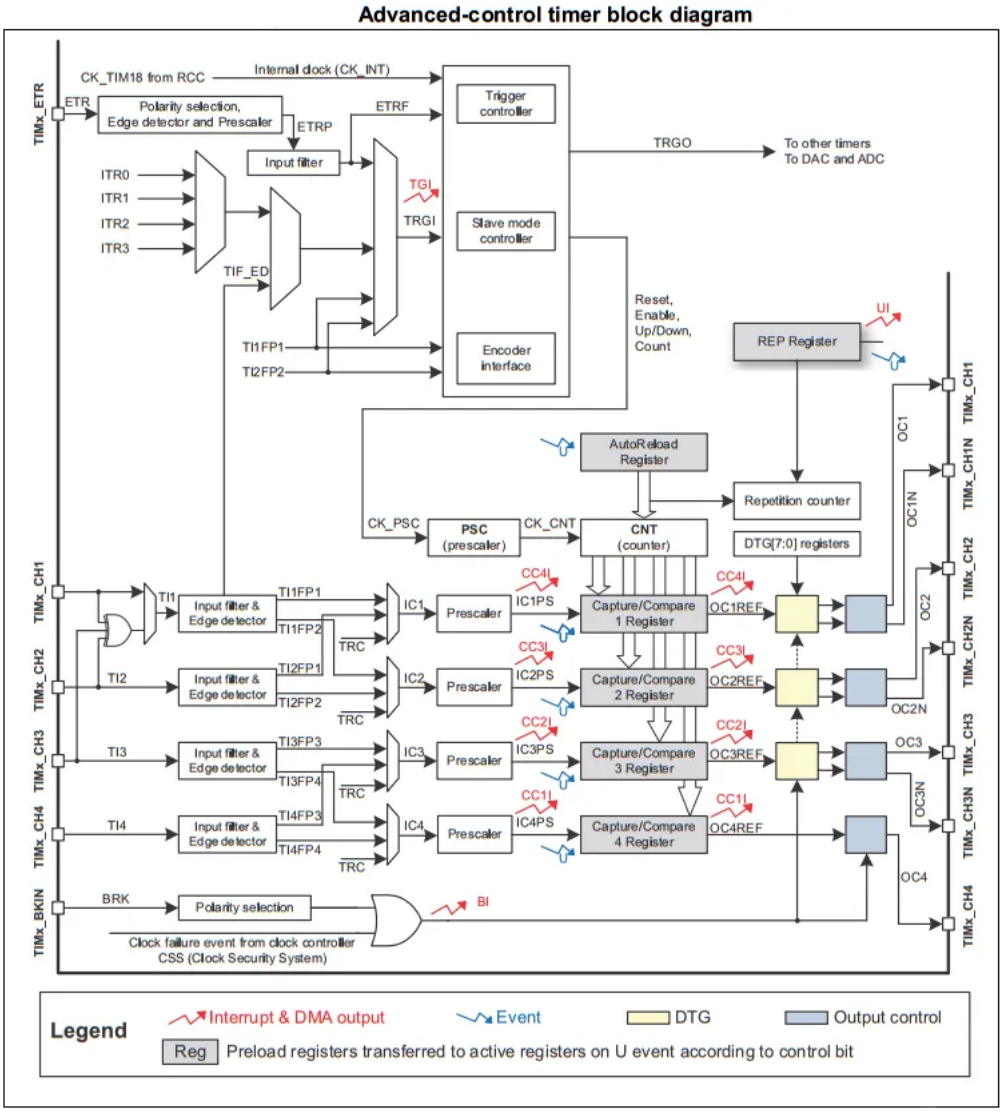
General-purpose TIMx timer features include:

- 16-bit up, down, up/down auto-reload counter.
- 16-bit programmable Prescaler used to divide (also “on the fly”) the counter clock frequency by any factor between 1 and 65536.
- Up to 4 independent channels for:
  – Input capture
  – Output compare

– PWM generation (Edge- and Center-aligned modes)
– One-pulse mode output

- Synchronization circuit to control the timer with external signals and to interconnect several timers.
- nterrupt/DMA generation on the following events:
  – Update: counter overflow/underflow, counter initialization (by software or internal/external trigger)
  – Trigger event (counter start, stop, initialization or count by internal/external trigger)
  – Input capture
  – Output compare
- Supports incremental (quadrature) encoder and hall-sensor circuitry for positioning purposes
- Trigger input for an external clock or cycle-by-cycle current management

2.4 Advanced-Control Timers Modules



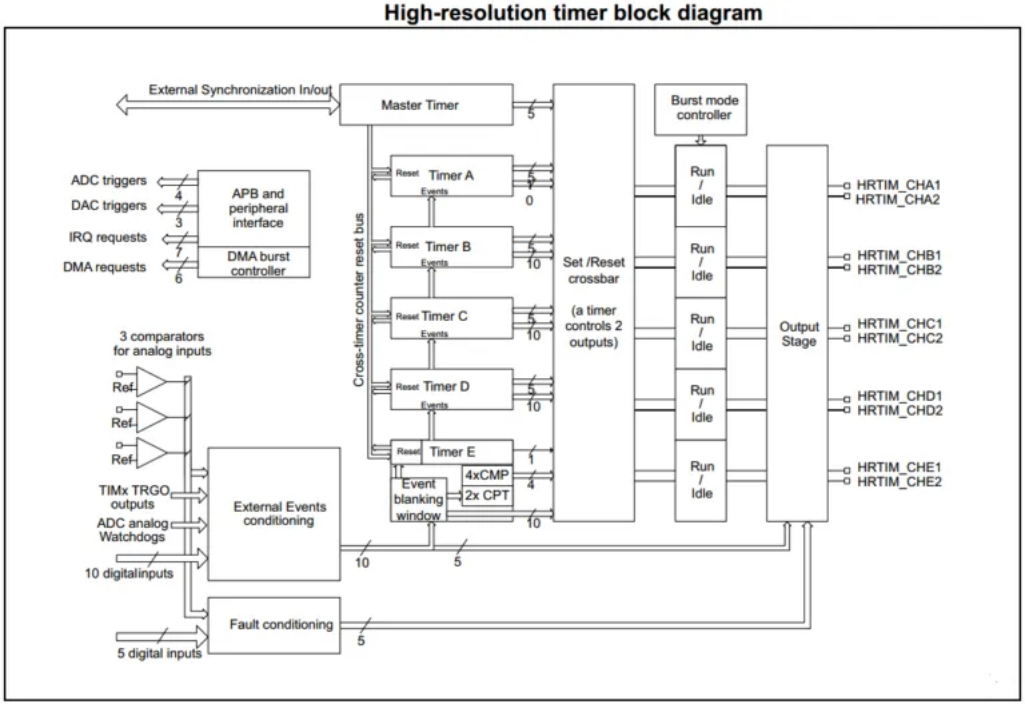
Advanced-Control Timers Features:

- Up to 4 independent channels for:
  – Input Capture
  – Output Compare
  – PWM generation (Edge and Center-aligned Mode)

– One-pulse mode output

- Repetition counter to update the timer registers only after a given number of cycles of the counter.
- Break input to put the timer's output signals in a reset state or in a known state.
- Interrupt/DMA generation on the following events:
  – Update: counter overflow/underflow, counter initialization (by software or internal/external trigger)
  – Trigger event (counter start, stop, initialization or count by internal/external trigger)
  – Input capture
  – Output compare
  – Break input
- Supports incremental (quadrature) encoder and hall-sensor circuitry for positioning purposes
- Trigger input for an external clock or cycle-by-cycle current management

2.5 High-Resolution Timers Modules



High-Resolution Timers (HRTIM) Features Include:

- High-resolution timing units
  – 217 ps resolution, compensated against voltage and temperature variations
  – High-resolution available on all outputs, possibility to adjust duty-cycle, frequency and pulse width in the triggered one-pulse mode
  – 6 16-bit timing units (each one with an independent counter and 4 compare units)
  – 10 outputs that can be controlled by any timing unit, up to 32 set/reset sources per channel
  – Modular architecture to address either multiple independent converters with 1 or 2 switches or few large multi-switch topologies
- Multiple links to built-in analog peripherals
  – 4 triggers to ADC converters

  - 3 triggers to DAC converters
  - 3 comparators for analog signal conditioning
- Multiple HRTIM instances can be synchronized with external synchronization inputs/outputs
- Versatile output stage
  - High-resolution Deadtime insertion (down to 868 pSec)
  - Programmable output polarity
  - Chopper mode
- Burst mode controller to handle light-load operation synchronously on multiple converters
- 7 interrupt vectors, each one with up to 14 sources
- 6 DMA requests with up to 14 sources, with a burst mode for multiple registers update

3. STM32 Timers Modes OF Operation

3.1 Timers Mode

In timer mode, the timer module gets clocked from an internal clock source with a known frequency.So the timer counts up or down on each rising or falling edge of the external input.

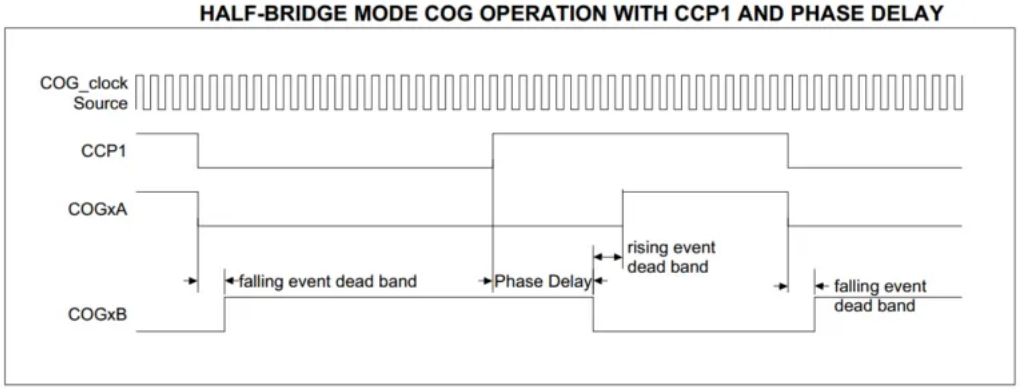
3.2 PWM Mode

In PWM mode, the timer module is clocked from an internal clock source and produces a digital waveform on the output channel pin called the PWM signal.
The timer in PWM mode will produce a PWM signal at the specified frequency the user chose. The duty cycle is also programmatically controlled by its register.

3.3 Advanced PWM Mode

The advanced PWM signal generation refers to the hardware ability to control more parameters and add some hardware circuitry to support extra features for the PWM signal generation. Which includes:

- The ability to produce a complementary PWM signal that is typically the same as the PWM on the main channel but logically inverted (high portion becomes low and vice versa).
- The ability to inject dead-time band in the PWM signal for motor driving applications to prevent shoot-through currents that result from PWM signals overlapping.
- The ability to perform auto-shutdown for the PWM signal, it's also called "auto brake" which an important feature for safety-critical applications.
- And the ability to phase-adjust the PWM signal, and much more! All of this is referred to as advanced-PWM control.

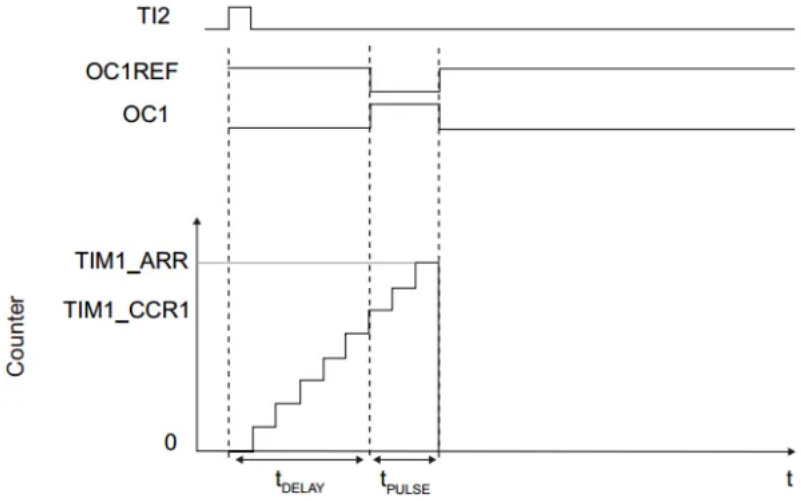


3.4 Output Compare Mode

In output compare mode, a timer module controls an output waveform or indicates when a period of time has elapsed. When a match is detected between the output compare register (OCR) and the counter, the output compare function assigns the corresponding output pin to a programmable value defined by the output compare mode defined by the programmer.

3.5 One-Pulse Mode

It allows the counter to be started in response to a stimulus and to generate a pulse with a programmable length after a programmable delay.



3.6 Input Capture Mode

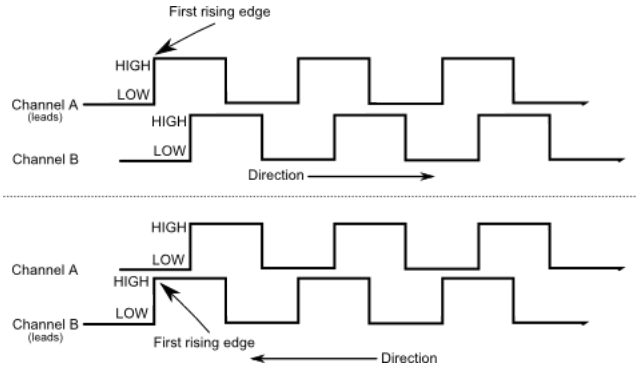
In Input capture mode, the Capture/Compare Registers (TIMx_CCRx) are used to latch the value of the counter after a transition detected by the corresponding ICx signal. When a capture occurs, the corresponding CCXIF flag (TIMx_SR register) is set and an interrupt or a DMA request can be sent if they are enabled.

The current value of the timer counts is captured when an external event occurs and an interrupt is fired. So, we can use this feature for a wide range of measurement applications.

3.7 Encoder Mode

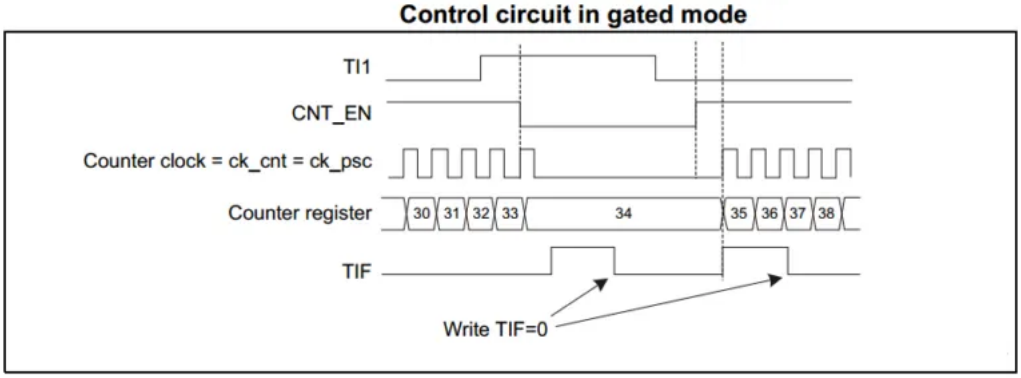
In the encoder interface mode, the timer module operates as a digital counter with two inputs. The counter is clocked by each valid transition on both input pins. The sequence of transitions of the two inputs is evaluated and generates count pulses as well as the direction signal. Depending on the sequence the counter counts up or down. So you don't have to detect these pulses individually and see which came first to detect rotation direction and this kind of work. Now, all of this is done by hardware thanks to the encoder mode hardware support.

The timer, when configured in Encoder Interface mode provides information on the sensor's current position. The user can obtain dynamic information (speed, acceleration, deceleration) by measuring the period between two encoder events using a second timer configured in capture mode. The output of the encoder which indicates the mechanical zero can be used for this purpose. Depending on the time between two events, the counter can also be read at regular times.



3.8 Timer Gate Mode

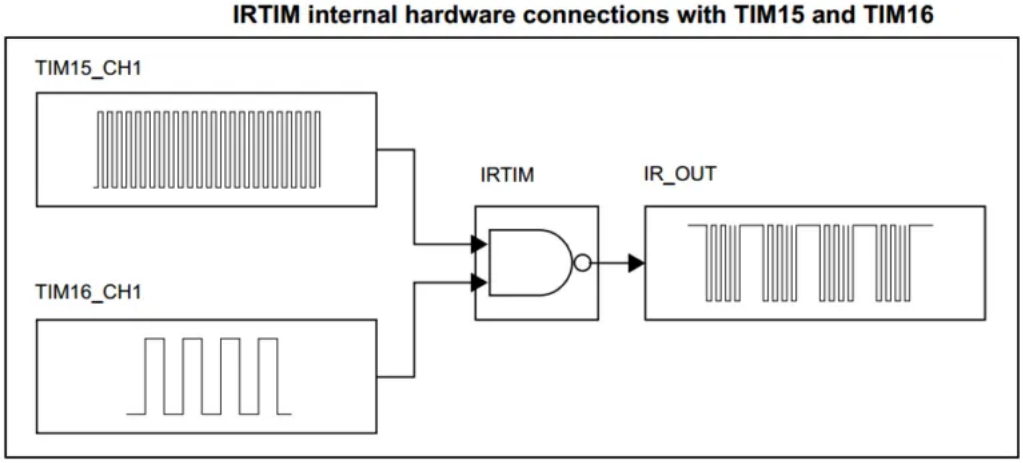
This mode can be used in a wide range of applications and signal measurements. It can help you measure extremely short pulses with a very high resolution. And also trigger the timer to count on external events from sensors or other MCUs.



3.9 Timer DMA Burst Mode

3.10 IRTIM Infrared Mode

An infrared interface (IRTIM) for remote control can be used with an infrared LED to perform remote control functions. It uses internal connections with TIM15 and TIM16 as shown in the diagram down below. To generate the infrared remote control signals, the IR interface must be enabled and TIM15 channel 1 (TIM15_OC1) and TIM16 channel 1 (TIM16_OC1) must be properly configured to generate correct waveforms. The infrared receiver can be implemented easily through a basic input capture mode.



TIM15 is used to generate the high-frequency carrier signal, while TIM16 generates the modulation envelope. The infrared function is output on the IR_OUT pin. The activation of this function is done through the GPIOx_AFRx register by enabling the related alternate function bit.

4. **STM32 Timers – Timer Mode**

4.1 **Block Diagram**

4.2 **Functional Description**

4.2.1 Time-Base Unit

The time-base unit includes:

- Counter Register (TIMx_CNT):The counter can count up, down, or both up and down.
- Prescaler Register (TIMx_PSC):The counter clock can be divided by a Prescaler.
- Auto-Reload Register (TIMx_ARR)

4.2.2 Timer Prescaler

The Prescaler can divide the counter clock frequency by any factor between 1 and 65536.

4.3 **Applications Examples**

The timer module in timer mode can be used to generate a time delay interval between specific events. Or to repeatedly perform a specific task each specific time interval. And also to measure the time between predefined events.

4.4 **Different Possible Configurations**

The time interval, which is a combination of the following settings:

- The timer clock source frequency
- The Prescaler value
- The time counting mode (up or down)
- Auto-Reload register value
- Autor preload enable/disable
- Overflow interrupt enable/disable

XIV STM32 Timer Interrupt HAL Example

1. **STM32 Timer Mode LAB Preface**

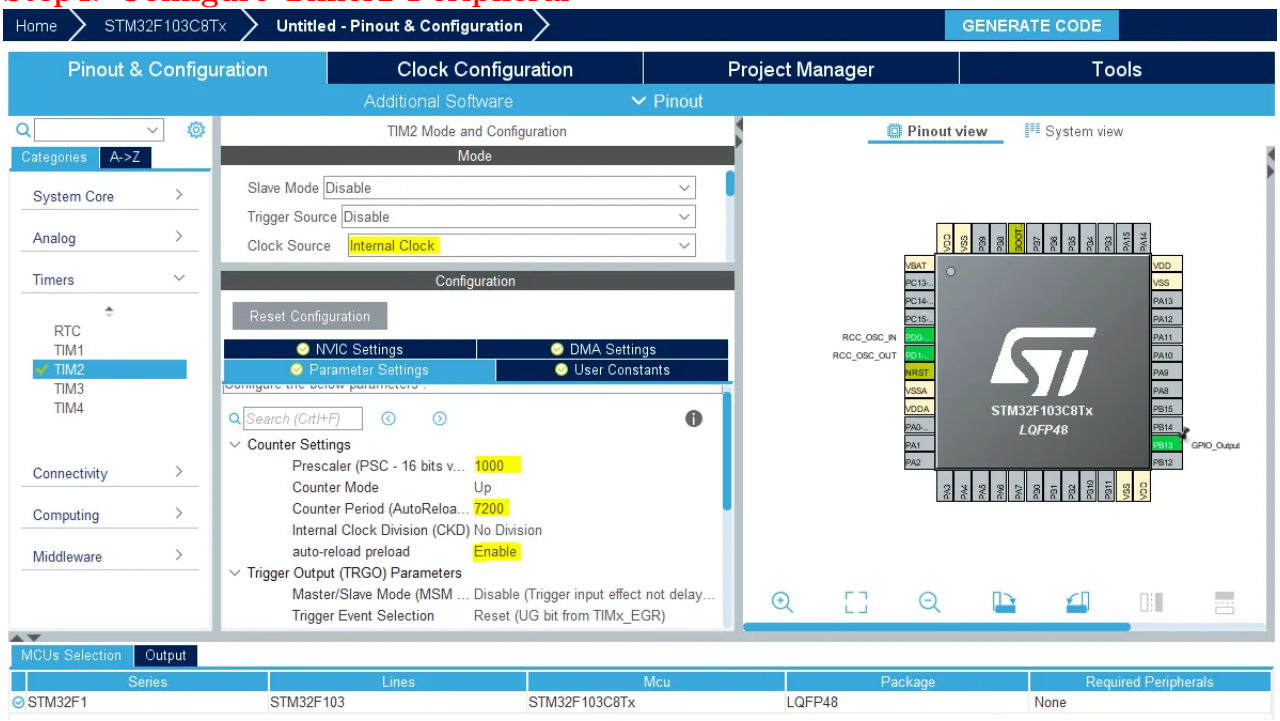
$$T_{OUT} = TicksCount \times TicksTime$$

$$T_{OUT} = \frac{PSC \times Preload}{F_{CLK}}$$

2. **STM32 Timer – Timer Mode LAB Config**

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Click On The Pin You Want To Configure As An Output & Select Output Option**
  Let it be B13 pin for example! (The LED Pin)

- **Step4: Configure Timer2 Peripheral**



- **Step5: Enable The Timer Interrupt Signal In NVIC Tab**
- **Step6: Set The RCC External Clock Source**
- **Step7: Go To The Clock Configuration**
- **Step8: Set The System Clock To Be 72MHz Or Whatever You Want**
- **Step9: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

3. **The Application Code In CubeIDE**
   Full LAB Code (main.c)

```c
#include "main.h"

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

int main(void)
{

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_TIM2_Init();
  HAL_TIM_Base_Start_IT(&htim2);

  while (1)
  {

  }

}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim)
{
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_13);
}
```

4. **Prototyping & Testing**

- **Step0: Refer To The Blue Pill Board Schematic & Pinout**
- **Step1: Connect She ST-Link To The USB Port & SWD Pins On Board**

- **Step2: Click The Debug Button To Compile The Code & Flash It To The Board & Start A Debugging Session**
- **Step3: You Can Stop The Debugging Session Or Keep It Going. But You Need To Restart The MCU Once To Run The New Application At The Booting Process.**

5. **The result for this lab:** https://cdn-0.deepbluembedded.com/wp-content/uploads/2020/06/STM32-Timer-Interrupt-HAL-Example.mp4?_=1

XV STM32 Counter Mode Example – Frequency Counter

1. **STM32 Counter Mode LAB Preface**

   1.1 **Up-counting Mode**
   In up-counting mode, the counter counts from 0 to the auto-reload value (the content of the TIMx_ARR register), then restart from 0 and generates a counter overflow event. An Update event can be generated at each counter overflow or by setting the UG bit in the TIMx_EGR register (by software or by using the slave mode controller).

   1.2 **Down-counting Mode**
   In down counting mode, the counter counts from the auto-reload value (the content of the TIMx_ARR register) down to 0, then restarts from the auto-reload value and generates a counter underflow event. An Update event can be generated at each counter underflow or by setting the UG bit in the TIMx_EGR register (by software or by using the slave mode controller).

   1.3 **Center-Aligned Mode (Up/Down)**
   In center-aligned mode, the counter counts from 0 to the auto-reload value (the content of the TIMx_ARR register) – 1, generates a counter overflow event, then counts from the auto-reload value down to 1 and generates a counter underflow event.

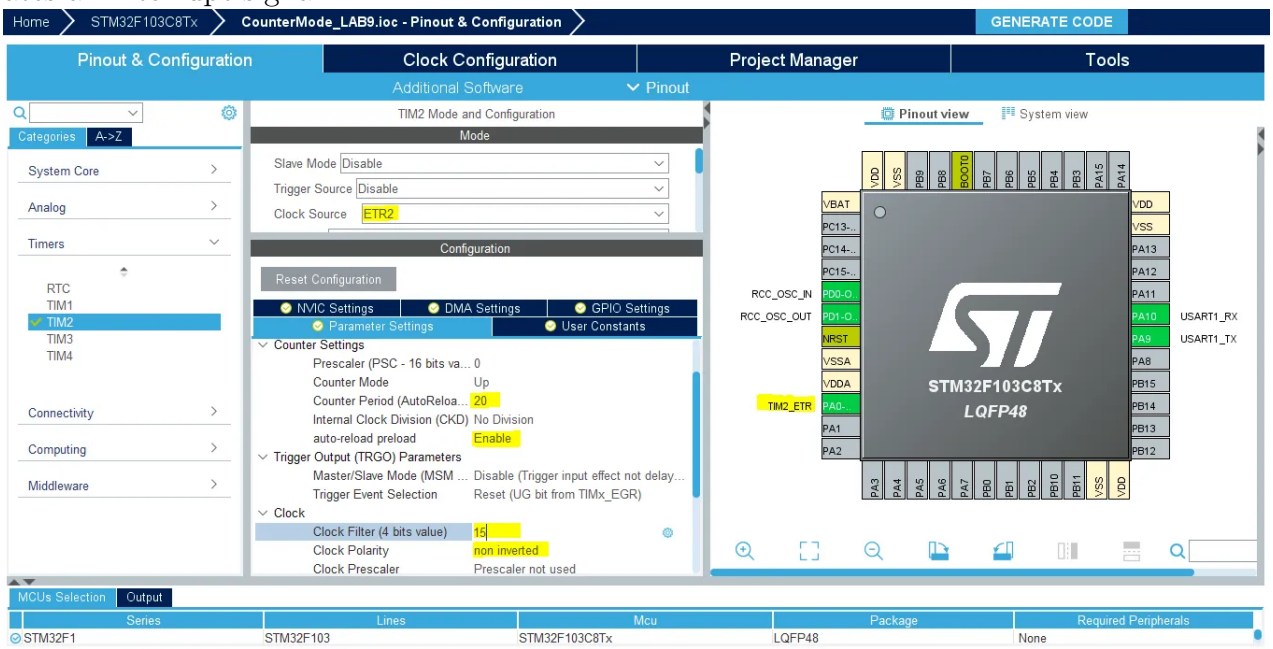
2. **STM32 Counter LAB9 Objectives**

   - Configure the general-purpose timer (TIM2) to operate in counter mode
   - Set The Preload value to 20, so the counter overflows after 20 ticks and generates an interrupt
   - Set The Timer input GPIO pin
   - Configure the USART1 module to print the counter ticks value for monitoring it
   - Print special message within the ISR routine (counter overflow event) to check that the interrupt firing and handling functionality is being done properly

3. **STM32 Timer – Counter Mode LAB Config.**

   - **Step1: Open CubeMX & Create New Project**
   - **Step2: Choose The Target MCU & Double-Click Its Name**
   - **Step3: Configure Timer2 Peripheral To Operate In Counter Mode**
     Note:that now the clock source is an external pin (timer2 input pin ETR2) which is highlighted as A0 as you can see. We can also configure a digital filter for this input channel to reject noise due to switch bouncing. The filter value can range from

0 upto 15. And last but not least, the edge selection for the counter. I want the counter to count on each rising edge of the input pin.
_I'll also enable the auto-reload function and set the upper count limit to 20 "the Preload value". Now, whenever the counter counts 20 ticks, it overflows and generates an interrupt signal!



- **Step4: Enable The Timer Interrupt Signal In NVIC Tab**
- **Step5: Configure USART1 Module To Operate In Async Mode With 9600bps**
- **Step6: Set The RCC External Clock Source**
- **Step7: Go To The Clock Configuration**
- **Step8: Set The System Clock To Be 72MHz Or Whatever You Want**
- **Step9: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

4. **The Application Code In CubeIDE**
   Full LAB Code (main.c)

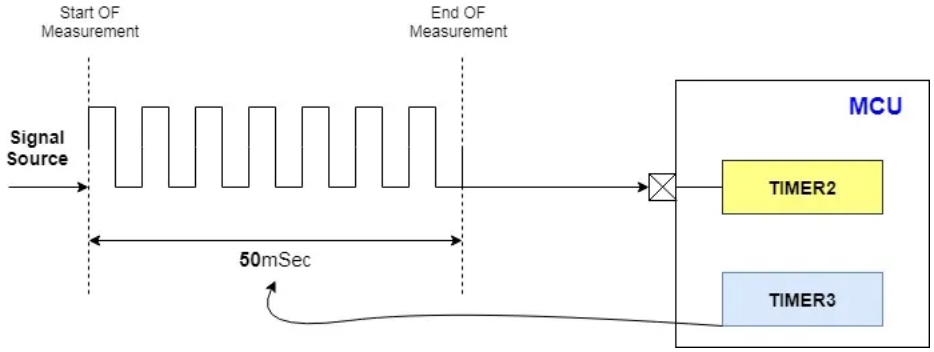
```c
#include "main.h"

TIM_HandleTypeDef htim2;

UART_HandleTypeDef huart1;

uint8_t END_MSG[35] = "Overflow Reached! Counter Reset!\n\r";

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    uint8_t MSG[20] = {'\0'};
    uint16_t CounterTicks = 0;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
    MX_USART1_UART_Init();
    HAL_TIM_Base_Start_IT(&htim2);

    while (1)
    {
        // Read The Counter Ticks Register
        CounterTicks = TIM2->CNT;
        // Print The Ticks Count Via UART1
        sprintf(MSG, "Ticks = %d\n\r", CounterTicks);
        HAL_UART_Transmit(&huart1, MSG, sizeof(MSG), 100);
        HAL_Delay(100);
    }
}

// Counter Overflow ISR Handler
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim)
{
    HAL_UART_Transmit(&huart1, END_MSG, sizeof(END_MSG), 100);
}
```

NOTE:it's worth noting that you have to start the timer in interrupt mode so it gets clocked and fires an interrupt signal on overflow and so. Otherwise, the timer will not work at all.

5. **STM32 Frequency Counter – LAB10**

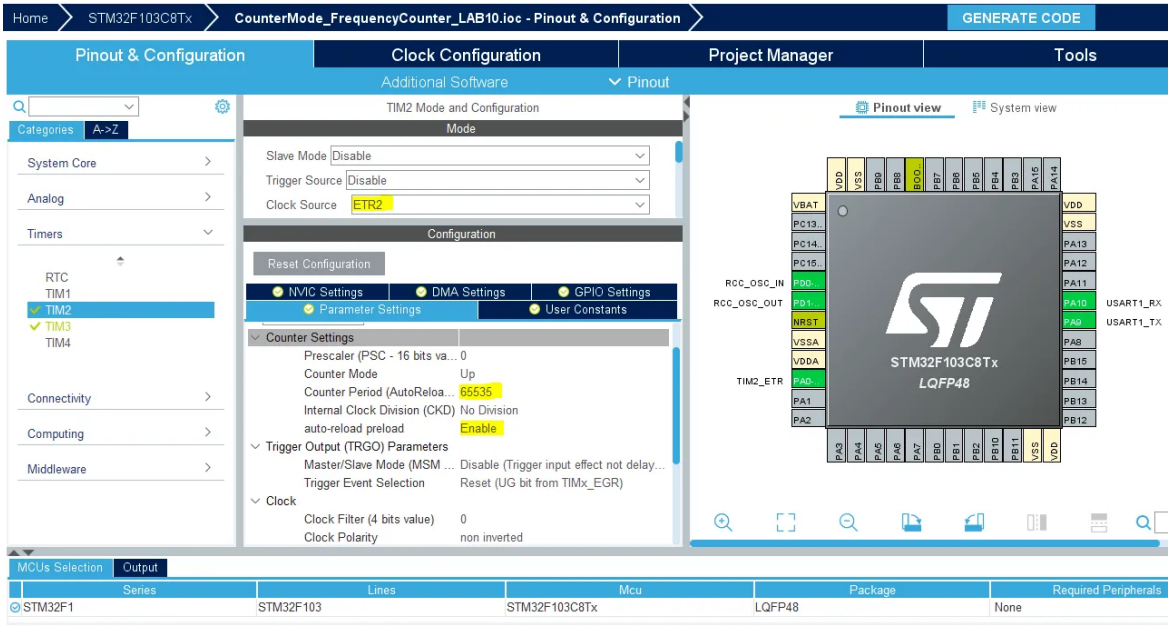
   - Set up TIM2 as a counter and hook its input pin to the signal source
   - Set up TIM3 as the measurement timer, and it should overflow every 50mSec (or any time interval you choose)
   - Count how many clock cycles in the ISR of TIM3 by reading TIM2 counter ticks and by multiplying this value by 20, you'll get the actual signal frequency.
   - Configure USART1 in Async mode with baud rate = 9600bps.
   - Print the calculated frequency to the serial port terminal.

6. **STM32 Frequency Counter Project LAB**



NOTE:this is not the best way to measure frequency. However, it should be a proof of concept and give you an idea for how to cooperatively set up multiple peripherals to do a specific task. The downsides for this technique will be mentioned at the end of the LAB after seeing the final results.

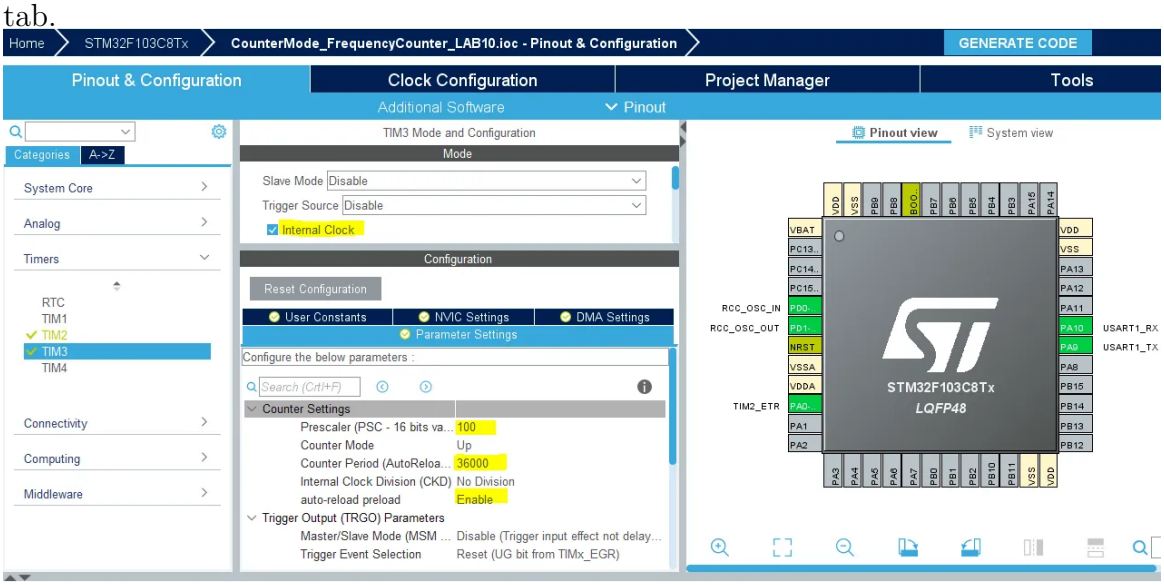
- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Configure Timer2 Peripheral To Operate In Counter Mode**



- **Step4: Configure Timer3 Peripheral To Operate In Timer Mode (Tout = 0.05 sec)**

$$T_{OUT} = \frac{PSC \times Preload}{F_{CLK}}$$

FCLK is 72MHz, I'll use Prescaler of 100. So, by solving for the preload register value. It turns out to be 36000. Therefore, we'll write this value in the configurations tab.



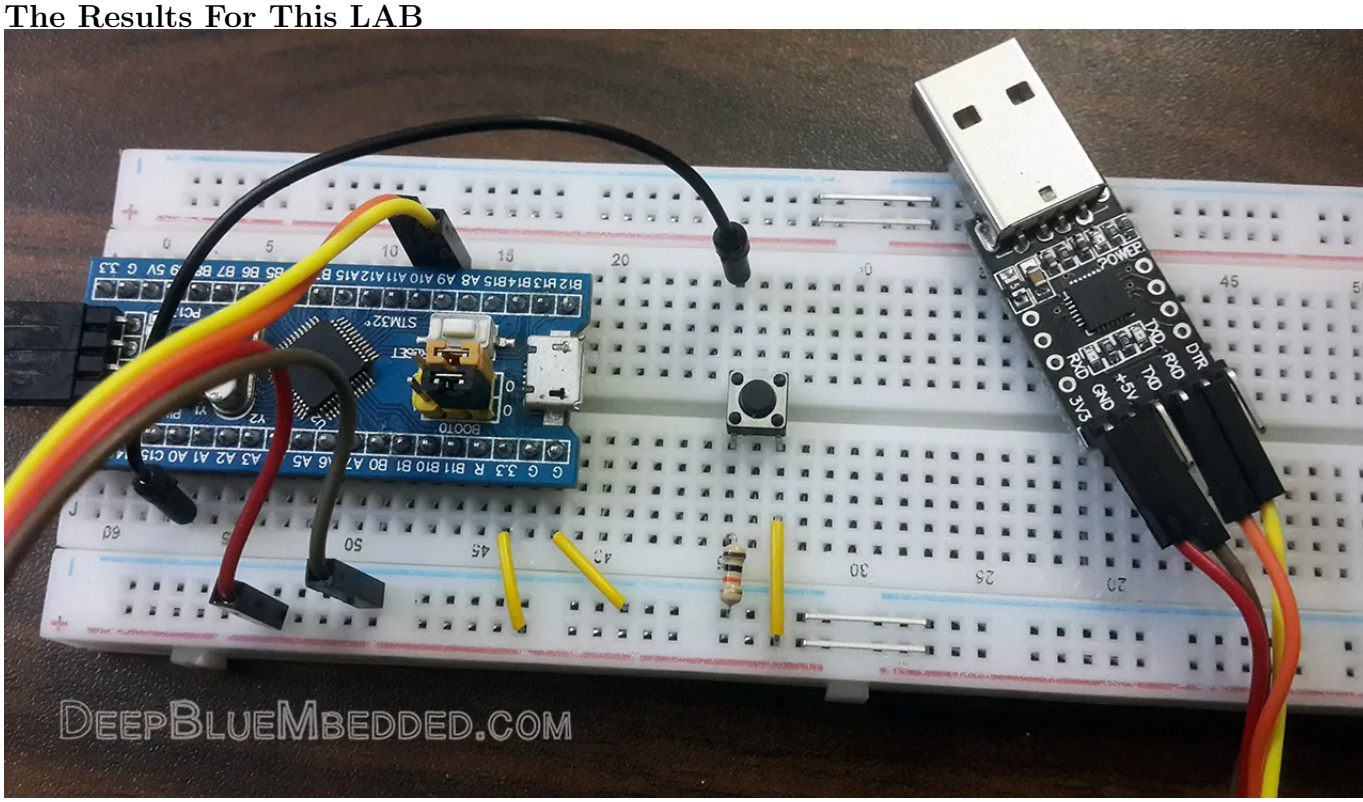
Enable Timer3 interrupts from the NVIC control tab.

- **Step5: Configure USART1 Module To Operate In Async Mode With 9600bps**
- **Step6: Set The RCC External Clock Source**
- **Step7: Set The System Clock To Be 72MHz**
- **Step8: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

Here is The Application Code For This LAB

```c
#include "main.h"

uint32_t gu32_CounterTicks = 0x00;
uint32_t gu32_Freq = 0x00;
uint8_t gu8_MSG[40] = {'\0'};

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
    MX_TIM3_Init();
    MX_USART1_UART_Init();
    HAL_TIM_Base_Start(&htim2);
    HAL_TIM_Base_Start_IT(&htim3);
    while (1)
    {

    }
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim)
{
    if(htim->Instance == TIM3)
    {
        gu32_CounterTicks = TIM2->CNT;
        gu32_Freq = gu32_CounterTicks * 20;
        sprintf(gu8_MSG, "Frequency = %d Hz\n\r", gu32_Freq);
        HAL_UART_Transmit(&huart1, gu8_MSG, sizeof(gu8_MSG), 100);
        TIM3->CNT = 0;
        TIM2->CNT = 0;
    }
}
```

The Results For This LAB



This is Video: https://youtu.be/yEmjfY9fhTE

XVI STM32 Input Capture & Frequency Measurement

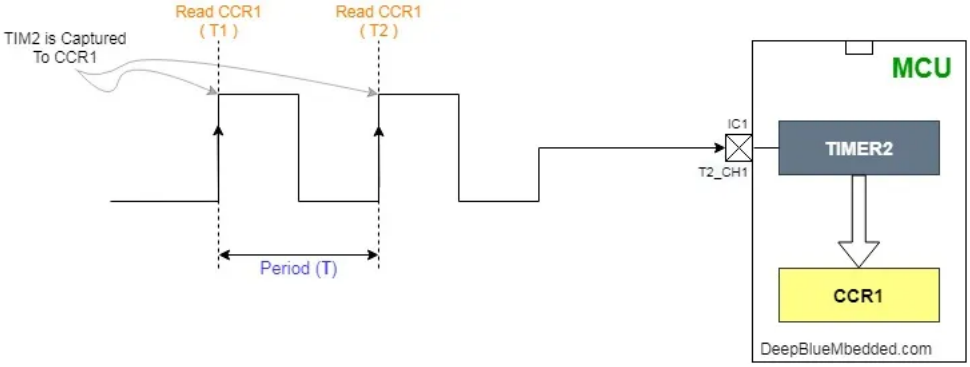
1. **STM32 Input Capture Frequency Counter**

   - Set up TIM2 to use the internal clock and configure CH1 to be input capture on every rising edge
   - Read the CCR1 register and save it to variable T1 on the first edge, on second edge read CCR1 and save it to T2. And now, the input signal's period is T2 – T1, and therefore, the frequency is easily calculated from 1/period.
   - Configure USART1 in Async mode with baud rate = 9600bps
   - Print the calculated frequency to the serial port terminal

2. **STM32 Input Capture Mode Frequency Counter**
   The system will go through a couple of states I've chosen to name them (IDLE, DONE).

   - In the IDLE state, the system is ready to capture the first timestamp using TIM2 that is running in the background until the external signal does a rising edge transition on the IC1 input channel pin.
   - In DONE state, the system will be running normally until a rising edge happens again on the IC1 pin. Therefore, in the ISR, the captured value is saved to the T2 variable.

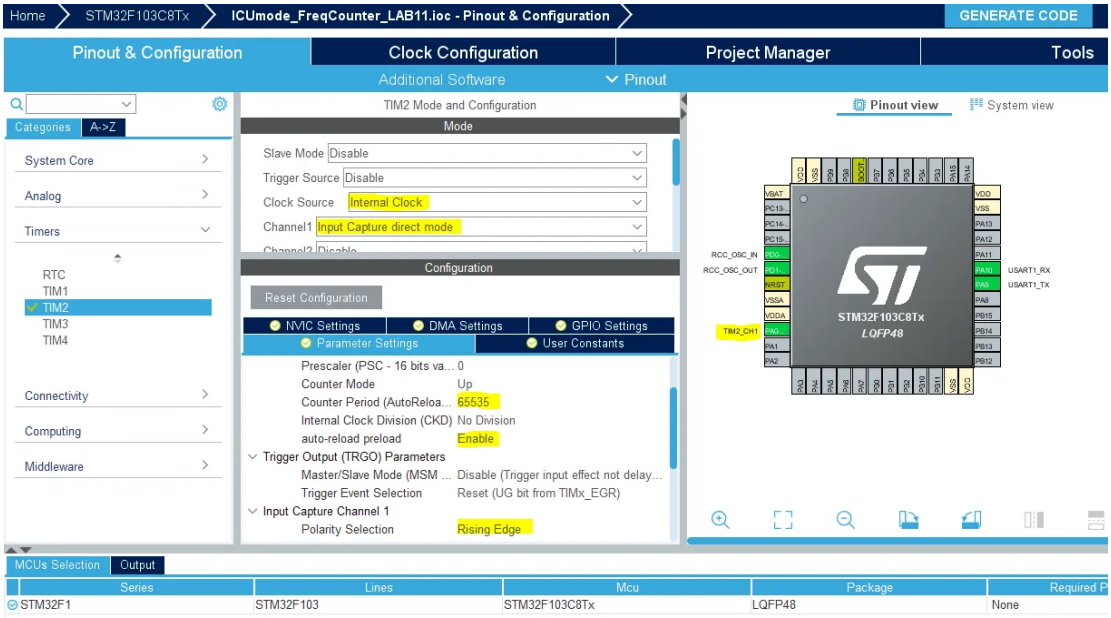
   And the measured period is (T2-T1), so the frequency is calculated as (1/(T2-T1)). And then, it's printed to the serial port using UART1. Finally, the system state is flipped again to IDLE, and the whole process is repeated.
   On the first rising edge, the value of timer2 is captured to the CCR1 register and an interrupt is fired. In the ISR, we'll save the CCR1 value to a variable called T1. And the system state is changed from IDLE to DONE,



   - **Step1: Open CubeMX & Create New Project**
   - **Step2: Choose The Target MCU & Double-Click Its Name**
   - **Step3: Configure Timer2 Peripheral To Operate In Input Capture Mode**

```c
#include "main.h"

#define IDLE   0
#define DONE   1
#define F_CLK  72000000UL

volatile uint8_t gu8_State = IDLE;
volatile uint8_t gu8_MSG[35] = {'\0'};
volatile uint32_t gu32_T1 = 0;
volatile uint32_t gu32_T2 = 0;
volatile uint32_t gu32_Ticks = 0;
volatile uint16_t gu16_TIM2_OVC = 0;
volatile uint32_t gu32_Freq = 0;

TIM_HandleTypeDef htim2;
UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
    MX_USART1_UART_Init();
    HAL_TIM_Base_Start_IT(&htim2);
    HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1);

    while (1)
    {
```

```c
}

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef* htim)
{
    if(gu8_State == IDLE)
    {
        gu32_T1 = TIM2->CCR1;
        gu16_TIM2_OVC = 0;
        gu8_State = DONE;
    }
    else if(gu8_State == DONE)
    {
        gu32_T2 = TIM2->CCR1;
        gu32_Ticks = (gu32_T2 + (gu16_TIM2_OVC * 65536)) - gu32_T1;
        gu32_Freq = (uint32_t)(F_CLK/gu32_Ticks);
        if(gu32_Freq != 0)
        {
          sprintf(gu8_MSG, "Frequency = %lu Hz\n\r", gu32_Freq);
          HAL_UART_Transmit(&huart1, gu8_MSG, sizeof(gu8_MSG), 100);
        }
        gu8_State = IDLE;
    }
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim)
{
    gu16_TIM2_OVC++;
}
```

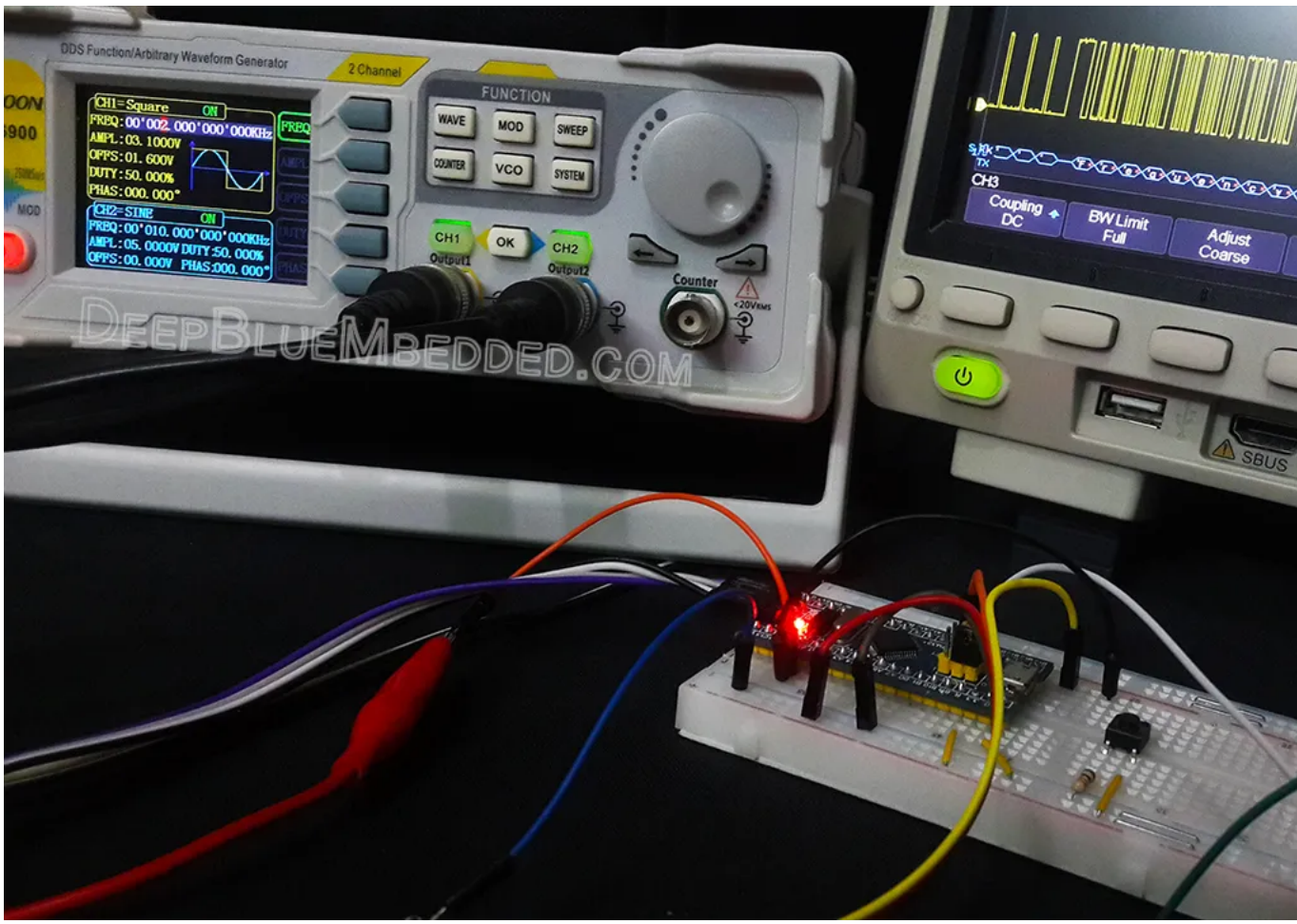


Enable Timer2 interrupts from the NVIC control tab.

- **Step4: Configure USART1 Module To Operate In Async Mode With 9600bps**
- **Step5: Set The RCC External Clock Source**
- **Step6: Set The System Clock To Be 72MHz**
- **Step7: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

Here is The Application Code For This LAB

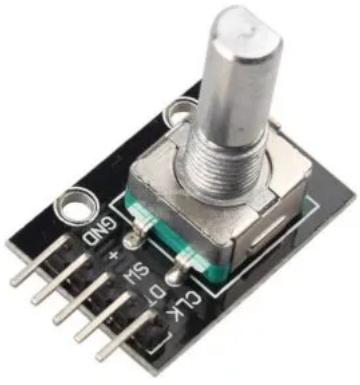
The Result:



The Video:https://youtu.be/h8jpNotihUY

XVII STM32 Timer Encoder Mode – STM32 Rotary Encoder Interfacing

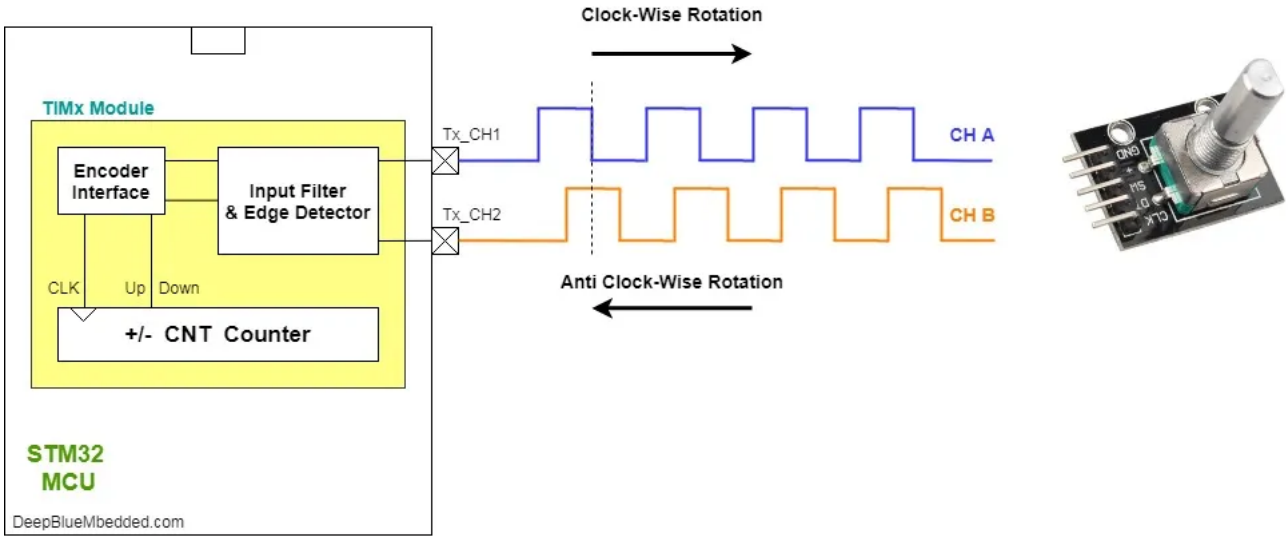
1. **Rotary Encoder Interfacing**



An encoder can be used as a sensor for motor speed and position sensing applications, for distances and length measurements, and more. And also an encoder can be used as an input device that the user can turn around with no limitations unlike the potentiometers. And you also (as a programmer) can detect how fast the user is turning the know, so your system can respond correspondingly.

2. **STM32 Encoder Mode Preface STM32 Timers In Encoder Mode**
The two inputs TI1 and TI2 are used to interface to an incremental encoder. The counter

is clocked by each valid transition on TI1FP1 or TI2FP2 (TI1 and TI2 after input filter and polarity selection). The sequence of transitions of the two inputs is evaluated and generates count pulses as well as the direction signal. Depending on the sequence the counter counts up or down, the DIR bit in the TIMx_CR1 register is modified by hardware accordingly. The DIR bit is calculated at each transition on any input (TI1 or TI2), whatever the counter is counting on TI1 only, TI2 only, or both TI1 and TI2.



Encoder interface mode acts simply as an external clock with direction selection. This means that the counter just counts continuously between 0 and the auto-reload value in the TIMx_ARR register.

3. **STM32 Encoder Example LABs STM32 Timer Encoder Mode Basic LAB**

   - Set up timer2 to operate in encoder mode with 2 input channels (combined)
   - Set up a GPIO input pin to be connected to encoder's SW switch button pin
   - Set up UART1 module to operate in async mode @ 9600bps
   - Read the timer2 counter register value and print the number via serial port as well as the button state

   STM32 Timer Encoder Mode – Rotary Encoder LED Dimmer

   - Set up timer2 to operate in encoder mode with 2 input channels (combined)
   - Set up timer3 to operate in PWM mode with output channel1 (LED control signal)
   - Read the timer2 counter value and map it to PWM duty cycle to control LED brightness

4. **STM32 Rotary Encoder Example LAB13**

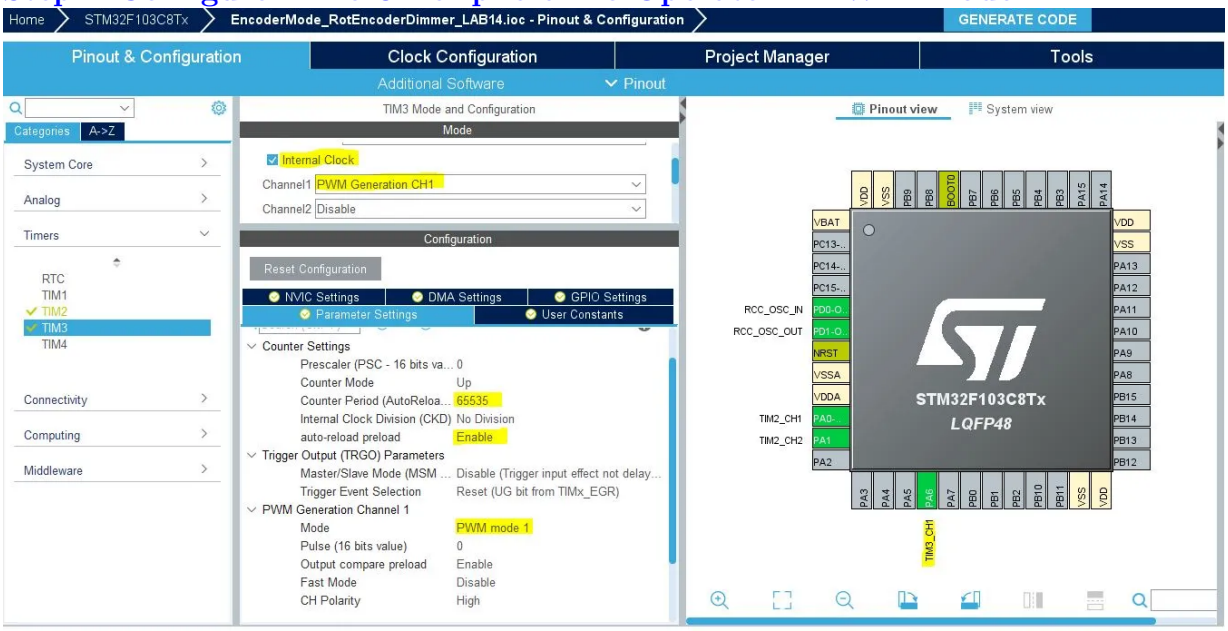
   - **Step1: Open CubeMX & Create New Project**
   - **Step2: Choose The Target MCU & Double-Click Its Name**
   - **Step3: Configure Timer2 Peripheral To Operate In Encoder Mode With Combined Channels**
   - **Step4: Set The RCC External Clock Source**
   - **Step5: Set The System Clock To Be 72MHz**
   - **Step6: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

5. **Here is The Application Code For This LAB**

```c
#include "main.h"

TIM_HandleTypeDef htim2;
UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    uint8_t MSG[50] = {'\0'};

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
    MX_USART1_UART_Init();
    HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_ALL);

    while (1)
    {
        if(HAL_GPIO_ReadPin (GPIOA, GPIO_PIN_2))
        {
            sprintf(MSG, "Encoder Switch Released, Encoder Ticks = %d\n\r", ((TIM2->CNT)>>2));
            HAL_UART_Transmit(&huart1, MSG, sizeof(MSG), 100);
        }
        else
        {
            sprintf(MSG, "Encoder Switch Pressed,  Encoder Ticks = %d\n\r", ((TIM2->CNT)>>2));
            HAL_UART_Transmit(&huart1, MSG, sizeof(MSG), 100);
        }
        HAL_Delay(100);
    }
}
```

6. **STM32 Rotary Encoder Example Dimmer LAB14**

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Configure Timer2 Peripheral To Operate In Encoder Mode With Combined Channels**
- **Step4: Configure Timer3 Peripheral To Operate In PWM Mode**



- **Step5: Set The RCC External Clock Source**
- **Step6: Set The System Clock To Be 72MHz**
- **Step7: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

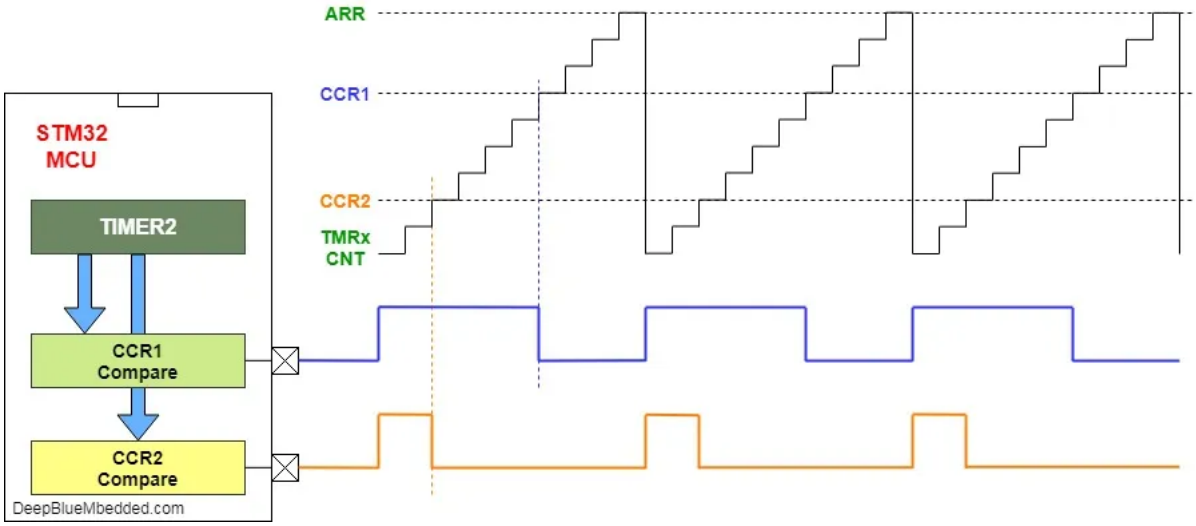
7. **Here is The Application Code For This LAB**

```c
#include "main.h"

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);

int main(void)
{
    uint16_t LED_DutyCycle = 0;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
    MX_TIM3_Init();
    HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_ALL);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);

    while (1)
    {
        if(TIM2->CNT < 1023)
        {
            LED_DutyCycle = ((TIM2->CNT)<<6);
        }
        TIM3->CCR1 = LED_DutyCycle;
        HAL_Delay(10);
    }
}
```

The Video:https://youtu.be/5FTNUQxjv5Y

XVIII Timer PWM Mode Tutorial

1. **STM32 PWM Mode Preface**



1.1 STM32 Timers – PWM Output Channels

Each Capture/Compare channel is built around a capture/compare register (including a shadow register), an input stage for capture (with a digital filter, multiplexing, and Prescaler) and an output stage (with comparator and output control). The output stage generates an intermediate waveform which is then used for reference: OCxRef (active high). The polarity acts at the end of the chain.

1.2 STM32 Timers In PWM Mode

In PWM mode (1 or 2), TIMx_CNT and TIMx_CCRx are always compared to determine whether TIMx_CCRx <= TIMx_CNT (depending on the direction of the counter).

NOTE:PWM signals have a lot of properties that we need to control in various applications. First of which is the frequency of the signal. And secondly, and

probably the most important one, is the duty cycle. Third, is the PWM resolution. And much more to be discussed in later tutorials, we'll get into those 3 properties in the next sections down below.

1.3 STM32 PWM Frequency

In various applications, you'll be in need to generate a PWM signal with a specific frequency. In servo motor control, LED drivers, motor drivers, and many more situations where you'll be in need to set your desired frequency for the output PWM signal.

$$F_{PWM} = \frac{F_{CLK}}{(ARR + 1) \times (PSC + 1)}$$

1.4 STM32 PWM Duty Cycle

The duty cycle percentage is controlled by changing the value of the CCRx register. And the duty cycle equals (CCRx/ARR) [%].

$$DutyCycle_{PWM}[\%] = \frac{CCRx}{ARRx}[\%]$$

1.5 STM32 PWM Resolution

$$Resolution_{PWM}[Bits] = \frac{log(\frac{F_{CLK}}{F_{PWM}})}{log(2)}[Bit]$$

1.6 STM32 PWM Different Modes

The PWM signal generation can be done in different modes, I'll be discussing two of them in this section. The edge-aligned and the center-aligned modes.

1.6.1 Edge-Aligned Mode

- Up-Counting Configuration
- Down-Counting Configuration

1.6.2 Center-Aligned Mode

The compare flag is set when the counter counts up when it counts down or both when it counts up and down depending on the CMS bits configuration. The direction bit (DIR) in the TIMx_CR1 register is updated by hardware and must not be changed by software.

2. STM32 PWM Example LED Dimmer

- Set up timer 2 to operate in PWM mode with the internal clock. And enable CH1 to be the PWM output channel.
- Set the ARR value to the maximum 65535 for example, so the frequency should be 1098Hz
- Control the duty cycle by writing to the CCR1 register
- Make Duty Cycle sweep from 0

In this LAB, our goal is to build a system that sweeps the duty cycle of the PWM channel1 from 0 up to 100% back and forth. So that the LED brightness follows the same pattern. The auto-reload register will be set to a maximum value which is 65535, for no particular reason. But you should know that the output FPWM frequency is expected to be 1098.6Hz from the equation we've seen earlier. And the PWM resolution is estimated to be 16-Bit which is the maximum possible value for this module.

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Configure Timer2 Peripheral To Operate In PWM Mode With CH1 Output**
- **Step4: Set The RCC External Clock Source**
- **Step5: Set The System Clock To Be 72MHz**
- **Step7: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

3. Here is The Application Code For This LAB

```c
#include "main.h"

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

int main(void)
{
    int32_t CH1_DC = 0;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
    while (1)
    {
        while(CH1_DC < 65535)
        {
            TIM2->CCR1 = CH1_DC;
            CH1_DC += 70;
            HAL_Delay(1);
        }
        while(CH1_DC > 0)
        {
            TIM2->CCR1 = CH1_DC;
            CH1_DC -= 70;
            HAL_Delay(1);
        }
    }
}
```

XIX STM32 Delay Microsecond Millisecond Utility — DWT Delay & Timer Delay

1. **STM32 Delay Functions**
   In earlier tutorials, we've been using the HAL_Delay utility function to get milliseconds time delay. And as we've discussed it's built on the SysTick timer that ticks at a rate of 1000Hz and can only give you multiples of 1ms time delay. Using time delays despite being a bad practice to though out everywhere in the code, it can be mandatory in many cases and can be justified.

2. **STM32 DWT Delay**

   - SWJ-DP: Serial wire / JTAG debug port
   - AHP-AP: AHB access port
   - ITM: Instrumentation trace macrocell
   - FPB: Flash patch breakpoint
   - **DWT**: Data watchpoint trigger
   - TPUI: Trace port unit interface (available on larger packages, where the corresponding pins are mapped)

- ETM: Embedded Trace Macrocell (available on larger packages, where the corresponding pins are mapped)
  We are interested in the Data Watchpoint Trigger which provides some means to give some profiling information. For this, some counters are accessible to give the number of:
- Clock cycle
- Folded instructions
- Load store unit (LSU) operations
- Sleep cycles
- CPI (clock per instructions)
- Interrupt overhead
  we'll read the core speed from the RCC while initialization and use it for further work.

DWT_InitializationFunction

Enable = To turn On . Disable = To Turn Off

```c
uint32_t DWT_Delay_Init(void)
{
    /* Disable TRC */
    CoreDebug->DEMCR &= ~CoreDebug_DEMCR_TRCENA_Msk; // ~0x01000000;
    /* Enable TRC */
    CoreDebug->DEMCR |=  CoreDebug_DEMCR_TRCENA_Msk; // 0x01000000;

    /* Disable clock cycle counter */
    DWT->CTRL &= ~DWT_CTRL_CYCCNTENA_Msk; //~0x00000001;
    /* Enable  clock cycle counter */
    DWT->CTRL |=  DWT_CTRL_CYCCNTENA_Msk; //0x00000001;

    /* Reset the clock cycle counter value */
    DWT->CYCCNT = 0;

    /* 3 NO OPERATION instructions */
    __ASM volatile ("NOP");
    __ASM volatile ("NOP");
    __ASM volatile ("NOP");

    /* Check if clock cycle counter has started */
    if(DWT->CYCCNT)
    {
       return 0; /*clock cycle counter started*/
    }
    else
    {
      return 1; /*clock cycle counter not started*/
    }
}
```

DWT_Delay_usFunction

```c
// This Function Provides Delay In Microseconds Using DWT

__STATIC_INLINE void DWT_Delay_us(volatile uint32_t au32_microseconds)
{
  uint32_t au32_initial_ticks = DWT->CYCCNT;
  uint32_t au32_ticks = (HAL_RCC_GetHCLKFreq() / 1000000);
  au32_microseconds *= au32_ticks;
  while ((DWT->CYCCNT - au32_initial_ticks) < au32_microseconds-au32_ticks);
}
```

DWT_Delay_msFunction

```c
// This Function Provides Delay In Milliseconds Using DWT

__STATIC_INLINE void DWT_Delay_ms(volatile uint32_t au32_milliseconds)
{
  uint32_t au32_initial_ticks = DWT->CYCCNT;
  uint32_t au32_ticks = (HAL_RCC_GetHCLKFreq() / 1000);
  au32_milliseconds *= au32_ticks;
  while ((DWT->CYCCNT - au32_initial_ticks) < au32_milliseconds);
}
```

3. **STM32 Hardware Timer Delay**
   An alternative way for the DWT to achieve time delay in milliseconds, microseconds, and even nanoseconds (require SysCLK ¿= 100MHz) is the hardware timers in the microcontroller.

The timer Prescaler is set up depending on the Fsys clock so that each timer tick accounts for a 1-microsecond time unit. This information is stored in a static global variable in the code file and you don't have to manually figure anything out. Jut initialize the timer delay function and you're good to go. If another timer to be used for this task, you'll only have to change this line down below.

TimerDelay_Initialization() Function

```c
void TimerDelay_Init(void)
{
    gu32_ticks = (HAL_RCC_GetHCLKFreq() / 1000000);

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

    HTIMx.Instance = TIMER;
    HTIMx.Init.Prescaler = gu32_ticks-1;
    HTIMx.Init.CounterMode = TIM_COUNTERMODE_UP;
    HTIMx.Init.Period = 65535;
    HTIMx.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HTIMx.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    if (HAL_TIM_Base_Init(&HTIMx) != HAL_OK)
    {
      Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&HTIMx, &sClockSourceConfig) != HAL_OK)
    {
      Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&HTIMx, &sMasterConfig) != HAL_OK)
    {
      Error_Handler();
    }

    HAL_TIM_Base_Start(&HTIMx);
}
```

TimerDelay_us() Function

```c
void delay_us(uint16_t au16_us)
{
    HTIMx.Instance->CNT = 0;
    while (HTIMx.Instance->CNT < au16_us);
}
```

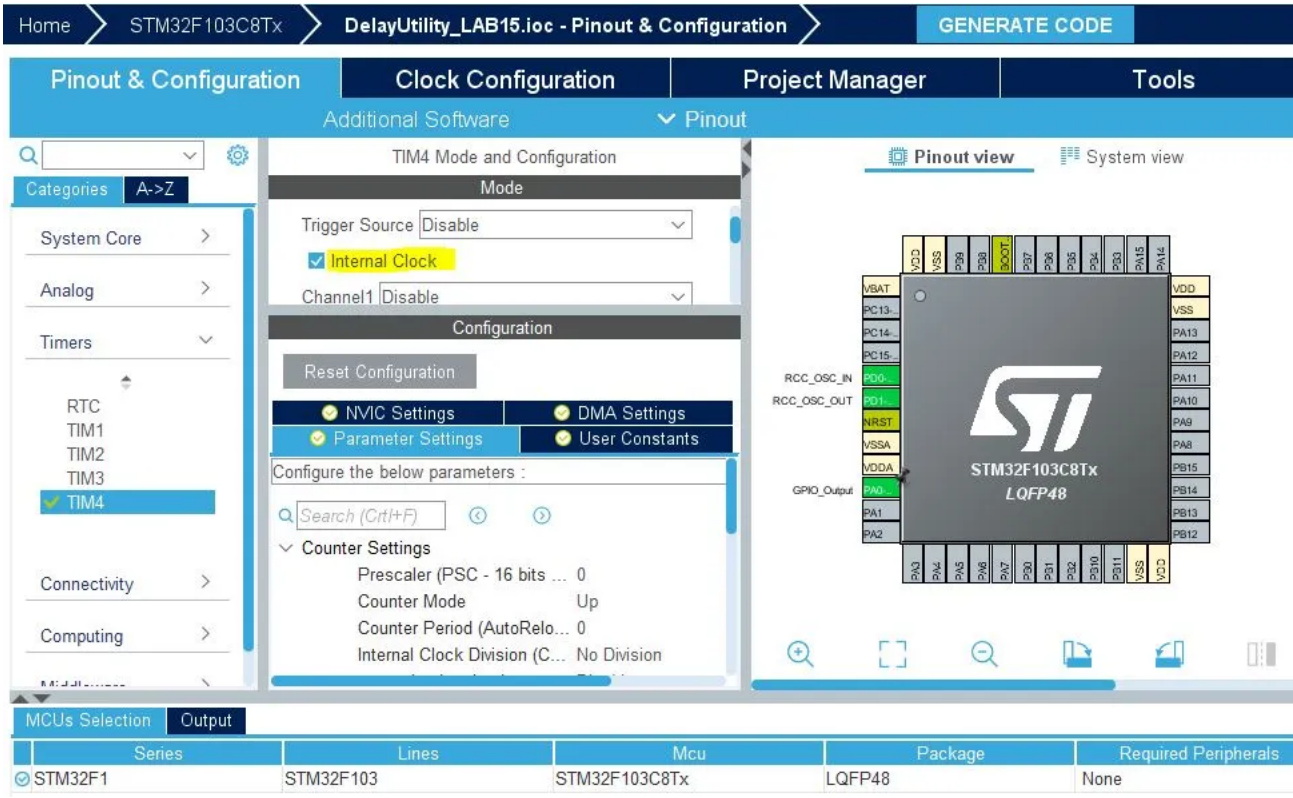
TimerDelay_ms() Function

```c
void delay_ms(uint16_t au16_ms)
{
    while(au16_ms > 0)
    {
    HTIMx.Instance->CNT = 0;
    au16_ms--;
    while (HTIMx.Instance->CNT < 1000);
    }
}
```

4. **STM32 Delay Microsecond & Millisecond Utility**
5. **STM32 Delay us & ms Example LAB**

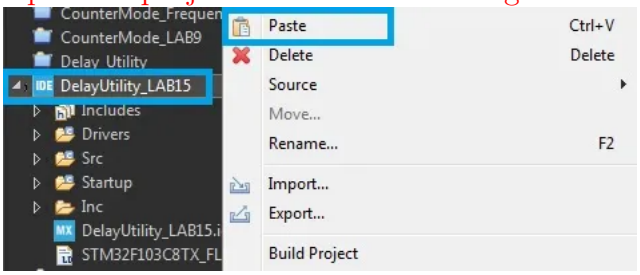
- Set up a new project as usual with a GPIO output pin for testing
- Add the util to our project directory
- include the delay utilities (timer or DWT) and test it

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Configure A0 pin to be a GPIO output pin**
- **Step4: Select the timer you're willing to use for the delay. Let it be TIMER4, just enable the clock**

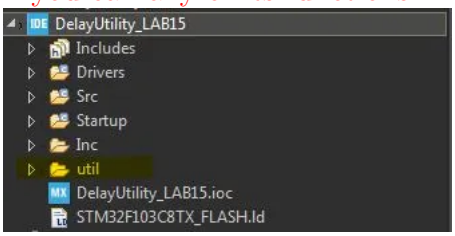


- **Step5: Set The RCC External Clock Source**
- **Step6: Set The System Clock To Be 72MHz**
- **Step7: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**
- **Step8: Add The util directory & add it to the path as a source code directory**

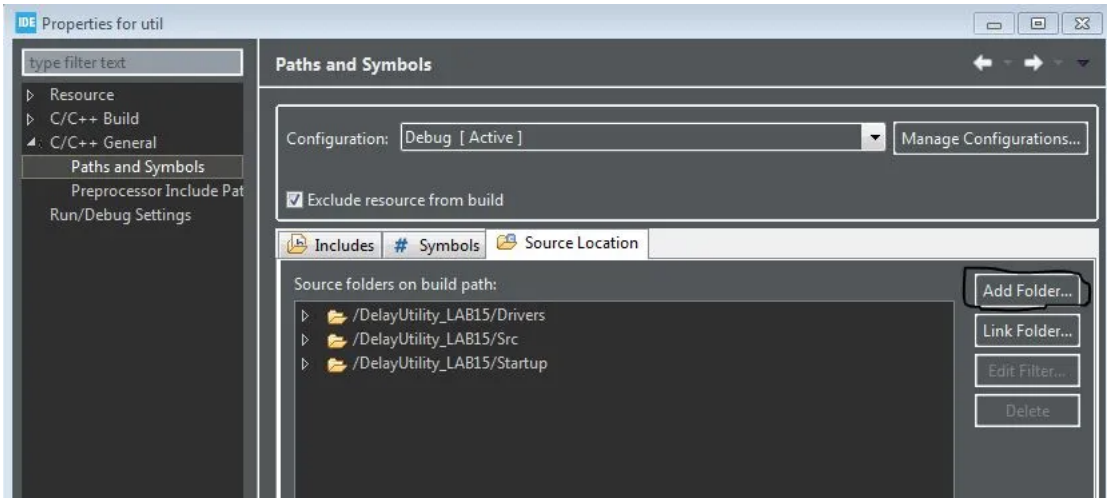
  Open the project in CubeIDE and right-click the project name and click paste



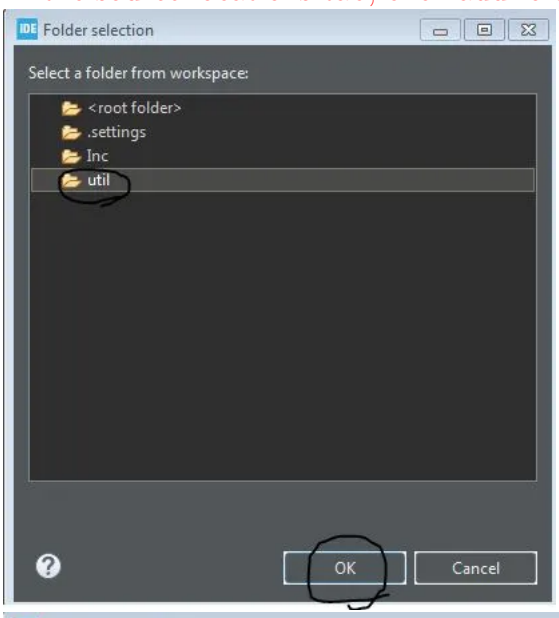
  The folder will be added to your project. However, it's not considered as a source code directory, so it won't be compiled and give you linking error in the linking stage if you call any of its functions.

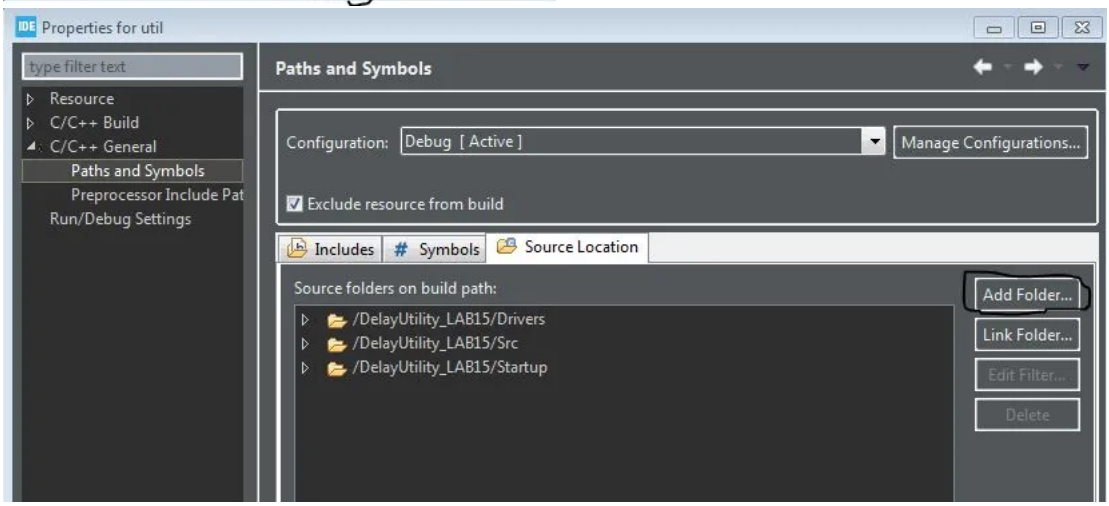


  Right-click the util folder and click properties. And navigate to C/C++ paths and symbols, and source locations tab.



In the source locations tab, click add folder and add our util folder.





- **Step9: The Last step is to remove the timer config function from the main.c file & make sure to name the timer instance correctly in the TimerDelay.c file**

6. Here is The Application Code For This LAB **Note** that the hal_timer_config function is removed and never called. The timer initialization is now done in the TimerDelay_Init() routine.

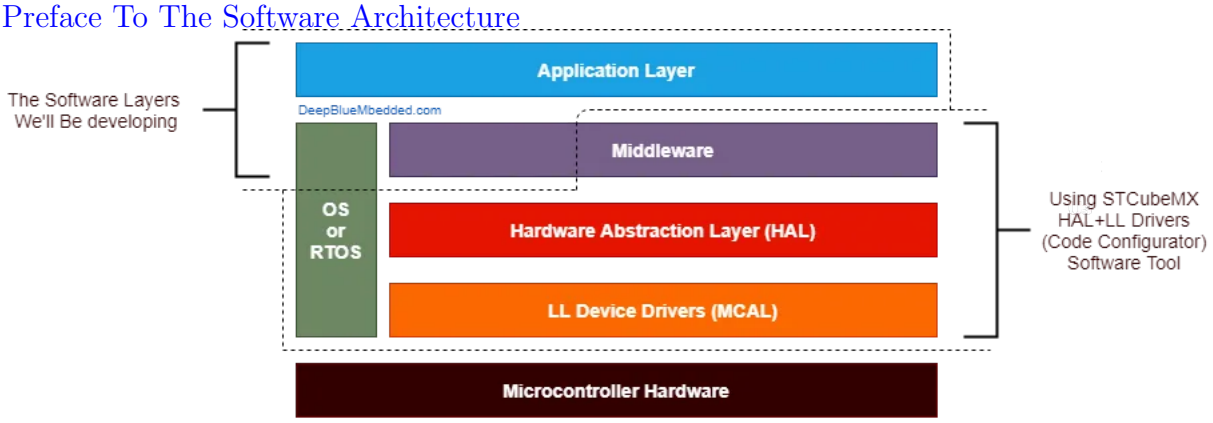
```c
#include "main.h"
#include "../util/Timer_Delay.h"
#include "../util/DWT_Delay.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    /* Initialize The TimerDelay*/
    TimerDelay_Init();

    while (1)
    {
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_0);
        delay_ms(100);
    }
}
```

or you use initialize DWT: DWT_Delay_Init();

XX Adding ECUAL Drivers To Your STM32 Project & Configurations Options

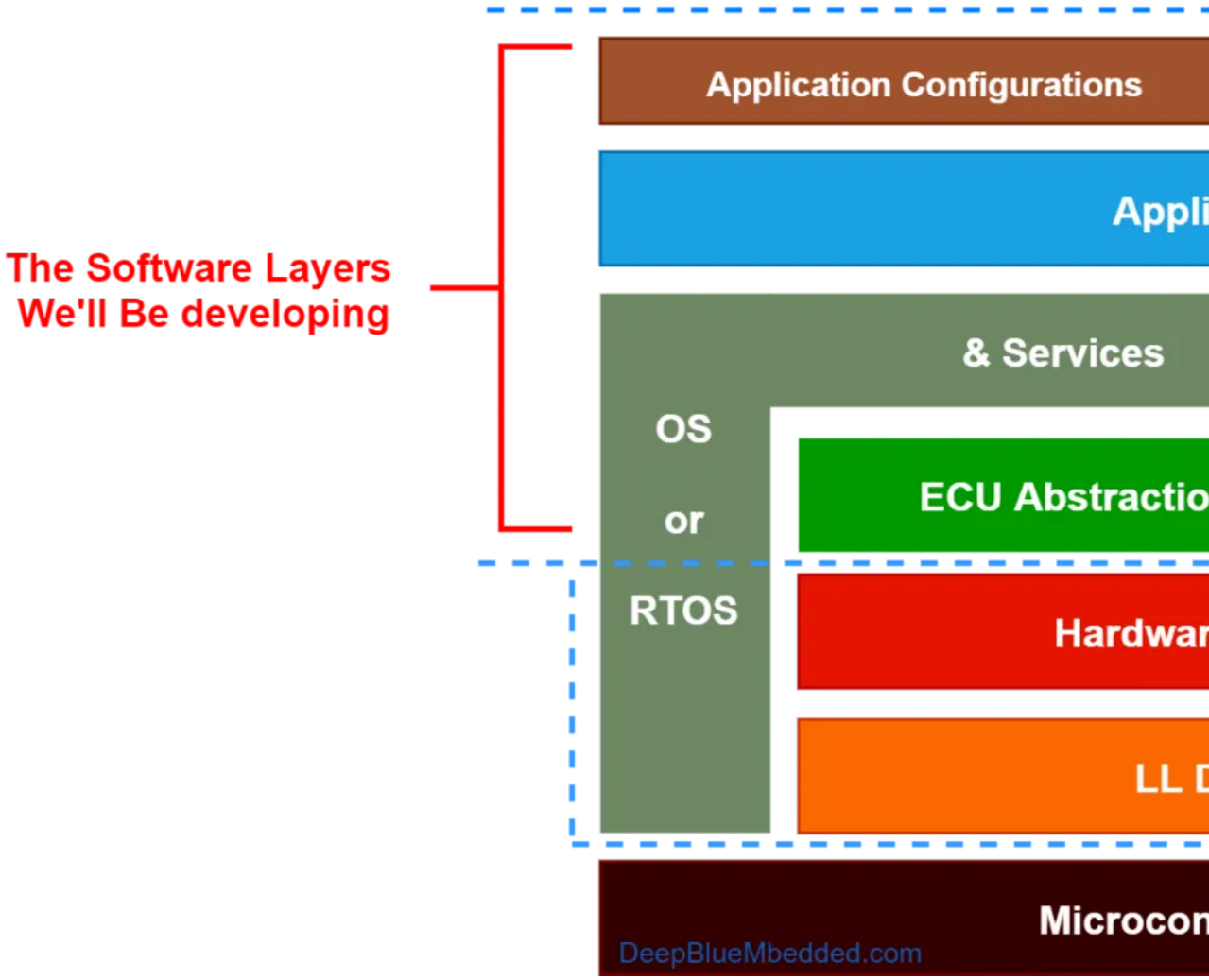
1. **Preface To The Software Architecture**

1.1 Preface To The Software Architecture



There were no issues in this and you can justify this by saying its a simple architecture and looks clear enough. However, proceeding with this setting will result in more problems in the future. You'll end up with an application code that is tied up to the underlying HAL drivers (like a servo motor that is hooked to TIMER2, or whatever). This significantly reduces the application code portability across multiple platforms and even introduces the possibility of clashes on the same target MCU itself (like a servo motor being hooked to TIMER2 CH1 meanwhile ultrasonic driver is hooked to the same hardware TIMER).

1.2 Adding ECUAL Layer To The Software Layered Architecture



An ECUAL driver is basically a set of functions that initialized the MCU HW via the HAL and does the calling of HAL functions, necessary calculations, algorithms, and utilities, in order to abstract the hardware handling from the application layer. So the application code doesn't talk directly to TIMER1 or USART3 or whatever. The ECUAL driver for the servo motor for example will check its configuration file to know which timer is assigned to it by the user (the application programmer) and therefore, initialize this timer and do its work. Now, the application turns ON and controls the servo motor without knowing or worrying about which timer is being used in the background. And the user (the programmer) can at any time open the servo driver configuration files and assign another timer or change the resolution or whatever.

1.3 Examples For ECUAL Drivers

- Sevormotor
- Stepper Motor
- I2C_LCD
- KeyPAD
- CapTouch
- 7Segments Display
- MPU6050
- OLED
- RTC
- ... The list goes on!

1.4 ECUAL Drivers Configurations

The drivers will have a couple of files for configuration parameters encapsulated in a structure that gets initialized by the user in the cfg.c file. This parameter will, therefore, be externed to my driver source code file like servo.c for example. The global cfg variable is now seen by the driver source file and can be accessed.

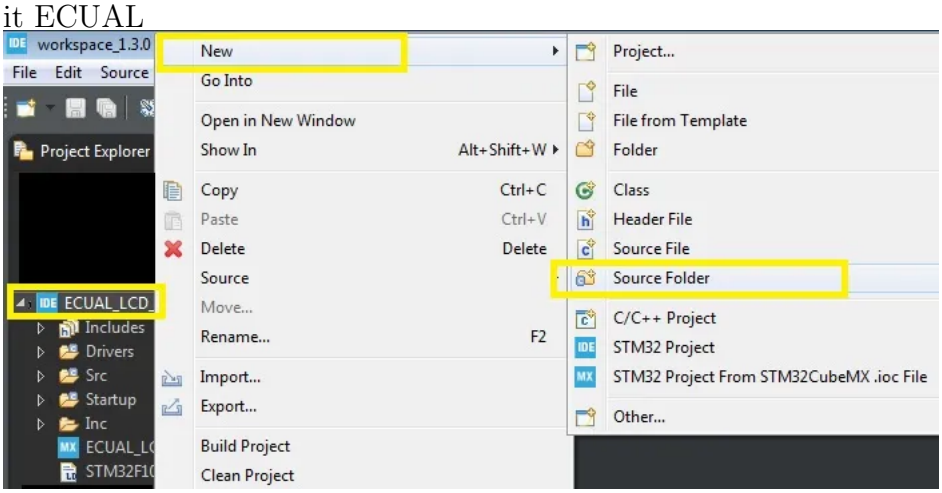
1.5 ECUAL Utilities & Math Dependency

The ECUAL drivers will also at some points require some math functions or algorithms for filtering signals and some DSP stuff. In this case, we'll develop the necessary math work and embed that into our projects in a separate math directory and that's the way I'll go in these cases.

2. **How To Add An ECUAL Driver To A New STM32 Project?** Example To Add ECUAL Driver "LCD16x2"

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Set The RCC External Clock Source**
- **Step4: Set The System Clock To Be 72MHz Or Whatever**
- **Step5: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**
- **Step6: Open The Project In CubeIDE & Create A Source Code Directory Called "ECUAL"**

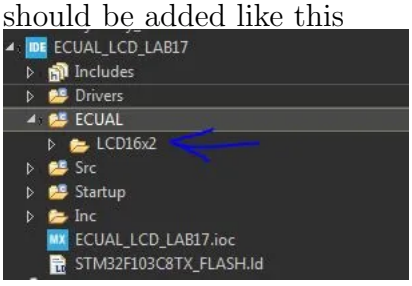
  Right-Click The Project Name in the navigator and create the new directory & name it ECUAL



- **Step7: Now, Copy The Driver Folder You've Downloaded And Paste It Into ECUAL**

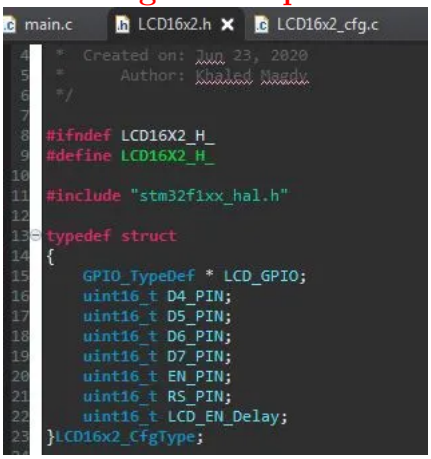
  The file Ecual download:https://drive.google.com/file/d/1exG5ua_rftnKutJX8mMl-B-NU6RKl
  view?usp=sharing

  Right-Click the ECUAL in the CubeIDE project navigator and paste the folder. It should be added like this



Note:that you'll have to do the same steps for other drivers other than the LCD. It could be servo, stepper, or whatever. The steps are the same though.

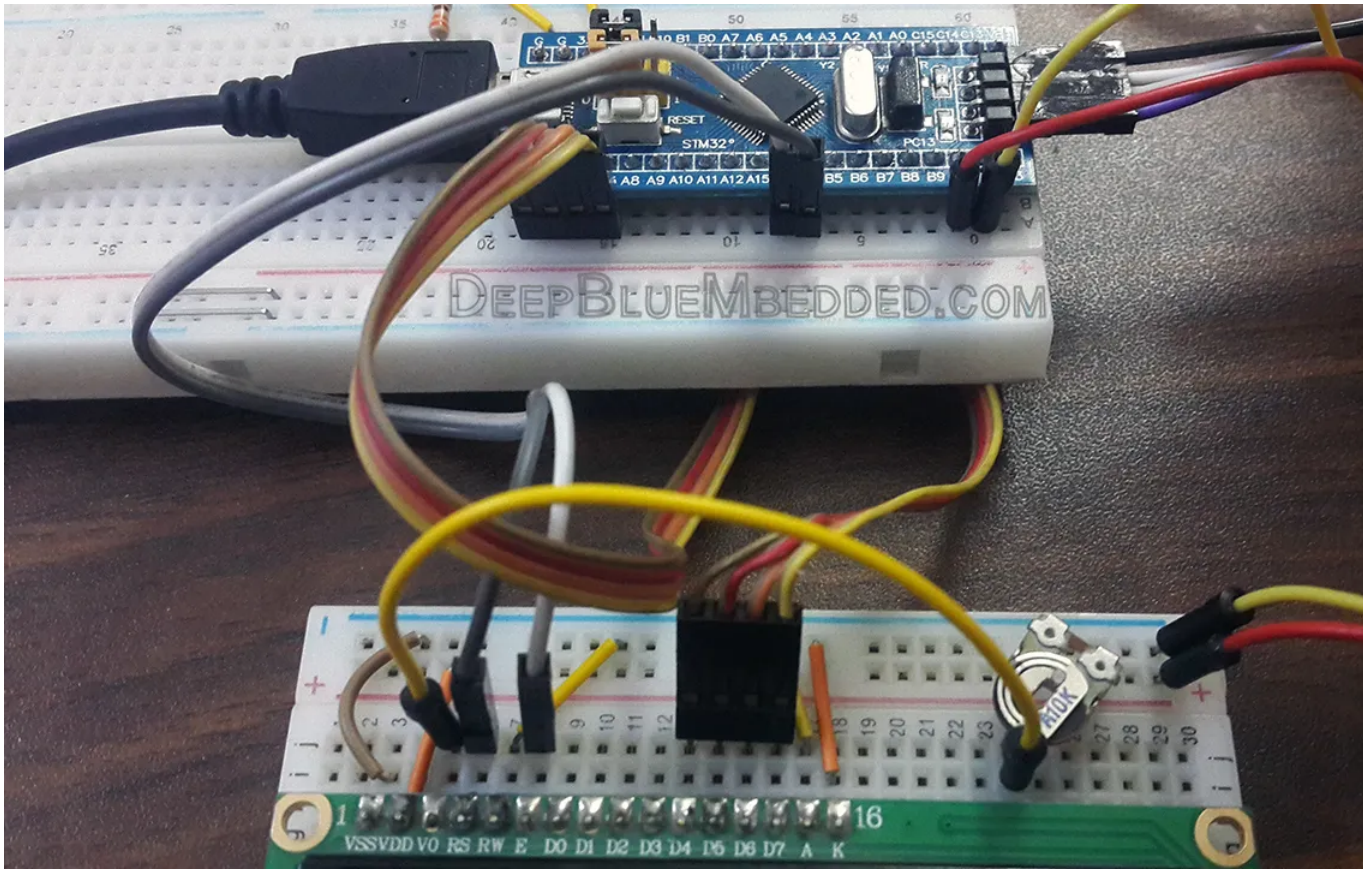
- **Step9: Add The util Directory in the same manner as you've done with the ECUAL**
  Paste the util files. Those files are for delay functions used by the LCD16x2 driver
  The file util_downloads:https://drive.google.com/file/d/1x6N19YExuAgTW3Cooq95-RqU2xL4view?usp=sharing
- **Step10: You can start developing your application now. But first, check the configuration parameters file**



3. **STM32 LCD Display 16x2 LAB17**
   Here is The Application Code For This LAB (main.c)

```c
#include "main.h"
#include "../ECUAL/LCD16x2/LCD16x2.h"
void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    LCD_Init();
    LCD_Clear();
    LCD_Set_Cursor(1, 1);
    LCD_Write_String("It Works! GG izi");

    while (1)
    {

    }
}
```

The LAB Connections

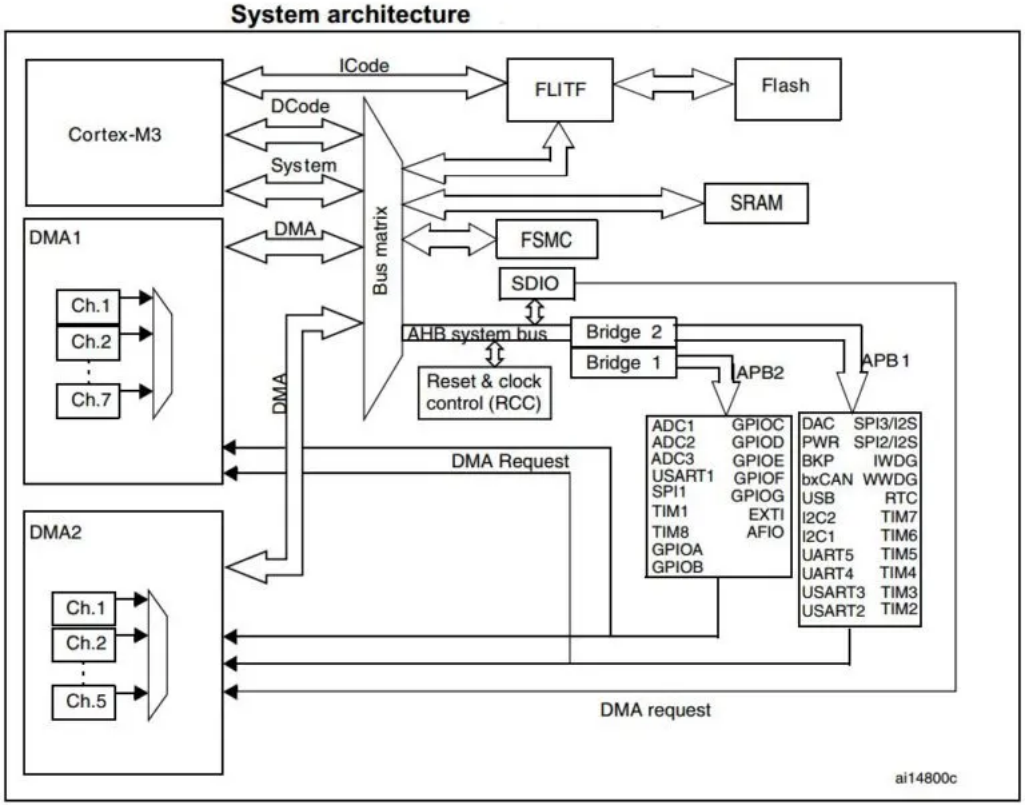


XXI STM32 DMA Tutorial – Using Direct Memory Access (DMA) In STM32

1. **What Is Direct Memory Access (DMA)?**
   A Direct Memory Access (DMA) unit is a digital logic element in computer architecture that can be used in conjunction with the main microprocessor on the same chip in order to offload the memory transfer operations

2. **STM32 DMA Hardware**

   2.1 **For STM32F103C8T6 (The Blue Pill MCU)**
   Direct memory access (DMA) is used in order to provide high-speed data transfer between peripherals and memory as well as memory to memory. Data can be quickly moved by DMA without any CPU actions. This keeps CPU resources free for other operations.



2.2 The DMA Units In STM32F103 Have The Following Features

- 12 independently configurable channels (requests): 7 for DMA1 and 5 for DMA2
- Each of the 12 channels is connected to dedicated hardware DMA requests, software trigger is also supported on each channel. This configuration is done by software.
- Independent source and destination transfer size (byte, half-word, word), emulating packing, and unpacking. Source/destination addresses must be aligned on the data size.
- Support for circular buffer management
- 3 event flags (DMA Half Transfer, DMA Transfer complete and DMA Transfer Error) logically ORed together in a single interrupt request for each channel
- Memory-to-memory transfer
- Peripheral-to-memory and memory-to-peripheral, and peripheral-to-peripheral transfers
- Access to Flash, SRAM, APB1, APB2 and AHB peripherals as source and destination
- Programmable number of data to be transferred: up to 65536

2.3 DMA Data Transactions

DMA transfer consists of three operations:

- The loading of data from the peripheral data register or a location in memory addressed through an internal current peripheral/memory address register. The start address used for the first transfer is the base peripheral/memory address programmed in the DMA_CPARx or DMA_CMARx register.
- The storage of the data loaded to the peripheral data register or a location in memory addressed through an internal current peripheral/memory address register. The start address used for the first transfer is the base peripheral/memory address programmed in the DMA_CPARx or DMA_CMARx register

- The post-decrementing of the DMA_CNDTRx register, which contains the number of transactions that have still to be performed.

2.4 **DMA Arbiter**

The arbiter manages the channel requests based on their priority and launches the peripheral/memory access sequences. The priorities are managed in two stages:

- Software: each channel priority can be configured in the DMA_CCRx register. There are four levels:
  – Very high priority
  – High priority
  – Medium priority
  – Low priority
- Hardware: if 2 requests have the same software priority level, the channel with the lowest number will get priority versus the channel with the highest number. For example, channel 2 gets priority over channel 4.

2.5 **DMA Channels**

2.6 **DMA Circular Mode**

The circular mode is available to handle circular buffers and continuous data flows (e.g. ADC scan mode). This feature can be enabled using the CIRC bit in the DMA_CCRx register. When the circular mode is activated, the number of data to be transferred is automatically reloaded with the initial value programmed during the channel configuration phase, and the DMA requests continue to be served.

2.7 **DMA Memory-To-Memory Mode**

The DMA channels can also work without being triggered by a request from a peripheral. This mode is called Memory to Memory mode. Memory to Memory mode may not be used at the same time as Circular mode.

2.8 **STM32 DMA Interrupts**

An interrupt can be produced on a Half-transfer, Transfer complete, or Transfer error for each DMA channel. Separate interrupt enable bits are available for flexibility.

2.9 **DMA Request Mapping**

The peripheral DMA requests can be independently activated/de-activated by programming the DMA control bit in the registers of the corresponding peripheral.

Summary of DMA1 requests for each channel

Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
ADC1	ADC1	-	-	-	-	-	-
SPI/I^2S	-	SPI1_RX	SPI1_TX	SPI2/I2S2_RX	SPI2/I2S2_TX	-	-
USART	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C	-	-	-	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1	-	TIM1_CH1	-	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	-
TIM2	TIM2_CH3	TIM2_UP	-	-	TIM2_CH1	-	TIM2_CH2 TIM2_CH4
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	-	-	TIM3_CH1 TIM3_TRIG	-
TIM4	TIM4_CH1	-	-	TIM4_CH2	TIM4_CH3	-	TIM4_UP

3. **STM32 DMA Configuration**

The following sequence should be followed to configure a DMA CHANNELx (where x is the channel number).

- Set the peripheral register address in the DMA_CPARx register. The data will be moved from/ to this address to/ from the memory after the peripheral event.
- Set the memory address in the DMA_CMARx register. The data will be written to or read from this memory after the peripheral event.
- Configure the total number of data to be transferred in the DMA_CNDTRx register. After each peripheral event, this value will be decremented.
- Configure the channel priority using the PL[1:0] bits in the DMA_CCRx register
- Configure data transfer direction, circular mode, peripheral & memory incremented mode, peripheral & memory data size, and interrupt after half and/or full transfer in the DMA_CCRx register
- Activate the channel by setting the ENABLE bit in the DMA_CCRx register.
  The half-transfer flag (HTIF) is set and an interrupt is generated if the Half Transfer Interrupt Enable bit (HTIE) is set. At the end of the transfer, the Transfer Complete Flag (TCIF) is set and an interrupt is generated if the Transfer Complete Interrupt Enable bit (TCIE) is set.

4. **STM32 DMA Examples**
   There are several use cases for the DMA units in STM32 microcontrollers. We'll implement some of them in the upcoming tutorials' LABs and projects. However, here are a handful of possible scenarios:

   - UART data reception from a terminal to a local buffer.
   - ADC circular buffer conversions for multiple channels with capacitive touchPADs connected.
   - SPI external memory interfacing for high-speed data logging.
   - SPI camera interfacing.
   - and much more...

5. **How To Receive UART Serial Data With STM32 – DMA / Interrupt / Polling**

5.1 STM32 UART Receive

5.1.1 The Polling Method

The polling method is essentially a blocking function being called from the main routine and it does block the CPU so it can't proceed in the main task execution until a certain amount of UART data bytes are received.After receiving the required amount of data, the function ends and the CPU continues the main code execution. Otherwise, and if the UART peripheral for some reason didn't receive the expected amount of data, the function will keep blocking the CPU for a certain amount of time "time-out" after which it gets terminated and gives back the control to the CPU to resume the main code.
This function for example
HAL_UART_Receive (&huart1, UART1_rxBuffer, 12, 5000);

5.1.2 The Interrupt Method

Using interrupt signals is a convenient way to achieve serial UART data reception.
HAL_UART_Receive_IT(&huart1, UART1_rxBuffer, 12);

5.1.3 The DMA Method

Using the DMA unit in order to direct the received serial UART data from the UART peripheral directly to the memory is considered to be the most efficient way to do such a task.

HAL_UART_Receive_DMA (&huart1, UART1_rxBuffer, 12);

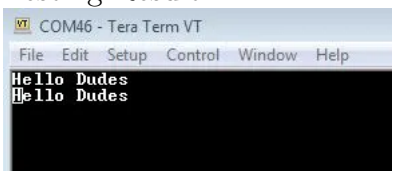
Note:Make sure you've completed the STM32 UART tutorial for more information about how it works in detail. And for more basic information about serial communication and UART absolute guide check this out.

5.2 STM32 UART Receive Polling Example

Step1:Start New CubeMX Project & Setup The Clock
Step2:Setup The UART1 Peripheral in Async Mode @ 9600bps
Step3:Generate Code & Open CubeIDE or Any Other IDE You're Using
Step4:Write This Application Code (main.c)

```c
#include "main.h"

uint8_t UART1_rxBuffer[12] = {0};

UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive (&huart1, UART1_rxBuffer, 12, 5000);
    HAL_UART_Transmit(&huart1, UART1_rxBuffer, 12, 100);

    while (1)
    {

    }

}
```

Testing Result

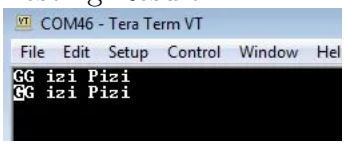


5.3 STM32 UART Recieve Interrupt Example

Step1:Start New CubeMX Project & Setup The Clock
Step2:Setup The UART1 Peripheral in Async Mode @ 9600bps
Step3:Enable The UART1 Global Interrupt From NVIC Controller Tab
Step4:Generate Code & Open CubeIDE or Any Other IDE You're Using
Step5:Write This Application Code (main.c)

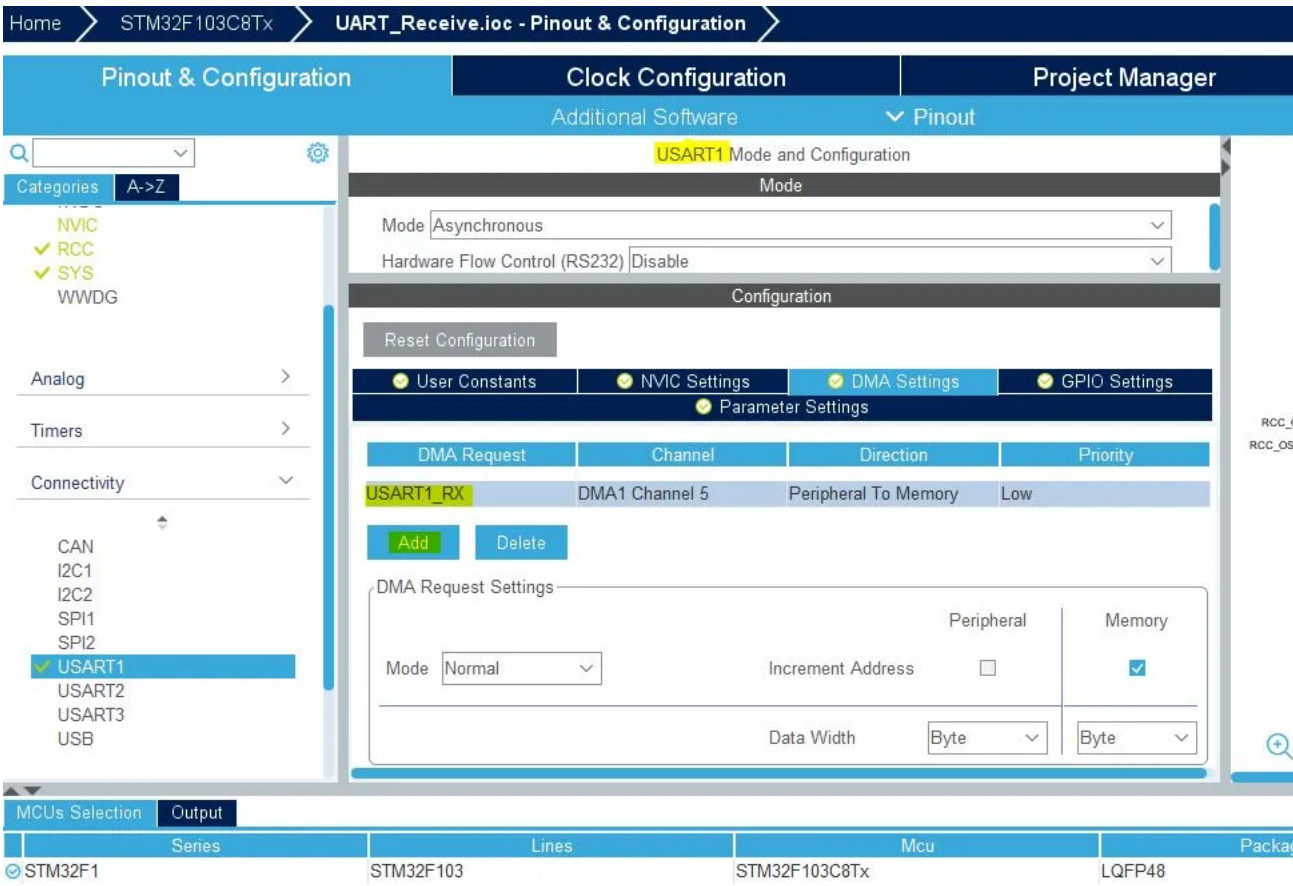
```c
#include "main.h"

uint8_t UART1_rxBuffer[12] = {0};

UART_HandleTypeDef huart1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive_IT (&huart1, UART1_rxBuffer, 12);

    while (1)
    {

    }

}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Transmit(&huart1, UART1_rxBuffer, 12, 100);
    HAL_UART_Receive_IT(&huart1, UART1_rxBuffer, 12);
}
```

Testing Result



5.4 STM32 UART Receive DMA Example

Step1:Start New CubeMX Project & Setup The Clock
Step2:Setup The UART1 Peripheral in Async Mode @ 9600bps
Step3:Add A DMA Channel For UART RX From The DMA Tab

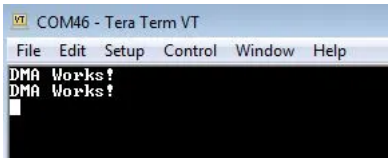


Step4:Generate Code & Open CubeIDE or Any Other IDE You're Using

Step5:Write This Application Code (main.c)

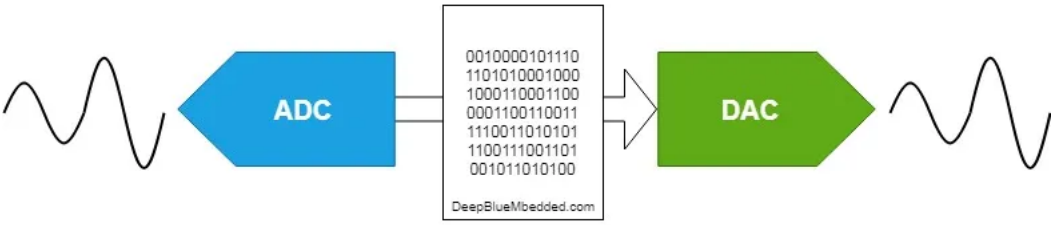
```c
#include "main.h"

uint8_t UART1_rxBuffer[12] = {0};

UART_HandleTypeDef huart1;
DMA_HandleTypeDef hdma_usart1_rx;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART1_UART_Init();

    HAL_UART_Receive_DMA (&huart1, UART1_rxBuffer, 12);

    while (1)
    {

    }

}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Transmit(&huart1, UART1_rxBuffer, 12, 100);
    HAL_UART_Receive_DMA(&huart1, UART1_rxBuffer, 12);
}
```

Testing Result



XXII STM32 ADC Tutorial – Complete Guide With Examples

1. **Analog-To-Digital Converters (ADC) Preface**
   An ADC (Analog-To-Digital) converter is an electronic circuit that takes in an analog voltage as input and converts it into digital data, a value that represents the voltage level in binary code.
   The ADC does the counter operation that of a DAC, while an ADC (A/D) converts analog voltage to digital data the DAC (D/A) converts digital numbers to the analog voltage on the output pin.



2. **STM32 ADC Brief**
   ADC Features
   - 12-bit resolution
   - Interrupt generation at End of Conversion, End of Injected conversion and Analog watchdog event
   - Single and continuous conversion modes
   - Scan mode for automatic conversion of channel 0 to channel ‘n’
   - Self-calibration
   - Data alignment with in-built data coherency
   - Channel by channel programmable sampling time
   - External trigger option for both regular and injected conversion
   - Discontinuous mode
   - Dual-mode (on devices with 2 ADCs or more)
   - ADC conversion time: 1 µs at 56 MHz (1.17 µs at 72 MHz)
   - ADC supply requirement: 2.4 V to 3.6 V
   - ADC input range: $V_{REF}- <= V_{IN} <= V_{REF}+$
   - DMA request generation during regular channel conversion
3. **STM32 ADC Functional Description**

3.1 STM32 ADC Block Diagram

3.2 The ADC Clock

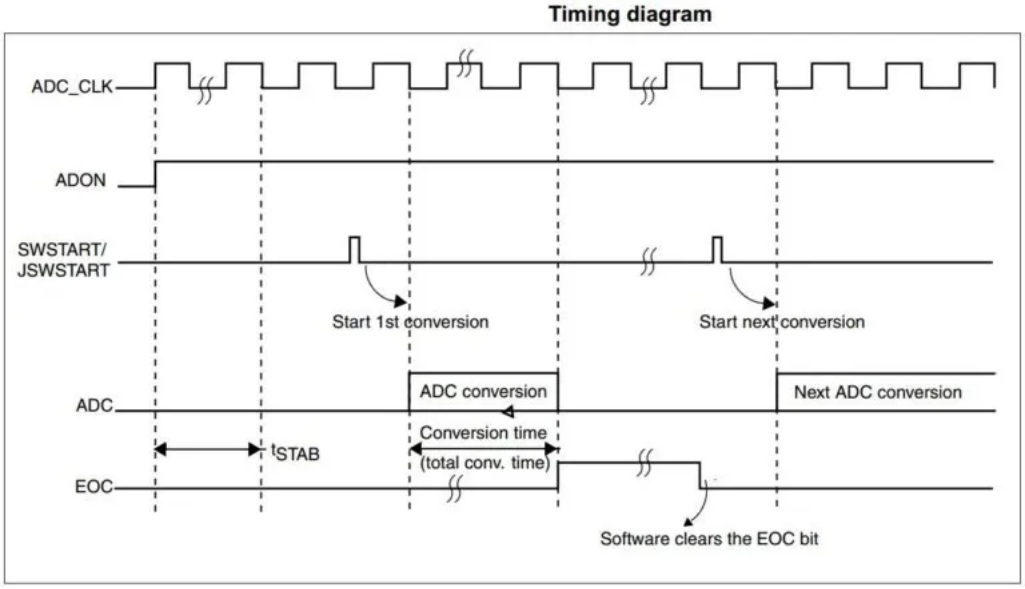
The ADCCLK clock provided by the Clock Controller is synchronous with the PCLK2 (APB2 clock). The RCC controller has a dedicated programmable Prescaler for the ADC clock, and it must not exceed 14 MHz.

3.3 Channel Selection

The Regular Group is composed of up to 16 conversions. The regular channels and their order in the conversion sequence must be selected in the ADC_SQRx registers. The total number of conversions in the regular group must be written in the L[3:0] bits in the ADC_SQR1 register.

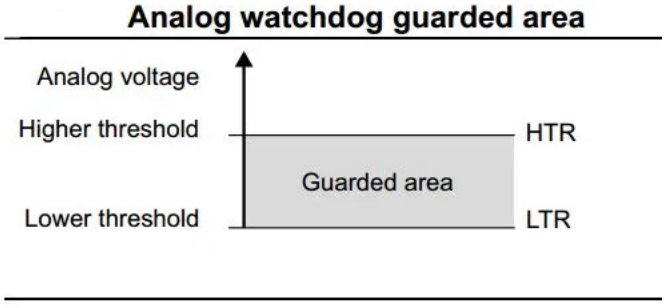
The Injected Groupis composed of up to 4 conversions. The injected channels and their order in the conversion sequence must be selected in the ADC_JSQR register. The total number of conversions in the injected group must be written in the L[1:0] bits in the ADC_JSQR register.

3.4 ADC Conversion Time & Timing Diagram



3.5 The ADC Analog Watchdog (AWD)

The AWD analog watchdog status bit is set if the analog voltage converted by the ADC is below a low threshold or above a high threshold.



3.6 ADC Result Data Alignment

ALIGN bit in the ADC_CR2 register selects the alignment of data stored after conversion. Data can be left or right-aligned as shown in the diagram below.

4. STM32 ADC Modes of Operation

4.1 Single Conversion Mode

In Single Conversion mode, the ADC does one conversion. This mode is started either by setting the ADON bit in the ADC_CR2 register (for a regular channel only) or by an external trigger (for a regular or injected channel), while the CONT bit is 0.

4.2 Continuous Conversion Mode

In continuous conversion mode, ADC starts another conversion as soon as it finishes one. This mode is started either by an external trigger or by setting the ADON bit in the ADC_CR2 register, while the CONT bit is 1.

4.3 Scan Mode

When using scan mode, DMA bit must be set and the direct memory access controller is used to transfer the converted data of regular group channels to SRAM after each update of the ADC_DR register. The injected channel converted data is always stored in the ADC_JDRx registers.

4.4 Discontinuous Mode

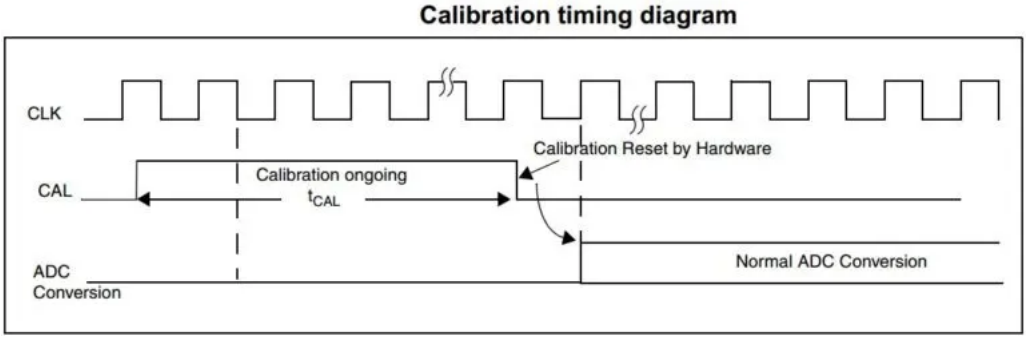
This mode is enabled by setting the DISCEN bit in the ADC_CR1 register. It can be used to convert a short sequence of n conversions (n <=8) which is a part of the sequence of conversions selected in the ADC_SQRx registers. The value of n is specified by writing to the DISCNUM[2:0] bits in the ADC_CR1 register.

5. ADC Conversion On External Triggers

External trigger for regular channels for ADC1 and ADC2

Source	Type	EXTSEL[2:0]
TIM1_CC1 event	Internal signal from on-chip timers	000
TIM1_CC2 event		001
TIM1_CC3 event		010
TIM2_CC2 event		011
TIM3_TRGO event		100
TIM4_CC4 event		101
EXTI line 11/TIM8_TRGO event(1)(2)	External pin/Internal signal from on-chip timers	110
SWSTART	Software control bit	111

6. **STM32 ADC Calibration**



The STM32 HAL does provide a function within the ADC APIs dedicated to starting the calibration process and as said before it's a recommended step after initializing the ADC hardware at the system power-up.

7. **STM32 ADC Sampling Time** ADC samples the input voltage for a number of ADC_CLK cycles which can be modified using the SMP[2:0] bits in the ADC_SMPR1 and ADC_SMPR2 registers. Each channel can be sampled with different sample times.
The Total ADC Conversion Time is calculated as follows:
Tconv = Sampling time + 12.5 cycles
The ADC Sampling Rate (Frequency) is calculated using this formula:
SamplingRate = 1 / Tconv

8. **STM32 ADC Resolution, Reference, Formulas**

8.1 **STM32 ADC Resolution**

The STM32 ADC has a resolution of 12-Bit which results in a total conversion time of SamplingTime+12.5 clock cycles. However, higher sampling rates can be achieved by sacrificing the high-resolution. Therefore, the resolution can be dropped down to 10-Bit, 8-Bit, or 6-Bit, and hence the conversion time is much shorter and the sampling rate increases. This can be configured and implemented in software by the programmer and the STM32 HAL does provide APIs to set all the ADC parameters including its resolution.

8.2 **ADC Reference Voltage**

Name	Signal type	Remarks
V_{REF+}	Input, analog reference positive	The higher/positive reference voltage for the ADC, 2.4 V $\leq V_{REF+} \leq V_{DDA}$
V_{DDA} (1)	Input, analog supply	Analog power supply equal to V_{DD} and 2.4 V $\leq V_{DDA} \leq$ 3.6 V
V_{REF-}	Input, analog reference negative	The lower/negative reference voltage for the ADC, $V_{REF-} = V_{SSA}$
V_{SSA} (1)	Input, analog supply ground	Ground for analog power supply equal to V_{SS}

If you're using a development board, you may need to check out its schematic diagram as it may not be connecting the ADC Vref at all or connecting it to a 2.5v for example, so the ADC will saturate and give you 4096 before the input analog voltage reaches 3.3v and you're wondering why! it may be because the reference voltage is set to a value less than 3.3v, so it's something to consider. **STM32 ADC Formulas**
ADC Result Voltage (Analog Input Value)
Vin = ADC_Res x (Reference Voltage / 4096)
where Reference Voltage = (V_{REF+})–(V_{REF-})

9. **ADC & DMA**
   Since converted regular channels value are stored in a unique data register, it is necessary to use DMA for the conversion of more than one regular channel. This avoids the loss of data already stored in the ADC_DR register.

10. **STM32 ADC Interrupts**
    An interrupt can be produced on the end of conversion for regular and injected groups and when the analog watchdog status bit is set. Separate interrupt enable bits are available for flexibility.

ADC interrupts

Interrupt event	Event flag	Enable Control bit
End of conversion regular group	EOC	EOCIE
End of conversion injected group	JEOC	JEOCIE
Analog watchdog status bit is set	AWD	AWDIE

11. **Different Ways To Read STM32 ADC**

    11.1 **The Polling Method**
    It's the easiest way in code in order to perform an analog to digital conversion using the ADC on an analog input channel. However, it's not an efficient way in all cases as it's considered to be a blocking way of using the ADC. As in this way, we start the A/D conversion and wait for the ADC until it completes the conversion so the CPU can resume processing the main code.

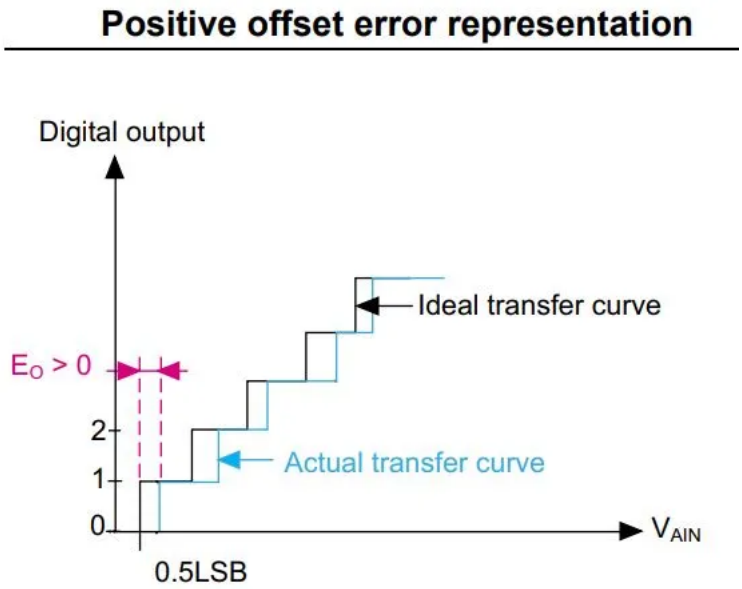
    11.2 **The Interrupt Method**
    The interrupt method is an efficient way to do ADC conversion in a non-blocking manner, so the CPU can resume executing the main code routine until the ADC completes the conversion and fires an interrupt signal so the CPU can switch to the ISR context and save the conversion results for further processing.

    11.3 **The DMA Method**
    Lastly, the DMA method is the most efficient way of converting multiple ADC channels at very high rates and still transfers the results to the memory without CPU intervention which is so cool and time-saving technique.
    NOTES: Code Examples For The 3 Methods Are Provided in The Next Tutorial. After completion, click the next tutorial to see the examples.

12. **STM32 ADC Conversion Errors**

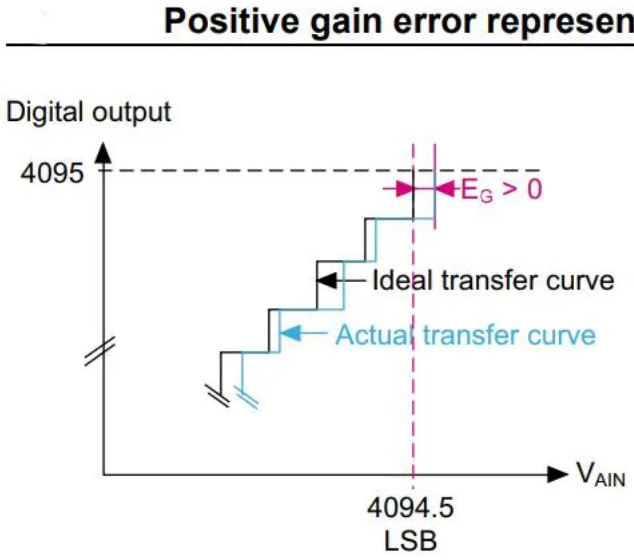
    12.1 **ADC Errors Due To The ADC Itself**

        12.1.1 ADC Offset Error
        The offset error is the deviation between the first actual transition and the first ideal transition. The first transition occurs when the digital ADC output changes from 0 to 1. Ideally, when the analog input ranges between 0.5 LSB and 1.5 LSB, the digital output should be 1. Still, ideally, the first transition occurs at 0.5 LSB. The offset error is denoted by EO.



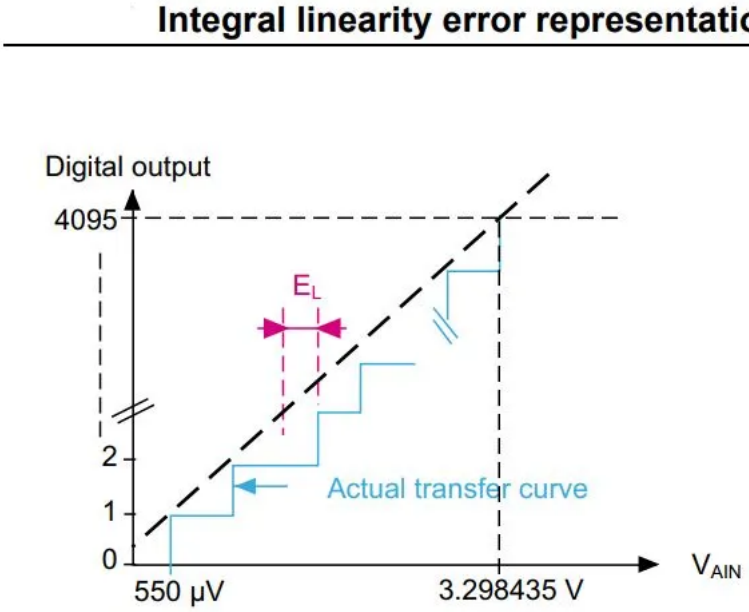
12.1.2 ADC Gain Error

The gain error is the deviation between the last actual transition and the last ideal transition. It is denoted by EG. The last actual transition is the transition from 0xFFE to 0xFFF. Ideally, there should be a transition from 0xFFE to 0xFFF when the analog input is equal to V_{REF+} – 0.5 LSB. So for VREF+= 3.3 V, the last ideal transition should occur at 3.299597 V. If the ADC provides the 0xFFF reading for $V_{AIN} < V_{REF+}$ – 0.5 LSB, then a negative gain error is obtained. The gain error is obtained by the formula below:
EG = Last actual transition – ideal transition



12.1.3 Integral Linearity Error

The integral linearity error is the maximum deviation between any actual transition and the endpoint correlation line. The ILE is denoted by EL. The endpoint correlation line can be defined as the line on the A/D transfer curve that connects the first actual transition with the last actual transition.



12.2 ADC Errors Due To The Environment

12.2.1 ADC Reference Voltage Noise

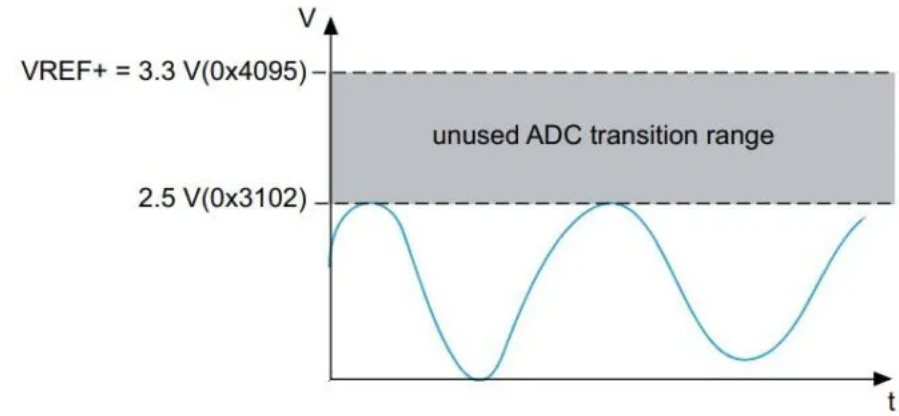
As the ADC output is the ratio between the analog signal voltage and the reference voltage, any noise on the analog reference causes a change in the converted digital value. V_{DDA} analog power supply is used on some packages as the reference voltage (V_{REF+}), so the quality of the V_{DDA} power supply has an influence on ADC error.

12.2.2 Analog Input Signal Noise

Small but high-frequency signal variation can result in big conversion errors during sampling time. This noise is generated by electrical devices, such as motors, engine ignition, power lines. It affects the source signal (such as sensors) by adding an unwanted signal. As a consequence, the ADC conversion results are not accurate.

12.2.3 ADC Dynamic Range Bad Matching

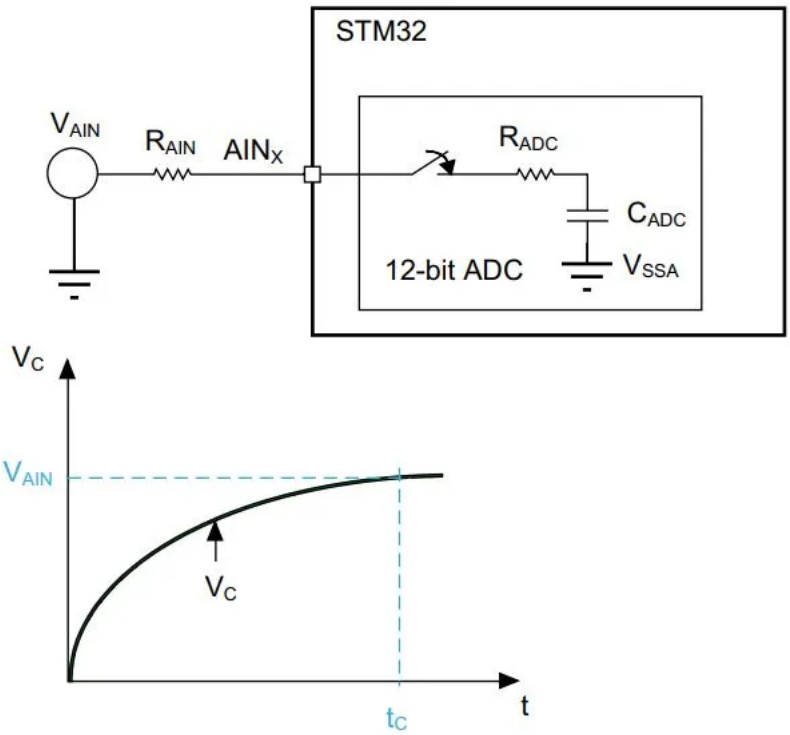
To obtain the maximum ADC conversion precision, it is very important that the ADC dynamic range matches the maximum amplitude of the signal to be converted. Let us assume that the signal to be converted varies between 0 V and 2.5 V and that V_{REF+} is equal to 3.3 V. The maximum signal value converted by the ADC is 3102 (2.5 V) as shown in the diagram down below. In this case, there are 993 unused transitions (4095 – 3102 = 993). This implies a loss in the converted



signal accuracy.

12.2.4 Analog Signal Source Impedance (Resistance)

The impedance of the analog signal source, or series resistance R_{AIN}, between the source and pin, causes a voltage drop across it because of the current flowing into the pin. The charging of the internal sampling capacitor C_{ADC} is controlled by switches with a resistance R_{ADC}. With the addition of source resistance (with R_{ADC}), the time required to fully charge the hold capacitor increases.



12.2.5 Analog Signal Source Capacitance & Parasitics

When converting analog signals, it is necessary to account for the capacitance at the source and the parasitic capacitance seen on the analog input pin. The source resistance and capacitance form an RC network. In addition, the ADC conversion results may not be accurate unless the external capacitor $(C_{AIN}+C_p)$ is fully charged to the level of the input voltage.

12.2.6 Injection Current Effect

A negative injection current on any analog pin (or a closely positioned digital input pin) may introduce leakage current into the ADC input. The worst case is the adjacent analog channel. A negative injection current is introduced when VAIN ¡ VSS, causing current to flow out from the I/O pin. Which can potentially

shift the voltage level on the pin and distort the measurement result.

12.2.7 IO Pins Cross-Talking

Switching the I/Os may induce some noise in the analog input of the ADC due to capacitive coupling between I/Os. Crosstalk may be introduced by PCB tracks that run close to each other or that cross each other.
Internally switching digital signals and IOs introduces high-frequency noise. Switching high sink I/Os may induce some voltage dips in the power supply caused by current surges. A digital track that crosses an analog input track on the PCB may affect the analog signal.

12.2.8 EMI-Induced Noise

Electromagnetic emissions from neighboring circuits may introduce high-frequency noise in an analog signal because the PCB tracks may act as an antenna. It's called electromagnetic interference (EMI) noise.

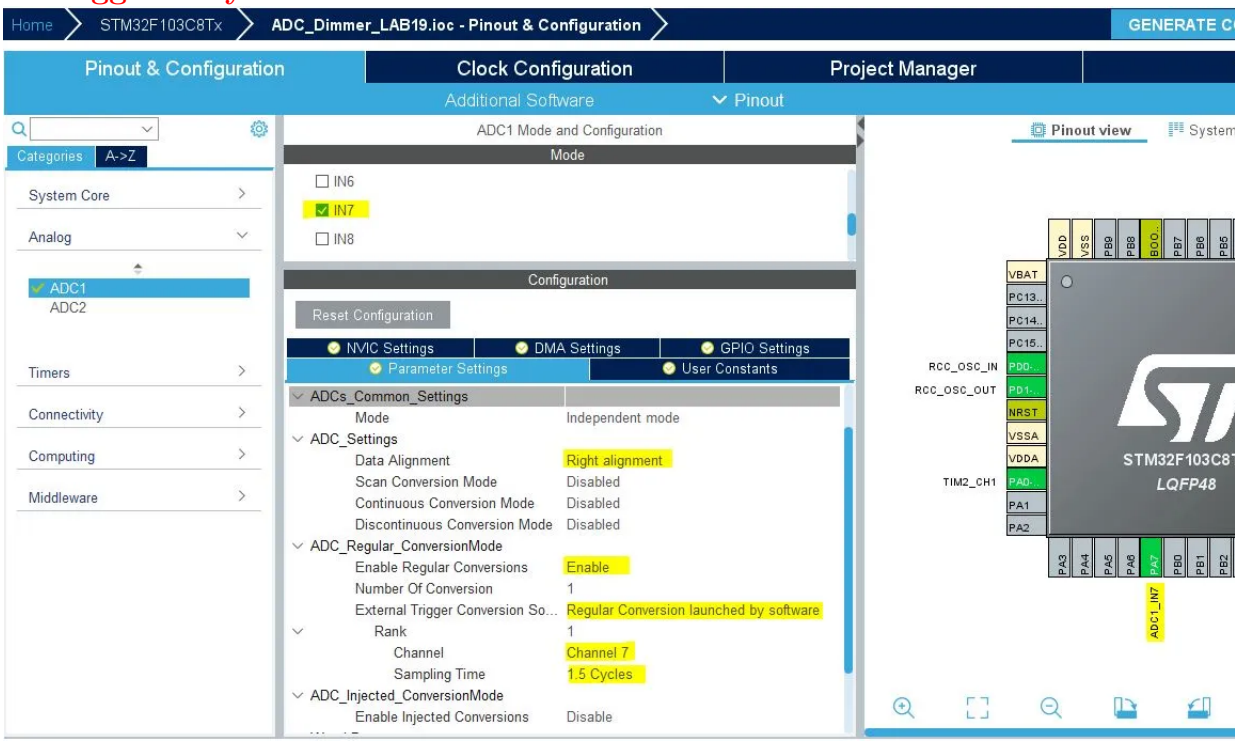
13. STM32 ADC Read Example – 3 Methods To Read ADC

We can read actually configure the ADC module to take samples (conversions) in 3 different ways.

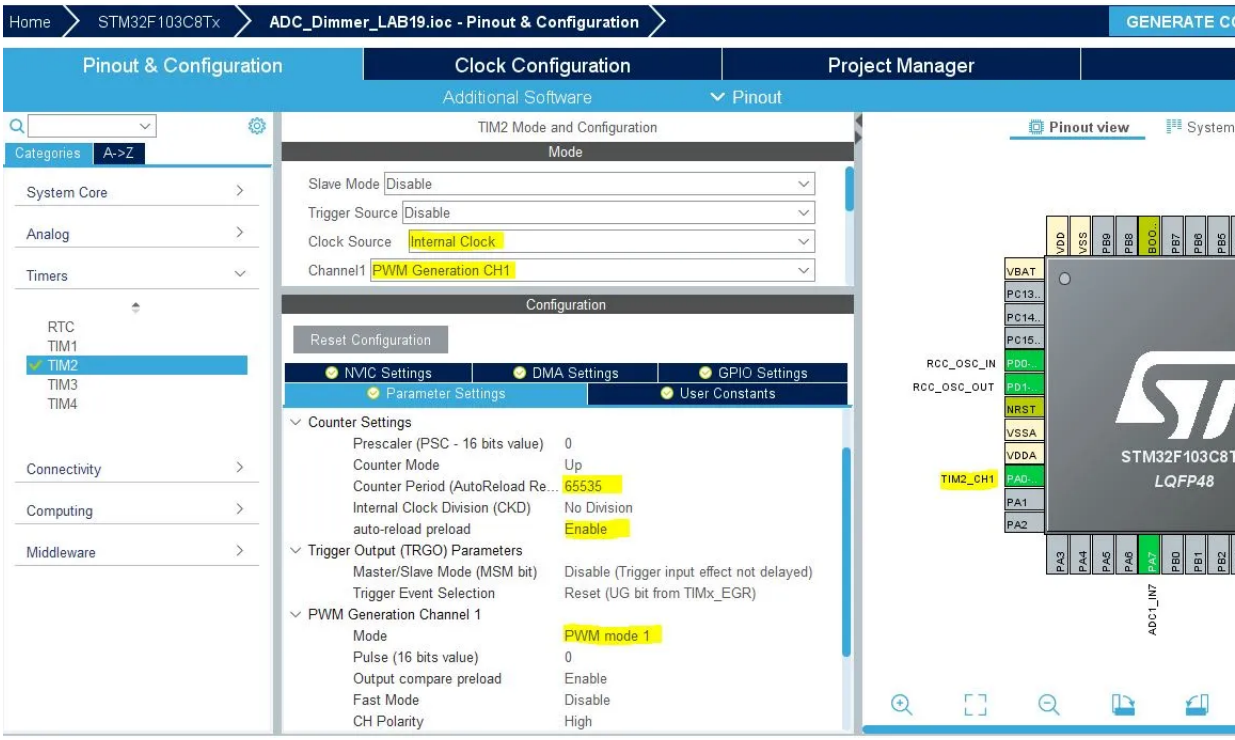
13.1 STM32 ADC Polling Example

In this LAB, our goal is to build a system that initializes the ADC with an analog input pin (channel 7). And also configure a timer module to operate in PWM mode with output on channel 1 pin (LED pin). Therefore, we can start an ADC conversion and map the result to the PWM duty cycle and repeat the whole process over and over again.
let's build this system step-by-step

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name**
- **Step3: Configure The ADC1 Peripheral, Enable Channel7 & Set it to be triggered by software**



- **Step4: Configure Timer2 To Operate In PWM Mode With Output On CH1**



- **Step5: Set The RCC External Clock Source**
- **Step6: Set The System Clock To Be 72MHz**
- **Step7: Name & Generate The Project Initialization Code For CubeIDE or The IDE You're Using**

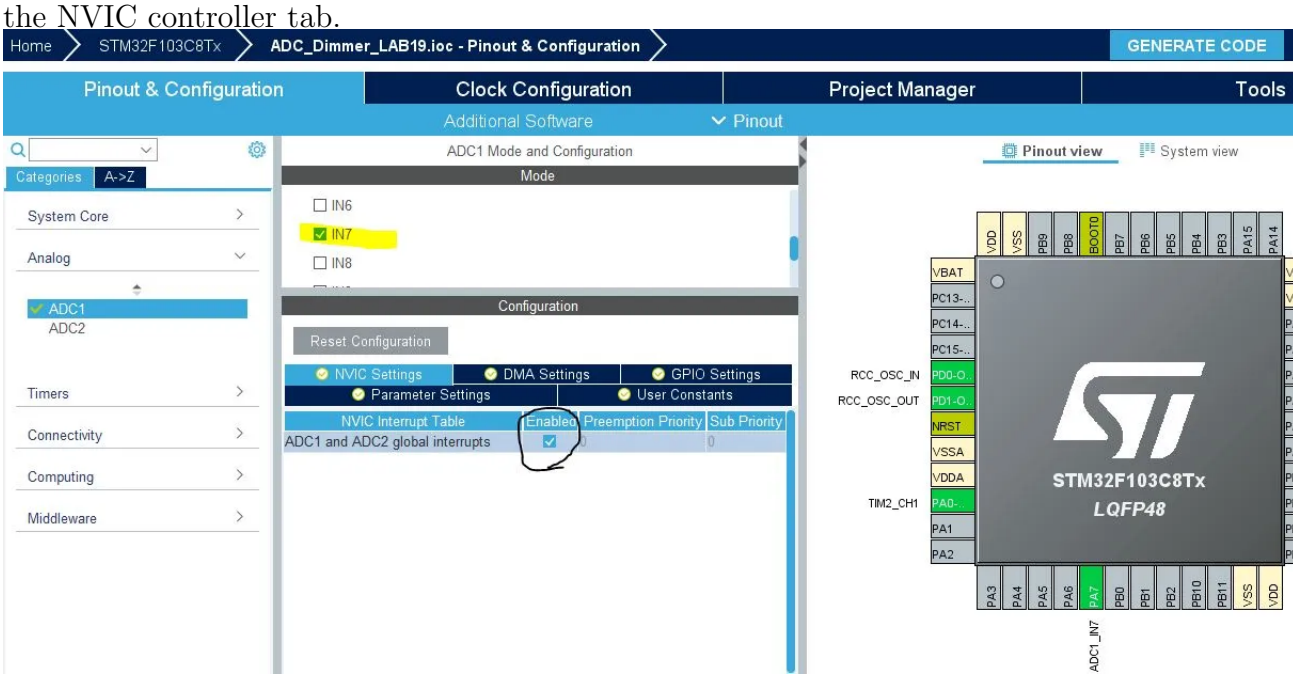
Here is The Application Code For This LAB (main.c)

```c
#include "main.h"

ADC_HandleTypeDef hadc1;
TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);

int main(void)
{
    uint16_t AD_RES = 0;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_TIM2_Init();
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
    // Calibrate The ADC On Power-Up For Better Accuracy
    HAL_ADCEx_Calibration_Start(&hadc1);

    while (1)
    {
       // Start ADC Conversion
        HAL_ADC_Start(&hadc1);
       // Poll ADC1 Perihperal & TimeOut = 1mSec
        HAL_ADC_PollForConversion(&hadc1, 1);
       // Read The ADC Conversion Result & Map It To PWM DutyCycle
        AD_RES = HAL_ADC_GetValue(&hadc1);
        TIM2->CCR1 = (AD_RES<<4);
        HAL_Delay(1);
    }
}
```

13.2 STM32 ADC Interrupt Example

The Exact Same Steps As The Previous Example Except For Step 3. The ADC Configuration Will Be As Follows:

All ADC settings will remain the same but we'll need to enable the interrupt from the NVIC controller tab.

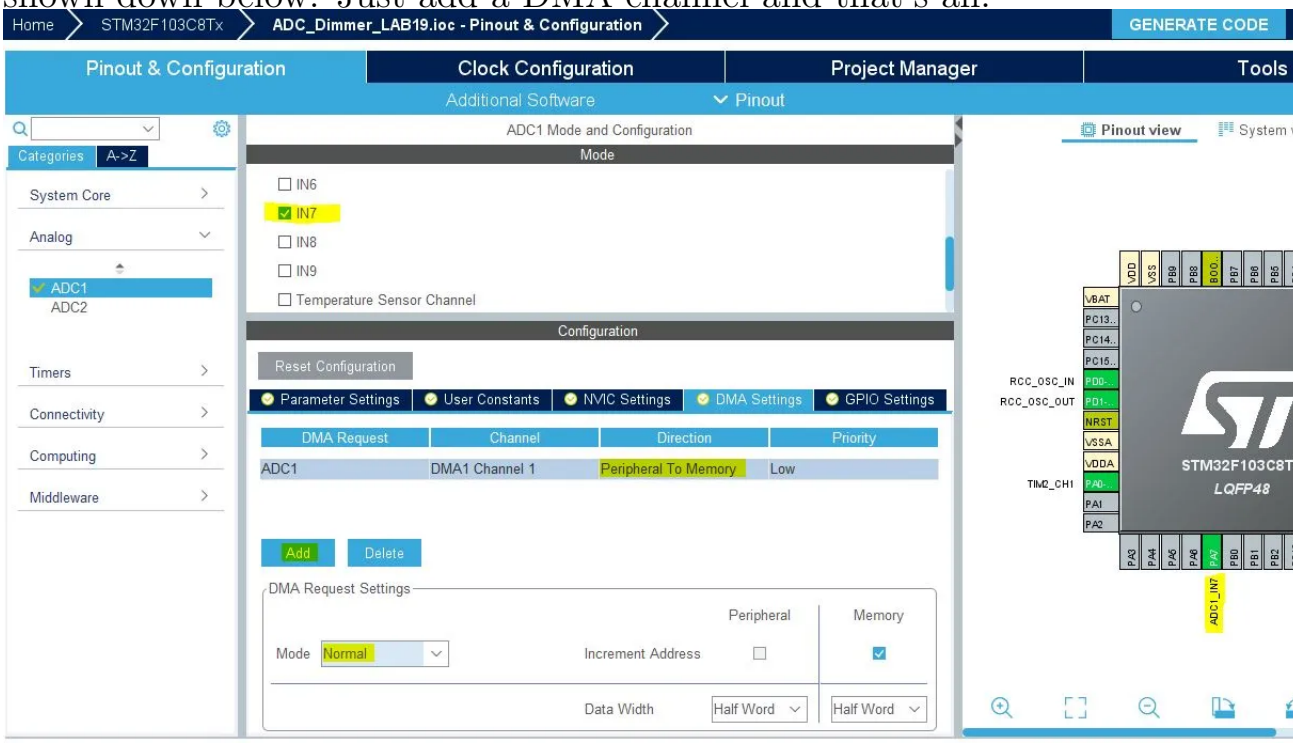


Here Is The Application Code Using The Interrupt Method

```c
#include "main.h"

uint16_t AD_RES = 0;

ADC_HandleTypeDef hadc1;
TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_TIM2_Init();
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
    // Calibrate The ADC On Power-Up For Better Accuracy
    HAL_ADCEx_Calibration_Start(&hadc1);

    while (1)
    {
        // Start ADC Conversion
        HAL_ADC_Start_IT(&hadc1);
        // Update The PWM Duty Cycle With Latest ADC Conversion Result
        TIM2->CCR1 = (AD_RES<<4);
        HAL_Delay(1);
    }
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    // Read & Update The ADC Result
    AD_RES = HAL_ADC_GetValue(&hadc1);
}
```

13.3 STM32 ADC DMA Example

Also The Exact Same Steps As The First Example Except For Step 3. The ADC Configuration Will Be As Follows:
Everything in ADC configurations will be as default in normal mode. However, this time the ADC interrupts are not activated and the DMA is configured instead and DMA interrupt is enabled by default in the NVIC controller tab. The DMA configurations for the ADC will be as shown down below. Just add a DMA channel and that's all.



Here Is The Application Code Using The DMA Method

```c
#include "main.h"

uint16_t AD_RES = 0;

ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
```



```c
static void MX_TIM2_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_ADC1_Init();
    MX_TIM2_Init();
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
    // Calibrate The ADC On Power-Up For Better Accuracy
    HAL_ADCEx_Calibration_Start(&hadc1);

    while (1)
    {
        // Start ADC Conversion
        // Pass (The ADC Instance, Result Buffer Address, Buffer Length)
        HAL_ADC_Start_DMA(&hadc1, &AD_RES, 1);
        HAL_Delay(1);
    }
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    // Conversion Complete & DMA Transfer Complete As Well
    // So The AD_RES Is Now Updated & Let's Move IT To The PWM CCR1
    // Update The PWM Duty Cycle With Latest ADC Conversion Result
    TIM2->CCR1 = (AD_RES<<4);
}
```

XXIII STM32 DAC Tutorial

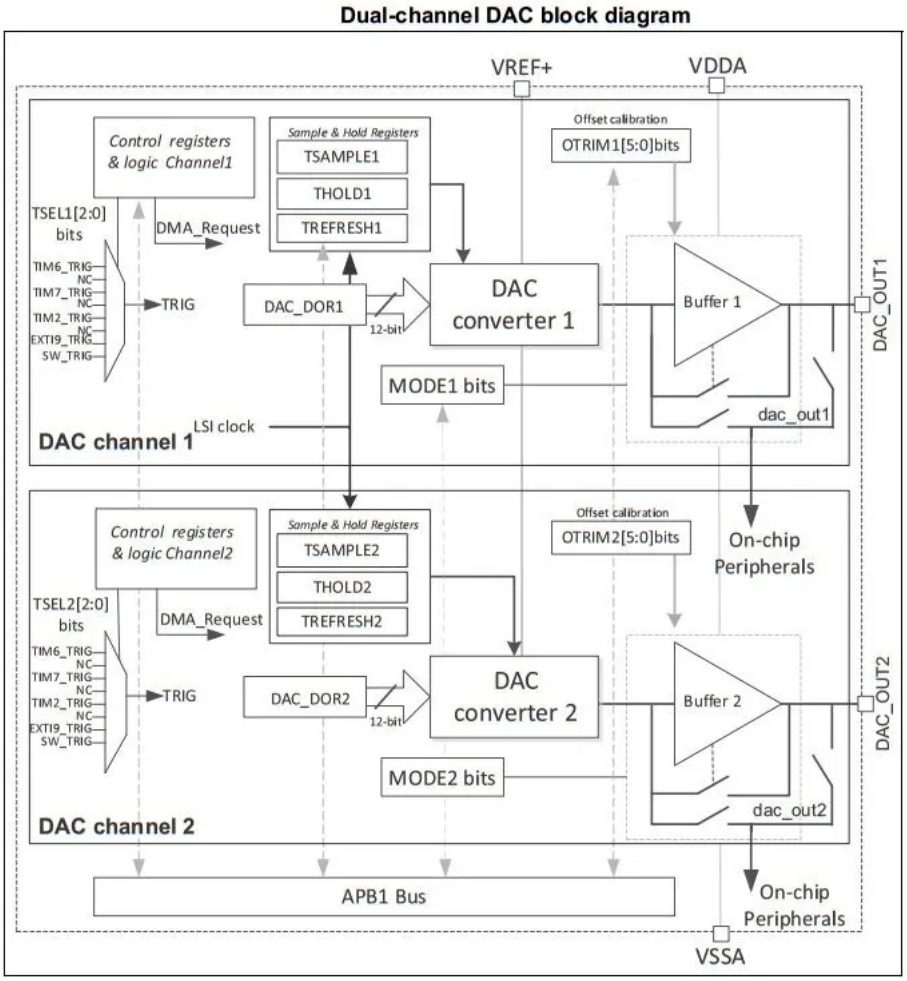
1. **Digital-To-Analog Converters (DAC) Preface**
   A DAC (Digital-To-Analog) converter is an electronic circuit that takes in a digital number or value as an input and converts it into an analog voltage, the voltage level that corresponds to the binary number in the DAC output register. The DAC output voltage changes whenever you change the DAC output register value, and this sampling process can be triggered in multiple ways as we'll see hereafter.

2. **STM32 DAC Brief**
   DAC Features

   - One DAC interface, maximum of two output channels
   - Left or right data alignment in 12-bit mode
   - Synchronized update capability
   - Noise-wave and Triangular-wave generation
   - Dual DAC channel for independent or simultaneous conversions
   - DMA capability for each channel including DMA underrun error detection
   - External triggers for conversion
   - DAC output channel buffered/unbuffered modes
   - Buffer offset calibration
   - Each DAC output can be disconnected from the DAC_OUTx output pin
   - DAC output connection to on-chip peripherals
   - Sample and Hold mode for low power operation in Stop mode

- Input voltage reference, $V_{REF}+$

3. **STM32 DAC Functional Description**
   STM32 DAC Block Diagram



The DAC includes up to two separate output channels. Each output channel can be connected to on-chip peripherals such as comparator, operational amplifier, and ADC (if available). In this case, the DAC output channel can be disconnected from the DAC_OUTx output pin and the corresponding GPIO can be used for another purpose. The DAC output can be buffered or not.

STM32 DAC Data Format

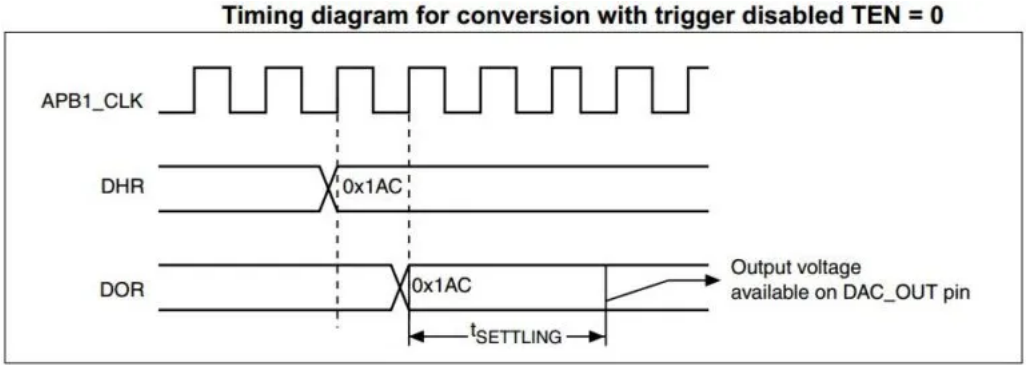
Single DAC channel

There are three possibilities:

- 8-bit right alignment: the software has to load data into the DAC_DHR8Rx[7:0] bits (stored into the DHRx[11:4] bits)
- 12-bit left alignment: the software has to load data into the DAC_DHR12Lx [15:4] bits (stored into the DHRx[11:0] bits)
- 12-bit right alignment: the software has to load data into the DAC_DHR12Rx [11:0] bits (stored into the DHRx[11:0] bits)

STM32 DAC Conversion

The DAC_DORx cannot be written directly and any data transfer to the DAC channelx must be performed by loading the DAC_DHRx register (write operation to DAC_DHR8Rx, DAC_DHR12Lx, DAC_DHR12Rx, DAC_DHR8RD, DAC_DHR12RD or DAC_DHR12LD).



STM32 DAC

Output Voltage Equation (Formula)
Digital inputs are converted to output voltages on a linear conversion between 0 and VREF+. The analog output voltages on each DAC channel pin are determined by the following equation: DACoutput = (V_{REF}+) x (DOR/4096)

DAC Noise Generation
In order to generate a variable-amplitude pseudo-noise, an LFSR (linear feedback shift register) is available. DAC noise generation is selected by setting WAVEx[1:0] to 01". The preloaded value in LFSR is 0xAAA. This register is updated three APB1 clock cycles after each trigger event, following a specific calculation algorithm.

4. **STM32 DAC Channel Modes of Operation**
Each DAC channel can be configured in Normal mode or Sample and Hold mode. The output buffer can be enabled to allow a high drive capability. Before enabling the output buffer, the voltage offset needs to be calibrated. This calibration is performed at the factory (loaded after reset) and can be adjusted by software during application operation.

Normal Mode
In Normal mode, there are four combinations, by changing the buffer state, and by changing the DAC_OUTx pin interconnections.
To enable the output buffer, the MODEx[2:0] bits in DAC_MCR register should be:

- 000: DAC is connected to the external pin
- 001: DAC is connected to the external pin and to on-chip peripherals

To disable the output buffer, the MODEx[2:0] bits in DAC_MCR register should be:

- 010: DAC is connected to the external pin
- 011: DAC is connected to on-chip peripherals

Sample And Hold Mode
In sample and Hold mode, the DAC core converts data on a triggered conversion, then, holds the converted voltage on a capacitor. When not converting, the DAC cores and buffer are completely turned off between samples and the DAC output is tri-stated, therefore reducing the overall power consumption. A new stabilization period, which value depends on the buffer state, is required before each new conversion.

5. **DAC Conversion On External Triggers**
If the software trigger is selected, the conversion starts once the SWTRIG bit is set. SWTRIG is reset by hardware once the DAC_DORx register has been loaded with the DAC_DHRx register contents.

DAC trigger selection

Source	Type	TSELx[2:0]
TIM6_TRGO	Internal signal from on-chip timers	000
TIM8_TRGO	Internal signal from on-chip timers	001
TIM7_TRGO(1)	Internal signal from on-chip timers	010
TIM5_TRGO	Internal signal from on-chip timers	011
TIM2_TRGO	Internal signal from on-chip timers	100
TIM4_TRGO	Internal signal from on-chip timers	101
EXTI9	External pin	110
SWTRIG	Software control bit	111

6. **STM32 DAC Calibration**
   The transfer function for an N-bit digital-to-analog converter (DAC) is:
   V_{OUT}=(($D/2^{N-1}$ x G x V_{ref}) + V_{os}
   Where VOUT is the analog output, D is the digital input, G is the gain, Vref is the nominal fullscale voltage, and Vos is the offset voltage. For an ideal DAC channel, G = 1 and Vos = 0.
   f applied in other modes when the buffer is off, it has no effect. During the calibration:

   - The buffer output will be disconnected from the pin internal/external connections and put in tristate mode (HiZ),
   - The buffer will act as a comparator, to sense the middle-code value 0x800 and compare it to VREF+/2 signal through an internal bridge, then toggle its output signal to 0 or 1 depending on the comparison result (CAL_FLAGx bit)

   Two calibration techniques are provided:

   - Factory trimming: (always enabled) The DAC buffer offset is factory trimmed. The default value of OTRIMx[4:0] bits in the DAC_CCR register is the factory trimming value and it is loaded once DAC digital interface is reset.
   - User trimming: The user trimming can be done when the operating conditions differ from nominal factory trimming conditions and in particular when V_{DD}/V_{DDA} voltage, temperature, VREF+ values change and can be done at any point during application by software.

7. **STM32 DAC Resolution, Reference, Formulas**
   STM32 DAC Resolution
   The STM32 DAC has a resolution of 12-Bit that can be configured to be 8-bit as well. Depending on your application you can choose between the two options and obviously the 12-Bit resolution is great for audio applications.
   DAC Reference Voltage
   The DAC reference voltage can be configured to be from an external pin or provided internally using the internal VREFBUF module. The reference voltage you'll choose for the DAC operation will define the maximum allowable voltage swing as well as the output voltage resolution.
   STM32 DAC Formulas
   DAC Output Voltage
   VOUT = DOR x (Reference Voltage / 4096)
   Where Reference Voltage = (VREF+) – (VREF-)

8. **DAC & DMA**
   n dual-mode, if both DMAENx bits are set, two DMA requests are generated. If only one DMA request is needed, only the corresponding DMAENx bit should be set. In this way, the application can manage both DAC channels in dual mode by using one DMA request and a unique DMA channel.
   DAC DMA Underrun
   The DAC DMA request is not queued so that if a second external trigger arrives before the acknowledgment for the first external trigger is received (first request), then no new request is issued and the DMA channelx underrun flag DMAUDRx in the DAC_SR register is set, reporting the error condition. The DAC channelx continues to convert old data.

9. **STM32 DAC Interrupts**
   Only one interrupt signal can be fired on a DAC DMA Underrun condition.

DAC interrupts

Interrupt event	Event flag	Enable control bit
DMA underrun	DMAUDRx	DMAUDRIEx

10. **STM32 DAC Buffered-Output Vs Non-Buffered**
    Every electronic circuit has an input impedance that represents a load resistance for whatever stage that precedes this circuit. For a DAC, the output voltage will be loaded if you connect the DAC_OUT pin to whatever circuit maybe to amplify it or filter it or whatever. The existence of output load resistance will draw a little bit of current that results in a voltage shift (drop) on the DAC output.
    Therefore, we always need to buffer the DAC output if we're going to hook it up to another electronic circuit. An output buffer is an op-amp that works in a voltage follower configuration which will source current to the load resistance at the output and do whatever it takes to make sure that the DAC_OUT voltage is what it should be (identical to its input). Down below is an example comparison between a buffered DAC and non-buffered DAC output. Both of which are being tested in no-load and loaded configurations. And both of which are programmed to produce 1.5v on the DAC_OUT pin.

11. **STM32 DAC Speed (Sampling Rate)**
    Output Buffer Effect & Output Impedance, Capacitance
    When the output buffer is enabled on the DAC output, the speed is specified by the output buffer performance. This number is indicated in the $T_{settling}$ or Update rate in the product datasheet.
    When the output buffer is disabled, the output signal speed simply follows the RC constant, which is determined by the DAC output impedance R_{DAC} (= 2 x Ra) and the capacitive load on the DAC_OUT pad.
    Solving for T gives T = CR x N x ln(2) = 0.693 x RC x N = 1.8 µs, hence, in this configuration, the conversion time cannot be smaller than 1.8 µs (equivalent to a frequency of 555 kHz). This analysis does not include any effect of the switching speed of the DAC itself and its transient. When using high speed, these factors cannot be ignored, they degrade the performance.
    Digital Data Update Rate
    The STM32 DAC output data need to be written to the DAC holding register (DHR), then the data is moved to the DAC output register (DOR) for the conversion. Generally, the data are saved in a RAM, and the CPU is in charge of transferring the data from RAM to DAC.
    The transfer speed from memory to the DAC is limited by several factors, among them:

- The clock cycle of the APB or of the AHB (DAC clock)
- The DMA transfer cycle from memory to the DAC (includes the AHB to APB bridge)
- The trigger mechanism itself

Maximum sampling time for different STM32 microcontrollers

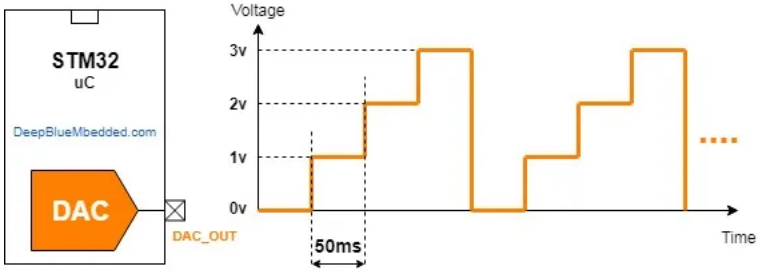
Product	Max bus speed	DAC max sampling rate
STM32F0 Series	48 MHz	4.8 Msps
STM32F100xx	24 MHz	2.4 Msps
STM32F101xx STM32F103xx STM32F105xx STM32F107xx	36 MHz	4.5 Msps
STM32F2 Series	30 MHz	7.5 Msps
STM32F3 Series	36 MHz	4.5 Msps
STM32F40x STM32F41x	42 MHz	10.5 Msps
STM32F42x	45 MHz	11.25 Msps
STM32F7 Series	54 MHz	13.5 Msps
STM32G0 Series	64 MHz	8.0 Msps
STM32G4 Series	150 MHz	16.67 Msps 30 Msps (DMA double data mode)
STM32H7 Series	100 MHz	18.18 Msps
STM32L0 Series	32 MHz	4.0 Msps
STM32L1 Series	32 MHz	3.2 Msps
STM32L4 Series	80 MHz	10 Msps
STM32L4+ Series	120 MHz	12 Msps
STM32L5 Series	110 MHz	11 Msps

12. **DAC Example Applications**
The DAC example applications will include the following:

- Analog Voltage Output
- Analog Waveform Generation (Sine, Sawtooth, Triangular)
- Audio Production

13. **STM32 DAC Example – LAB22**

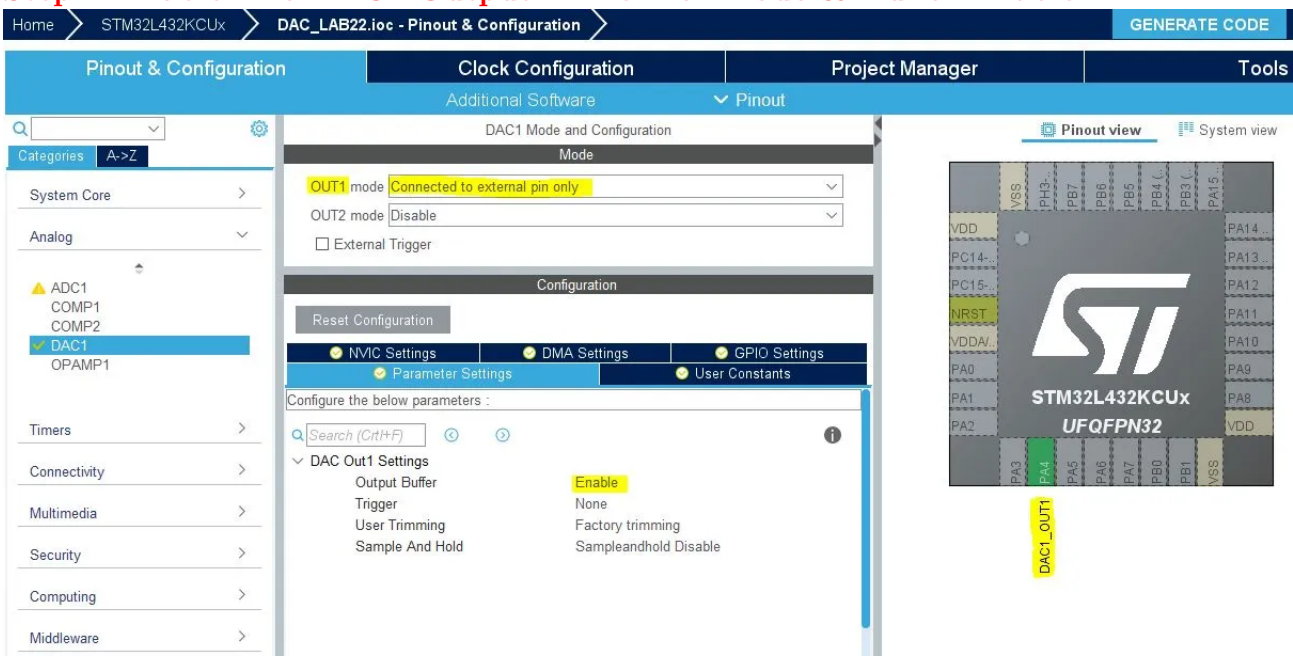
- Set up a new project as usual with system clock @ 80MHz
- Set up the DAC1 peripheral to work in normal mode with output buffer enabled
- Output the following voltage pattern 0v, 1v, 2v, 3v, every 50ms and repeat



For 1v output, the DOR is calculated from
Vout = DOR x (V_{REF} / 4096)

1v = DOR x (3.3 / 4096)
=> DOR = 1241

- **Step1: Open CubeMX & Create New Project**
- **Step2: Choose The Target MCU & Double-Click Its Name STM32L432KC**
- **Step3: Set The System Clock To Be 80MHz**
- **Step4: Enable The DAC1 Output In Normal Mode & Buffer Enable**



- **Step5: Generate The Initialization Code & Open The Project In Your IDE**

Here is The Application Code For This LAB (main.c)

```c
#include "main.h"

DAC_HandleTypeDef hdac1;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DAC1_Init(void);

int main(void)
{
    uint32_t DAC_OUT[4] = {0, 1241, 2482, 3723};
    uint8_t i = 0;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_DAC1_Init();

    while (1)
    {
        DAC1->DHR12R1 = DAC_OUT[i++];
        if(i == 4)
        {
            i = 0;
        }
        HAL_Delay(50);
    }
}
```

XXIV STM32 DAC Waveform Generation (Sine, Sawtooth, Sinc, etc)

1. **STM32 DAC For Waveform Generation**
   This process can be divided into two major steps:
   Step1: Generating The Lookup Table
   The lookup table is an array of unsigned integer values that represents the sample points of a specific waveform for a complete cycle (from 0 to 2π). The length of the lookup table is denoted as Ns or the number of sample points per complete cycle. The more sample points per cycle, the better the output waveform.
   Step2: Moving The Data To The DAC Output
   The output waveform's frequency is controlled by changing the delay period (x).
   while(1)
   {
   DAC_Out(Wave_LUT[i++]);
   DWT_Delay_us(x);
   }

2. **Generating The Waveform Lookup Table**
   you can use a **MATLAB** script to generate and print the waveform sample points. You can configure the script below to set the required sample points number (Ns) and give it an offset from 0 in case you want that. And also the resolution of the DAC in your system (in bits).
   Here is the MATLAB script that generates the lookup table for you.

```
clear; clc;   % Clear The Previous Points
Ns     = 128; % Set The Number of Sample Points
RES    = 12;  % Set The DAC Resolution
OFFSET = 0;   % Set An Offset Value For The DAC Output
%------------[ Calculate The Sample Points ]-------------
T = 0:((2*pi/(Ns-1))):(2*pi);
Y = sin(T);
Y = Y + 1;
Y = Y*((2^RES-1)-2*OFFSET)/(2+OFFSET);
Y = round(Y);
plot(T, Y);
grid
%--------------[ Print The Sample Points ]---------------
fprintf('%d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, \n', Y);

%-------------------[ Other Examples To Try ]-------------------
%  Y = diric(T, 13);     % Periodic Sinc
%  Y = sawtooth(T)       % Sawtooth
%  Y = sawtooth(T, 0.5); % Triangular
```

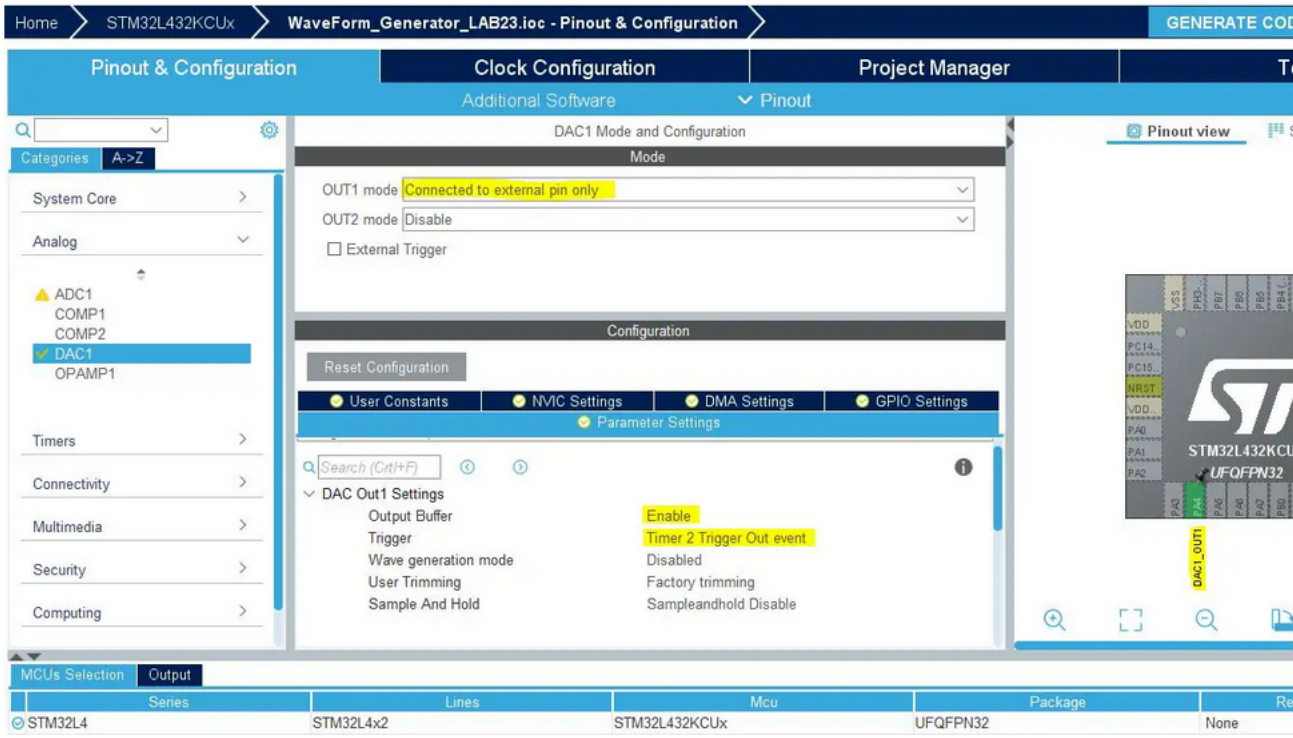
3. **DAC DMA Sine Waveform Generation**
   Here are the formulas to be used

$$TriggerFrequency = \frac{F_{CLK}}{(PSC+1)(ARR+1)} // OutWaveFrequency = \frac{TriggerFrequency}{Ns} \tag{1}$$

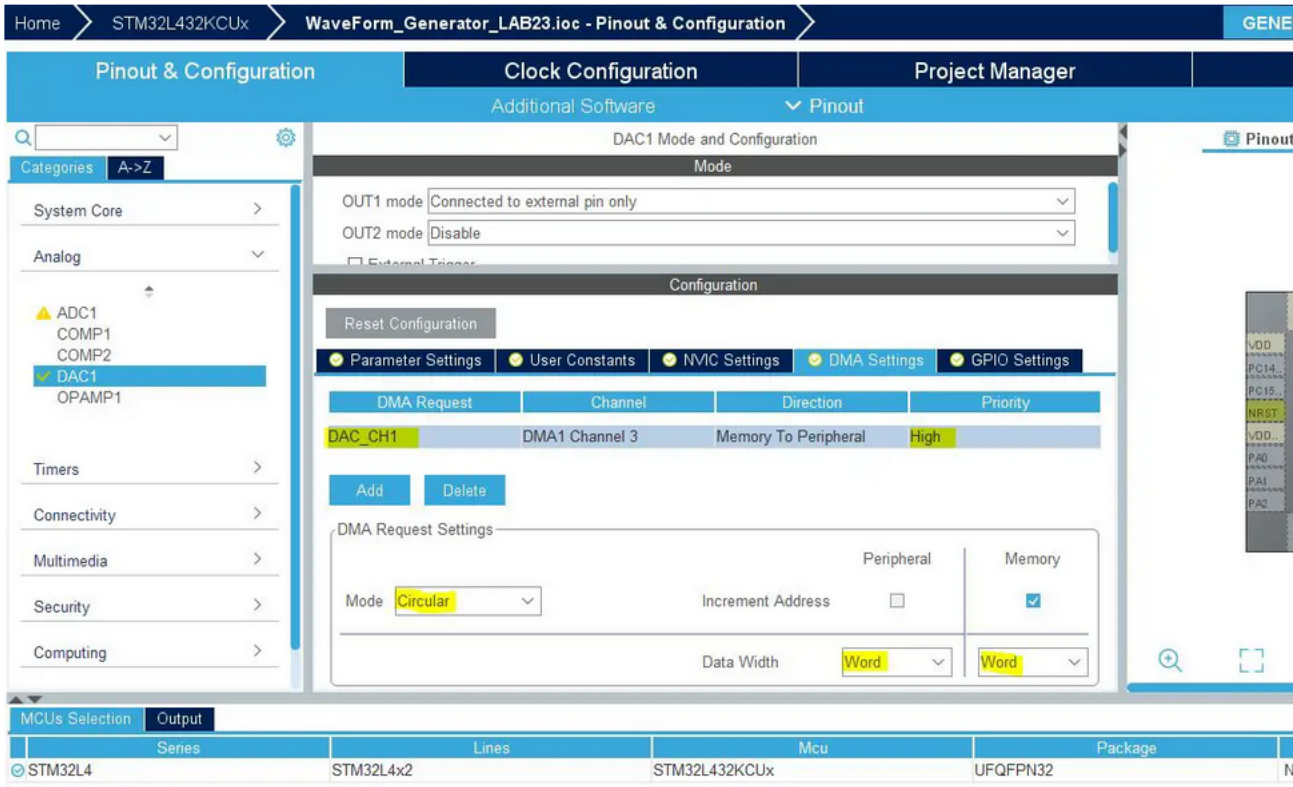
4. **STM32 DAC WaveForm Generator – LAB23**

   - **Step1: Open CubeMX & Create New Project**

- **Step2: Choose The Target MCU & Double-Click Its Nam**
- **Step3: Go To The Clock Configuration**
- **Step4: Set The System Clock To Be 80MHz**
- **Step5: Enable The DAC1 Output In Normal Mode & Buffer Enable**

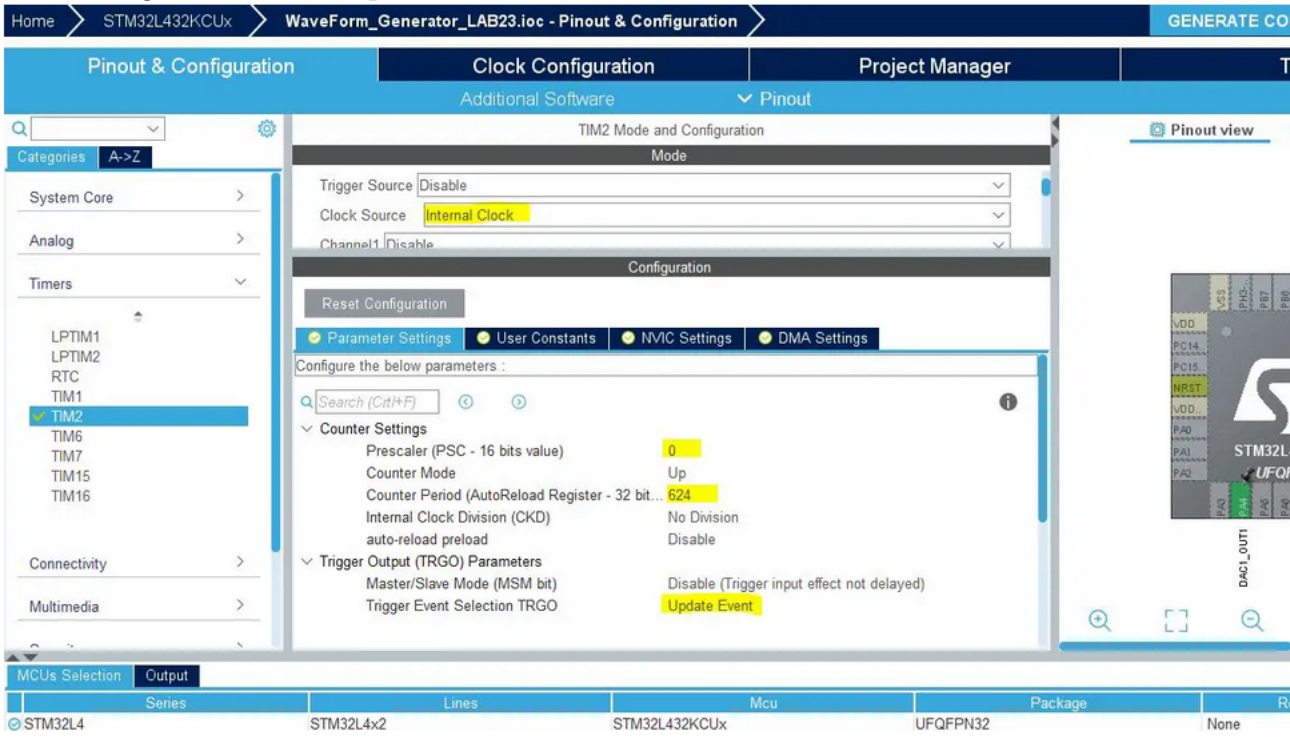


- **Step6: Enable The DAC1 DMA Channel & Configure It As Shown Below**



- **Step7: Now, Configure Timer2 Peripheral As Shown**
  Do you remember the example of the calculation shown earlier in this article? Where we want to generate a 1kHz sine wave with a system clock of 80MHz and lookup table with 128 entries.

The PCS was chosen to be 0, and the ARR was calculated to be 624. That's what we're doing here in this step.



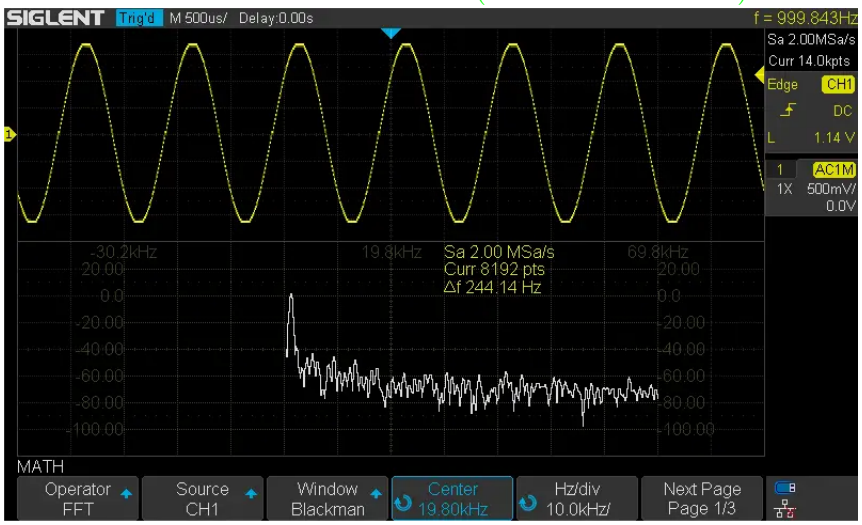
- **Step8: Generate The Initialization Code & Open The Project In Your IDE**

Here is The Application Code For This LAB (main.c)

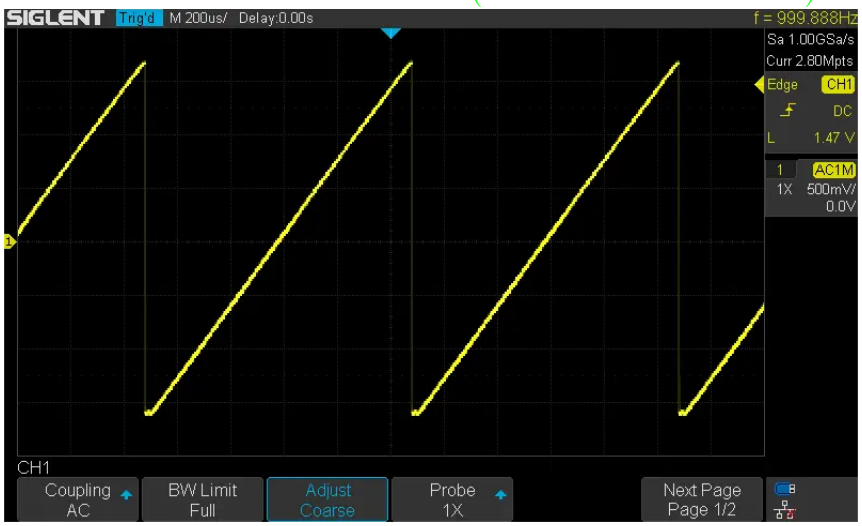
```
#include "main.h"

#define NS  128

uint32_t Wave_LUT[NS] = {
    2048, 2149, 2250, 2350, 2450, 2549, 2646, 2742, 2837, 2929, 3020, 3108, 3193, 3275, 3355,
    3431, 3504, 3574, 3639, 3701, 3759, 3812, 3861, 3906, 3946, 3982, 4013, 4039, 4060, 4076,
    4087, 4094, 4095, 4091, 4082, 4069, 4050, 4026, 3998, 3965, 3927, 3884, 3837, 3786, 3730,
    3671, 3607, 3539, 3468, 3394, 3316, 3235, 3151, 3064, 2975, 2883, 2790, 2695, 2598, 2500,
    2400, 2300, 2199, 2098, 1997, 1896, 1795, 1695, 1595, 1497, 1400, 1305, 1212, 1120, 1031,
    944, 860, 779, 701, 627, 556, 488, 424, 365, 309, 258, 211, 168, 130, 97,
    69, 45, 26, 13, 4, 0, 1, 8, 19, 35, 56, 82, 113, 149, 189,
    234, 283, 336, 394, 456, 521, 591, 664, 740, 820, 902, 987, 1075, 1166, 1258,
    1353, 1449, 1546, 1645, 1745, 1845, 1946, 2047
};

DAC_HandleTypeDef hdac1;
DMA_HandleTypeDef hdma_dac_ch1;
TIM_HandleTypeDef htim2;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_DAC1_Init();
    MX_TIM2_Init();
    HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)Wave_LUT, 128, DAC_ALIGN_12B_R);
    HAL_TIM_Base_Start(&htim2);

    while (1)
    {

    }
}
```

The Result For This LAB Testing On My DSO

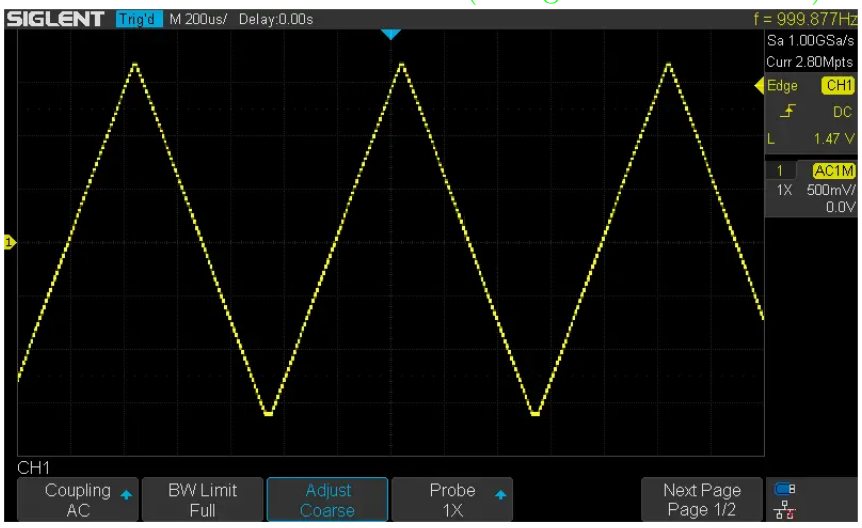
Here is the result for the first test (sine wave @ 1KHz)



Here is the result for the 2nd test (sawtooth wave @ 1KHz)



Here is the result for the 3rd test (triangular wave @ 1KHz)



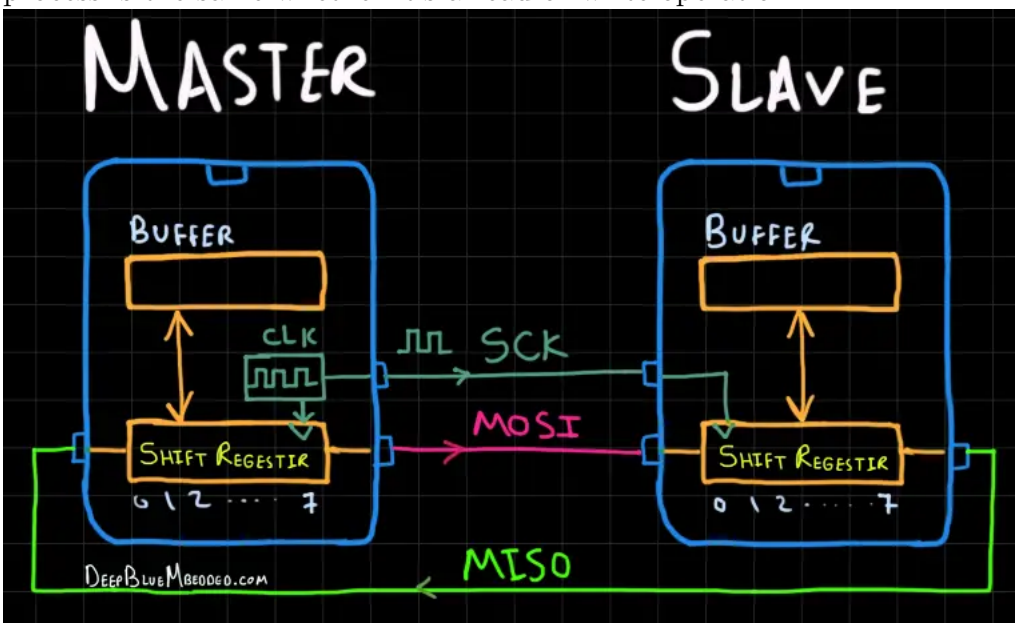
XXV SPI Tutorial

1. **Introduction To SPI Communication**
   SPI is a synchronous communication protocol that transmits and receives information si-

multaneously with high data transfer rates and is designed for board-level communication over short distances.

1.1 SPI Pin Conventions Connections

The master initiates communication by generating a serial clock signal to shift a data frame out, at the same time serial data is being shifted-in to the master. This process is the same whether it's a read or write operation.



- MOSI:Master output slave input (D_{OUT} From Master).
- MISO:Master input slave output(D_{OUT} From Slave).
- SCLK:Serial Clock,generated by the master and goes to the slave.
- SS:Slave Select. Generated by the master to control which slave to talk to. It's usually an active-low signal.

1.2 SPI Modes of Operation

- Devices on the SPI bus can operate in either mode of the following: Master or Slave.
- There must be at least one master who initiates the serial communication process (For Reading/Writing).
- On the other hand, there can be single or multiple devices operating in slave mode.
- The master device can select which slave to talk to by setting the SS (slave select) pin to logic low.
- If a single slave is being addressed, you can tie the SS pin of this slave device to logic low without the need to control this line by the master.

1.3 SPI Clock Configurations

The SPI clock has two more parameters to control which are the Clock Phase (CPHA) and the Clock Polarity (CPOL).

- The clock phase determines the phase at which the data latching occurs at each bit transfer whether it's the leading edge or the trailing edge.
- The clock polarity determines the IDLE state of the clock line whether it's a HIGH or LOW.

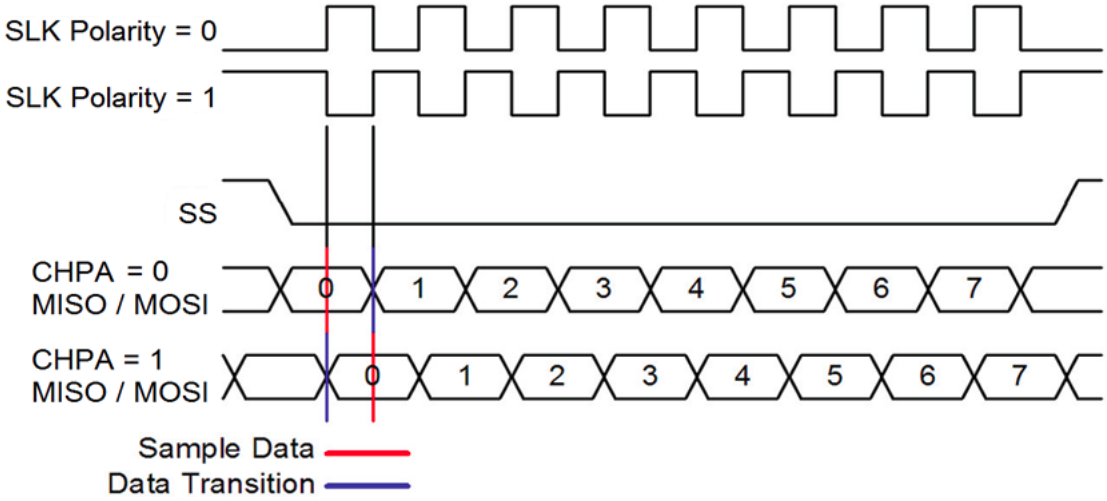


Figure 2: Clock Polarity and Phase Timing Diagram

2. SPI Hardware In STM32

2.1 STM32 SPI Hardware Overview

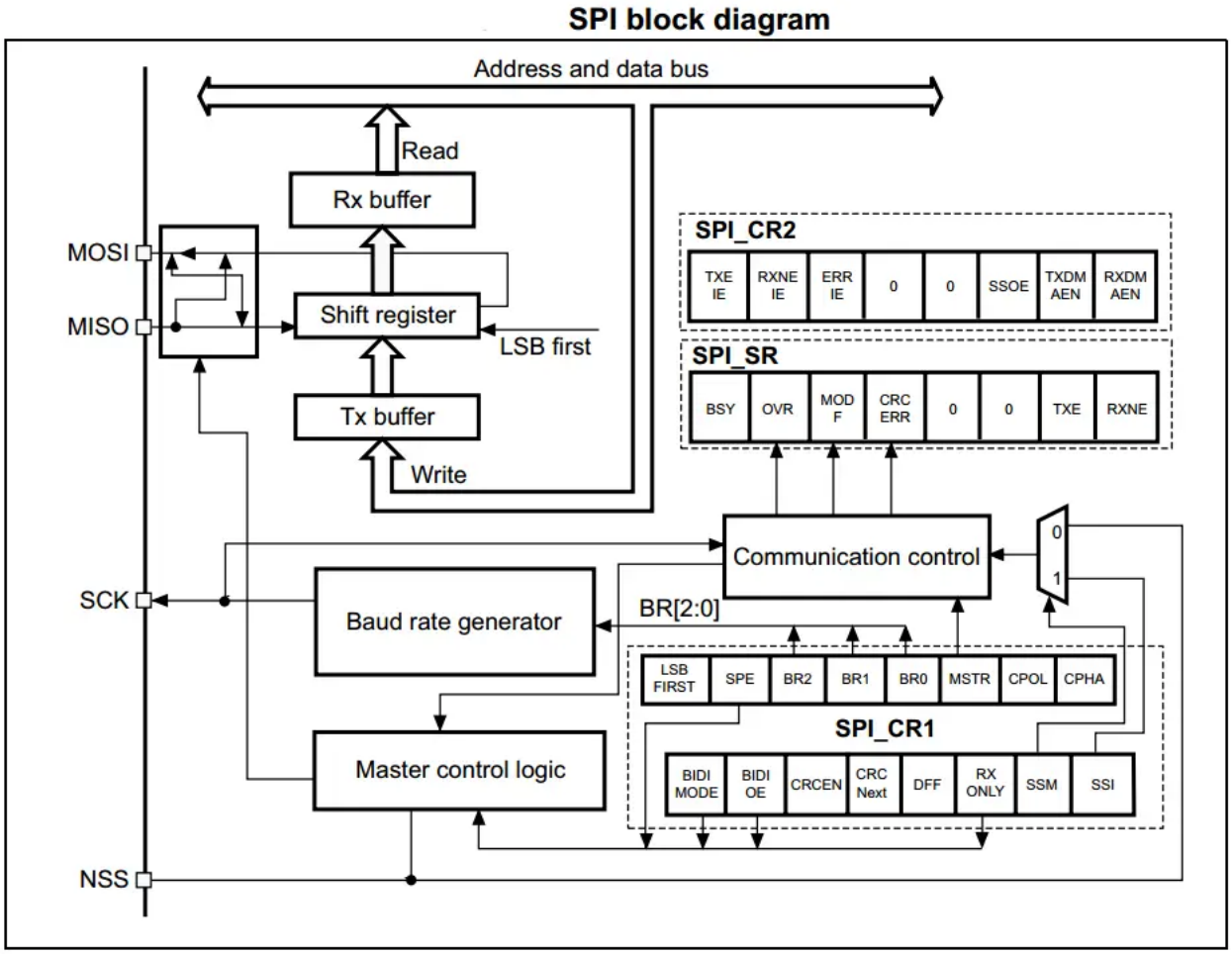
The serial peripheral interface (SPI) allows half/ full-duplex, synchronous, serial communication with external devices. The interface can be configured as the master and in this case, it provides the communication clock (SCK) to the external slave device. The interface is also capable of operating in a multi-master configuration.

2.2 STM32 SPI Main Features

- Full-duplex synchronous transfers on three lines.
- Simplex synchronous transfers on two lines with or without a bidirectional data line
- 8- or 16-bit transfer frame format selection
- Master or slave operation
- Multimaster mode capability
- 8 master mode baud rate prescalers (fPCLK/2 max.)
- Slave mode frequency (fPCLK/2 max)
- NSS pin management by hardware or software for both master and slave: dynamic change of master/slave operations
- Programmable clock polarity and phase
- Programmable data order with MSB-first or LSB-first shifting
- Dedicated transmission and reception flags with interrupt capability
- SPI bus busy status flag
- Hardware CRC feature for reliable communication: [CRC value can be transmitted as the last byte in Tx mode – Automatic CRC error checking for last received byte]
- Master mode fault, overrun, and CRC error flags with interrupt capability
- 1-byte transmission and reception buffer with DMA capability: Tx and Rx requests

3. STM32 SPI Hardware Functionalities

3.1 STM32 SPI Block Diagram



3.2 STM32 SPI Data Frame Format

- Data can be shifted out either MSB-first or LSB-first depending on the value of the LSBFIRST bit in the SPI_CR1 Register.
- Each data frame is 8 or 16 bits long depending on the size of the data programmed using the DFF bit in the SPI_CR1 register. The selected data frame format is applicable for transmission and/or reception.

3.3 STM32 SPI In Slave Mode

In the slave configuration, the serial clock is received on the SCK pin from the master device. The baud rate value set in the BR[2:0] bits in the SPI_CR1 register, does not affect the data transfer rate. In this configuration, the MOSI pin is a data input pin and the MISO pin is a data output.

Note: that you need to have the SPI slave device running and configured before the master starts to send anything just to avoid any possible data corruption at the beginning of the first communication session.

- **Transmit Sequence in Slave Mode**
  The data byte is parallel-loaded into the Tx buffer during a write cycle. The transmit sequence begins when the slave device receives the clock signal and the most significant bit of the data on its MOSI pin. The remaining bits (the 7 bits in 8-bit data frame format, and the 15 bits in 16-bit data frame format) are loaded into the shift-register. The TXE flag in the SPI_SR register is set on the transfer of data from the Tx Buffer to the shift register and an interrupt is generated if the TXEIE bit in the SPI_CR2 register is set.

- **Receive Sequence in Slave Mode**
  _For the receiver, when data transfer is complete: The Data in the shift register is transferred to Rx Buffer and the RXNE flag (SPI_SR register) is set. An Interrupt is generated if the RXNEIE bit is set in the SPI_CR2 register.
  _After the last sampling clock edge, the RXNE bit is set, a copy of the data byte received in the shift register is moved to the Rx buffer. When the SPI_DR register is read, the SPI peripheral returns this buffered value. Clearing of the RXNE bit is performed by reading the SPI_DR register.

3.4 STM32 SPI In Master Mode

- **Transmit Sequence in Master Mode**
  The transmit sequence begins when a byte is written in the Tx Buffer. The data byte is parallel-loaded into the shift register (from the internal bus) during the first-bit transmission and then shifted out serially to the MOSI pin MSB first or LSB first depending on the LSBFIRST bit in the SPI_CR1 register.
- **Receive Sequence in Master Mode**
  - For the receiver, when data transfer is complete: The data in the shift register is transferred to the RX Buffer and the RXNE flag is set. An interrupt is generated if the RXNEIE bit is set in the SPI_CR2 register.
  - At the last sampling clock edge, the RXNE bit is set, a copy of the data byte received in the shift register is moved to the Rx buffer. When the SPI_DR register is read, the SPI peripheral returns this buffered value.
  - Clearing the RXNE bit is performed by reading the SPI_DR register. A continuous transmit stream can be maintained if the next data to be transmitted is put in the Tx buffer once the transmission is started. Note that the TXE flag should be ‘1 before any attempt to write the Tx buffer is made.

3.5 SPI Half-Duplex Communication Configuration

The STM32 SPI Hardware is capable of operating in half-duplex mode in 2 configurations.

- 1 clock and 1 bidirectional data wire
- 1 clock and 1 data wire (receive-only or transmit-only)

3.6 STM32 SPI Data Transmission & Reception

- In reception, data are received and then stored into an internal Rx buffer while In transmission, data are first stored into an internal Tx buffer before being transmitted.
- A read access to the SPI_DR register returns the Rx buffered value, whereas a write access to the SPI_DR stores the written data into the Tx buffer.

3.7 STM32 SPI CRC Calculation

- A CRC calculator has been implemented in STM32 SPI hardware for communication reliability.
- Separate CRC calculators are implemented for transmitted data and received data.
- The CRC is calculated using a programmable polynomial serially on each bit.
- It is calculated on the sampling clock edge defined by the CPHA and CPOL bits in the SPI_CR1 register.
- At the end of data and CRC transfers, the CRCERR flag in the SPI_SR register is set if corruption occurs during the transfer.

- This is similar to a checksum if you're familiar with error detection procedures for communication.
- The CRC (cyclic redundancy check) is a much "mathematically" powerful operation than a normal checksum that's much harder to fool.

4. **STM32 SPI Flags (Status & Errors)**
   There are some status flags provided for the application to completely monitor the state of the SPI bus.

   - **Tx buffer empty flag (TXE)**: When it is set, this flag indicates that the Tx buffer is empty and the next data to be transmitted can be loaded into the buffer. The TXE flag is cleared when writing to the SPI_DR register.
   - **Rx buffer not empty (RXNE)**: When set, this flag indicates that there are valid received data in the Rx buffer. It is cleared when SPI_DR is read.
   - **BUSY flag**: The BSY flag is useful to detect the end of a transfer if the software wants to disable the SPI and enter Halt mode (or disable the peripheral clock). This avoids corrupting the last transfer. For this, the procedure described below must be strictly respected. The BSY flag is also useful to avoid write collisions in a multi-master system.

   There are also other SPI flags that indicate whether a specific type of error has occurred or not.

   - **SPI Master mode fault (MODF)**: Master mode fault occurs when the master device has its NSS pin pulled low (in NSS hardware mode) or SSI bit low (in NSS software mode), this automatically sets the MODF bit.
   - **SPI Overrun condition**: An overrun condition occurs when the master device has sent data bytes and the slave device has not cleared the RXNE bit resulting from the previous data byte transmitted.
   - **SPI CRC error** This flag is used to verify the validity of the value received when the CRCEN bit in the SPI_CR1 register is set. The CRCERR flag in the SPI_SR register is set if the value received in the shift register does not match the receiver SPI_RXCRCR value.

5. **STM32 SPI Interrupts**
   The SPI interrupt events are connected to the same interrupt vector. So the SPI fires a single interrupt signal regardless of the source of it. The software will have to detect it. These events generate an interrupt if the corresponding Enable Control Bit is set.

SPI interrupt requests

Interrupt event	Event flag	Enable Control bit
Transmit buffer empty flag	TXE	TXEIE
Receive buffer not empty flag	RXNE	RXNEIE
Master Mode fault event	MODF	ERRIE
Overrun error	OVR	
CRC error flag	CRCERR	

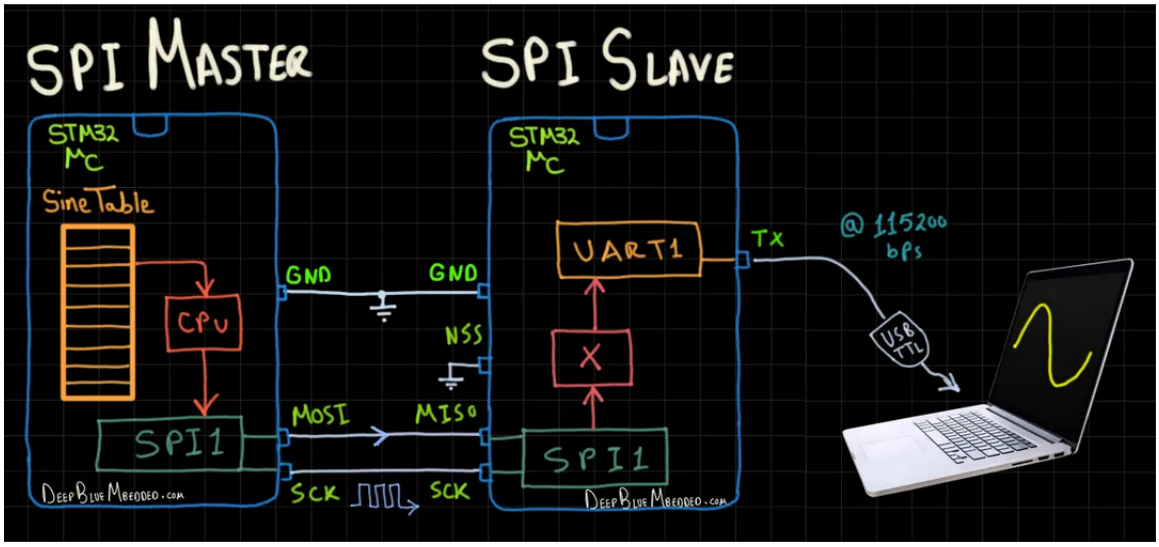
6. **STM32 SPI Transmit & Receive Modes**

   6.1 **SPI With Polling**

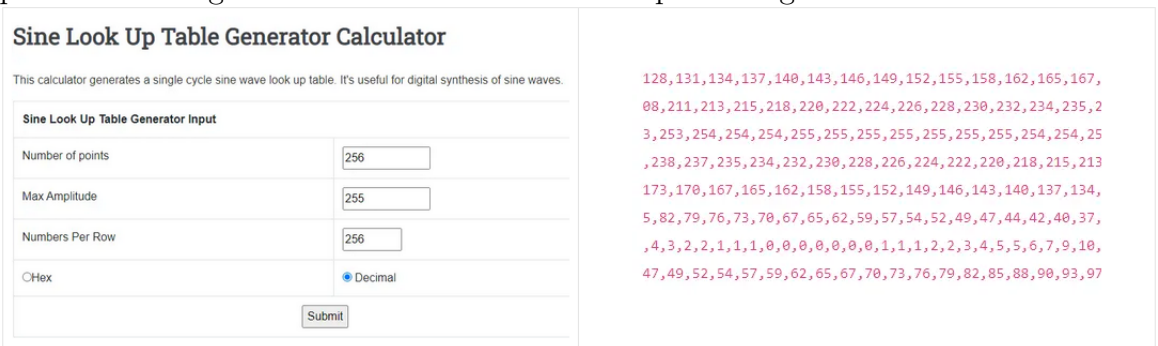
- The first method for sending a data buffer over the SPI bus is by using the Polling method which is a blocking piece of code that waits for the current byte to be completely transmitted then it sends the next and so on.
- This method is the easiest to implement and the most time-consuming for the CPU which will end up being in the "busy waiting" state for some time unnecessarily.
- Here is an example code for sending a data buffer over SPI in the blocking mode (polling):
  uint8_t TX_Data[] = "Hello World!";
  ...
  HAL_SPI_Transmit(&hspi1, TX_Data, sizeof(TX_Data), 1);
  ...

STM32 SPI TX-RX Test Setup

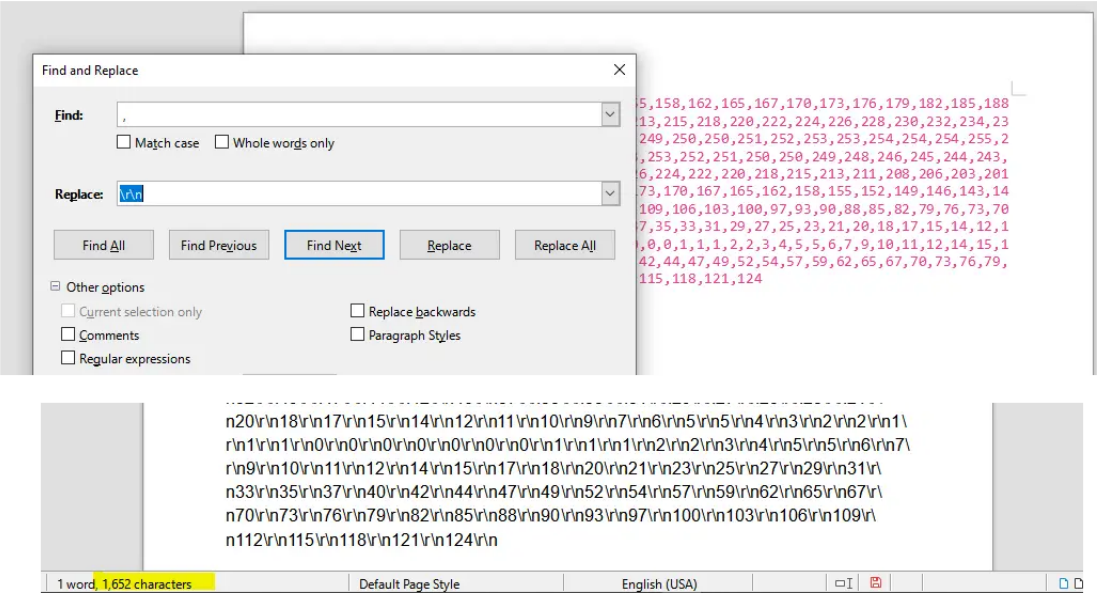
- **Test Setup Overview**



- **Sine Table Data Buffer**
  _You can use a MATLAB script to generate the sine table data points or use this online simple calculator(https://www.daycounter.com/Calculators/Sine-Generator-Calcula phtml. I want the maximum value to be 255, will generate 256 data points for the complete cycle, make numbers in the decimal format, and will get them printed in a single line to be easier for the next processing.

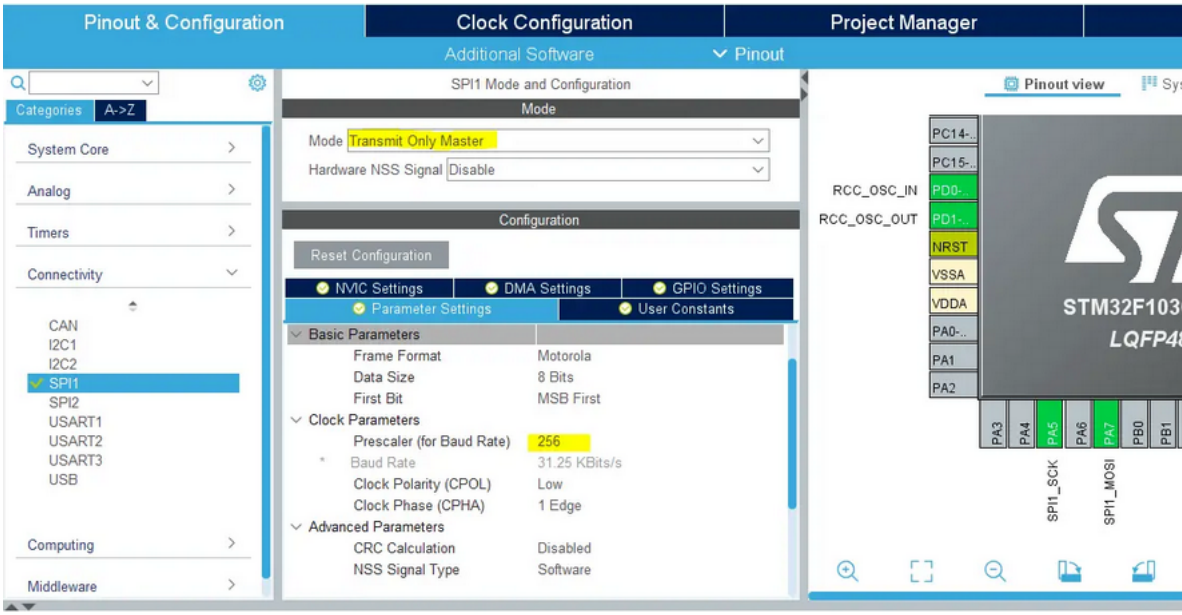


  _Copy the text that has been generated to any word processor like LibreOffice. And there you'll need to click edit and choose the "Find and Replace" tool. Then replace every comma "," with a " ° " so that we can easily plot that data when it's received at the PC side.



The total string length is now 1652 bytes. This is going to be a global variable at the SPI master device, so it's going to take up a lot of the flash memory when it's flashing and the data buffer will be initialized in RAM during startup. That's a lot of memory indeed but anyway it's a tester code. We want to test the receiver functionality and not really concerned about the master device.

- **STM32 SPI Master (TX) Project**
  - Step1:Open CubeMX Create New Project
  - Step2:Choose The Target MCU & Double-Click Its Name
  - Step3:Go To The RCC Clock Configuration
  - Step4: Set The System Clock To Be 72MHz or whatever your uC board supports
    And set the APB2 bus clock Prescaler to 16 or whatever value that makes the SPI bit rate a little bit lower than the UART baud rate just to give the slave SPI devices enough headroom to be able to send out the data packets they will receive
  - Step5: Enable The SPI Module (Transmitter Only Master Mode)

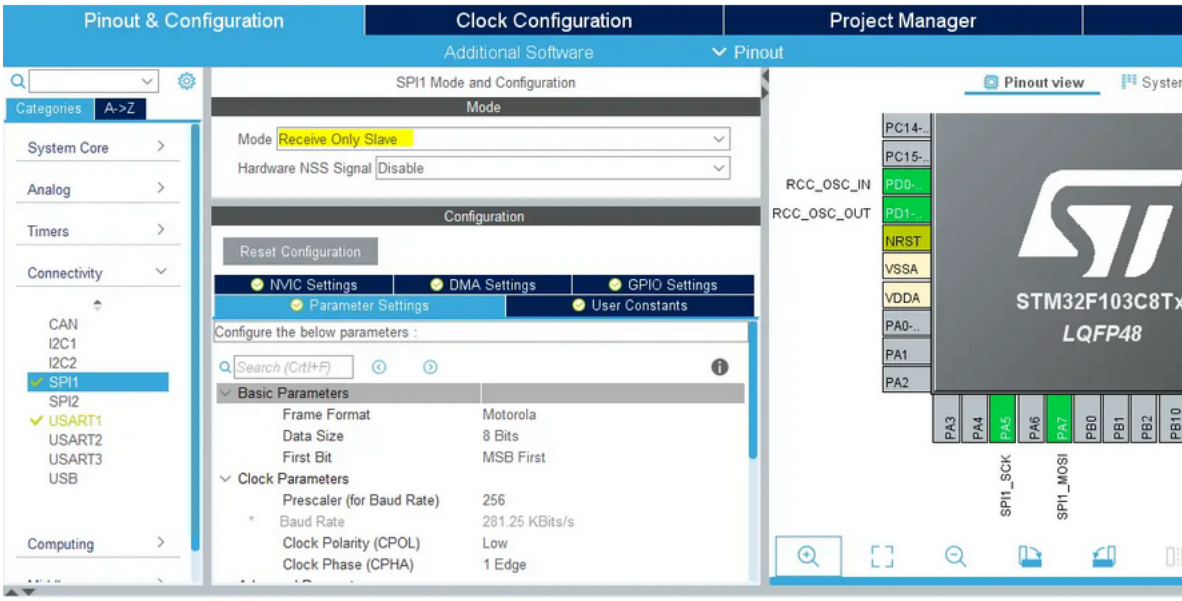


  - Step6: Generate The Initialization Code Open The Project In Your IDE

- Step7: The Application Code For The SPI Master Device (main.c)

```c
#include "main.h"

SPI_HandleTypeDef hspi1;

uint8_t TX_Data[] = "128\r\n131\r\n134\r\n137\r\n140\r\n143\r\n146\r\n149\r\n152\r\n15

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);

int main(void)
{

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI1_Init();

    // Send The TX Data Buffer (Blocking Mode)
    HAL_SPI_Transmit(&hspi1, TX_Data, sizeof(TX_Data), 5000);

    while (1)
    {

    }
}
```

- **STM32 SPI Slave Receiver Polling Mode – LAB**
  - Step1:Open CubeMX Create New Project
  - Step2:Choose The Target MCU & Double-Click Its Name
  - Step3:Go To The RCC Clock Configuration
  - Step4: Set The System Clock To Be 70MHz or whatever your uC board supports
  - Step5: Enable The SPI Module (Receiver Only Slave Mode)



  - Step6: Enable Any UART Module (Async Mode) @ 115200 bps.
  - Step7: Generate The Initialization Code Open The Project In Your IDE
    At this point, we can pretend as if we don't know that incoming SPI data length from the master device and we'd make the receiver buffer large enough to get all the incoming data bytes and set the timeout to any reasonable value

after which we'll stop SPI reception and start processing the available data, if any, has been received.

- Step8: The Application Code For This LAB (main.c)

```c
#include "main.h"

SPI_HandleTypeDef hspi1;
UART_HandleTypeDef huart1;

uint8_t RX_Data[1652] = {0};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI1_Init();
    MX_USART1_UART_Init();

    // Receive SPI Data (Blocking Mode) Polling
    HAL_SPI_Receive(&hspi1, RX_Data, sizeof(RX_Data), 5000);
    HAL_UART_Transmit(&huart1, RX_Data, sizeof(RX_Data), 5000);

    while (1)
    {

    }
}
```

- **The Result For This LAB Testing (Video)**
  I did restart the slave device first, then restart the transmitter and wait for the 1.6Kbytes to be completely received over SPI and sent over UART to my PC. The Arduino Serial Plotter did catch all the data bytes successfully and the result is a pretty sine wave.

6.2 SPI Transmitter With Interrupt

we can use the interrupt signal to free-up some of the CPU time. So it does start the data transmission process and goes to handle other logic parts of the firmware until the transmission completion interrupt signal is fired. Then, the CPU goes to handle the interrupt and sends the next byte of data to be transmitted.
Here is an example code for sending a data buffer over SPI in the interrupt mode

```c
SPI_HandleTypeDef hspi1;

uint8_t TX_Data[] = "Hello World!";

int main(void)
{
    ..
    HAL_SPI_Transmit_IT(&hspi1, TX_Data, sizeof(TX_Data));
    ..
    while (1)
    {
      ..
    }
}
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef * hspi)
{
    // TX Done .. Do Something ...
}
```

STM32 SPI Slave Receiver Interrupt Mode – LAB

- Step1:Open CubeMX Create New Project
- Step2:Choose The Target MCU & Double-Click Its Name
- Step3:Go To The RCC Clock Configuration
- Step4: Set The System Clock To Be 70MHz or whatever your uC board supports.
- Step5: Enable The SPI Module (Receiver Only Slave Mode) + Enable NVIC Interrupt For SPI.
- Step6: Enable Any UART Module (Async Mode) @ 115200 bps + Enable UART Interrupt in NVIC tab.
- Step7: Generate The Initialization Code Open The Project In Your IDE.
- Step8: The Application Code For This LAB (main.c):
  ⎵At this point, we can pretend as if we don't know the incoming SPI data length from the master device and we'd make the receiver buffer may be 100 bytes. Whenever 100 bytes are received by SPI in interrupt mode, the Rx completion callback function is called. At which, we'll process the received data and repeat to get the next 100 bytes.
  ⎵The issue with this approach is we'll be receiving a total of 1652 bytes. After 16 callbacks we'll have received and processed 1600 bytes of data. There will be 52 bytes remaining. The SPI will receive those bytes in interrupt mode. But the application will not be notified as the callback function gets called only when the provided buffer (100 bytes) is full.
  ⎵Those remaining 52 bytes won't be lost or anything. They'll roll over to the next communication session. If we did restart the master device, it'll attempt to send another 1652bytes data packet. After getting the first 48 bytes, the Rx callback will be called because there are already another 52 bytes in the buffer from the last communication session.

```c
#include "main.h"

#define BUFFER_SIZE 100

SPI_HandleTypeDef hspi1;
UART_HandleTypeDef huart1;

uint8_t RX_Buffer[BUFFER_SIZE] = {0};

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI1_Init();
    MX_USART1_UART_Init();

    HAL_SPI_Receive_IT(&hspi1, RX_Buffer, BUFFER_SIZE);

    while (1)
    {

    }
}

void HAL_SPI_RxCpltCallback(SPI_HandleTypeDef * hspi)
{
    HAL_SPI_Receive_IT(&hspi1, RX_Buffer, BUFFER_SIZE);
    HAL_UART_Transmit_IT(&huart1, RX_Buffer, BUFFER_SIZE);
}
```

XXVI I2C Tutorial

1. **Introduction To I2C Communication**
   I2C (i-square-c) is an acronym for "Inter-Integrated-Circuit" which was originally created by Philips Semiconductors (now NXP) back in 1982. I2CTM is a registered trademark for its respective owner and maybe it was the reason they call it "Two Wire Interface (TWI)" in some microcontrollers like Atmel AVR. The I2C is a multi-master, multi-slave, synchronous, bidirectional, half-duplex serial communication bus. It's widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication.

   1.1 **I2C Modes & Bus Speeds**
   Originally, the I2C-bus was limited to 100 kbit/s operations. Over time there have been several additions to the specification so that there are now five operating speed categories. Standard-mode, Fast-mode (Fm), Fast-mode Plus (Fm+), and High-speed mode (Hs-mode) devices are downward-compatible. This means any device may be operated at a lower bus speed. Ultra Fast-mode devices are not compatible with previous versions since the bus is unidirectional.
   Bidirectional bus:

   - **Standard-Mode (Sm)**, with a bit rate up to 100 kbit/s
   - **Fast-Mode (Fm)**, with a bit rate up to 400 kbit/s
   - **Fast-Mode Plus (Fm+)**, with a bit rate up to 1 Mbit/s.

- **High-speed Mode (Hs-mode)**, with a bit rate up to 3.4 Mbit/s.

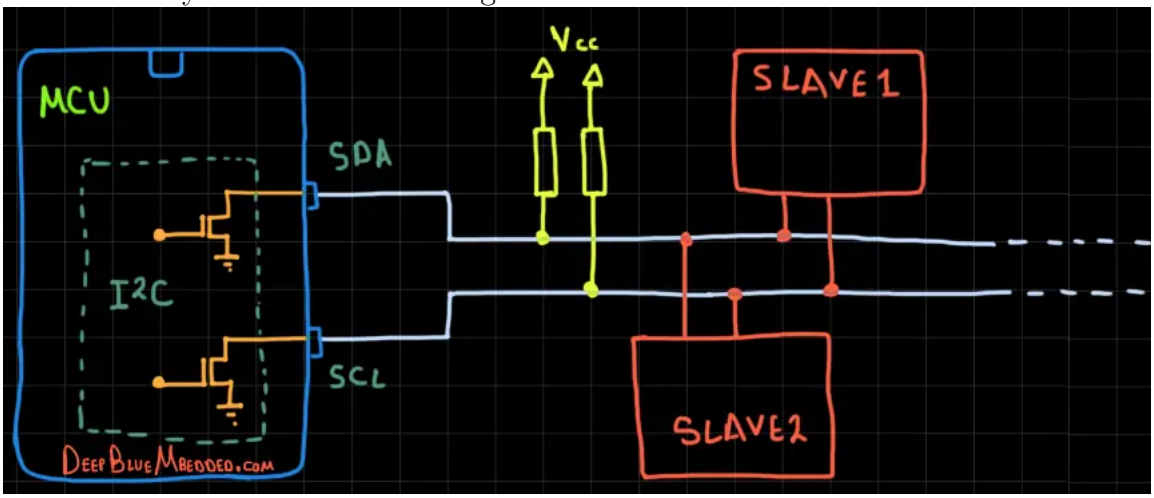
Unidirectional bus:

- **Ultra Fast-Mode (UFm)**, with a bit rate up to 5 Mbit/s

Note: You have to refer to the specific device datasheet to check the typical details for the i2c hardware specifications that have actually been implemented on-chip.

1.2 I2C Physical Layer (Hardware)

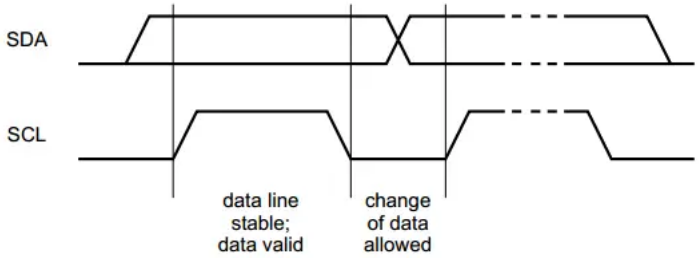
The I2C bus uses what's known as an open-drain (open-collector) output driver for both SDA and SCL lines. Which as the name suggests is having each IO pin connected to the collector of the output driver transistor internally, while having it pulled up to Vcc with a resistor eternally. That's why the default (IDLE) state for each line is HIGH when the open-drain driver is turned OFF. However, if we turn ON the output driver, the IO pin is driven LOW to the ground by the output driver transistor as you can see in the diagram below.



Anyone will write a 0 first while the other is writing a 1, will win the arbitration and continue its message and the other master will stop and wait till the end. That's because of the nature of "Open-drain" output. To write a 0, we turn ON the output driver to pull the signal line to LOW. To write a 1, we turn OFF the output driver and the line will be pulled up to HIGH by the effect of the external resistors. That's why bus arbitration is a very powerful feature for I2C communication.

1.3 SDA & SCL, Data Validity

Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a current-source or pull-up resistor. When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function.

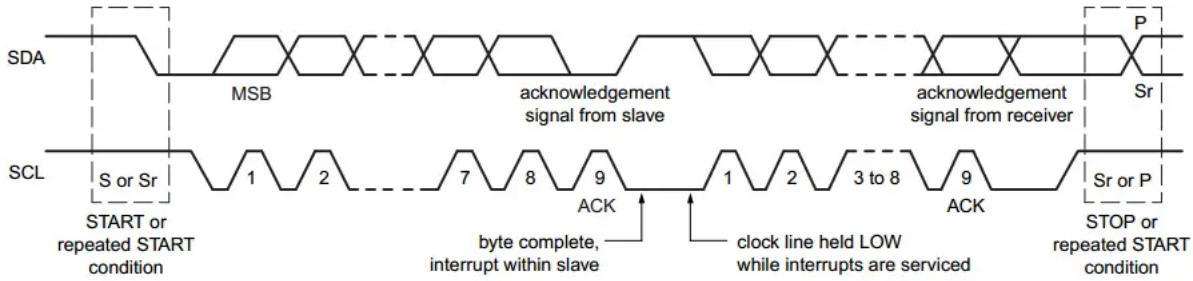


Bit transfer on the I²C-bus

1.4 Elements of I2C Transactions

A typical I2C message consists of some basic elements (conditions) that take place on the I2C bus sequentially and it always starts with a **start condition (s)**. Followed

by the desired slave device **address (7-Bits or 10-Bits)**, then a **R/W** bit to determine whether the master (who initiated the S condition for communication) wants to read or write to this slave having that address. Then if the slave exists and works OK, it'll acknowledge back to the master by sending an **Acknowledge bit ACK** otherwise, it's considered a **Negative Acknowledge NACK**. Afterward, the byte of Data is sent, followed by an acknowledge from the slave. And finally, the master can terminate the communication by sending the **Stop Condition (P)** sequence.



Data transfer on the I²C-bus

2. I2C Hardware In STM32

2.1 STM32 I2C Hardware Overview

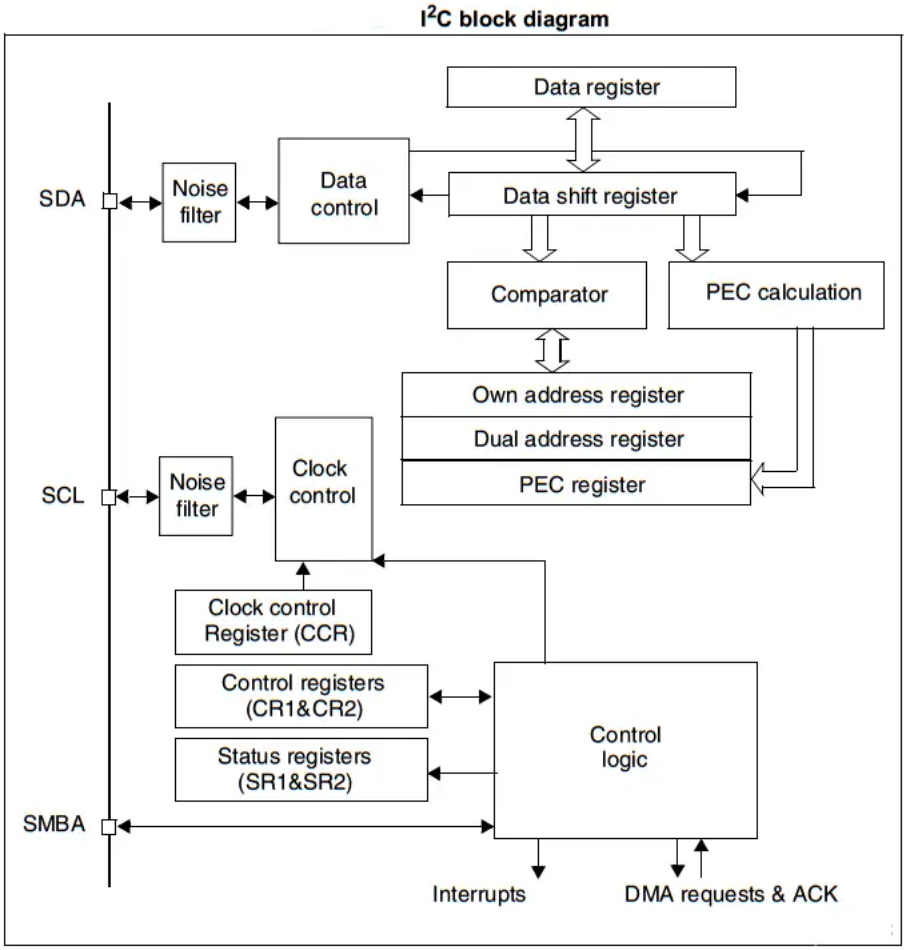
- I2C (inter-integrated circuit) bus Interface serves as an interface between the microcontroller and the serial I2C bus.
- It provides multi-master capability and controls all I2C bus-specific sequencing, protocol, arbitration, and timing.
- It supports the standard mode (Sm, up to 100 kHz) and Fm mode (Fm, up to 400 kHz).
- It may be used for a variety of purposes, including CRC generation and verification, SMBus (system management bus), and PMBus (power management bus).

2.2 STM32 I2C Main Features

- Multimaster capability: the same interface can act as Master or Slave
- I2C Master features: [Clock generation – Start and Stop generation]
- I2C Slave features: [Programmable I2C Address detection – Dual Addressing Capability to acknowledge 2 slave addresses – Stop bit detection]
- Generation and detection of 7-bit/10-bit addressing and General Call
- Supports different communication speeds:
  – Standard Speed (up to 100 kHz)
  – Fast Speed (up to 400 kHz)
- Analog noise filter
- 2 Interrupt vectors:
  – 1 Interrupt for successful address/ data communication
  – 1 Interrupt for error condition
- Optional clock stretching
- 1-byte buffer with DMA capability
- Configurable PEC (packet error checking) generation or verification:
- SMBus 2.0 Compatibility
- PMBus Compatibility

3. STM32 I2C Hardware Functionalities

3.1 STM32 I2C Block Diagram



3.2 STM32 I2C Mode Selection

The interface can operate in one of the four following modes:

- Slave transmitter
- Slave receiver
- Master transmitter
- Master receiver

3.3 STM32 I2C In Slave Mode

By default, the I2C interface operates in Slave mode. To switch from default Slave mode to Master mode a Start condition generation is needed. The peripheral input clock must be programmed in the I2C_CR2 register in order to generate correct timings. The peripheral input clock frequency must be at least:

- 2 MHz in Sm mode
- 4 MHz in Fm mode

Process of Slave Mode:

- As soon as a start condition is detected, the address is received from the SDA line and sent to the shift register.
- Then it is compared with the address of the interface.
- Following the address reception and after clearing ADDR, the slave receives bytes from the SDA line into the DR register via the internal shift register.

- After each byte, the interface generates an acknowledge pulse if the ACK bit is set.
- If RxNE is set and the data in the DR register is not read before the end of the next data reception, the BTF bit is set and the interface waits until BTF is cleared by a read from I2C_SR1 followed by a read from the I2C_DR register, stretching SCL low.
- Clock stretching is essentially holding the clock line SCL LOW by the slave device which prevents any master device on the I2C bus from initiating any new transaction until that slave releases the SCL back again.
- After the last data byte is transferred a Stop Condition is generated by the master.
- The interface detects this condition and sets the STOPF bit and generates an interrupt if the ITEVFEN bit is set.
- The STOPF bit is cleared by a read of the SR1 register followed by a write to the CR1 register.

3.4 **STM32 I2C In Master Mode**

In Master mode, the I2C interface initiates a data transfer and generates the clock signal. A serial data transfer always begins with a Start condition and ends with a Stop condition.
The following is the required sequence in master mode:

- Program the peripheral input clock in I2C_CR2 Register in order to generate correct timings.
- Configure the clock control registers.
- Configure the rise time register.
- Program the I2C_CR1 register to enable the peripheral.
- Set the START bit in the I2C_CR1 register to generate a Start condition.

The peripheral input clock frequency must be at least:

- 2 MHz in Sm mode
- 4 MHz in Fm mode

3.5 **STM32 I2C PEC (Packet Error Checking)**

A PEC calculator has been implemented by STMicroelectronics in I2C hardware to improve the reliability of communication. The PEC is calculated by using the C(x) = x8 + x2 + x + 1 CRC-8 polynomial serially on each bit. By enabling PEC, you can have automatic error checking for large data packet transactions all done by hardware without adding any overhead on the software side.
However, you should also know that if the master device did lose the “arbitration” at any instance, this will corrupt the PEC. And the device will be set back to slave mode and waits until the arbitration “winner” master device finishes its message. Only then, you can start the process all over again.

4. **STM32 I2C Error Conditions**

There are some error conditions that could be detected by the I2C interface hardware to indicate some issues on the hardware level. The software can easily detect those error conditions by reading the corresponding flag bits for each error signal.
The error conditions include:

- **Bus Error (BERR)** – This error occurs when the I2C interface detects an external Stop or Start condition during an address or a data transfer.

- **Acknowledge Failure (AF)** – This error occurs when the interface detects a non-acknowledge bit.
- **Arbitration Lost (ARLO)** – This error occurs when the I2C interface detects an arbitration lost condition.
- **Overrun/Underrun Error (OVR)** – An overrun error can occur in slave mode when clock stretching is disabled and the I2C interface is receiving data. The interface has received a byte and the data in DR has not been read before the next byte is received by the interface. Underrun error can occur in slave mode when clock stretching is disabled and the I2C interface is transmitting data. The interface has not updated the DR with the next byte before the clock comes for the next byte.

5. **STM32 I2C Interrupts**
The I2C interrupt events are connected to the same interrupt vector. So the I2C fires a single interrupt signal regardless of the source of it. The software will have to detect it. These events generate an interrupt if the corresponding Enable Control Bit is set.

I^2C Interrupt requests

Interrupt event	Event flag	Enable control bit
Start bit sent (Master)	SB	ITEVFEN
Address sent (Master) or Address matched (Slave)	ADDR	
10-bit header sent (Master)	ADD10	
Stop received (Slave)	STOPF	
Data byte transfer finished	BTF	
Receive buffer not empty	RxNE	ITEVFEN and ITBUFEN
Transmit buffer empty	TxE	

6. **STM32 I2C Master – Slave Modes TX & RX**
I2C With Polling
The first and the easiest way to do anything in embedded software is just to poll for the hardware resource until it's ready to move on to the next step in your program instructions. However, it's the least efficient way to do things and the CPU will end up wasting so much time in a "busy-waiting" state.
It's the same thing for both transmission and reception. You just wait until the current byte of data to be transmitted so you can start the next one and so on.
I2C With Interrupts

- We can, however, enable the I2C interrupts and have a signal when it's done and ready for servicing by CPU.
- Either for data that has been sent or received.
- Which saves a lot of time and has been always the best way to handle events like that.
- However, in some "Time Critical" applications we need everything to be as deterministic, in time, as possible.
- And a major problem with interrupts is that we can't expect when it'd arrive or during which task.

- That can potentially screw up the timing behavior of the system.

I2C With DMA

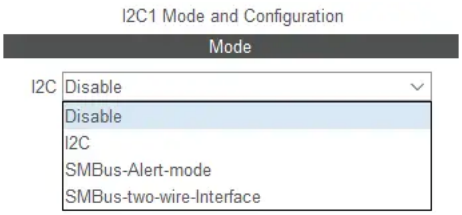
- To operate at its maximum speed, the I2C needs to be fed with the data for transmission and the data received on the Rx buffer should be read to avoid overrun.
- To facilitate the transfers, the I2C features a DMA capability implementing a simple request/acknowledge protocol.
- DMA requests are generated by Data Register becoming empty in transmission and Data Register becoming full in reception.
- Using the DMA will also free the CPU from doing the data transfers "peripheral ¡-¿ memory".
- This will end up saving a lot of time and is considered to be the most efficient way to handle this peripheral to memory data transfer and vice versa.

7. **STM32 I2C Device Memory Read / Write**
In this section, I'll explain a useful feature that has been implemented in HAL APIs for the I2C driver firmware library which is the device memory read/write.
The following are the (Blocking) version for both of the 2 functions:
HAL_I2C_Mem_Write();
HAL_I2C_Mem_Read();
The basic functionality for transmitting data over I2C as a Master device is handled by the following HAL function.
HAL_I2C_Master_Transmit();
Example the MPU6050 IMU device.
The IMU has an internal I2C device address so that any master on the bus can easily address this sensor module. Internally, the MPU6050 itself has some registers with a unique address for each of them. Generally speaking, we need to set some values for the internal registers of the IMU in order to configure it as per our application, and also read some registers in which the IMU data is located.
Single-Byte Read Sequence

Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

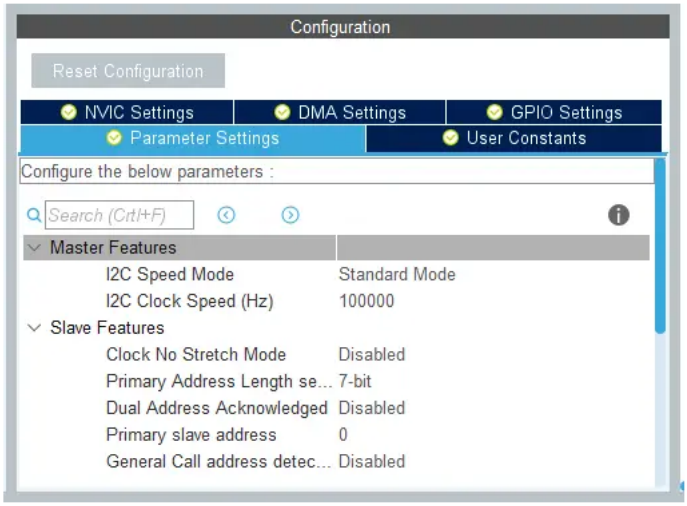
You can do all of that manually with the provided basic HAL APIs or use the Mem_Write / Mem_Read collection of functions that support all modes (blocking – interrupt – DMA).

8. **I2C Configuration In CubeMX**
Here is the configuration tab for the I2C peripheral in CubeMX. And those are the possible modes for I2C.



We can also enable/disable interrupts for I2C in the NVIC tab if you're willing to use interrupt mode instead of polling the peripheral. And the same goes for the DMA, you can click the DMA tab to "Add" a DMA channel dedicated to I2C transfer and configure

its parameters.



9. STM32 I2C HAL Functions APIs

9.1 STM32 I2C "Blocking" HAL Functions (Blocking Mode)

- **Master Transmission**
  HAL_I2C_Master_Transmit (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint8_t* pData, uint16_t Size, uint32_t Timeout);
- **Master Reception**
  HAL_I2C_Master_Receive (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint8_t* pData, uint16_t Size, uint32_t Timeout);
- **Slave Transmission**
  HAL_I2C_Slave_Transmit (I2C_HandleTypeDef * hi2c, uint8_t * pData, uint16_t Size, uint32_t Timeout);
- **Slave Reception**
  HAL_I2C_Slave_Receive (I2C_HandleTypeDef * hi2c, uint8_t * pData, uint16_t Size, uint32_t Timeout);
- **Device Memory Write**
  HAL_I2C_Mem_Write (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t * pData, uint16_t Size, uint32_t Timeout);
- **Device Memory Read**
  HAL_I2C_Mem_Read (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint16_t MemAddress, uint16_t MemAddSize, uint8_t * pData, uint16_t Size, uint32_t Timeout);

9.2 STM32 I2C Interrupt Mode HAL Functions (Non-Blocking Mode)

- **Master Transmission**
  HAL_I2C_Master_Transmit_DMA (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint8_t * pData, uint16_t Size);
  After calling the above function, the I2C peripheral will start sending all the data bytes in the buffer one by one until it's done (in DMA Mode).
  void HAL_I2C_MasterTxCpltCallback (I2C_HandleTypeDef * hi2c)

  // TX Done .. Do Something!

- **Master Reception**
  HAL_I2C_Master_Receive_DMA (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint8_t * pData, uint16_t Size);
  After calling the above function, the I2C peripheral will start receiving all the incoming data bytes in the buffer one by one until it's done (in DMA Mode).
  void HAL_I2C_MasterRxCpltCallback (I2C_HandleTypeDef * hi2c)

  // RX Done .. Do Something!

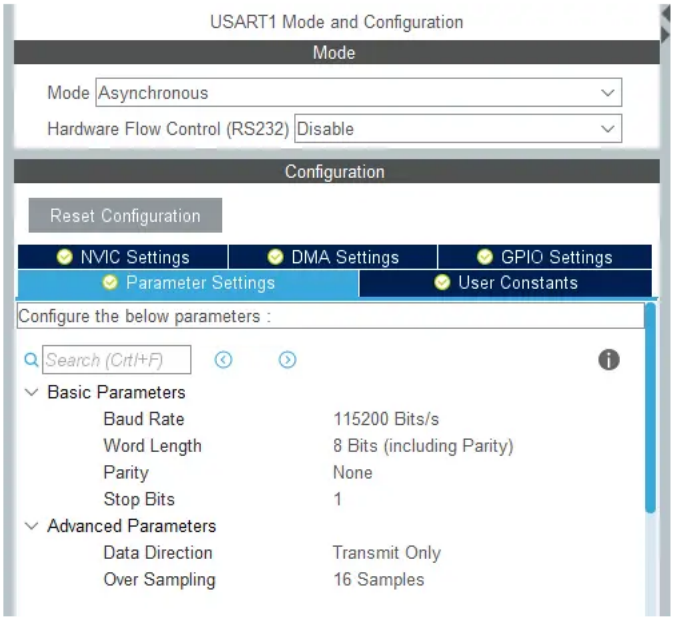
9.3 STM32 I2C Device Check

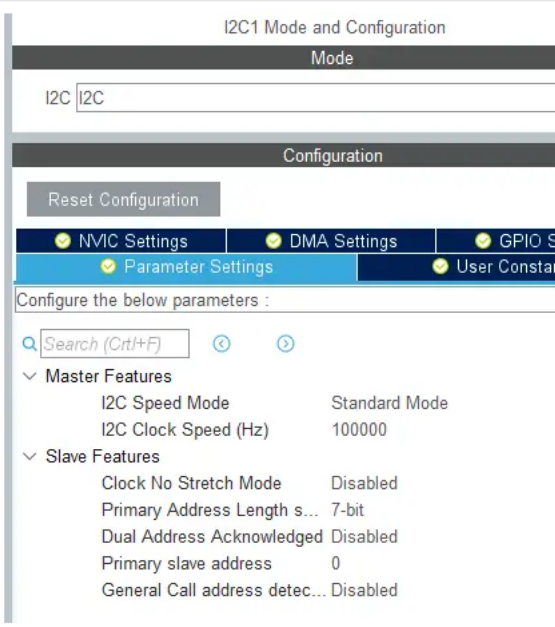
This is a useful utility to check if a slave device on the I2C is actually existing and working or not.
AL_I2C_IsDeviceReady (I2C_HandleTypeDef * hi2c, uint16_t DevAddress, uint32_t Trials, uint32_t Timeout);

10. STM32 I2C Scanner HAL Code Example

For this example project, you'll need to configure one UART peripheral and one I2C interface in master mode. The UART will be used to send the address readings to the PC (with USB-TTL), and the I2C master will do the address scanning as we'll see next.

- **I2C Scanner Project Configurations**
  Those are the CubeMX configurations I've used for UART1 I2C1 peripherals





- **I2C Scanner Project Code Listing**

```c
#include "main.h"
#include "stdio.h"

I2C_HandleTypeDef hi2c1;
UART_HandleTypeDef huart1;

uint8_t Buffer[25] = {0};
uint8_t Space[] = " - ";
uint8_t StartMSG[] = "Starting I2C Scanning: \r\n";
uint8_t EndMSG[] = "Done! \r\n\r\n";

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_I2C1_Init(void);
static void MX_USART1_UART_Init(void);

int main(void)
{
    uint8_t i = 0, ret;

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_I2C1_Init();
    MX_USART1_UART_Init();

    HAL_Delay(1000);
    /*-[ I2C Bus Scanning ]-*/
    HAL_UART_Transmit(&huart1, StartMSG, sizeof(StartMSG), 10000);
    for(i=1; i<128; i++)
    {
        ret = HAL_I2C_IsDeviceReady(&hi2c1, (uint16_t)(i<<1), 3, 5);
        if (ret != HAL_OK) /* No ACK Received At That Address */
        {
            HAL_UART_Transmit(&huart1, Space, sizeof(Space), 10000);
        }
        else if(ret == HAL_OK)
        {
            sprintf(Buffer, "0x%X", i);
            HAL_UART_Transmit(&huart1, Buffer, sizeof(Buffer), 10000);
        }
    }
    HAL_UART_Transmit(&huart1, EndMSG, sizeof(EndMSG), 10000);
    /*--[ Scanning Done ]--*/

    while (1)
    {
    }
}
```

11. **Example2:IMU (MPU6050)**
MPU6050 Have a lot of functions and its not possible to cover all of them in just one tutorial. So in part 1, we will only cover the following:-

11.1 How to Initialize the device
Let's start with the initialization of the MPU6050. In order to initialize the sensor, we need to perform the following actions:- We need to check if the sensor is responding by reading the "WHO_AM_I (0x75)" Register. If the sensor responds with 0x68, this means it's available and good to go.
I am using HAL_I2C_Mem_Read function to directly read from the given memory register
HAL_I2C_Mem_Read (hi2c1, MPU6050_ADDR,WHO_AM_I_REG,1, check, 1, 1000);

Next we will wake the sensor up and in order to do that we will write to the "PWR_MGMT_1 (0x6B)" Register. See below the register content.

4.28 Register 107 – Power Management 1
PWR_MGMT_1

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6B	107	DEVICE _RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2:0]		

On writing (0x00) to the PWR_MGMT_1 Register, sensor wakes up and the Clock sets up to 8 MHz
// power management register 0X6B we should write all 0's to wake the sensor up
Data = 0;
HAL_I2C_Mem_Write(hi2c1, MPU6050_ADDR, PWR_MGMT_1_REG, 1,&Data, 1, 1000);
Now we have to set the Data output Rate or Sample Rate. This can be done by writing into "SMPLRT_DIV (0x19)" Register. This register specifies the divider from the gyroscope output rate used to generate the Sample Rate for the MPU6050. As the formula says Sample Rate = Gyroscope Output Rate / (1 + SMPLRT_DIV). Where Gyroscope Output Rate is 8KHz, To get the sample rate of 1KHz, we need to use the SMPLRT_DIV as '7'.

4.2 Register 25 – Sample Rate Divider
SMPRT_DIV

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1
19	25	SMPLRT_DIV[7:0]						

Sample Rate = Gyroscope Output Rate / (1 + *SMPLRT_DIV*)

// Set DATA RATE of 1KHz by writing SMPLRT_DIV register
Data = 0x07;
HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, SMPLRT_DIV_REG, 1, Data, 1, 1000);
Now configure the Accelerometer and Gyroscope registers and to do so, we need to modify "GYRO_CONFIG (0x1B)" and "ACCEL_CONFIG (0x1C)"Registers.

4.4 Register 27 – Gyroscope Configuration
GYRO_CONFIG

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-

FS_SEL	Full Scale Range
0	± 250 °/s
1	± 500 °/s
2	± 1000 °/s
3	± 2000 °/s

XG_ST Setting this bit causes the X axis gyroscope to perf
YG_ST Setting this bit causes the Y axis gyroscope to perf
ZG_ST Setting this bit causes the Z axis gyroscope to perf

4.5 Register 28 – Accelerometer Configuration ACCEL_CONFIG

Type: Read/Write

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1
1C	28	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]			-

AFS_SEL	Full Scale Range
0	± 2*g*
1	± 4*g*
2	± 8*g*
3	± 16*g*

XA_ST When set to 1, the X- Axis accelerometer perform

YA_ST When set to 1, the Y- Axis accelerometer perform

ZA_ST When set to 1, the Z- Axis accelerometer perform

Writing (0x00) to both of these registers would set the Full scale range of ± 2g in **ACCEL_CONFIG** Register and a Full scale range of ± 250 °/s in **GYRO_CONFIG** Register along with Self-test disabled.

11.2 How to read RAW values

We can read 1 BYTE from each Register separately or we can just read 6 BYTES all together starting from **ACCEL_XOUT_H**Register.

4.17 Registers 59 to 64 – Accelerometer Measurements ACCEL_XOUT_H, ACCEL_XOUT_L, ACCEL_YOUT_H, ACCEL_YOUT_L, ACCEL_ ACCEL_ZOUT_L

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1
3B	59	ACCEL_XOUT[15:8]						
3C	60	ACCEL_XOUT[7:0]						
3D	61	ACCEL_YOUT[15:8]						
3E	62	ACCEL_YOUT[7:0]						
3F	63	ACCEL_ZOUT[15:8]						
40	64	ACCEL_ZOUT[7:0]						

AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2*g*	16384 LSB/*g*
1	±4*g*	8192 LSB/*g*
2	±8*g*	4096 LSB/*g*
3	±16*g*	2048 LSB/*g*

11.3 How to convert acceleration values in ‘g’ and Gyroscope values in dps (°/s)

The ACCEL_XOUT_H (0x3B) Register stores the higher Byte for the acceleration data along X-Axis and Lower Byte is stored in ACCEL_XOUT_L Register. So we need to combine these 2 BYTES into a 16 bit integer value. Below is the process to do that:-

The ACCEL_XOUT_H (0x3B) Register stores the higher Byte for the acceleration data along X-Axis and Lower Byte is stored in ACCEL_XOUT_L Register. So we need to combine these 2 BYTES into a 16 bit integer value. Below is the process to do that:-

ACCEL_X = (ACCEL_XOUT_H ¡¡8 — ACCEL_XOUT_L)

we are shifting the higher 8 bits to the left and than ‘OR’ it with the lower 8 bits.

For Example,if ACCEL_XOUT_H = 11101110 and ACCEL_XOUT_L = 10101010, we will get the resultant 16 bit value as 1110111010101010.
You can see in the picture above that for the Full-Scale range of ± 2g, the sensitivity is 16384 LSB/g. So to get the 'g' value, we need to divide the RAW from 16384. Look at the code below:

4.17 Registers 59 to 64 – Accelerometer Measurements ACCEL_XOUT_H, ACCEL_XOUT_L, ACCEL_YOUT_H, ACCEL_YOUT_L, ACCEL ACCEL_ZOUT_L

Type: Read Only

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1
3B	59	ACCEL_XOUT[15:8]						
3C	60	ACCEL_XOUT[7:0]						
3D	61	ACCEL_YOUT[15:8]						
3E	62	ACCEL_YOUT[7:0]						
3F	63	ACCEL_ZOUT[15:8]						
40	64	ACCEL_ZOUT[7:0]						

AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2*g*	16384 LSB/*g*
1	±4*g*	8192 LSB/*g*
2	±8*g*	4096 LSB/*g*
3	±16*g*	2048 LSB/*g*

```
HAL_I2C_Mem_Read (&hi2c1, MPU6050_ADDR, ACCEL_XOUT_H_REG, 1, Rec_Data, 6, 1000);

Accel_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data [1]);
Accel_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data [3]);
Accel_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data [5]);

/*** convert the RAW values into acceleration in 'g'
     we have to divide according to the Full scale value set in FS_SEL
     I have configured FS_SEL = 0. So I am dividing by 16384.0
     for more details check ACCEL_CONFIG Register              ****/

Ax = Accel_X_RAW/16384.0;  // get the float g
Ay = Accel_Y_RAW/16384.0;
Az = Accel_Z_RAW/16384.0;
```

Reading Gyro Data is similar to the Acceleration case. We will start reading 6 BYTES of data from the **GYRO_XOUT_H** Register, Combine the 2 Bytes to get 16 bit integer RAW values. As we have selected the Full-Scale range of ± 250 °/s, for which the sensitivity is 131 LSB /°/s, we have to divide the RAW values by 131.0 to get the values in dps (°/s). Check below

```
// Read 6 BYTES of data starting from GYRO_XOUT_H register

HAL_I2C_Mem_Read (&hi2c1, MPU6050_ADDR, GYRO_XOUT_H_REG, 1, Rec_Data, 6, 1000);

Gyro_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data [1]);
Gyro_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data [3]);
Gyro_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data [5]);

/*** convert the RAW values into dps (°/s)
     we have to divide according to the Full scale value set in FS_SEL
     I have configured FS_SEL = 0. So I am dividing by 131.0
     for more details check GYRO_CONFIG Register              ****/

Gx = Gyro_X_RAW/131.0;
Gy = Gyro_Y_RAW/131.0;
Gz = Gyro_Z_RAW/131.0;
```

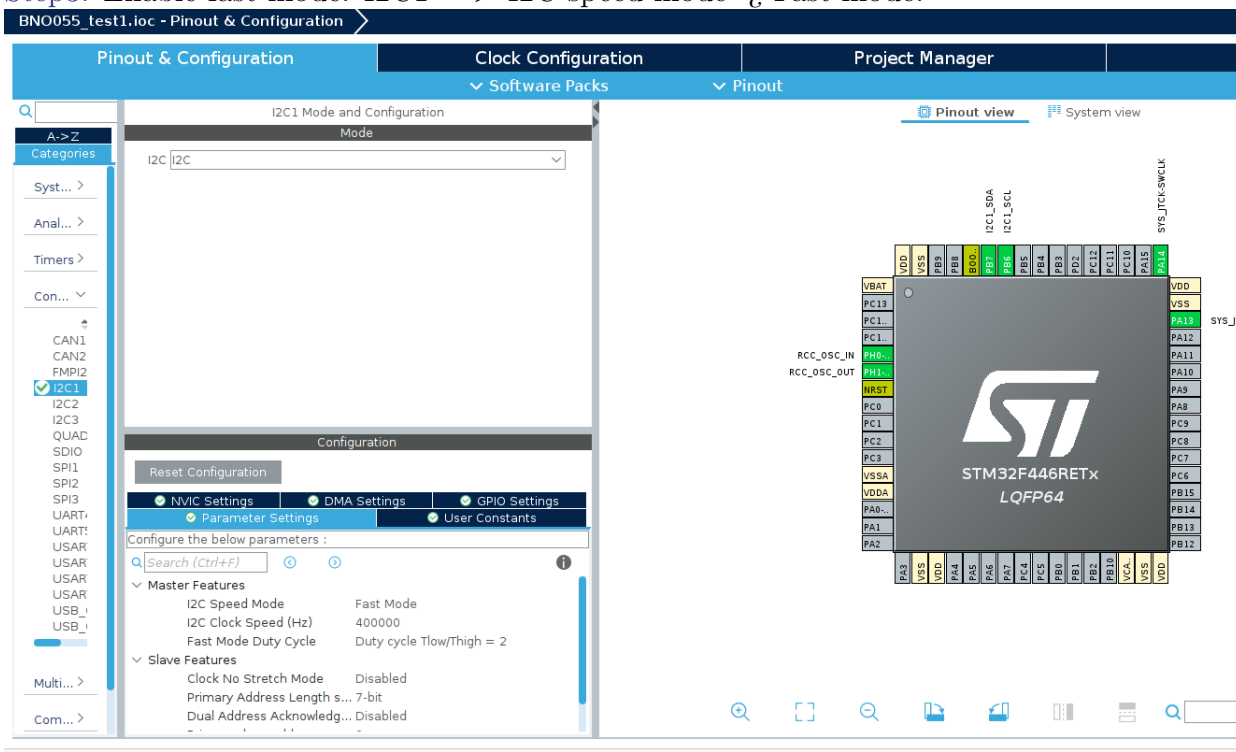
12. **Example3: IMU (BNO055)**

Bosch BNO055 is 9-axis orientation sensor with 3-axis gyro, 3-axis accelerometer and 3-axis magnetometer. It's smart sensor, that support sensor fusion and self-diagnostics. We made a library to use it with STM32 micro-controllers.
We needed only roll, pitch and heading, so we wrote it in the first place, but adding other sensor values are easy. If you want to test the library, you will need STCubeMX and IAR (demo) to build simple project. Here is how to do it.

12.1 How to Configuration

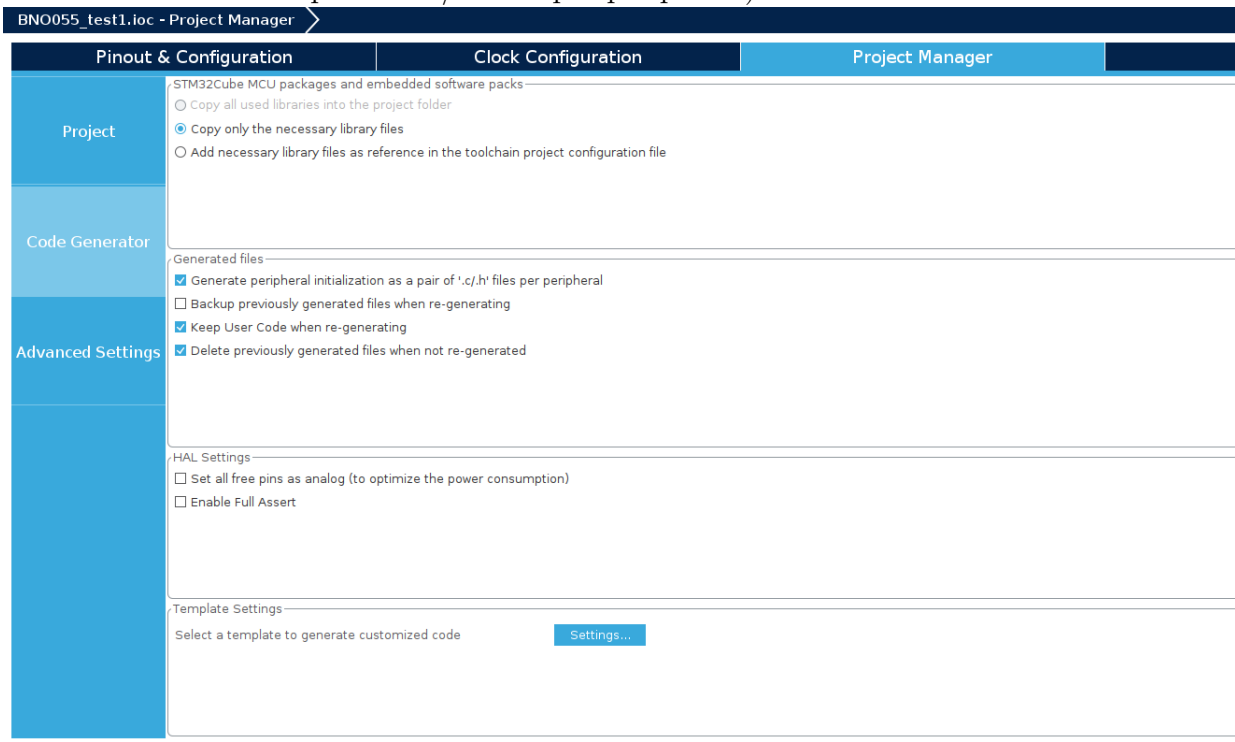
Open STCubeMX and create new project. Select board or chip. I used Nucleo(stm32F446RETx) in my avionics project and will show periphery configuration as example. Here is the minimal settings you need to configure:

- Step1: Enable debug: Sys $->$ Debug $->$ Serial Wire.
- Step2: Enable I2C: Connectivity $->$ I2C1 $->$ I2C1.
- Step3: Enable fast mode: I2C1 $->$ I2C speed mode -¿ Fast mode.



- Step4: Go to Clock Configuration and set Frequency CPU (180 MHz).

- Step5: Go to Project Manager − > Code Generate and Click (Generate peripheral initialization as a pair of '.c/.h' file per peripheral)



12.2 How to Write Code

- Step1: Go to github (https://github.com/ivyknob/bno055_stm32) and download 3 files:
  - bno055.c - main library code
  - bno055.h - common header for every platform
  - bno055_stm32.h - stm32 specific I2C header files
- Step2: Copy all files above to your project. Using IAR as example, you need to copy header files to ./Core/Inc and bno055.c to ./Core/Src and then add file with right click on Application -¿ User -¿ Core in project tree and Add -¿ Add files ... (select bno055.c).
- Step3: Write Code in main.c
  - In USER CODE BEGIN Includes code block add:
    include "bno055_stm32.h"
  - In USER CODE BEGIN PTD
    bno055_vector_t v;
    bno055_vector_t V;
  - In USER CODE BEGIN 2 code block add:
    bno055_assignI2C(&hi2c1);
    bno055_setup();
    bno055_setOperationModeNDOF();
  - After USER CODE Begin WHILE add:
    v= bno055_getVectorEuler();
    V = bno055_getVectorQuaternion();
    Wherer Euller= (roll,pitch,yaw) , Quaternion=(angle=w,rotation=(x,y,z))

XXVII CAN BUS Tutorial

1. **Introduction**
   The Basic Extended CAN peripheral, named bxCAN, interfaces the CAN network. It supports the CAN protocols version 2.0A and B. It has been designed to manage a high number of incoming messages efficiently with a minimum CPU load. It also meets the priority requirements for transmit messages. For safety-critical applications, the CAN controller provides all hardware functions for supporting the CAN Time Triggered Communication option.

2. **bxCAN main**

   - Supports CAN protocol version 2.0A, B Active
   - Bit rates up to 1Mbit/s
   - Supports the Time Triggered Communication option

   2.1 **Transmission**

   - Three transmit mailboxes
   - Configurable transmit priority
   - Time Stamp on SOF transmission

   2.2 **Reception**

   - Two receive FIFOs with three stages
   - Scalable filter banks:
     - 28 filter banks shared between CAN1 and CAN2 in connectivity line devices
     - 14 filter banks in other STM32F10xxx devices
   - Identifier list feature
   - Configurable FIFO overrun
   - Time Stamp on SOF reception

   2.3 **Time-triggered communication option**

   - Disable automatic retransmission mode
   - 16-bit free running timer
   - Time Stamp sent in last two data bytes

   2.4 **Management**

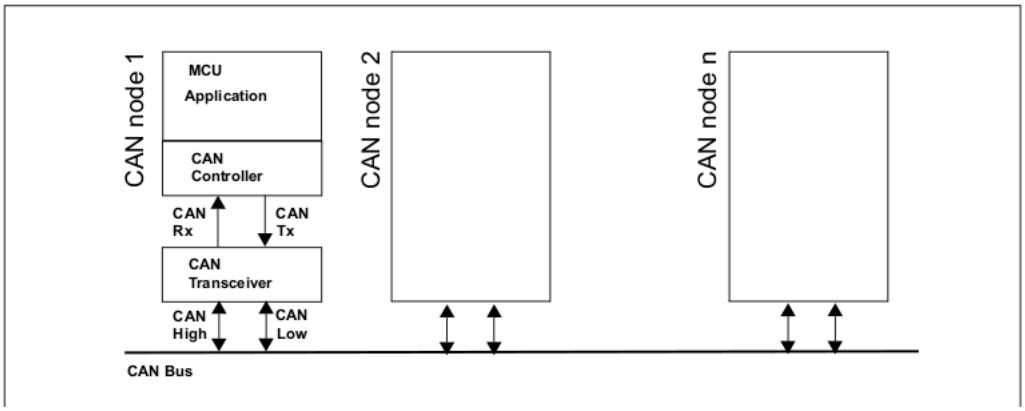
   - Maskable interrupts
   - Software-efficient mailbox mapping at a unique address space

   2.5 **Dual CAN**

   - CAN1: Master bxCAN for managing the communication between a Slave bxCAN and the 512-byte SRAM memory.
   - CAN2: Slave bxCAN, with no direct access to the SRAM memory.
   - The two bxCAN cells share the 512-byte SRAM memory

3. **bxCAN general description**
   In addition to the application messages, Network Management and Diagnostic messages have been introduced.

- An enhanced filtering mechanism is required to handle each type of message. Furthermore, application tasks require more CPU time, therefore real-time constraints caused by message reception have to be reduced.
- A receive FIFO scheme allows the CPU to be dedicated to application tasks for a long time period without losing messages.



3.1 CAN 2.0B active core

The bxCAN module handles the transmission and the reception of CAN messages fully autonomously. Standard identifiers (11-bit) and extended identifiers (29-bit) are fully supported by hardware.

3.2 Control, status and configuration registers

The application uses these registers to:

- Configure CAN parameters, e.g. baud rate
- Request transmissions
- Handle receptions
- Manage interrupts
- Get diagnostic information

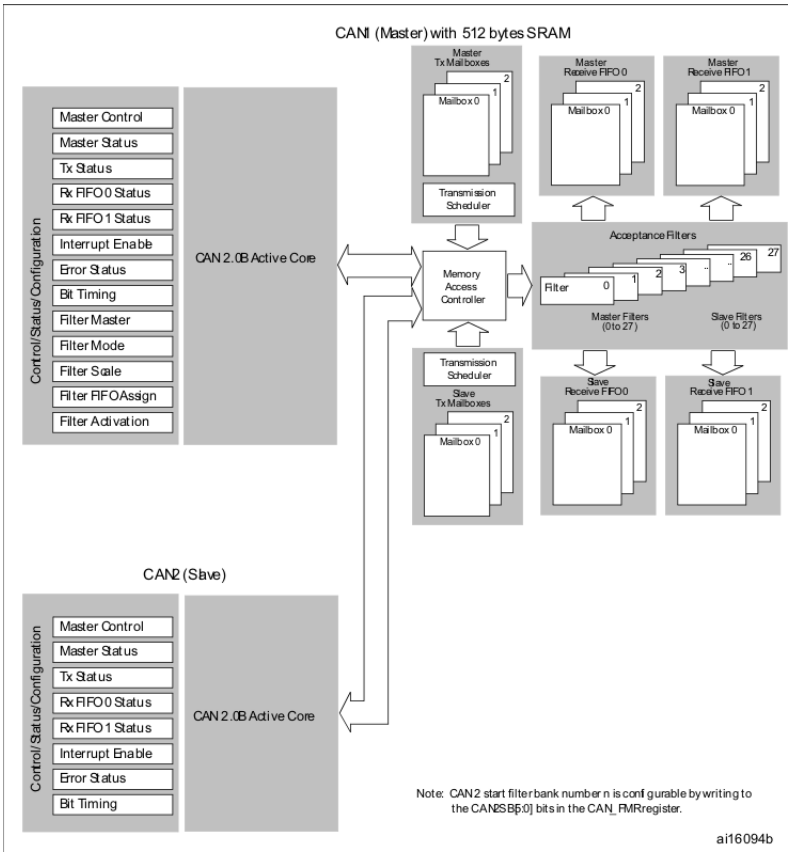
3.3 Tx mailboxes

Three transmit mailboxes are provided to the software for setting up messages. The transmission Scheduler decides which mailbox has to be transmitted first.

3.4 Acceptance filters

- The bxCAN provides 28 scalable/configurable identifier filter banks for selecting the incoming messages the software needs and discarding the others.
- In other devices there are 14 scalable/configurable identifier filter banks.
- Two receive FIFOs are used by hardware to store the incoming messages.
- Three complete messages can be stored in each FIFO.
- The FIFOs are managed completely by hardware.

Figurre:Dual CAN block diagram (connectivity devices)



4. bxCAN operating modes

4.1 Initialization mode

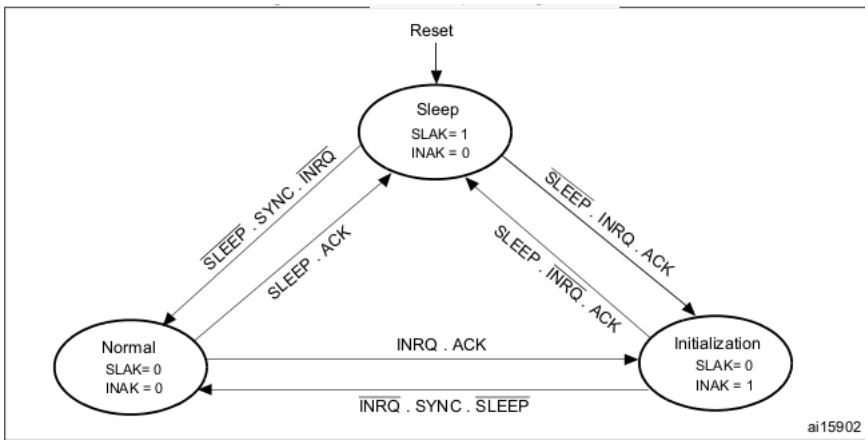
- While in Initialization Mode, all message transfers to and from the CAN bus are stopped and the status of the CAN bus output CANTX is recessive (high).
- To initialize the CAN Controller, software has to set up the Bit Timing (CAN_BTR) and CAN options (CAN_MCR) registers.
- To initialize the registers associated with the CAN filter banks (mode, scale, FIFO assignment, activation and filter values), software has to set the FINIT bit (CAN_FMR). Filter initialization also can be done outside the initialization mode.

4.2 Normal mode

- Once the initialization is complete, the software must request the hardware to enter Normal mode to be able to synchronize on the CAN bus and start reception and transmission.
- The bxCAN enters Normal mode and is ready to take part in bus activities when it has synchronized with the data transfer on the CAN bus.

4.3 Sleep mode (low-power)

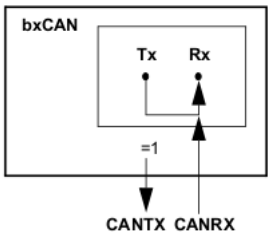
- This mode is entered on software request by setting the SLEEP bit in the CAN_MCR register.
- In this mode, the bxCAN clock is stopped, however software can still access the bxCAN mailboxes.
  Figure:bxCAN operating modes



5. Test mode

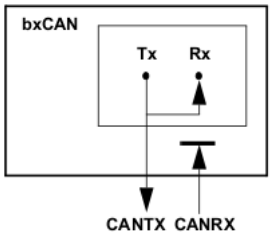
5.1 Silent mode

- The bxCAN can be put in Silent mode by setting the SILM bit in the CAN_BTR register.
- In Silent mode, the bxCAN is able to receive valid data frames and valid remote frames, but it sends only recessive bits on the CAN bus and it cannot start a transmission.
- Silent mode can be used to analyze the traffic on a CAN bus without affecting it by the transmission of dominant bits (Acknowledge Bits, Error Frames).
  Figure: bxCAN in silent mode



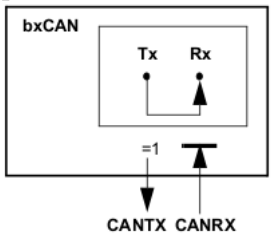
5.2 Loop back mode

- The bxCAN can be set in Loop Back Mode by setting the LBKM bit in the CAN_BTR register.
- n Loop Back Mode, the bxCAN treats its own transmitted messages as received messages and stores them (if they pass acceptance filtering) in a Receive mailbox.
- In this mode, the bxCAN performs an internal feedback from its Tx output to its Rx input.
- The actual value of the CANRX input pin is disregarded by the bxCAN. The transmitted messages can be monitored on the CANTX pin.
  Figure: bxCAN in loop back mode



5.3 Loop back combined with silent mode

- This mode can be used for a “Hot Selftest”, meaning the bxCAN can be tested like in Loop Back mode but without affecting a running CAN system connected to the CANTX and CANRX pins.
- In this mode, the CANRX pin is disconnected from the bxCAN and the CANTX pin is held recessive.



6. bxCAN functional description

6.1 Transmission handling

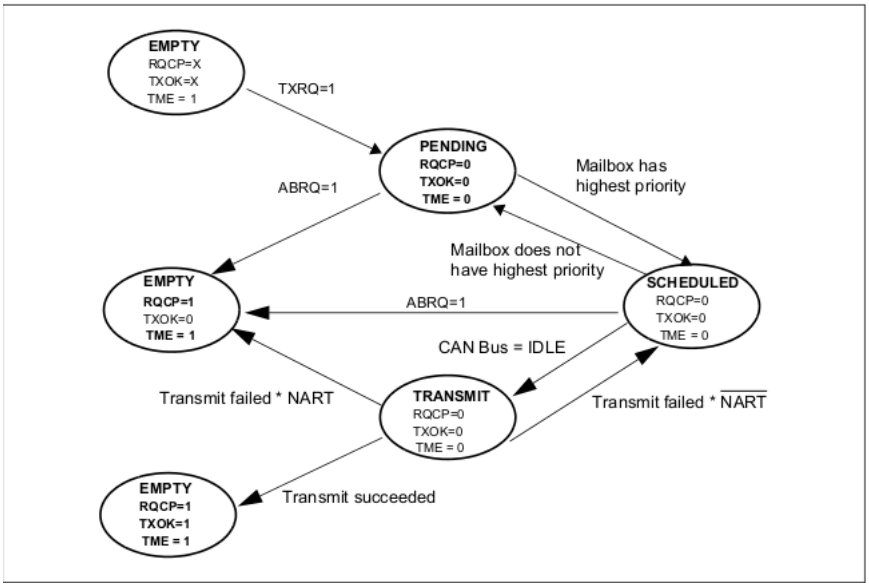
6.1.1 Transmit priority

- By identifier:When more than one transmit mailbox is pending, the transmission order is given by the identifier of the message stored in the mailbox. The message with the lowest identifier value has the highest priority according to the arbitration of the CAN protocol. If the identifier values are equal, the lower mailbox number will be scheduled first.
- By transmit request order: The transmit mailboxes can be configured as a transmit FIFO by setting the TXFP bit in the CAN_MCR register. In this mode the priority order is given by the transmit request order. This mode is very useful for segmented transmission.

6.1.2 Abort

- In pending or scheduled state, the mailbox is aborted immediately.
- An abort request while the mailbox is in transmit state can have two results. If the mailbox is transmitted successfully the mailbox becomes empty with the TXOK bit set in the CAN_TSR register.
- If the transmission fails, the mailbox becomes scheduled, the transmission is aborted and becomes empty with TXOK cleared.
- In all cases the mailbox will become empty again at least at the end of the current transmission.

Figure:Transmit mailbox states.



Time triggered communication mode

- In this mode, the internal counter of the CAN hardware is activated and used to generate the Time Stamp value stored in the CAN_RDTxR/CAN_TDTxR registers, respectively (for Rx and Tx mailboxes).
- The internal counter is captured on the sample point of the Start Of Frame bit in both reception and transmission.

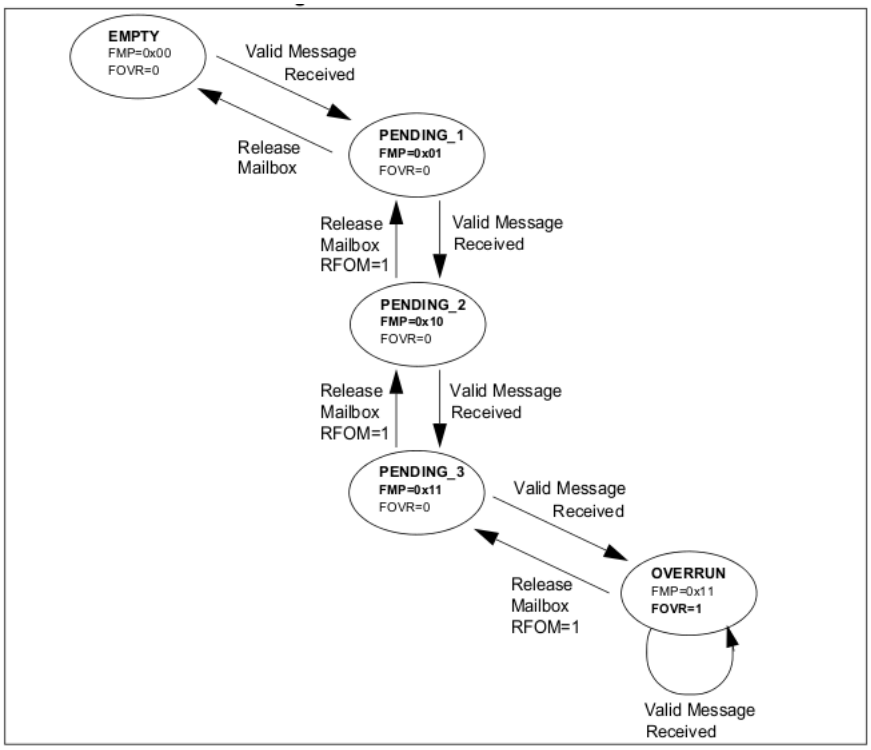
Reception handling

- For the reception of CAN messages, three mailboxes organized as a FIFO are provided.
- In order to save CPU load, simplify the software and guarantee data consistency, the FIFO is managed completely by hardware.
- The application accesses the messages stored in the FIFO through the FIFO output mailbox.

Valid message

A received message is considered as valid when it has been received correctly according to the CAN protocol (no error until the last but one bit of the EOF field) and It passed through the identifier filtering successfully,

Figure:Receive FIFO states



FIFO management

- Starting from the empty state, the first valid message received is stored in the FIFO which becomes pending_1.
- The message is available in the FIFO output mailbox.
- The software reads out the mailbox content and releases it by setting the RFOM bit in the CAN_RFR register.
- If a new valid message has been received in the meantime, the FIFO stays in pending_1 state and the new message is available in the output mailbox.
- If the application does not release the mailbox, the next valid message will be stored in the FIFO which enters pending_2 state (FMP[1:0] = 10b).
- The storage process is repeated for the next valid message putting the FIFO into pending_3 state (FMP[1:0] = 11b).

Overrun

Once the FIFO is in pending_3 state (i.e. the three mailboxes are full) the next valid message reception will lead to an overrun and a message will be lost. The hardware signals the overrun condition by setting the FOVR bit in the CAN_RFR register. Which message is lost depends on the configuration of the FIFO:

- If the FIFO lock function is disabled (RFLM bit in the CAN_MCR register cleared) the last message stored in the FIFO will be overwritten by the new incoming message. In this case the latest messages will be always available to the application.
- If the FIFO lock function is enabled (RFLM bit in the CAN_MCR register set) the most recent message will be discarded and the software will have the three oldest messages in the FIFO available.

Reception related interrupts

- Once a message has been stored in the FIFO, the FMP[1:0] bits are updated and an interrupt request is generated if the FMPIE bit in the CAN_IER register is set.

- When the FIFO becomes full (i.e. a third message is stored) the FULL bit in the CAN_RFR register is set and an interrupt is generated if the FFIE bit in the CAN_IER register is set.
- On overrun condition, the FOVR bit is set and an interrupt is generated if the FOVIE bit in the CAN_IER register is set.

6.2 Identifier filtering

- In the CAN protocol the identifier of a message is not associated with the address of a node but related to the content of the message.
- On message reception a receiver node decides - depending on the identifier value - whether the software needs the message or not,If not, the message must be discarded without intervention by the software.
- To fulfill this requirement, the bxCAN Controller provides 28 configurable and scalable filter banks (27-0) to the application.
- In other devices the bxCAN Controller provides 14 configurable and scalable filter banks (13-0) to the application in order to receive only the messages the software needs.
- This hardware filtering saves CPU resources which would be otherwise needed to perform filtering by software. Each filter bank x consists of two 32-bit registers, CAN_FxR0 and CAN_FxR1.

Scalable width

- One 32-bit filter for the STDID[10:0], EXTID[17:0], IDE and RTR bits.
- Two 16-bit filters for the STDID[10:0], RTR, IDE and EXTID[17:15] bits.

Mask mode

In mask mode the identifier registers are associated with mask registers specifying which bits of the identifier are handled as “must match” or as “don’t care”.

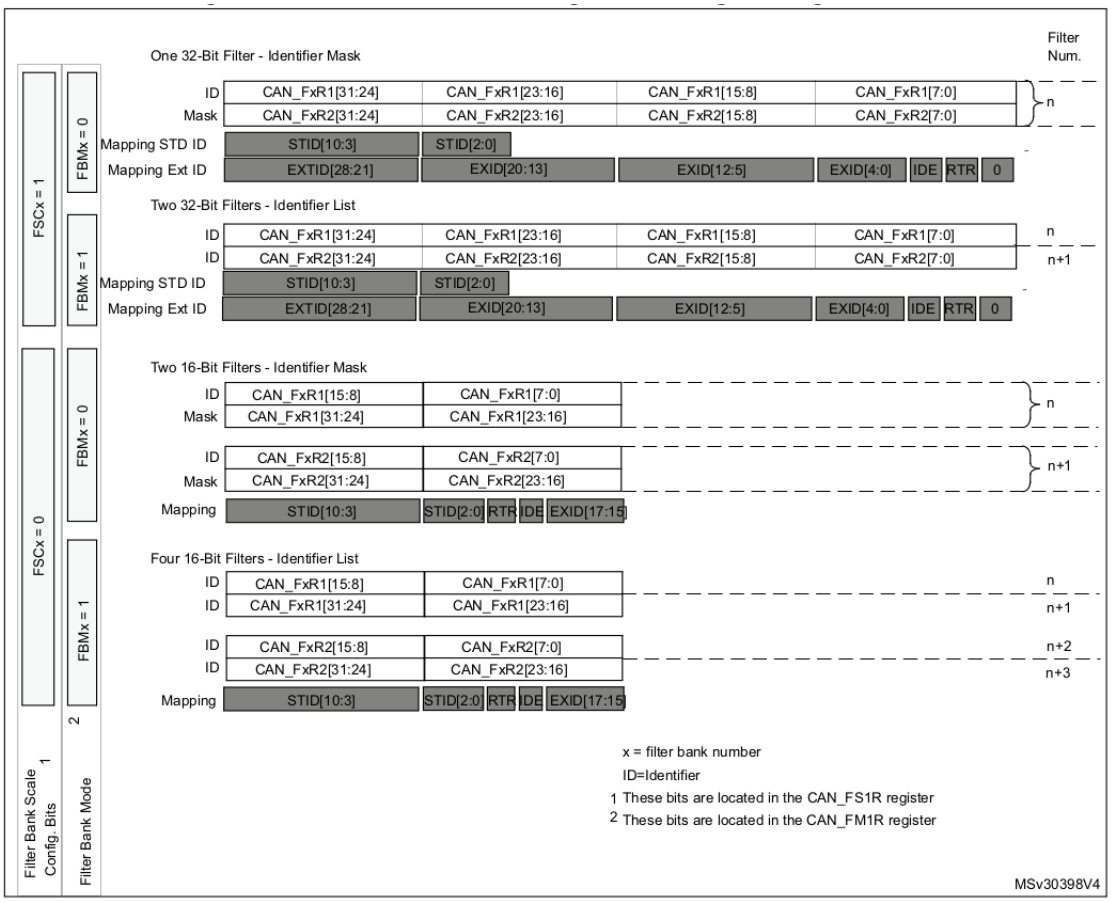
Identifier list mode

In identifier list mode, the mask registers are used as identifier registers. Thus instead of defining an identifier and a mask, two identifiers are specified, doubling the number of single identifiers. All bits of the incoming identifier must match the bits specified in the filter registers.

Filter bank scale and mode configuration

- The filter banks are configured by means of the corresponding CAN_FMR register.
- To configure a filter bank it must be deactivated by clearing the FACT bit in the CAN_FAR register.
- The filter scale is configured by means of the corresponding FSCx bit in the CAN_FS1R register
- The identifier list or identifier mask mode for the corresponding Mask/Identifier registers is configured by means of the FBMx bits in the CAN_FMR register.

Figure: Filter bank scale configuration - Register organization



Filter match index

This index is stored in the mailbox together with the message according to the filter priority rules. Thus each received message has its associated filter match index. The Filter Match index can be used in two ways:

- Compare the Filter Match index with a list of expected values.
- Use the Filter Match Index as an index on an array to access the data destination location.

Note:

- For nonmasked filters, the software no longer has to compare the identifier.
- If the filter is masked the software reduces the comparison to the masked bits only.
- The index value of the filter number does not take into account the activation state of the filter banks. In addition, two independent numbering schemes are used, one for each FIFO.

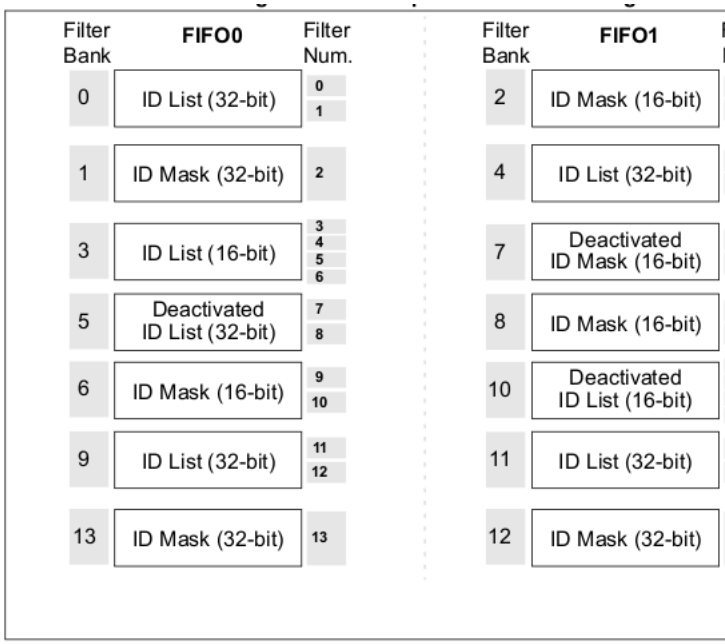


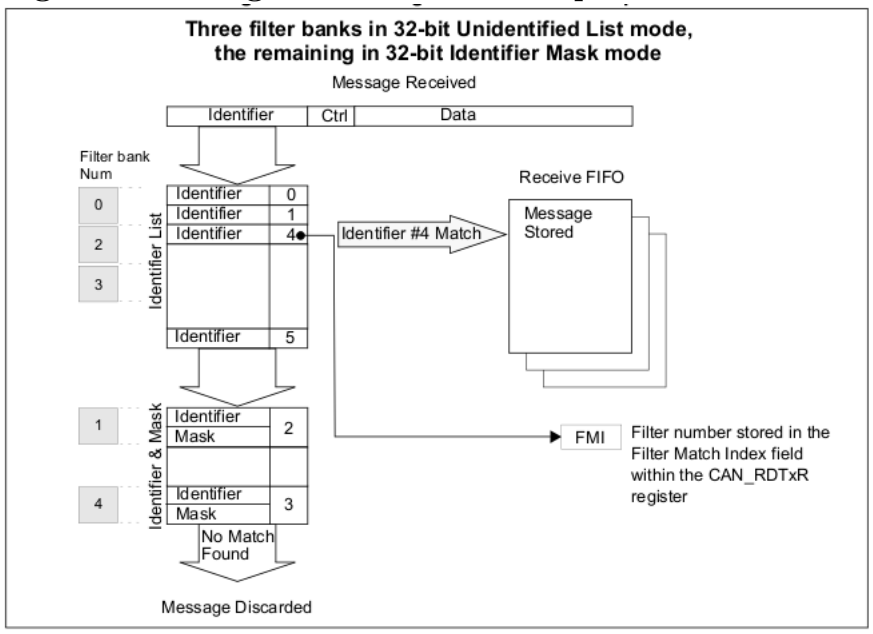
Figure: Example of filter numbering

Filter priority rules

Depending on the filter combination it may occur that an identifier passes successfully through several filters. In this case the filter match value stored in the receive mailbox is chosen according to the following priority rules:

- A 32-bit filter takes priority over a 16-bit filter.
- For filters of equal scale, priority is given to the Identifier List mode over the Identifier Mask mode
- For filters of equal scale and mode, priority is given by the filter number (the lower the number, the higher the priority).

Figure: Filtering mechanism - Example



6.3 Message storage

The interface between the software and the hardware for the CAN messages is implemented by means of mailboxes. A mailbox contains all information related to a message; identifier, data, control, status and time stamp information.

Transmit mailbox

The software sets up the message to be transmitted in an empty transmit mailbox.

The status of the transmission is indicated by hardware in the CAN_TSR register.

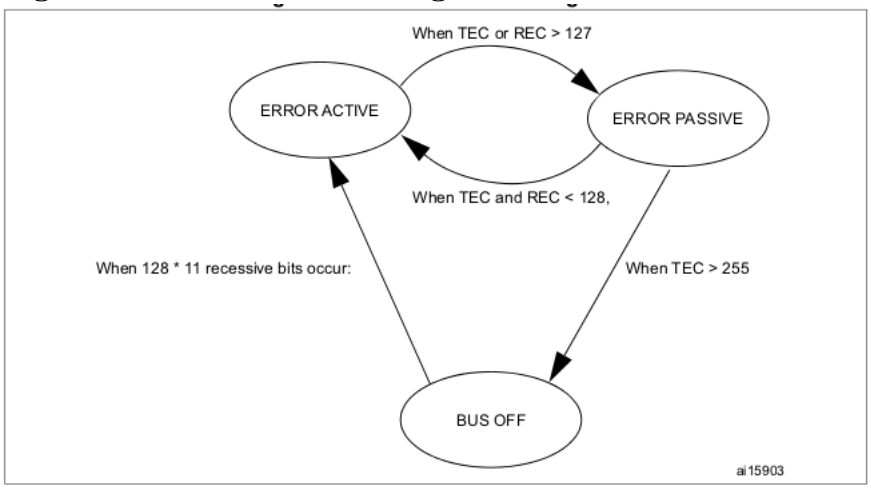
Receive mailbox

When a message has been received, it is available to the software in the FIFO output mailbox. Once the software has handled the message (e.g. read it) the software must release the FIFO output mailbox by means of the RFOM bit in the CAN_RFR register to make the next incoming message available. The filter match index is stored in the MFMI field of the CAN_RDTxR register. The 16-bit time stamp value is stored in the TIME[15:0] field of CAN_RDTxR.

6.4 Error management

The error management as described in the CAN protocol is handled entirely by hardware using a Transmit Error Counter (TEC value, in CAN_ESR register) and a Receive Error Counter (REC value, in the CAN_ESR register), which get incremented or decremented according to the error condition.

Figure:CAN error state diagram



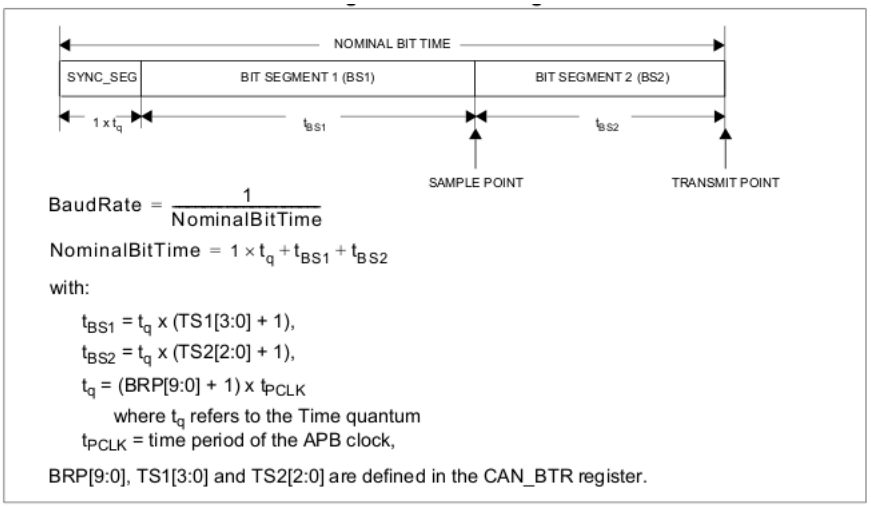
6.5 Bit timing

The bit timing logic monitors the serial bus-line and performs sampling and adjustment of the sample point by synchronizing on the start-bit edge and resynchronizing on the following edges.

Its operation may be explained simply by splitting nominal bit time into three segments as follows:

- **Synchronization segment(SYNC_SEG):** a bit change is expected to occur time segment.It has a fixed length of one time quantum (1 X tq)
- **Bit segment (BS1):** defines the location of the sample point. It includes the PROP_SEG and PHASE_SEG1 of the CAN standard. Its duration is programmable between 1 and 16 time quanta but may be automatically lengthened to compensate for positive phase drifts due to differences in the frequency of the various nodes of the network.
- **Bit segment 2 (BS2):**defines the location of the transmit point. It represents the PHASE_SEG2 of the CAN standard. Its duration is programmable between 1 and 8 time quanta but may also be automatically shortened to compensate for negative phase drifts.

  Figure: Bit timing



7. **bxCAN interrupts**
   Four interrupt vectors are dedicated to bxCAN. Each interrupt source can be independently enabled or disabled by means of the CAN Interrupt Enable register (CAN_IER).

   - The **transmit interrupt** can be generated by the following events:
     - Transmit mailbox 0 becomes empty, RQCP0 bit in the CAN_TSR register set.
     - Transmit mailbox 1 becomes empty, RQCP1 bit in the CAN_TSR register set.
     - Transmit mailbox 2 becomes empty, RQCP2 bit in the CAN_TSR register set.
   - The **FIFO 0 interrupt** can be generated by the following events:
     - Reception of a new message, FMP0 bits in the CAN_RF0R register are not '00'.
     - FIFO0 full condition, FULL0 bit in the CAN_RF0R register set.
     - FIFO0 overrun condition, FOVR0 bit in the CAN_RF0R register set.
   - The **FIFO 1 interrupt** can be generated by the following events:
     - Reception of a new message, FMP0 bits in the CAN_RF0R register are not '00'.
     - FIFO0 full condition, FULL0 bit in the CAN_RF0R register set.
     - FIFO0 overrun condition, FOVR0 bit in the CAN_RF0R register set.
   - The **error and status change interrupt** can be generated by the following events:
     - Error condition, for more details on error conditions refer to the CAN Error Status register (CAN_ESR).
     - Wakeup condition, SOF monitored on the CAN Rx signal.
     - Entry into Sleep mode.

8. **Syntax CAN STM32 to write code**

8.1 Modifying the code

- **Create structures for managing CAN (filters, tranmission message, reception message)**

```
/* USER CODE BEGIN PV*/
/* Private variables */
CAN_FilterTypeDef    sFilterConfig;
CAN_TxHeaderTypeDef    TxHeader;
CAN_RxHeaderTypeDef    RxHeader;
uint8_t    TxData[8];
```

```
uint8_t    RxData[8];
uint32_t   TxMailbox;
/* USER CODE END PV*
```

- **Configure filters in the way, that all received messages are accepted**

```
//CAN filter strucutre items configuration
/* USER CODE BEGIN 2 */
sFilterConfig.FilterBank = 0;
sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
sFilterConfig.FilterIdHigh = 0x0000;
sFilterConfig.FilterIdLow = 0x0000;
sFilterConfig.FilterMaskIdHigh = 0x0000;
sFilterConfig.FilterMaskIdLow = 0x0000;
sFilterConfig.FilterFIFOAssignment = CAN_RX_FIFO0;
sFilterConfig.FilterActivation = ENABLE;
sFilterConfig.SlaveStartFilterBank = 14;
//CAN filter configuration function call
if (HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK)
{
/* Filter configuration Error */
Error_Handler();
}
```

- **Start CAN**

```
if (HAL_CAN_Start(&hcan1) != HAL_OK)
{
/* Start Error */
Error_Handler();
}
or
HAL_CAN_Start(&hcan1);
```

Enable notifications

```
if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING
| CAN_IT_TX_MAILBOX_EMPTY) != HAL_OK)
{
/* Notification Error */
Error_Handler();
}
```

- **Fill in TX message**

```
TxHeader.StdId = 0x321;
TxHeader.ExtId = 0x01;
TxHeader.RTR = CAN_RTR_DATA;
TxHeader.IDE = CAN_ID_STD;
TxHeader.DLC = 8;
TxHeader.TransmitGlobalTime = DISABLE;
TxData[0] = 1;
TxData[1] = 2;
TxData[2] = 3;
TxData[3] = 4;
TxData[4] = 5;
TxData[5] = 6;
```

```
TxData[6] = 7;
TxData[7] = 8;
/* USER CODE END 2 */
```

- **Send message in infinite loop**

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */
/* USER CODE BEGIN 3 */
HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox);
HAL_Delay(500);
TxData[7] = TxData[7] + 1;
}
/* USER CODE END 3 */
```

- **Interrupts callbacks**

```
/* USER CODE BEGIN 4 */
//CAN TX callback
void HAL_CAN_TxMailbox0CompleteCallback(CAN_HandleTypeDef *hcan)
{
HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_2);
}
//CAN RX callback
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader, RxData);
HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_2);
}
/* USER CODE END 4 */
```

8.2 The same application using Old CAN API − > New CAN API

- Fields of CAN_InitTypeDef structure are renamed : SJW to SyncJumpWidth, BS1 to TimeSeg1, BS2 to TimeSeg2, ABOM to AutoBusOff, AWUM to AutoWakeUp, NART to AutoRetransmission (inversed), RFLM to ReceiveFifoLocked and TXFP to TransmitFifoPriority.
- HAL_CAN_Init() is split into both HAL_CAN_Init() and HAL_CAN_Start()
- HAL_CAN_Transmit() is replaced by HAL_CAN_AddTxMessage() to place Tx request, then HAL_CAN_GetTxMailboxesFreeLevel() for polling until completion.
- HAL_CAN_Receive() is replaced by HAL_CAN_GetRxFifoFillLevel() for polling until reception, then HAL_CAN_GetRxMessage() to get Rx message
- HAL_CAN_Receive_IT() is replaced by HAL_CAN_ActivateNotification() to enable reception with interrupt mode, then HAL_CAN_GetRxMessage() in the receive callback to get Rx message.
- HAL_CAN_Sleep() is renamed to HAL_CAN_RequestSleep()
- HAL_CAN_TxCpltCallback() is split into HAL_CAN_TxMailbox0CompleteCallback(), HAL_CAN_TxMailbox1CompleteCallback() and HAL_CAN_TxMailbox2CompleteCallback
- HAL_CAN_RxCpltCallback() is split into HAL_CAN_RxFifo0MsgPendingCallback() and HAL_CAN_RxFifo1MsgPendingCallback()

9. Python CAN

9.1 **Installation**: Install the can package from PyPi with pip or similar:
pip3 install python-can (if python3)

9.2 **Interface Names**

Name	Documentation
"canalystii"	*CANalyst-II*
"cantact"	*CANtact CAN Interface*
"etas"	*ETAS*
"gs_usb"	*Geschwister Schneider and candleLight*
"iscan"	*isCAN*
"ixxat"	*IXXAT Virtual Communication Interface*
"kvaser"	*Kvaser's CANLIB*
"neousys"	*Neousys CAN Interface*
"neovi"	*Intrepid Control Systems neoVI*
"nican"	*National Instruments NI-CAN*
"nixnet"	*National Instruments NI-XNET*
"pcan"	*PCAN Basic API*
"robotell"	*Robotell CAN-USB interface*
"seeedstudio"	*Seeed Studio USB-CAN Analyzer*
"serial"	*CAN over Serial*
"slcan"	*CAN over Serial / SLCAN*
"socketcan"	*SocketCAN*
"socketcand"	*socketcand Interface*
"systec"	*SYSTEC interface*
"udp_multicast"	*Multicast IP Interface*
"usb2can"	*USB2CAN Interface*
"vector"	*Vector*
"virtual"	*Virtual*

9.3 **Bus**

The BusABC class provides a wrapper around a physical or virtual CAN Bus. An interface specific instance is created by calling the Bus() function with a particular interface, for example:
import can
bus=can.interface.Bus(channel='can0', interface ='socketcan', bitrate=1000000)

- **Transmitting**
  Writing individual messages to the bus is done by calling the send() method and passing a Message instance.
  msg = can.Message(arbitration_id=0x111, is_extended_id=False, data=self.TxData)
  bus.send(msg,0.1) (//0.1 is timeout)
- **Receiving**
  Reading from the bus is achieved by either calling the recv() method or by directly iterating over the bus:
  can_msg =self.bus.recv(0.1) (//0.1 is timeout)
  When print can_msg we have can_msg.(arbitration_id, DLC, data[ ](RxData))

- **Filtering**
  Message filtering can be set up for each bus. Where the interface supports it, this is carried out in the hardware or kernel layer - not in Python. All messages that match at least one filter are returned.
  Example defining two filters, one to pass 11-bit ID $0\times451, theothertopass29-bitID0\times$

$A0000:$
$filters = [\{"can\_id" : 0 \times 451, "can\_mask" : 0 \times 7FF, "extended" : False\}, \{"can\_id" : 0 \times A0$
$bus = can.interface.Bus(channel = "can0", interface = "socketcan", can\_filters = filters)$

9.4 SocketCAN

The SocketCAN documentation can be found in the Linux kernel docs in the networking directory. Quoting from the SocketCAN Linux documentation:
The socketcan package is an implementation of CAN protocols (Controller Area Network) for Linux. CAN is a networking technology which has widespread use in automation, embedded devices, and automotive fields. While there have been other CAN implementations for Linux based on character devices, Socket- CAN uses the Berkeley socket API, the Linux network stack and implements the CAN device drivers as network interfaces. The CAN socket API has been designed as similar as possible to the TCP/IP protocols to allow programmers, familiar with network programming, to easily learn how to use CAN sockets.

9.4.1 Socketcan Quickstart

The CAN network driver provides a generic interface to setup, configure and monitor CAN devices. To configure bit-timing parameters use the program ip.

- **The virtual CAN driver (vcan)**
  _The virtual CAN interfaces allow the transmission and reception of CAN frames without real CAN controller hardware. Virtual CAN network devices are usually named 'vcanX', like vcan0 vcan1 vcan2.
  _To create a virtual can interface using socketcan run the following:
  sudo modprobe vcan
  # Create a vcan network interface with a specific name
  sudo ip link add dev vcan0 type vcan sudo ip link set vcan0 up
- **Real Device**
  vcan should be substituted for can and vcan0 should be substituted for can0 if you are using real hardware. Setting the bitrate can also be done at the same time, for example to enable an existing can0 interface with a bitrate of 1MB:
  sudo ip link set can0 type can bitrate 1000000
  sudo ip link set up can0
- **CAN over Serial / SLCAN**
  SLCAN adapters can be used directly via CAN over Serial / SLCAN, or via SocketCAN with some help from the slcand utility which can be found in the can-utils package.
  To create a socketcan interface for an SLCAN adapter run the following:
  slcand -f -o -c -s5 /dev/ttyAMA0
  ip link set up slcan0
  Names of the interfaces created by slcand match the slcan+ regex. If a custom name is required, it can be specified as the last argument. E.g.:
  slcand -f -o -c -s5 /dev/ttyAMA0 can0
  ip link set up can0

ASCII Command	CAN Bitrate
s0	10 Kbit/s
s1	20 Kbit/s
s2	50 Kbit/s
s3	100 Kbit/s
s4	125 Kbit/s
s5	250 Kbit/s
s6	500 Kbit/s
s7	800 Kbit/s
s8	1000 Kbit/s

- **PCAN**
  Kernels >= 3.4 supports the PCAN adapters natively via SocketCAN, so there is no need to install any drivers. The CAN interface can be brought like so:
  sudo modprobe peak_usb
  sudo modprobe peak_pci
  sudo ip link set can0 up type can bitrate 500000
- **Basic tools to display,record,generate and replay CAN traffic**
  - candump : display, filter and log CAN data to files.
  - canplayer : replay CAN logfiles.
  - cansend : send a single frame
  - cangen : generate (random) CAN traffic.
  - cansequence : send and check sequence of CAN frames with incrementing payload
  - cansniffer : display CAN data content differences
- **CAN Errors**
  A device may enter the "bus-off" state if too many errors occurred on the CAN bus. Then no more messages are received or sent. An automatic bus-off recovery can be enabled by setting the "restart-ms" to a non-zero value, e.g.:
  sudo ip link set canX type can restart-ms 100
  Alternatively, the application may realize the "bus-off" condition by monitoring CAN error frames and do a restart when appropriate with the command:
  ip link set canX type can restart
  Note:that a restart will also create a CAN error frame.

9.4.2 Wireshark

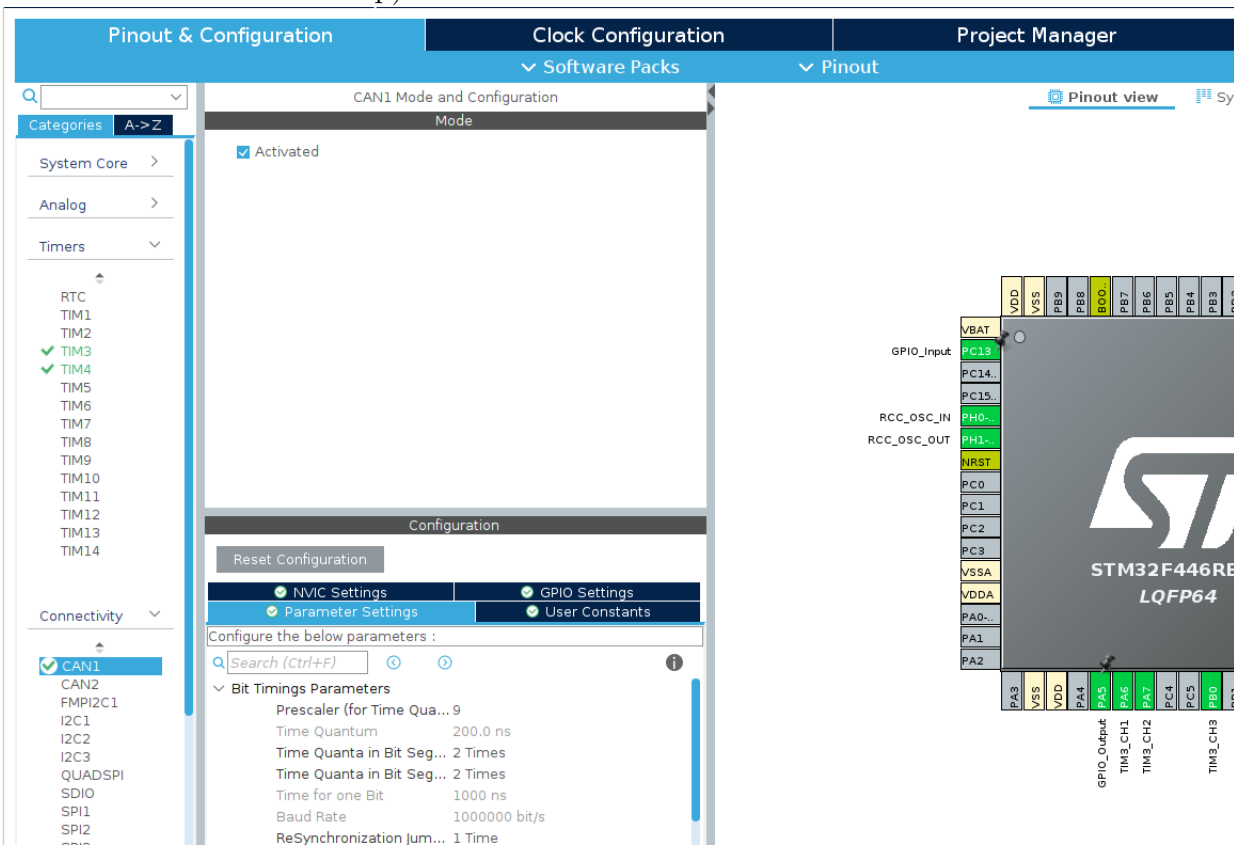
Wireshark supports socketcan and can be used to debug python-can messages. Fire it up and watch your new interface.
To spam a bus:

```python
import time
import can

bustype = 'socketcan'
channel = 'vcan0'

def producer(id):
    """:param id: Spam the bus with messages including the data id."""
    bus = can.Bus(channel=channel, interface=bustype)
    for i in range(10):
        msg = can.Message(arbitration_id=0xc0ffee, data=[id, i, 0, 1, 3, 1, 4, 1], is_extended_id=False)
        bus.send(msg)

    time.sleep(1)

producer(10)
```

10. Example1:Master(Nucleo446RET6) − > Slave(Nucleo446RET6)

1.1 How to Configure in STM32 for both (Master and Slave)

- **Step1:**Go to Clock Configure − > input 180MHz
- **Step2:**Go to Connectivity − > CAN1 − > Click Activated − > Input Value (Prescaler=9, tq1=2time,tq2=2time) to get buad_rate 1000000bit/s.
- **Step3:**Go to NVIC setting in CAN1 − > Click Interrupt Receive(CAN1 RX0 interrupt)
- **Step4:**Go to TIM3 − > chose PWM Generation(CH1,CH2,CH3) − > Set (PSC=0,ARR=65535,auto reload preload =Enable) to Blink LED RGB through PWM.
- **Step5:**Go to TIM4 − > Click Internal Clock − > Set (PSC=89,ARR=9999,auto reload preload =Enable) − > NVIC Setting(Click global Interrupt) to get Real time Clock 10ms of one loop)



1.2 How to Setup Code in main.c(Master)

- Step1:Input Variable

```
/* Private variables ---------------------------------------------------------*/
CAN_HandleTypeDef hcan1;

TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;

/* USER CODE BEGIN PV */
CAN_RxHeaderTypeDef RxHeader;
uint8_t RxData[8];
CAN_TxHeaderTypeDef TxHeader;
uint8_t TxData[8];
uint32_t TxMailbox;
int datacheck = 0;
float pwm_M1 = 0;
float pwm_M2 = 0;

float RxData1 = 0;
float RxData2 = 0;
float RxData3 = 0;
float RxData4 = 0;
float V1 = 0; // target speed of motor1
float V2 = 0; // target speed of motor2
float V3 = 0; // target speed of motor1
float V4 = 0; // target speed of motor2
uint16_t V1_out = 0;
uint16_t V2_out = 0;
uint16_t V3_out = 0;
uint16_t V4_out = 0;

uint8_t flag = 0;
uint8_t cntt;
uint8_t cnt;
/* USER CODE END PV */
```

- Step2:Input Function callback Receive

```
/* USER CODE BEGIN PFP */
float map(float Input, float Min_Input, float Max_Input, float Min_Output,
        float Max_Output) {

    return (float) ((Input - Min_Input) * (Max_Output - Min_Output)
            / (Max_Input - Min_Input) + Min_Output);
}
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0, &RxHeader, RxData);
    cntt++;
    while (cntt - 100 > 0) {
        //HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
        cntt = 0;
    }

    if (RxHeader.DLC == 8) {
        RxData1 = (RxData[0] << 8) | RxData[1];
        RxData2 = (RxData[2] << 8) | RxData[3];
        RxData3 = (RxData[4] << 8) | RxData[5];
        RxData4 = (RxData[6] << 8) | RxData[7];
        V1 = RxData1;
        V2 = RxData2;
        V3 = RxData3;
        V4 = RxData4;
        flag=1;
    }
}
```

- **Step3:** Input Frame for Transmition message

```
  /* USER CODE BEGIN 2 */
  HAL_CAN_Start(&hcan1);
  //Activate the notification
  HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);
  TxHeader.DLC =8; // data length
  TxHeader.IDE = CAN_ID_STD;
  TxHeader.RTR = CAN_RTR_DATA;
  TxHeader.StdId = 0x446; //Id 0x7FF
  HAL_TIM_Base_Start_IT(&htim4);
  HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
  HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
  V1_out = 65535;
  V2_out = 65535;
  V3_out = 65535;
  V4_out = 65535;
  /* USER CODE END 2 */
```

- **Step3:** Input filters in he way, that all received messages are accepted

```
304   /* USER CODE BEGIN CAN1_Init 2 */
305   CAN_FilterTypeDef canfilterconfig;
306
307     canfilterconfig.FilterActivation = CAN_FILTER_ENABLE;
308     canfilterconfig.FilterBank = 18; // anything between 0 to slaveStartFilterBank
309     canfilterconfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
310     canfilterconfig.FilterIdHigh = 0x445<< 5;
311     canfilterconfig.FilterIdLow = 0x0000;
312     canfilterconfig.FilterMaskIdHigh = 0x445<< 5;
313     canfilterconfig.FilterMaskIdLow = 0x0000;
314     canfilterconfig.FilterMode = CAN_FILTERMODE_IDMASK;
315     canfilterconfig.FilterScale = CAN_FILTERSCALE_32BIT;
316     canfilterconfig.SlaveStartFilterBank = 20; // how many filter to assign to the CAN1 (master Can)
317
318     HAL_CAN_ConfigFilter(&hcan1, &canfilterconfig);
319
320   /* USER CODE END CAN1_Init 2 */
```

- **Step4:**Push Button(GPIOC_PIN13) to Transmit data

```
470 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
471     if (htim->Instance == TIM4) {
472         TIM3->CCR1 = V1;
473         TIM3->CCR2 = V2;
474         TIM3->CCR3 = V3;
475         TxData[0] = ((V1_out & 0xFF00) >> 8);
476         TxData[1] = (V1_out & 0x00FF);
477         TxData[2] = ((V2_out & 0xFF00) >> 8);
478         TxData[3] = (V2_out & 0x00FF);
479         TxData[4] = ((V3_out & 0xFF00) >> 8);
480         TxData[5] = (V3_out & 0x00FF);
481         TxData[6] = ((V4_out & 0xFF00) >> 8);
482         TxData[7] = (V4_out & 0x00FF);
483         if (!(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)))
484         {
485             HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox);
486
487         }
488
489     }
490 }
```

1.3 How to Setup Code in main.c(Slave)

- Step1:Input Variable
  Same as Master
- Step2:Input Function callback Receive
  Same as Master
- **Step3:** Input Frame for Transmition message
  Same as Master but just change TxHeader.StdId from 0x446 to 0x445.
- **Step3:** Input filters in he way, that all received messages are accepted
  The same as to Master but just change (FilterIdHigh and FilterMaskIdHigh) from 0x445¡¡ 5 to 0x446¡¡ 5.
- **Step4:**Input Transmit data when this device Received.

```
469 /* USER CODE BEGIN 4 */
470 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
471     if (htim->Instance == TIM4) {
472         TIM3->CCR1 = V1;
473         TIM3->CCR2 = V2;
474         TIM3->CCR3 = V3;
475         TxData[0] = ((V1_out & 0xFF00) >> 8);
476         TxData[1] = (V1_out & 0x00FF);
477         TxData[2] = ((V2_out & 0xFF00) >> 8);
478         TxData[3] = (V2_out & 0x00FF);
479         TxData[4] = ((V3_out & 0xFF00) >> 8);
480         TxData[5] = (V3_out & 0x00FF);
481         TxData[6] = ((V4_out & 0xFF00) >> 8);
482         TxData[7] = (V4_out & 0x00FF);
483         if (flag==1)
484         {
485             HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox);
486             flag=0;
487
488         }
489
490     }
491 }
```

11. **Example2:Master(Nucleo446)− >Slave(STM32F103C8T6**

2.1 **How to Configure in STM32 for both (Master and Slave)**

- The Master as same Master(Example1).
- The Slave as same Slave(Example1) but just change Receive Interrupt to RX1 because Rx_Interrupt only one

2.2 **How to Setup Code in main.c(Master)**: As the same Master(Example1)

2.3 **How to Setup Code in main.c(Slave)**: As the same Slave(Example1) but you just change (filters in he way, that all received messages are accepted)

```
354   /* USER CODE BEGIN CAN_Init 2 */
355   CAN_FilterTypeDef canfilterconfig;
356
357     canfilterconfig.FilterActivation = CAN_FILTER_ENABLE;
358     canfilterconfig.FilterBank = 10;  // which filter bank to use from the assigned ones
359     canfilterconfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
360     canfilterconfig.FilterIdHigh = 0x446<<5;
361     canfilterconfig.FilterIdLow = 0;
362     canfilterconfig.FilterMaskIdHigh = 0x446<<5;
363     canfilterconfig.FilterMaskIdLow = 0x0000;
364     canfilterconfig.FilterMode = CAN_FILTERMODE_IDMASK;
365     canfilterconfig.FilterScale = CAN_FILTERSCALE_32BIT;
366     canfilterconfig.SlaveStartFilterBank = 0;  // doesn't matter in single can controllers
367
368     HAL_CAN_ConfigFilter(&hcan, &canfilterconfig);
369   /* USER CODE END CAN_Init 2 */
370
371 }
```

12. **Example3:Master(USB CAN Python)− > Slave(STM32F103C8T6)**

2.1 **How to configure USB CAN**: Use SocketCAN

- sudo ip link set can0 type can bitrate 1000000
- sudo ip link set up can0
- candump can0

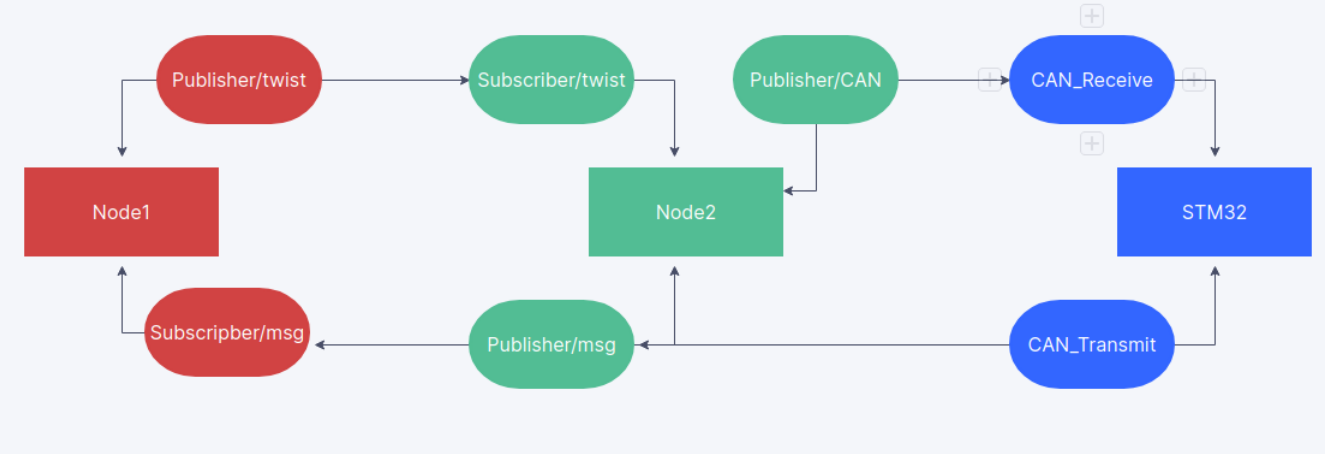
2.2 **How to Setup Code in python CAN**

```python
import can
import time
def map(Input, min_input, max_input, min_output, max_output):
    value = ((Input - min_input)*(max_output-min_output)/(max_input - min_input) + min_output)
    return int(value)
vx=int(input("Enter Velocity:\t"))
vy=int(input("Enter Velocity:\t"))
yaw=int(input("Enter Velocity:\t"))
pt=int(input("Enter point:\t"))
class can_node:
    def data(self,Vx,Vy,yaw,point):
        filters = [{"can_id": 0x103, "can_mask": 0x0000, "extended": False},]
        self.bus = can.interface.Bus(channel='can0', interface='socketcan',bitrate=1000000)
        self.TxData=[]
        self.TxData.append((Vx & 0xFF00) >> 8)
        self.TxData.append(Vx & 0x00FF)
        self.TxData.append((Vy & 0xFF00) >> 8)
        self.TxData.append(Vy & 0x00FF)
        self.TxData.append((yaw & 0xFF00) >> 8)
        self.TxData.append(yaw & 0x00FF)
        self.TxData.append((point & 0xFF00) >> 8)
        self.TxData.append(point & 0x00FF)
    def can_callback(self):
        mssage=can.Message(arbitration_id=0x111,data=self.TxData,is_extended_id=False)
        self.bus.send(mssage,0.1)
        V_back=[0,0,0,0]
        for i in range(2):
            try:
                print("Message sent on {}".format(self.bus.channel_info))
                msg=self.bus.recv(0.1) #time out
                print(msg)
                if (msg != None):
                    if msg.arbitration_id == 0x103:
                        V_back[0] = (msg.data[0] << 8) | msg.data[1]
                        V_back[1] = (msg.data[2] << 8) | msg.data[3]
                        V_back[2] = (msg.data[4] << 8) | msg.data[5]
                        V_back[3] = (msg.data[6] << 8) | msg.data[7]

                    elif msg.arbitration_id == 0x140:
                        V_back[2] = (msg.data[4] << 8) | msg.data[5]
                        V_back[3] = (msg.data[6] << 8) | msg.data[7]

            except can.CanOperationError:
                pass
        return V_back
if __name__ == "__main__":
    Vx=map(vx,0,255,0,65535)
    Vy=map(vy,0,255,0,65535)
    Yaw=map(yaw,0,255,0,65535)
    Pt=map(pt,0,255,0,65535)
    canode=can_node()
    canode.data(Vx,Vy,Yaw,Pt)
    Vback=canode.can_callback()
    print(Vback)
```

2.3 **How to Configure in STM32F103C8T6**:As the same Slave(Example2)

2.4 **How to setup code in main.c**:As the same Slave(Example2)

13. **Example4:Master(CAN_ROS2) − > Slave(STM32F103C8T6)**
Goall of this project : Node1(Publisher twist/cmd_vel) − > Node2(Subscriber twist/cmd_vel) take value linearX,Y,AngularZ and Iput this Value − > Value transmit of CAN_BUS to STM32 when CAN receive from STM32 It will convert value − > Node2(Pubisher Float32MultiArray/velocity − > Node1(Subscriber Float32MultiArray/velocity)



4.1 **How to configure USB CAN**: Use SocketCAN: As same Example3.

4.2 **How to Setup code in CAN_ROS2**

- **Step1:** Install and Create package python in ROS2(Foxy).
- **Step2:** Install library can and command to check can
  - sudo pip3 install canprog
  - sudo apt install python3-can
  - Then canprog seems to work:
    canprog -h
  - STM32 bootloader option:
    canprog stm32 [-h]
- **Step3:** Setup code in CAN_ROS2
  - First Create File in ROS2_python for Node1 which It has two topic such as:
    +Publisher twist(/cmd_vel)
    +Subscriber float32MultiArray(/velocity)

```python
import sys
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from std_msgs.msg import Float32MultiArray,Int32,Float32

#First:Publisher twist/cmd_vel and Seconde:Subsciber Int32/velocity
class PublishTwist(Node):
    def __init__(self):
        super().__init__('node1')

        #Subscripber Float32MultiArray/velocity
        self.sub_float_=self.create_subscription(Float32MultiArray, '/velocity',self.sub_float32,10)

        #Publisher twist/cmd_vel:
        timer_period = 0.01  # seconds
        self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 10)
        self.timer_ = self.create_timer(timer_period, self.publish_message)

    ###*** Publisher Twist message ***###
    def publish_message(self):
        message = Twist()
        message.linear.x = float(sys.argv[1])
        message.linear.y = float(sys.argv[2])
        message.angular.z = float(sys.argv[3])
        self.get_logger().info('Transmit to Node2 - Linear Velocity X : %f, Linear Velocity Y :
                               % (message.linear.x, message.linear.y, message.angular.z))
        self.publisher_.publish(message)

    ###*** Subscriber Float32MultiArray ***###
    def sub_float32(self,msg):
        self.x=msg.data[0]
        self.y=msg.data[1]
        self.yaw=msg.data[2]
        self.omega=msg.data[3]
        self.get_logger().info('Recieved From Node2 - Linear Velocity X : %f, Linear Velocity Y
                               % (self.x, self.y,self.yaw,self.omega))

def main(args=None):
    rclpy.init(args=args)
    Publisher_Twist = PublishTwist()
    rclpy.spin(Publisher_Twist)
    Publisher_Twist.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
      main()
```

– Second You create File in ROS2_python fo Node3 which It has two topic such as:
+Subsciber twist(/cmd_vel)
+Publisher CAN_BUS to STM32 through pythonCAN and receive from STM32
+Publisher Value that receive from STM32 float32MultiArray(/velocity)

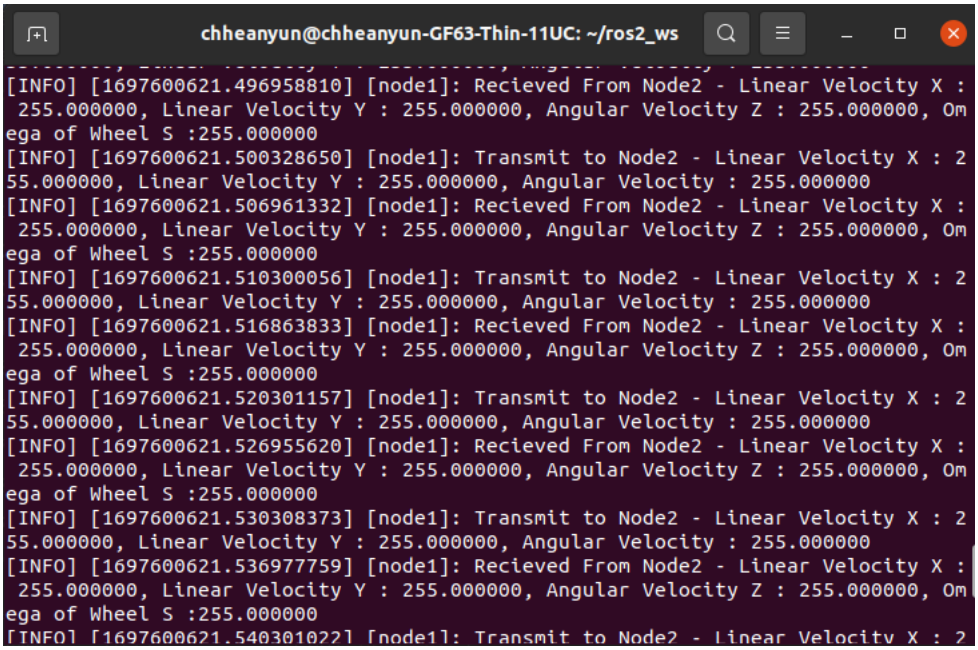
```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from std_msgs.msg import Float32MultiArray, UInt16MultiArray, Int8, Int16, Int32, UInt
import can
import numpy as np

##*First Subscripber Twist/cmd_vel and Second Publish msg CANBUS to STM32
##*And Third Publish Data Float32MultiArray/velocity
def map(Input, min_input, max_input, min_output, max_output):
    value = ((Input - min_input)*(max_output-min_output)/(max_input - min_input) + min_
    return value
class ros_node(Node):
    def __init__(self):
        super().__init__('can_node')
        #Publisher Float32MultiArray/velocity
        self.pub_timer = 0.01
        self.msg_pub_ = self.create_publisher(Float32MultiArray, '/velocity',10)
        self.msg_timer = self.create_timer(self.pub_timer, self.pub_callback)

        #Subscriber Twist/cmd_vel
        self.Input_ = self.create_subscription(Twist, '/cmd_vel', self.subCmdCB, 10)
        self.Input_

        #Publish CANBUS to STM32
        self.timer = 0.01
        self.bus =can.interface.Bus(channel='can0', interface ='socketcan', bitrate=100
        self.can_timer_ = self.create_timer(self.timer, self.timerCanCB)

        #self.publisher= self.create_publisher(String, 'velocity',10)
        # self.time =self.create_timer(self.pub_timer, self.pub_msg)
        self.i=0
        self.i=0
        self.Vx=0
        self.Vy=0
        self.Vyaw=0
        self.TxData = [128,0,128,0,128,0,128,0]
        self.V_back=[0,0,0,0]
        self.V_Back=[0,0,0,0]
        self.V1_back=10.0
        self.V2_back=10.0
        self.V3_back=10.0
        self.V4_back=10.0
        self.V1_Back=10.0
        self.V2_Back=10.0
        self.V3_Back=10.0
        self.V4_Back=10.0
    ###*** Subscriber Twist msg ***###
    def subCmdCB(self,msg):
        vx =msg.linear.x
        vy =msg.linear.y
        vyaw=msg.angular.z
        self.Vx = int(map(vx,0,255,0,65535))
        self.Vy = int(map(vy,0,255,0,65535))
        self.Vyaw = int(map(vyaw,0,255,0,65535))
        self.TxData[0] = ((self.Vx & 0xFF00) >> 8)
        self.TxData[1] = (self.Vx & 0x00FF)
        self.TxData[2] = ((self.Vy & 0xFF00) >> 8)
        self.TxData[3] = (self.Vy & 0x00FF)
        self.TxData[4] = ((self.Vyaw & 0xFF00) >> 8)
        self.TxData[5] = (self.Vyaw & 0x00FF)
        self.TxData[6] = ((self.Vx & 0xFF00) >> 8)
        self.TxData[7] = (self.Vx & 0x00FF)
```

```python
    ###*** Publisher CAN_BUS to STM32 ***###
    def timerCanCB(self):
        msg = can.Message(arbitration_id=0x111, is_extended_id=False, data=self.TxData
        self.bus.send(msg,0.01) #time out 10ms
        self.get_logger().info('Velocity transmit to STM32:[%f, %f, %f]'%(self.Vx,sel
        for i in range(2):
            try:
                can_msg =self.bus.recv(0.01)
                if(can_msg != None):
                    if can_msg.arbitration_id == 0x103:
                        self.V_back[0] = ((can_msg.data[0] << 8) | can_msg.data[1])
                        self.V_back[1] = ((can_msg.data[2] << 8) | can_msg.data[3])
                        self.V_back[2] = ((can_msg.data[4] << 8) | can_msg.data[5])
                        self.V_back[3] = ((can_msg.data[6] << 8)| can_msg.data[7])
                        self.V_back=[self.V_back[0],self.V_back[1],self.V_back[2],sel
                        print(self.V_back)
                    elif can_msg.arbitration_id == 0x104:
                        self.V_Back[0] = ((can_msg.data[0] << 8) | can_msg.data[1])
                        self.V_Back[1] = ((can_msg.data[2] << 8) | can_msg.data[3])
                        self.V_Back[2] = ((can_msg.data[4] << 8) | can_msg.data[5])
                        self.V_Back[3] = ((can_msg.data[6] << 8)| can_msg.data[7])
                        self.V_Back=[self.V_Back[0],self.V_Back[0],self.V_Back[0],sel
                        print(self.V_Back)
                    else:
                        self.get_logger().error('time out on msg recv!')

                except can.CanOperationError:
                    pass
                except can.CanOperationError:
                    pass
                self.V1_back=float(map(self.V_back[0],0,65535,0,255))
                self.V2_back=float(map(self.V_back[1],0,65535,0,255))
                self.V3_back=float(map(self.V_back[2],0,65535,0,255))
                self.V4_back=float(map(self.V_back[3],0,65535,0,255))
                self.V1_Back=float(map(self.V_Back[0],0,65535,0,255))
                self.V2_Back=float(map(self.V_Back[1],0,65535,0,255))
                self.V3_Back=float(map(self.V_Back[2],0,65535,0,255))
                self.V4_Back=float(map(self.V_Back[3],0,65535,0,255))
                V_BACK=[self.V1_back,self.V2_back,self.V1_Back,self.V2_Back]
                print(V_BACK)

    ###*** Publisher Float32MultiArray ***###
    def pub_callback(self):
        pub_msg=Float32MultiArray()
        pub_msg.data=[self.V1_back,self.V2_back,self.V1_Back,self.V2_Back]
        self.get_logger().info('publish Velocity Receive From STM32: [%f,%f,%f,%f]'
                               %(pub_msg.data[0],pub_msg.data[1],pub_msg.data[2],pub
        self.msg_pub_.publish(pub_msg)

def main(args=None):
    rclpy.init(args=args)
    ros = ros_node()
    rclpy.spin(ros)
    ros.destroy_node()
    rclpy.shutdown

if __name__=='__main__':
    main()
```

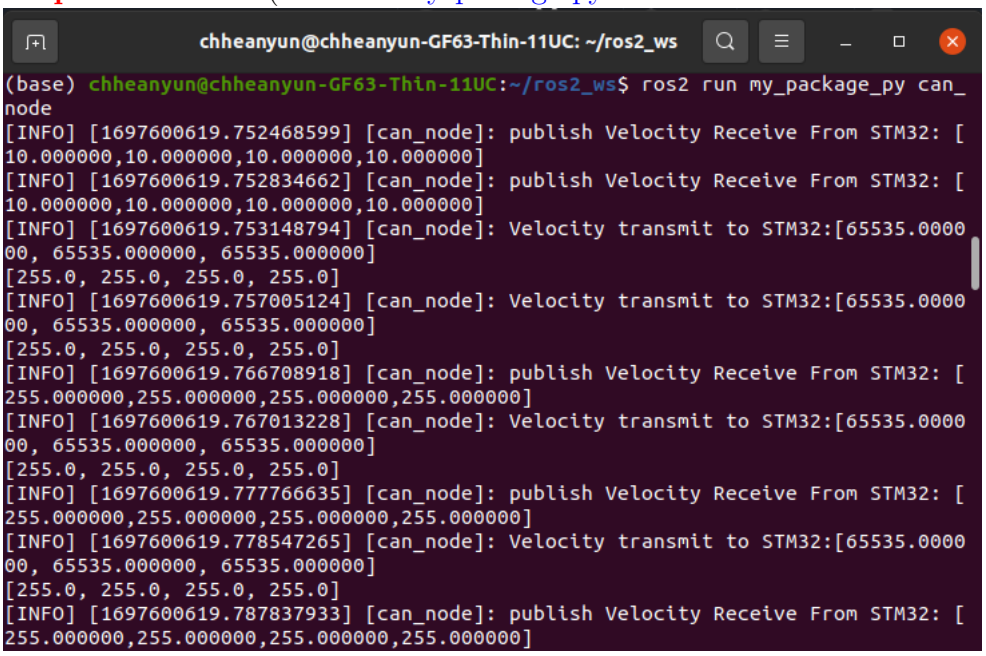
4.3 **How to Setup in STM32**:As the same Slave(Example2)

4.4 **Result**

- **Step1:**Run Node1 publish twist (ros2 run my_package_py can_node)



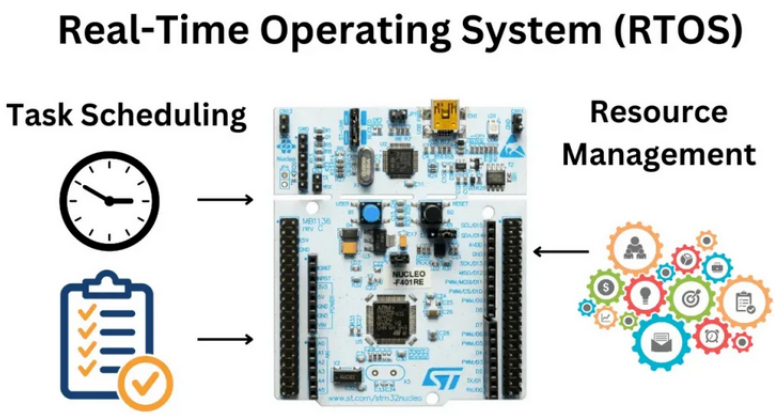
- **Step2:**Run Node2 (ros2 run my_package_py node1_float



XXVIII FreeRTOS Tutorial

1. **What is RTOS?**
   RTOS stands for Real-Time Operating System. It is a specialized operating system designed to handle real-time applications that have specific timing and responsiveness requirements

   - In an RTOS, tasks are scheduled and executed based on their priority and time requirements.
   - It ensures that critical tasks receive the necessary processing time and resources to meet their timing deadlines.
   - Real-time operating systems are commonly used in embedded systems, such as microcontrollers, where precise timing and responsiveness are crucial.



2. **How RTOS works?**
   Imagine you have an autonomous robot that performs various tasks in a warehouse, such as picking up items and placing them in specific locations.
   The robot's tasks include:

   - **Sensor Reading:** The robot needs to continuously read data from its sensors, such as distance sensors and cameras, to detect obstacles, recognize objects, and navigate the environment.
   - **Path Planning:** Based on the sensor data, the robot needs to determine the optimal path to navigate through the warehouse, avoiding obstacles and reaching the desired locations efficiently.
   - **Object Detection:** The robot needs to analyze the camera data to identify and classify objects in its surroundings, distinguishing between different items to be picked up or avoided.
   - **Motion Control:** Once the path is planned and objects are detected, the robot needs to control its motors and actuators to move smoothly and precisely, following the planned trajectory and performing tasks like grasping objects.

   In this scenario, an RTOS comes into play to manage the execution of these tasks efficiently and in a timely manner. Here's how it works:

   - **Task Scheduling:** The RTOS scheduler assigns priorities to each task based on their importance and time constraints. For example, the sensor reading task may have a higher priority to ensure real-time obstacle detection, while path planning can have a lower priority.
   - **Context Switching:** The RTOS handles context switching, which means it can pause the execution of one task and switch to another task seamlessly. This allows the robot to respond quickly to changing situations and events.
   - **Resource Management:** The RTOS manages the sharing of resources among tasks. For instance, if multiple tasks need to access the robot's motors simultaneously, the RTOS ensures that they can do so without conflicts by implementing synchronization mechanisms like semaphores or mutexes.
   - **Timeliness:** The RTOS guarantees that critical tasks are executed within their specified deadlines. For example, the motion control task needs to execute with precise timing to ensure the robot moves accurately and avoids collisions.

3. **Difference between super loop architecture and RTOS in Microcontroller**
When it comes to designing microcontroller-based systems, there are two common approaches for handling tasks and resources: super loop architecture and Real-Time Operating System (RTOS). Both have their advantages and drawbacks, and the choice between them depends on the specific application requirements and constraints.

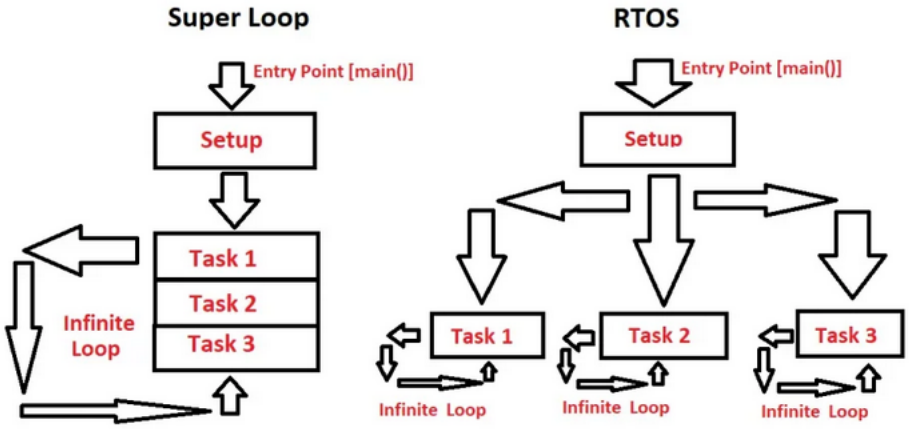


Figure: Super Loop vs Real Time Operating System (RTOS)

4. **What is FreeRTOS?**
FreeRTOS, which stands for Free Real-Time Operating System, is an open-source real-time operating system kernel designed specifically for embedded systems. It provides a compact and efficient solution for running multiple tasks concurrently while meeting strict timing requirements.
FreeRTOS supports a wide range of microcontrollers and microprocessors, including the popular STM32 family. It offers features such as task management, inter-task communication, synchronization mechanisms, timers, and memory management.
FreeRTOS also provides synchronization mechanisms like semaphores, mutexes, and queues, which allow tasks to communicate and share resources safely. This ensures coordinated execution and avoids conflicts between tasks.

4.1 The FreeRTOS Distribution

4.1.1 Understanding the FreeRTOS Distribution

- **FreeRTOS Port**:FreeRTOS can be built with approximately twenty different compilers, and can run on more than thirty different processor architectures.
- **FreeRTOS Source Files Common to All Ports**:
In addition to these two files, the following source files are located in the same directory:
  - queue.c: queue.c provides both queue and semaphore services, as described later in this book.
  - timers.c:timers.c provides software timer functionality, as described later in this book. It need only be included in the build if software timers are actually going to be used.
  - event_groups.c:event_groups.c provides event group functionality, as described later in this book. It need only be included in the build if event groups are actually going to be used.
  - croutine.c: croutine.c implements the FreeRTOS co-routine functionality. It need only be included in the build if co-routines are actually going to be

used. Co-routines were intended for use on very small microcontrollers, are rarely used now, and are therefore not maintained to the same level as other FreeRTOS features. Co-routines are not described in this book.

4.1.2 Demo Applications

- **The demo application has several purposes:**
  - To provide an example of a working and pre-configured project, with the correct files included, and the correct compiler options set.
  - To allow 'out of the box' experimentation with minimal setup or prior knowledge.
  - As a demonstration of how the FreeRTOS API can be used.
  - As a base from which real applications can be created.
- **The web page includes information on:**
  - How to locate the project file for the demo within the FreeRTOS directory structure.
  - Which hardware the project is configured to use.
  - How to set up the hardware for running the demo.
  - How to build the demo.
  - How the demo is expected to behave.

4.1.3 Creating a FreeRTOS Project

- **Creating a New Project from Scratch**
  As already mentioned, it is recommended that new projects are created from an existing demo project. If this is not desirable, then a new project can be created using the following procedure:
  - Using your chosen tool chain, create a new project that does not yet include any FreeRTOS source files.
  - Ensure the new project can be built, downloaded to your target hardware, and executed.
  - Only when you are sure you already have a working project, add the FreeRTOS source files detailed in Table 1 to the project.
  - Copy the FreeRTOSConfig.h header file used by the demo project provided for the port in use into the project directory.
  - Copy the compiler settings from the relevant demo project.
  - Install any FreeRTOS interrupt handlers that might be necessary. Use the web page that describes the port in use, and the demo project provided for the port in use, as a reference.

4.1.4 Data Types and Coding Style Guide

Data Types

- **TickType_t**
  - FreeRTOS configures a periodic interrupt called the tick interrupt.
  - The number of tick interrupts that have occurred since the FreeRTOS application started is called the tick count. The tick count is used as a measure of time.
  - The time between two tick interrupts is called the tick period. Times are specified as multiples of tick periods.
  - TickType_t is the data type used to hold the tick count value, and tospecify times.

- TickType_t can be either an unsigned 16-bit type, or an unsigned 32-bit type, depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. If configUSE_16_BIT_TICKS is set to 1, then TickType_t is defined as uint16_t. If configUSE_16_BIT_TICKS is set to 0 then TickType_t is defined as uint32_t.

- **BaseType_t**
  - This is always defined as the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture.
  - BaseType_t is generally used for return types that can take only a very limited range of values, and for pdTRUE/pdFALSE type Booleans.

  Variable Names
  Variables are prefixed with their type: 'c' for char, 's' for int16_t (short), 'l' int32_t (long), and 'x' for BaseType_t and any other non-standard types (structures, task handles, queue handles,etc.).
  Function Names
  Functions are prefixed with both the type they return, and the file they are defined within. For example:
  - vTaskPrioritySet() returns a void and is defined within task.c.
  - xQueueReceive() returns a variable of type BaseType_t and is defined within queue.c.
  - pvTimerGetTimerID() returns a pointer to void and is defined within timers.c.

4.2 Task Management

4.2.1 Chapter Introduction and Scope

Scope

This chapter aims to give readers a good understanding of:

- How FreeRTOS allocates processing time to each task within an application.
- How FreeRTOS chooses which task should execute at any given time.
- How the relative priority of each task affects system behavior.
- The states that a task can exist in.

Readers should also gain a good understanding of:

- How to implement tasks.
- How to create one or more instances of a task.
- How to use the task parameter.
- How to change the priority of a task that has already been created.
- How to delete a task.
- How to implement periodic processing using a task (software timers are discussed in a later chapter).
- When the idle task will execute and how it can be used.

4.2.2 Task Functions

- **The task function prototype**
  void ATaskFunction(void *pvParameters);
- **The xTaskCreate() API function prototype**
  BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,const char * const pcName,uint16_t usStackDepth,void *pvParameters,UBaseType_t uxPriority,TaskHandle_t *pxCreatedTask);

- **pvTaskCode**: The pvTaskCode parameter is simply a pointer to the function that implements the task (in effect, just the name of the function).
- **pcName**:A descriptive name for the task.
- **usStackDepth**: The usStackDepth value tells the kernel how large to make the stack. The value specifies the number of words the stack can hold, not the number of bytes. For example, if the stack is 32-bits wide and usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes).
- **pvParameters**: Task functions accept a parameter of type pointer to void (void*). The value assigned to pvParameters is the value passed into the task. Some examples in this book demonstrate how the parameter can be used.
- **uxPriority**: Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to(configMAX_PRIORITIES – 1), which is the highest priority.
- **pxCreatedTask**:pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL.
- **Returned value**:There are two possible return values:
  1.pdPASS:This indicates that the task has been created successfully.
  2.pdFAIL:his indicates that the task has not been created because there is insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.

- **The vTaskPrioritySet() API function prototype**
  void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority);
  - **pxTask**:The handle of the task whose priority is being modified (the subject task)—see the pxCreatedTask parameter of the xTaskCreate() API function for information on obtaining handles to tasks.
  - **uxNewPriority**:The priority to which the subject task is to be set.
- **The uxTaskPriorityGet() API function prototype**
  UBaseType_t uxTaskPriorityGet(TaskHandle_t pxTask);
- **The vTaskDelete() API function prototype**
  void vTaskDelete(TaskHandle_t pxTaskToDelete);

4.3 Queue Management

'Queues' provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism.

4.3.1 Chapter Introduction and Scope

This chapter aims to give readers a good understanding of:

- How to create a queue.
- How a queue manages the data it contains.
- How to send data to a queue.
- How to receive data from a queue.
- What it means to block on a queue.

- How to block on multiple queues.
- How to overwrite data in a queue.
- How to clear a queue.
- The effect of task priorities when writing to and reading from a queue.

4.3.2 Characteristics of a Queue

Data Storage

A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.

There are two ways in which queue behavior could have been implemented:
1. Queue by copy:Queuing by copy means the data sent to the queue is copied byte for byte into the queue.
2.Queue by reference:Queuing by reference means the queue only holds pointers to the data sent to the queue, not the data itself.

FreeRTOS uses the queue by copy method. Queuing by copy is considered to be simultaneously more powerful and simpler to use than queueing by reference because:

- Stack variable can be sent directly to a queue, even though the variable will not exist after the function in which it is declared has exited.
- Data can be sent to a queue without first allocating a buffer to hold the data, and then copying the data into the allocated buffer.
- The sending task can immediately re-use the variable or buffer that was sent to the queue.
- The sending task and the receiving task are completely de-coupled—the application designer does not need to concern themselves with which task 'owns' the data, or which task is responsible for releasing the data.
- Queuing by copy does not prevent the queue from also being used to queue by reference. For example, when the size of the data being queued makes it impractical to copy the data into the queue, then a pointer to the data can be copied into the queue instead.
- The RTOS takes complete responsibility for allocating the memory used to store data.
- In a memory protected system, the RAM that a task can access will be restricted. In that case queueing by reference could only be used if the sending and receiving task could both access the RAM in which the data was stored. Queuing by copy does not impose that restriction; the kernel always runs with full privileges, allowing a queue to be used to pass data across memory protection boundaries.

4.3.3 Using a Queue

- **The xQueueCreate() API function prototype**
  QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
  - uxQueueLength:The maximum number of items that the queue being created can hold at any one time.

- uxItemSize: The size in bytes of each data item that can be stored in the queue.
- Return Value: If NULL is returned, then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.

  A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

- **The xQueueSendToBack() and xQueueSendToFront() API Functions**
  1.BaseType_t xQueueSendToFront(QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait);
  2.BaseType_t xQueueSendToBack(QueueHandle_t xQueue,const void * pvItemToQueue,TickType_t xTicksToWait);
  - xQueue:The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.
  - pvItemToQueue:A pointer to the data to be copied into the queue.The size of each item that the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
  - xTicksToWait:The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.
  - Returned value:There are two possible return values:
    1. pdPASS:pdPASS will be returned only if data was successfully sent to the queue.
    2. errQUEUE_FULL:errQUEUE_FULL will be returned if data could not be written to the queue because the queue was already full.
- **The xQueueReceive() API Function**
  xQueueReceive() is used to receive (read) an item from a queue. The item that is received is removed from the queue.
  BaseType_t xQueueReceive(QueueHandle_t xQueue,void * const pvBuffer,TickType_t xTicksToWait);
  - pvBuffer:A pointer to the memory into which the received data will be copied.The size of each data item that the queue holds is set when the queue is created. The memory pointed to by pvBuffer must be at least large enough to hold that many bytes.
- **The uxQueueMessagesWaiting() API Function**
  UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue);
  - Returned value:The number of items that the queue being queried is currently holding. If zero is returned, then the queue is empty.

4.3.4 Receiving Data From Multiple Sources

It is common in FreeRTOS designs for a task to receive data from more than one source. The receiving task needs to know where the data came from to determine how the data should be processed. An easy design solution is to use a single queue to transfer structures with both the value of the data and the source of the data contained in the structure's fields.

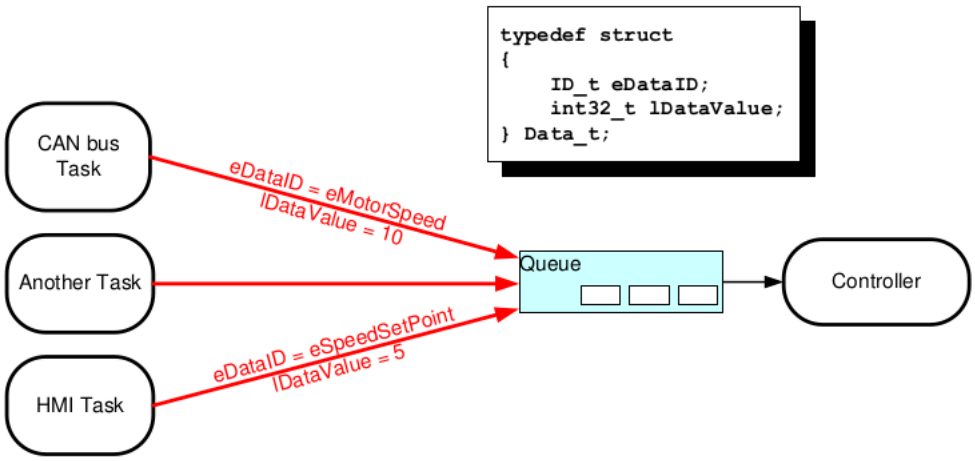


Figure 34. An example scenario where structures are sent on a queue

4.3.5 Working with Large or Variable Sized Data

Queuing Pointers

If the size of the data being stored in the queue is large, then it is preferable to use the queue to transfer pointers to the data, rather than copy the data itself into and out of the queue byte by byte. Transferring pointers is more efficient in both processing time and the amount of RAM required to create the queue. However, when queuing pointers, extreme care must be taken to ensure that:

- The owner of the RAM being pointed to is clearly defined.
- The RAM being pointed to remains valid.

How to use a queue to send a pointer to a buffer from one task to another:

- Creating a queue that holds pointers

/* Declare a variable of type QueueHandle/ito holdthehandleofthequeuebeingcreated.*
/
QueueHandle_t xPointerQueue;
/ * Createaqueuethatcanholdamaximumof5pointers, inthiscasecharacterpointers. * /
xPointerQueue = xQueueCreate(5, sizeof(char*)); Usingaqueuetosendapointertoabu

```
/* A task that obtains a buffer, writes a string to the buffer, then sends the address of the
buffer to the queue created in Listing 52. */
void vStringSendingTask( void *pvParameters )
{
char *pcStringToSend;
const size_t xMaxStringLength = 50;
BaseType_t xStringNumber = 0;

    for( ;; )
    {
        /* Obtain a buffer that is at least xMaxStringLength characters big.  The implementation
        of prvGetBuffer() is not shown - it might obtain the buffer from a pool of pre-allocated
        buffers, or just allocate the buffer dynamically. */
        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );

        /* Write a string into the buffer. */
        snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n", xStringNumber );

        /* Increment the counter so the string is different on each iteration of this task. */
        xStringNumber++;

        /* Send the address of the buffer to the queue that was created in Listing 52. The
        address of the buffer is stored in the pcStringToSend variable.*/
        xQueueSend( xPointerQueue,   /* The handle of the queue. */
                    &pcStringToSend, /* The address of the pointer that points to the buffer. */
                    portMAX_DELAY );
    }
}
```

- Using a queue to receive a pointer to a buffer

```
/* A task that receives the address of a buffer from the queue created in Listing 52, and
written to in Listing 53.  The buffer contains a string, which is printed out. */
void vStringReceivingTask( void *pvParameters )
{
char *pcReceivedString;

    for( ;; )
    {
        /* Receive the address of a buffer. */
        xQueueReceive( xPointerQueue,     /* The handle of the queue. */
                       &pcReceivedString, /* Store the buffer's address in pcReceivedString. */
                       portMAX_DELAY );

        /* The buffer holds a string, print it out. */
        vPrintString( pcReceivedString );

        /* The buffer is not required any more - release it so it can be freed, or re-used. */
        prvReleaseBuffer( pcReceivedString );
    }
}
```

Using a Queue to Send Different Types and Lengths of Data:
Previous sections have demonstrated two powerful design patterns; sending structures to a queue, and sending pointers to a queue. Combining those techniques allows a task to use a single queue to receive any data type from any data source.

- The structure used to send events to the TCP/IP stack task in FreeRTOS+TCP

```
/* A subset of the enumerated types used in the TCP/IP stack to identify events. */
typedef enum
{
    eNetworkDownEvent = 0, /* The network interface has been lost, or needs (re)connecting. */
    eNetworkRxEvent,       /* A packet has been received from the network. */
    eTCPAcceptEvent,       /* FreeRTOS_accept() called to accept or wait for a new client. */

    /* Other event types appear here but are not shown in this listing. */

} eIPEvent_t;


/* The structure that describes events, and is sent on a queue to the TCP/IP task. */
typedef struct IP_TASK_COMMANDS
{
    /* An enumerated type that identifies the event.  See the eIPEvent_t definition above. */
    eIPEvent_t eEventType;

    /* A generic pointer that can hold a value, or point to a buffer. */
    void *pvData;

} IPStackEvent_t;
```

- Pseudo code showing how an IPStackEvent_t structure is used to send data received from the network to the TCP/IP task

```
void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t *pxRxedData )
{
IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure.  The received data is stored in
    pxRxedData. */
    xEventStruct.eEventType = eNetworkRxEvent;
    xEventStruct.pvData = ( void * ) pxRxedData;

    /* Send the IPStackEvent_t structure to the TCP/IP task. */
    xSendEventStructToIPTask( &xEventStruct );
}
```

- Pseudo code showing how an IPStackEvent_t structure is used to send the handle of a socket that is accepting a connection to the TCP/IP task

```
void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )
{
IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure. */
    xEventStruct.eEventType = eTCPAcceptEvent;
    xEventStruct.pvData = ( void * ) xSocket;

    /* Send the IPStackEvent_t structure to the TCP/IP task. */
    xSendEventStructToIPTask( &xEventStruct );
}
```

- Pseudo code showing how an IPStackEvent_t structure is used to send a network down event to the TCP/IP task

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )
{
IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure. */
    xEventStruct.eEventType = eNetworkDownEvent;
    xEventStruct.pvData = NULL; /* Not used, but set to NULL for completeness. */

    /* Send the IPStackEvent_t structure to the TCP/IP task. */
    xSendEventStructToIPTask( &xEventStruct );
}
```

- Pseudo code showing how an IPStackEvent_t structure is received and processed

```
IPStackEvent_t xReceivedEvent;

    /* Block on the network event queue until either an event is received, or xNextIPSleep ticks
    pass without an event being received.  eEventType is set to eNoEvent in case the call to
    xQueueReceive() returns because it timed out, rather than because an event was received. */
    xReceivedEvent.eEventType = eNoEvent;
    xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );

    /* Which event was received, if any? */
    switch( xReceivedEvent.eEventType )
    {
        case eNetworkDownEvent :
            /* Attempt to (re)establish a connection.  This event is not associated with any
            data. */
            prvProcessNetworkDownEvent();
            break;

        case eNetworkRxEvent:
            /* The network interface has received a new packet.  A pointer to the received data
            is stored in the pvData member of the received IPStackEvent_t structure.  Process
            the received data. */
            prvHandleEthernetPacket( ( NetworkBufferDescriptor_t * )( xReceivedEvent.pvData ) );
            break;

        case eTCPAcceptEvent:
            /* The FreeRTOS_accept() API function was called.  The handle of the socket that is
            accepting a connection is stored in the pvData member of the received IPStackEvent_t
            structure. */
            xSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );
            xTCPCheckNewClient( pxSocket );
            break;

        /* Other event types are processed in the same way, but are not shown here. */

    }
```

4.3.6 Receiving From Multiple Queues

Queue Sets

Queue sets allow a task to receive data from more than one queue without the task polling each queue in turn to determine which, if any, contains data.

The following sections describe how to use a queue set by:

- Creating a queue set.
- Adding queues to the set.
- Reading from the queue set to determine which queues within the set contain data.

Note: If a queue is a member of a queue set then do not read data from the queue unless the queue's handle has first been read from the queue set.

- **The xQueueCreateSet() API Function**
  QueueSetHandle_t xQueueCreateSet(const UBaseType_t uxEventQueueLength);
  - uxEventQueueLength:
    When a queue that is a member of a queue set receives data, the handle of the receiving queue is sent to the queue set. uxEventQueueLength defines

the maximum number of queue handles the queue set being created can hold at any one time.

– Return Value:
If NULL is returned, then the queue set cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue set data structures and storage area.
A non-NULL value being returned indicates that the queue set has been created successfully. The returned value should be stored as the handle to the created queue set.

- **The xQueueAddToSet() API Function**
BaseType_t xQueueAddToSet(QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet);

– xQueueOrSemaphore: The handle of the queue or semaphore that is being added to the queue set.
Queue handles and semaphore handles can both be cast to the QueueSetMemberHandle_t type.

– xQueueSet: The handle of the queue set to which the queue or semaphore is being added.

– Return Value:
1. pdPASS= pdPASS will be returned only if the queue or semaphore was successfully added to the queue set.
2. pdFAIL= pdFAIL will be returned if the queue or semaphore could not be added to the queue set.
Queues and binary semaphores can only be added to a set when they are empty. Counting semaphores can only be added to a set when their count is zero. Queues and semaphores can only be a member of one set at a time.

The xQueueSelectFromSet() API Function
QueueSetMemberHandle_t xQueueSelectFromSet(QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait);

4.3.7 Using a Queue to Create a Mailbox

A queue may get described as a mailbox because of the way it is used in the application, rather than because it has a functional difference to a queue:

- A queue is used to send data from one task to another task, or from an interrupt service routine to a task. The sender places an item in the queue, and the receiver removes the item from the queue. The data passes through the queue from the sender to the receiver.
- A mailbox is used to hold data that can be read by any task, or any interrupt service routine. The data does not pass through the mailbox, but instead remains in the mailbox until it is overwritten. The sender overwrites the value in the mailbox. The receiver reads the value from the mailbox, but does not remove the value from the mailbox.

```
/* A mailbox can hold a fixed size data item.  The size of the data item is set
when the mailbox (queue) is created.  In this example the mailbox is created to
hold an Example_t structure.  Example_t includes a time stamp to allow the data held
in the mailbox to note the time at which the mailbox was last updated.  The time
stamp used in this example is for demonstration purposes only - a mailbox can hold
any data the application writer wants, and the data does not need to include a time
stamp. */
typedef struct xExampleStructure
{
    TickType_t xTimeStamp;
    uint32_t ulValue;
} Example_t;


/* A mailbox is a queue, so its handle is stored in a variable of type
QueueHandle_t. */
QueueHandle_t xMailbox;

void vAFunction( void )
{
    /* Create the queue that is going to be used as a mailbox.  The queue has a
    length of 1 to allow it to be used with the xQueueOverwrite() API function, which
    is described below. */
    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );

}
```

- **The xQueueOverwrite() API Function**
  BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const void * pvItemToQueue);
  - pvItemToQueue:A pointer to the data to be copied into the queue. The size of each item that the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
  - Returned value:xQueueOverwrite() will write to the queue even when the queue is full, so pdPASS is the only possible return value.
- **The xQueuePeek() API Function**
  xQueuePeek() is used to receive (read) an item from a queue without the item being removed from the queue. xQueuePeek() receives data from the head of the queue, without modifying the data stored in the queue, or the order in which data is stored in the queue.
  Note: Never call xQueuePeek() from an interrupt service routine. The interrupt-safe version xQueuePeekFromISR() should be used in its place.
  BaseType_t xQueuePeek(QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait);

```
/* A mailbox can hold a fixed size data item.  The size of the data item is set
when the mailbox (queue) is created.  In this example the mailbox is created to
hold an Example_t structure.  Example_t includes a time stamp to allow the data held
in the mailbox to note the time at which the mailbox was last updated.  The time
stamp used in this example is for demonstration purposes only - a mailbox can hold
any data the application writer wants, and the data does not need to include a time
stamp. */
typedef struct xExampleStructure
{
    TickType_t xTimeStamp;
    uint32_t ulValue;
} Example_t;


/* A mailbox is a queue, so its handle is stored in a variable of type
QueueHandle_t. */
QueueHandle_t xMailbox;

void vAFunction( void )
{
    /* Create the queue that is going to be used as a mailbox.  The queue has a
    length of 1 to allow it to be used with the xQueueOverwrite() API function, which
    is described below. */
    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );

}
```

4.4 Software Timer Management

4.4.1 Chapter Introduction and Scope

Software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency. The function executed by the software timer is called the software timer's callback function.

Software timers are implemented by, and are under the control of, the FreeRTOS kernel. They do not require hardware support, and are not related to hardware timers or hardware counters.

Note that, in line with the FreeRTOS philosophy of using innovative design to ensure maximum efficiency, software timers do not use any processing time unless a software timer callback function is actually executing.

This chapter aims to give readers a good understanding of:

- The characteristics of a software timer compared to the characteristics of a task.
- The RTOS daemon task.
- The timer command queue.
- The difference between a one shot software timer and a periodic software timer.
- How to create, start, reset and change the period of a software timer.

4.4.2 Software Timer Callback Functions

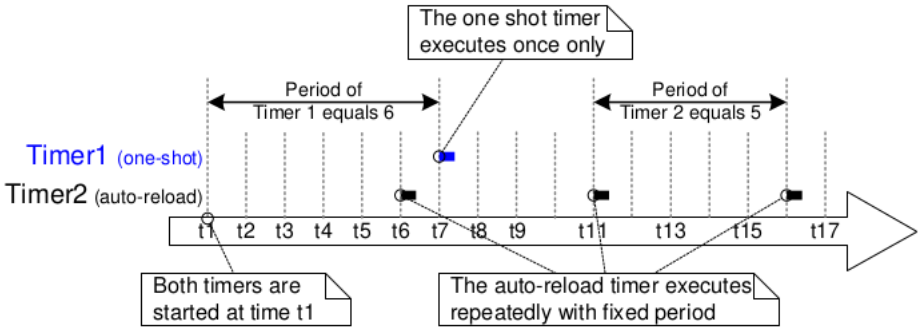
- **The software timer callback function**
  void ATimerCallback(TimerHandle_t xTimer);

4.4.3 Attributes and States of a Software Timer

- **Period of a Software Timer**: A software timer's 'period' is the time between the software timer being started, and the software timer's callback function executing.
- **One-shot and Auto-reload Timers**:
  1. One-shot timers= Once started, a one-shot timer will execute its callback

function once only. A one-shot timer can be restarted manually, but will not restart itself.
2. Auto-reload timers= Once started, an auto-reload timer will re-start itself each time it expires, resulting in periodic execution of its callback function.

The difference in behavior between one-shot and auto-reload software timers



- **Software Timer States**
  A software timer can be in one of the following two states:
  – Dormant=A Dormant software timer exists, and can be referenced by its handle, but is not running, so its callback functions will not execute.
  – Running=A Running software timer will execute its callback function after a time equal to its period has elapsed since the software timer entered the Running state, or since the software timer was last reset.

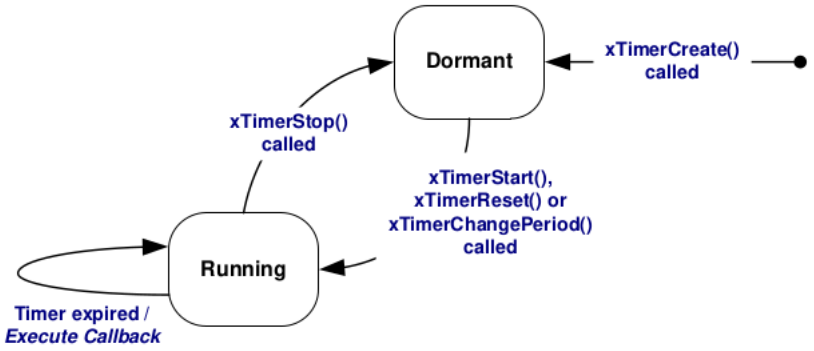


Figure 39 Auto-reload software timer states and transitions

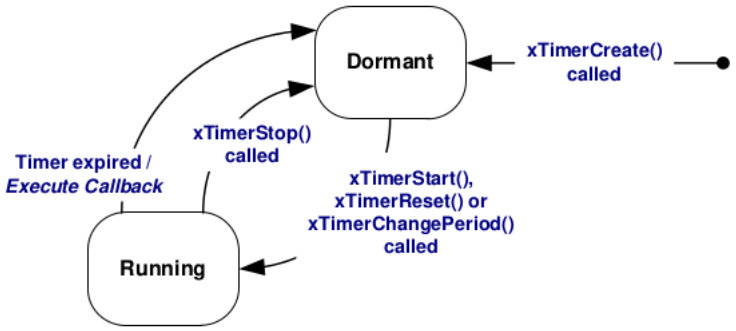


Figure 40 One-shot software timer states and transitions

4.4.4 The Context of a Software Timer

- **The RTOS Daemon (Timer Service) Task**
  The daemon task is a standard FreeRTOS task that is created automatically when the scheduler is started. Its priority and stack size are set by the configTIMER_TASK_PRIORITY and configcompile time configuration constants respectively. Both constants are defined within FreeRTOSConfig.h.
- **The Timer Command Queue**
  The timer command queue is a standard FreeRTOS queue that is created automatically when the scheduler is started. The length of the timer command queue is set by the configTIMER_QUEUE_LENGTH compile time configuration constant in FreeRTOSConfig.h.

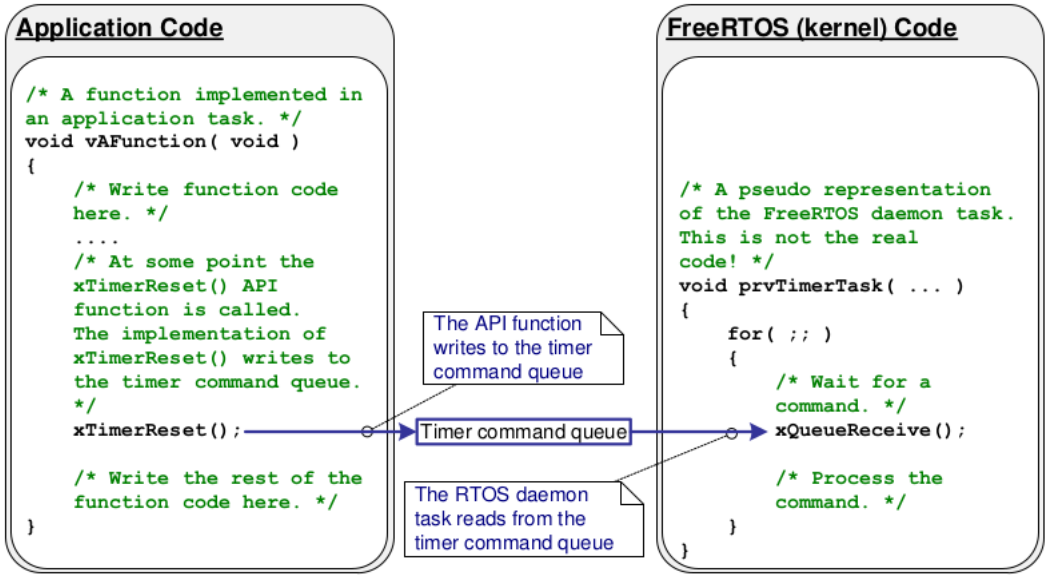


Figure 41 The timer command queue being used by a software timer API function to communicate with the RTOS daemon task

4.4.5 Creating and Starting a Software Timer

- **The xTimerCreate() API Function**
  xTimerCreate() is used to create a software timer and returns a TimerHandle_t to reference the software timer it creates. Software timers are created in the Dormant state.
  TimerHandle_t xTimerCreate(const char * const pcTimerName, TickType_t xTimerPeriodInTicks, UBaseType_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction);
  - pcTimerName: A descriptive name for the timer. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a timer by a human readable name is much simpler than attempting to identify it by its handle.
  - xTimerPeriodInTicks: The timer's period specified in ticks. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.
  - uxAutoReload: Set uxAutoReload to pdTRUE to create an auto-reload timer. Set uxAutoReload to pdFALSE to create a one-shot timer.
  - pvTimerID: pvTimerID sets an initial value for the ID of the task being created.
  - pxCallbackFunction: The pxCallbackFunction parameter is a pointer to

the function (in effect, just the function name) to use as the callback function for the software timer being created.

– Returned value:
1. If NULL is returned, then the software timer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the necessary data structure.
2. A non-NULL value being returned indicates that the software timer has been created successfully. The returned value is the handle of the created timer.

- **The xTimerStart() API Function**
xTimerStart() is used to start a software timer that is in the Dormant state, or reset (re-start) a software timer that is in the Running state.
BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);
  – xTimer: The handle of the software timer being started or reset. The handle will have been returned from the call to xTimerCreate() used to create the software timer.
  – xTicksToWait: xTicksToWait specifies the maximum amount of time the calling task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.
  – Returned value:
  1. pdPASS= pdPASS will be returned only if the 'start a timer' command was successfully sent to the timer command queue.
  2. pdFALSE= pdFALSE will be returned if the 'start a timer' command could not be written to the timer command queue because the queue was already full.
- **Example. Creating one-shot and auto-reload timers**
  – Creating and starting the timers used in Example

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
second respectively. */
#define mainONE_SHOT_TIMER_PERIOD     pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD  pdMS_TO_TICKS( 500 )

int main( void )
{
TimerHandle_t xAutoReloadTimer, xOneShotTimer;
BaseType_t xTimer1Started, xTimer2Started;

   /* Create the one shot timer, storing the handle to the created timer in xOneShotTimer. */
   xOneShotTimer = xTimerCreate(
                   /* Text name for the software timer - not used by FreeRTOS. */
                   "OneShot",
                   /* The software timer's period in ticks. */
                   mainONE_SHOT_TIMER_PERIOD,
                   /* Setting uxAutoRealod to pdFALSE creates a one-shot software timer. */
                   pdFALSE,
                   /* This example does not use the timer id. */
                   0,
                   /* The callback function to be used by the software timer being created. */
                   prvOneShotTimerCallback );

   /* Create the auto-reload timer, storing the handle to the created timer in xAutoReloadTimer. */
   xAutoReloadTimer = xTimerCreate(
                   /* Text name for the software timer - not used by FreeRTOS. */
                   "AutoReload",
                   /* The software timer's period in ticks. */
                   mainAUTO_RELOAD_TIMER_PERIOD,
                   /* Setting uxAutoRealod to pdTRUE creates an auto-reload timer. */
                   pdTRUE,
                   /* This example does not use the timer id. */
                   0,
                   /* The callback function to be used by the software timer being created. */
                   prvAutoReloadTimerCallback );

   /* Check the software timers were created. */
   if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
   {
       /* Start the software timers, using a block time of 0 (no block time).  The scheduler has
       not been started yet so any block time specified here would be ignored anyway. */
       xTimer1Started = xTimerStart( xOneShotTimer, 0 );
       xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

       /* The implementation of xTimerStart() uses the timer command queue, and xTimerStart()
       will fail if the timer command queue gets full.  The timer service task does not get
       created until the scheduler is started, so all commands sent to the command queue will
       stay in the queue until after the scheduler has been started.  Check both calls to
       xTimerStart() passed. */
       if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
       {
           /* Start the scheduler. */
           vTaskStartScheduler();
       }
   }

   /* As always, this line should not be reached. */
   for( ;; );
}
```

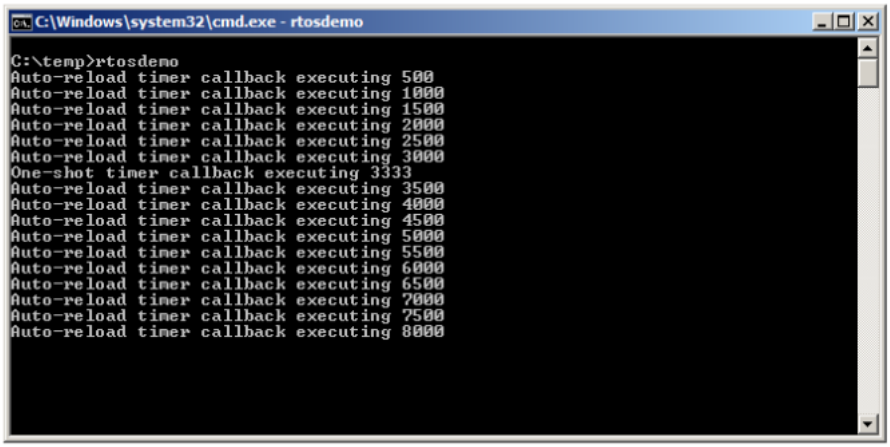
– The callback function used by the one -shot timer in Example

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
TickType_t xTimeNow;

    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

    /* File scope variable. */
    ulCallCount++;
}
```

– The callback function used by the auto -reload timer in Example

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
TickType_t xTimeNow;

    /* Obtain the current tick count. */
    xTimeNow = uxTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );

    ulCallCount++;
}
```

– The output produced when Example is executed



4.4.6 The Timer ID

Each software timer has an ID, which is a tag value that can be used by the application writer for any purpose. The ID is stored in a void pointer (void *), so can store an integer value directly, point to any other object, or be used as a function pointer.

- **The vTimerSetTimerID() API Function**
  void vTimerSetTimerID(const TimerHandle_t xTimer, void *pvNewID);
  _pvNewID: The value to which the software timer's ID will be set.
- **The pvTimerGetTimerID() API Function**
  void *pvTimerGetTimerID(TimerHandle_t xTimer);
  _Returned value: The ID of the software timer being queried.

4.5 Interrupt Management

5.1 Chapter Introduction and Scope

Event

In each case, a judgment has to be made as to the best event processing implementation strategy:

- How should the event be detected? Interrupts are normally used, but inputs can also be polled.
- When interrupts are used, how much processing should be performed inside the interrupt service routine (ISR), and how much outside? It is normally desirable to keep each ISR as short as possible.
- How events are communicated to the main (non-ISR) code, and how can this code be structured to best accommodate processing of potentially asynchronous occurrences?

It is important to draw a distinction between the priority of a task, and the priority of an interrupt:

- A task is a software feature that is unrelated to the hardware on which FreeRTOS is running. The priority of a task is assigned in software by the application writer, and a software algorithm (the scheduler) decides which task will be in the Running state.
- Although written in software, an interrupt service routine is a hardware feature because the hardware controls which interrupt service routine will run, and when it will run. Tasks will only run when there are no ISRs running, so the lowest priority interrupt will interrupt the highest priority task, and there is no way for a task to pre-empt an ISR.

Scope

This chapter aims to give readers a good understanding of:

- Which FreeRTOS API functions can be used from within an interrupt service routine.
- Methods of deferring interrupt processing to a task.
- How to create and use binary semaphores and counting semaphores.
- The differences between binary and counting semaphores.
- How to use a queue to pass data into and out of an interrupt service routine.
- The interrupt nesting model available with some FreeRTOS ports.

5.2 Using the FreeRTOS API from an ISR

The xHigherPriorityTaskWoken Parameter

There are several reasons why context switches do not occur automatically inside the interrupt safe version of an API function:

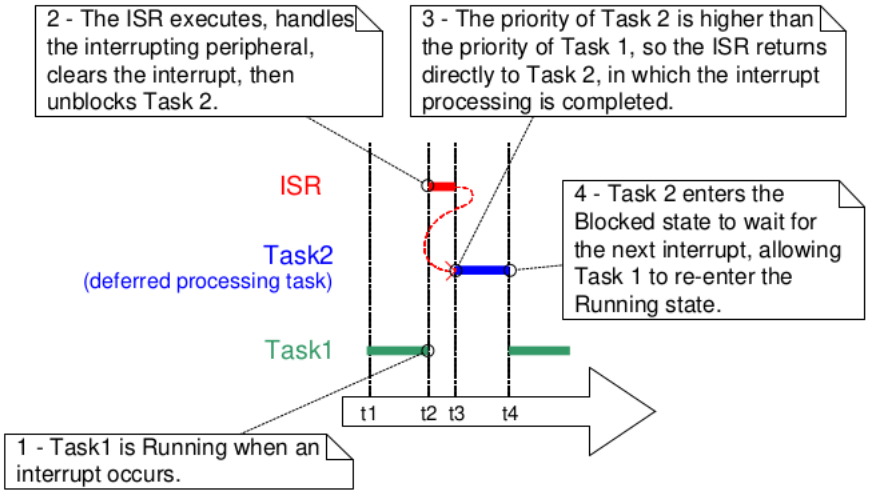
- Avoiding unnecessary context switches: An interrupt may execute more than once before it is necessary for a task to perform any processing. For example, consider a scenario where a task processes a string that was received by an interrupt driven UART; it would be wasteful for the UART ISR to switch to the task each time a character was received because the task would only have processing to perform after the complete string had been received.
- Control over the execution sequence:Interrupts can occur sporadically, and at unpredictable times. Expert FreeRTOS users may want to temporarily avoid an unpredictable switch to a different task at specific points in their application—although this can also be achieved using the FreeRTOS scheduler locking mechanism.
- Portability: It is the simplest mechanism that can be used across all FreeRTOS ports.
- Efficiency: Ports that target smaller processor architectures only allow a context switch to be requested at the very end of an ISR, and removing that restriction would require additional and more complex code. It also allows more than one call to a FreeRTOS API function within the same ISR without generating more than one request for a context switch within the same ISR.
- Execution in the RTOS tick interrupt: As will be seen later in this book, it is possible to add application code into the RTOS tick interrupt. The result of attempting a context switch inside the tick interrupt is dependent on the FreeRTOS port in use. At best, it will result in an unnecessary call to the scheduler.

5.3 Deferred Interrupt Processing

It is normally considered best practice to keep ISRs as short as possible. Reasons for this include:

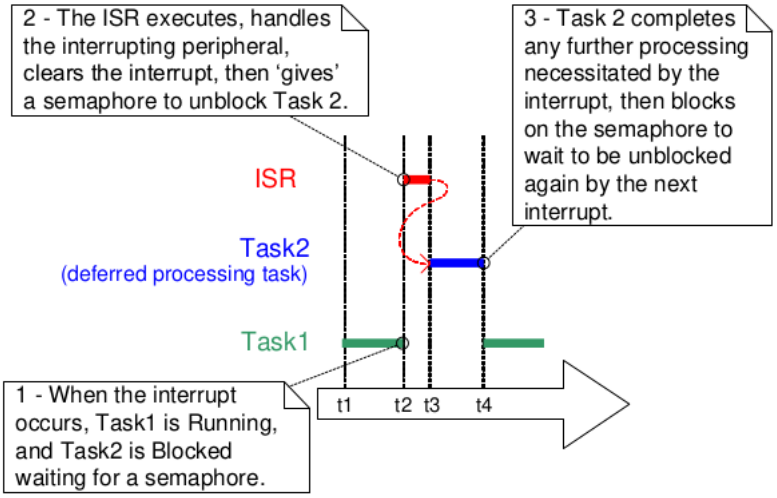
- Even if tasks have been assigned a very high priority, they will only run if no interrupts are being serviced by the hardware.
- ISRs can disrupt (add 'jitter' to) both the start time, and the execution time, of a task.
- Depending on the architecture on which FreeRTOS is running, it might not be possible to accept any new interrupts, or at least a subset of new interrupts, while an ISR is executing.

- The application writer needs to consider the consequences of, and guard against, resources such as variables, peripherals, and memory buffers being accessed by a task and an ISR at the same time.
- Some FreeRTOS ports allow interrupts to nest, but interrupt nesting can increase complexity and reduce predictability. The shorter an interrupt is, the less likely it is to nest.



5.4 Binary Semaphores Used for Synchronization

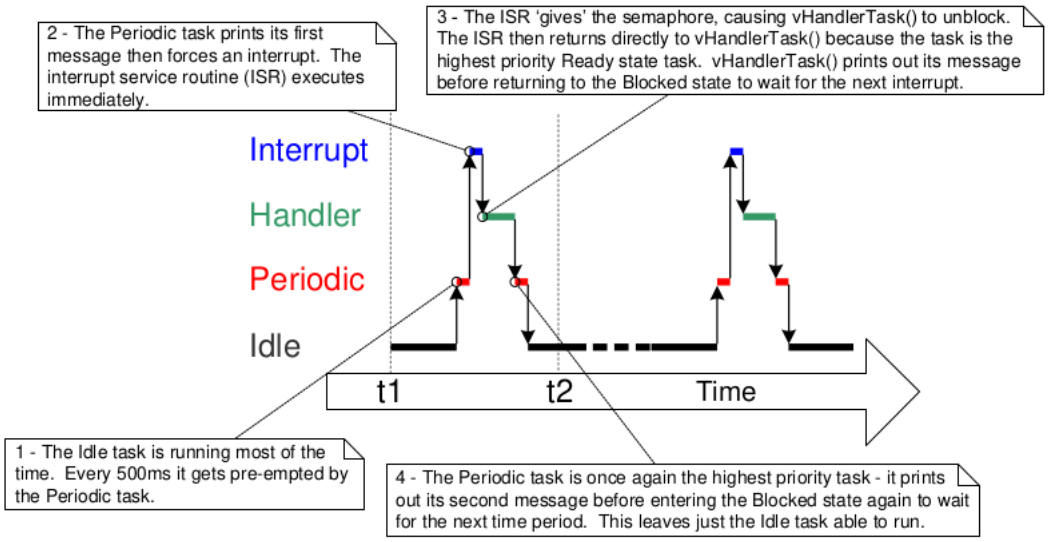
The interrupt safe version of the Binary Semaphore API can be used to unblock a task each time a particular interrupt occurs, effectively synchronizing the task with the interrupt. This allows the majority of the interrupt event processing to be implemented within the synchronized task, with only a very fast and short portion remaining directly in the ISR. As described in the previous section, the binary semaphore is used to 'defer' interrupt processing to a task .



- **The xSemaphoreCreateBinary() API Function**
  To create a binary semaphore, use the xSemaphoreCreateBinary() API function .
  SemaphoreHandle_t xSemaphoreCreateBinary(void);
  - Returned value:
    If NULL is returned, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the

semaphore data structures.
A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

- **The xSemaphoreTake() API Function**
  Taking' a semaphore means to 'obtain' or 'receive' the semaphore. The semaphore can be taken only if it is available.
  BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
  - xSemaphore:The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t. It must be explicitly created before it can be used.
  - xTicksToWait: The maximum amount of time the task should remain in the Blocked state to wait for the semaphore if it is not already available.
    1. If xTicksToWait is zero, then xSemaphoreTake() will return immediately if the semaphore is not available.
    2. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro **pdMS_TO_TICKS()** can be used to convert a time specified in milliseconds to a time specified in ticks.
    3. Setting xTicksToWait to **portMAX_DELAY** will cause the task to wait indefinitely (without a timeout) if INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.
  - Returned value:
    1. pdPASS=pdPASS is returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore.
    2. pdFALSE=The semaphore is not available.
- **The xSemaphoreGiveFromISR() API Function**
  Binary and counting semaphores can be 'given' using the xSemaphoreGiveFromISR() function.
  BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken);
  - pxHigherPriorityTaskWoken:It is possible that a single semaphore will have one or more tasks blocked on it waiting for the semaphore to become available.
  - Returned value:pdPASS and pdFAIL.
- **Example 16. Using a binary semaphore to synchronize a task with an interrupt**
  A simple periodic task is used to generate a software interrupt every 500 milliseconds. A software interrupt is used for convenience because of the complexity of hooking into a real interrupt in some target environments.

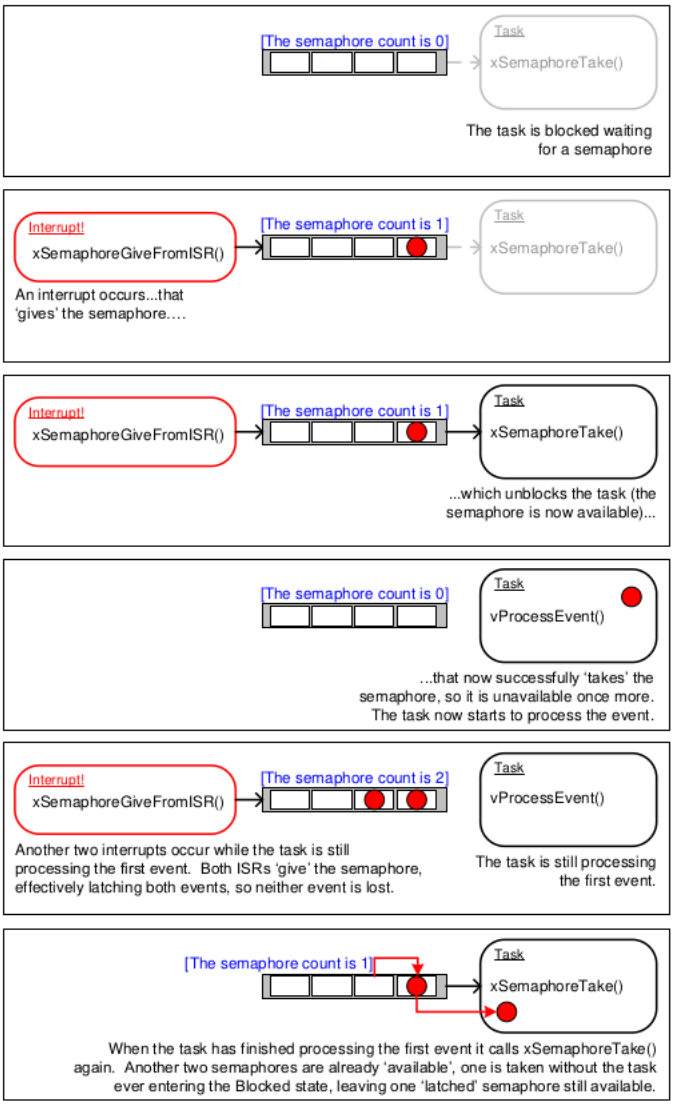


5.5 Counting Semaphores

Just as binary semaphores can be thought of as queues that have a length of one, counting semaphores can be thought of as queues that have a length of more than one. Tasks are not interested in the data that is stored in the queue—just the number of items in the queue. configUSE_COUNTING_SEMAPHORES must be set to 1 in FreeRTOSConfig.h for counting semaphores to be available.

Each time a counting semaphore is 'given', another space in its queue is used. The number of items in the queue is the semaphore's 'count' value.

Counting semaphores are typically used for two things:
1. Counting events: The count value is the difference between the number ofevents that have occurred and the number that have been processed.
Counting semaphores that are used to count events are created with an initial count value of zero.
2. Resource management: In this scenario, the count value indicates the number of resources available. To obtain control of a resource, a task must first obtain a semaphore—decrementing the semaphore's count value. resources. When the count value reaches zero, there are no free When a task finishes with the resource, it 'gives' the semaphore back— incrementing the semaphore's count value.
Figure: Using a counting semaphore to 'count' events



- **The xSemaphoreCreateCounting() API Function**
  SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);
  - uxMaxCount: The maximum value to which the semaphore will count. To continue the queue analogy, the uxMaxCount value is effectively the length of the queue.
    1. When the semaphore is to be used to count or latch events, uxMaxCount is the maximum number of events that can be latched.
    2. When the semaphore is to be used to manage access to a collection of resources, uxMaxCount should be set to the total number of resources that are available.
  - uxInitialCount: The initial count value of the semaphore after it has been created.
    1. When the semaphore is to be used to count or latch events, uxInitialCount should be set to zero—as, presumably, when the semaphore is created, no events have yet occurred.
    2. When the semaphore is to be used to manage access to a collection of resources, uxInitialCount should be set to equal uxMaxCount—as, presumably, when the semaphore is created, all the resources are available.
  - Returned value:
    1. If NULL is returned, the semaphore cannot be created because there is

insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
2. A non-NULL value being returned indicates that the semaphore has been created successfully. The returned value should be stored as the handle to the created semaphore.

5.6 Using Queues within an Interrupt Service Routine

The xQueueSendToFrontFromISR() and xQueueSendToBackFromISR() API Functions

- BaseType_t xQueueSendToFrontFromISR(QueueHandle_t xQueue, void *pvItemToQueue BaseType_t *pxHigherPriorityTaskWoken);
- BaseType_t xQueueSendToBackFromISR(QueueHandle_t xQueue, void *pvItemToQueue BaseType_t *pxHigherPriorityTaskWoken);
  - xQueue: The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.
  - pvItemToQueue: A pointer to the data that will be copied into the queue.The size of each item the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.
  - pxHigherPriorityTaskWoken: It is possible that a single queue will have one or more tasks blocked on it, waiting for data to become available.
  - Returned value: pdPASS and errQUEUE_FULL.

Considerations When Using a Queue From an ISR

The ISRs that use a queue in this manner are definitely not intended to represent an efficient design, and unless the data is arriving slowing, it is recommended that production code does not copy the technique. **Example : Sending and receiving on a queue from within an interrupt**

- The implementation of the task that writes to the queue

```
static void vIntegerGenerator( void *pvParameters )
{
TickType_t xLastExecutionTime;
uint32_t ulValueToSend = 0;
int i;

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* This is a periodic task.  Block until it is time to run again.  The task
        will execute every 200ms. */
        vTaskDelayUntil( &xLastExecutionTime, pdMS_TO_TICKS( 200 ) );

        /* Send five numbers to the queue, each value one higher than the previous
        value.  The numbers are read from the queue by the interrupt service routine.
        The interrupt service routine always empties the queue, so this task is
        guaranteed to be able to write all five values without needing to specify a
        block time. */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++;
        }

        /* Generate the interrupt so the interrupt service routine can read the
        values from the queue. The syntax used to generate a software interrupt is
        dependent on the FreeRTOS port being used.  The syntax used below can only be
        used with the FreeRTOS Windows port, in which such interrupts are only
        simulated.*/
        vPrintString( "Generator task - About to generate an interrupt.\r\n" );
        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );
        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

- The implementation of the interrupt service routine

```
static uint32_t ulExampleInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken;
uint32_t ulReceivedNumber;

/* The strings are declared static const to ensure they are not allocated on the
interrupt service routine's stack, and so exist even when the interrupt service
routine is not executing. */
static const char *pcStrings[] =
{
    "String 0\r\n",
    "String 1\r\n",
    "String 2\r\n",
    "String 3\r\n"
};

    /* As always, xHigherPriorityTaskWoken is initialized to pdFALSE to be able to
    detect it getting set to pdTRUE inside an interrupt safe API function. Note that
    as an interrupt safe API function can only set xHigherPriorityTaskWoken to
    pdTRUE, it is safe to use the same xHigherPriorityTaskWoken variable in both
    the call to xQueueReceiveFromISR() and the call to xQueueSendToBackFromISR(). */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Read from the queue until the queue is empty. */
    while( xQueueReceiveFromISR( xIntegerQueue,
                                 &ulReceivedNumber,
                                 &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
    {
        /* Truncate the received value to the last two bits (values 0 to 3
        inclusive), then use the truncated value as an index into the pcStrings[]
        array to select a string (char *) to send on the other queue. */
        ulReceivedNumber &= 0x03;
        xQueueSendToBackFromISR( xStringQueue,
                                 &pcStrings[ ulReceivedNumber ],
                                 &xHigherPriorityTaskWoken );
    }

    /* If receiving from xIntegerQueue caused a task to leave the Blocked state, and
    if the priority of the task that left the Blocked state is higher than the
    priority of the task in the Running state, then xHigherPriorityTaskWoken will
    have been set to pdTRUE inside xQueueReceiveFromISR().

    If sending to xStringQueue caused a task to leave the Blocked state, and if the
    priority of the task that left the Blocked state is higher than the priority of
    the task in the Running state, then xHigherPriorityTaskWoken will have been set
    to pdTRUE inside xQueueSendToBackFromISR().

    xHigherPriorityTaskWoken is used as the parameter to portYIELD_FROM_ISR().  If
    xHigherPriorityTaskWoken equals pdTRUE then calling portYIELD_FROM_ISR() will
    request a context switch.  If xHigherPriorityTaskWoken is still pdFALSE then
    calling portYIELD_FROM_ISR() will have no effect.

    The implementation of portYIELD_FROM_ISR() used by the Windows port includes a
    return statement, which is why this function does not explicitly return a
    value. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

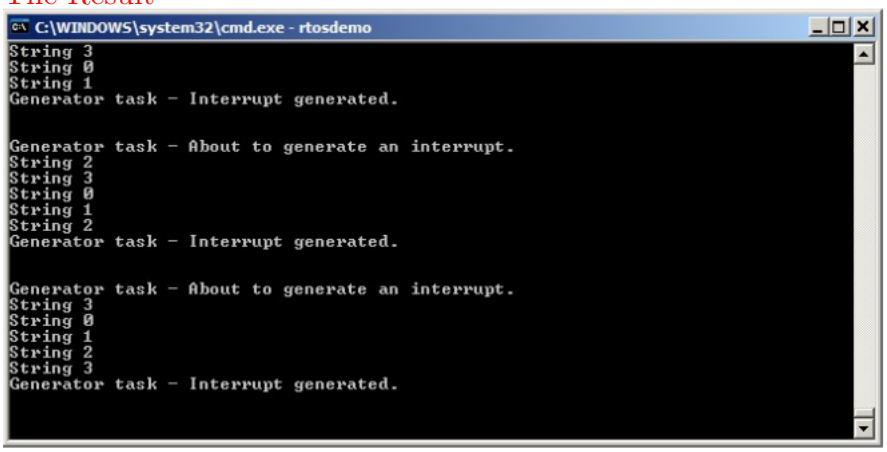
- The task that prints out the strings received from the interrupt service routine

```
static void vStringPrinter( void *pvParameters )
{
char *pcString;

    for( ;; )
    {
        /* Block on the queue to wait for data to arrive. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Print out the string received. */
        vPrintString( pcString );
    }
}
```

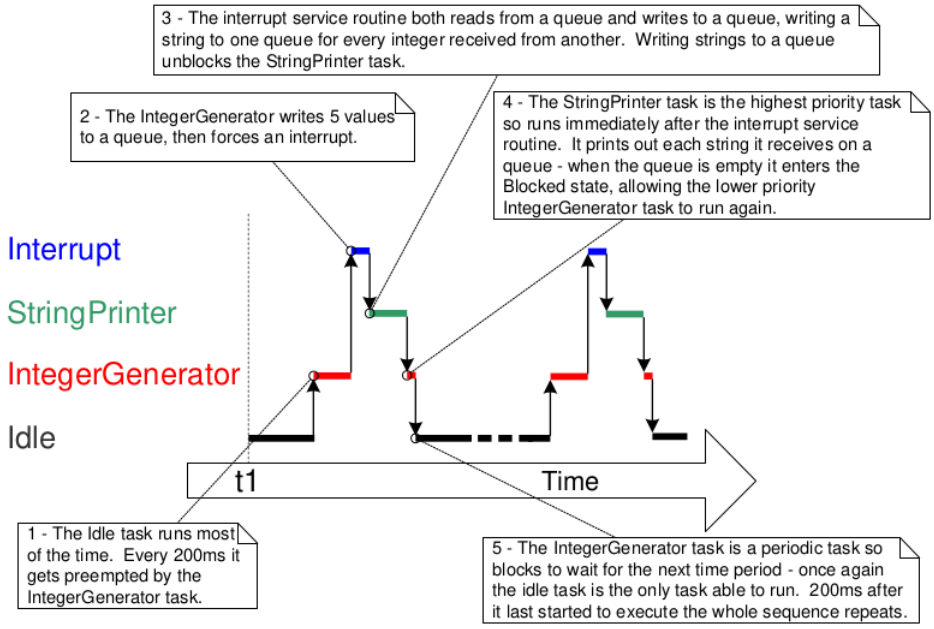
- The main() function

```
int main( void )
{
    /* Before a queue can be used it must first be created.  Create both queues used
    by this example.  One queue can hold variables of type uint32_t, the other queue
    can hold variables of type char*.  Both queues can hold a maximum of 10 items.  A
    real application should check the return values to ensure the queues have been
    successfully created. */
    xIntegerQueue = xQueueCreate( 10, sizeof( uint32_t ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Create the task that uses a queue to pass integers to the interrupt service
    routine.  The task is created at priority 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

    /* Create the task that prints out the strings sent to it from the interrupt
    service routine.  This task is created at the higher priority of 2. */
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

    /* Install the handler for the software interrupt.  The syntax necessary to do
    this is dependent on the FreeRTOS port being used.  The syntax shown here can
    only be used with the FreeRTOS Windows port, where such interrupts are only
    simulated. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks.  If main() does reach here then it is likely that there was
    insufficient heap memory available for the idle task to be created.  Chapter 2
    provides more information on heap memory management. */
    for( ;; );
}
```

- The Result



- The sequence of execution produced

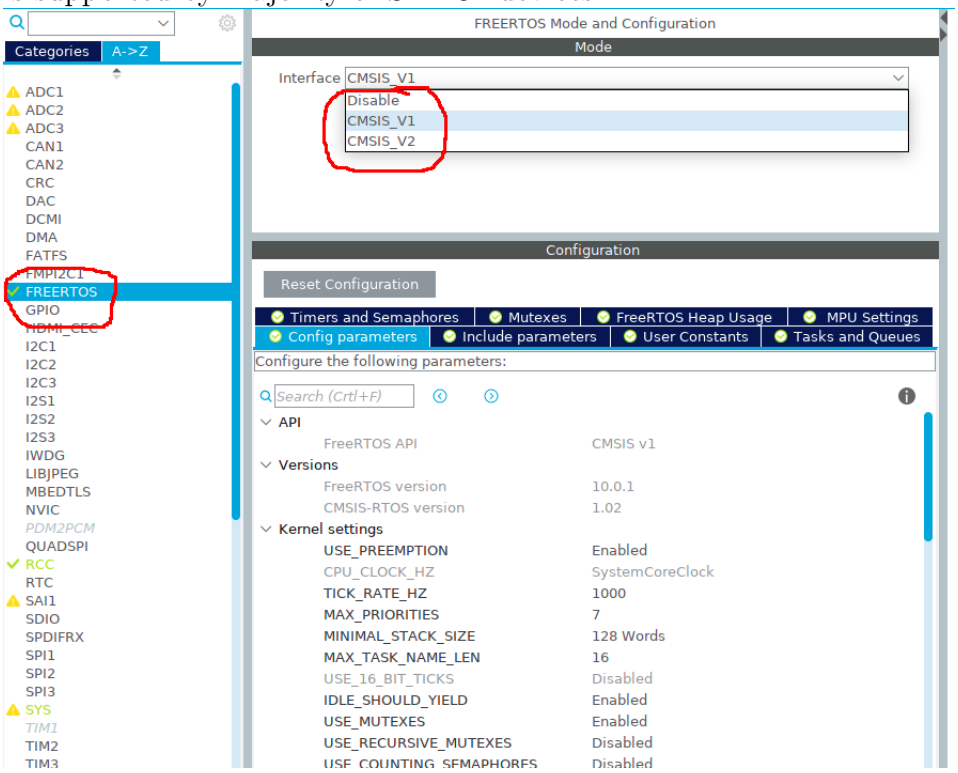


4.6 Resource Management

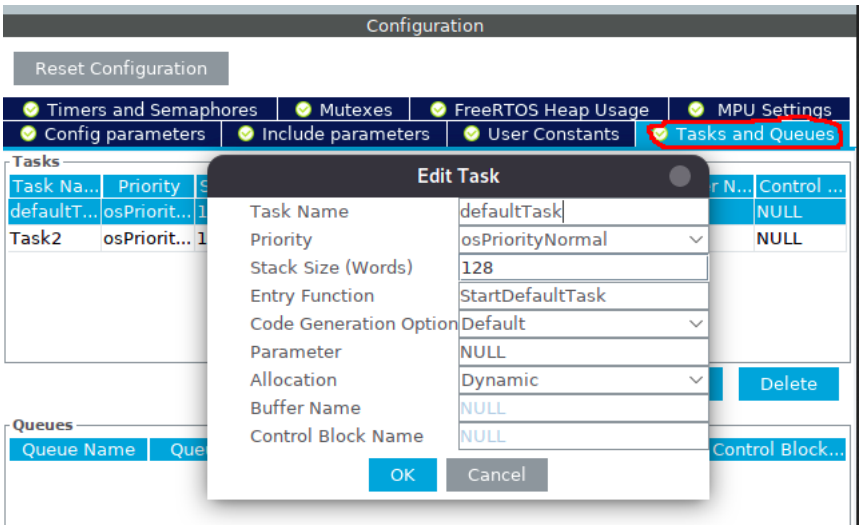
5. Example1:Basics of FreeRTOS

1.1 CubeMX Setup

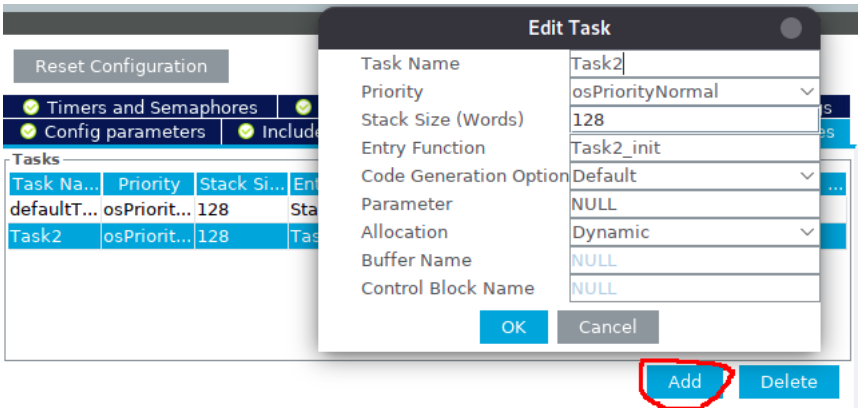
- Step1: Go to Middleware− >FreeRTOS after you choose CMSIS_V1 because it is supported by majority of STM32 devices.



- Step2: Go to 'tasks and queues'

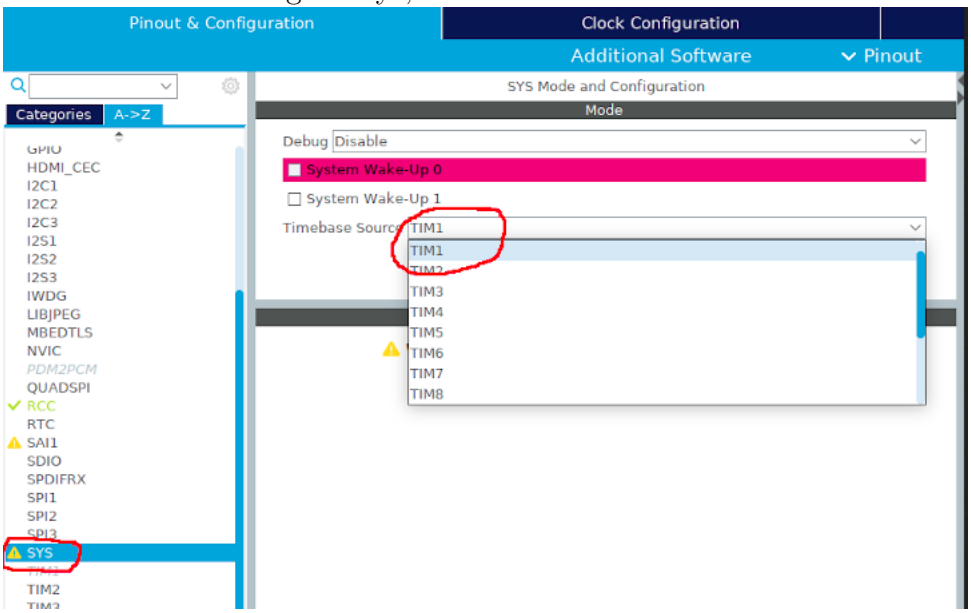


- Step3: For now, you have to focus only on the Task name, priority, and the entry function.



So I am calling it task 2, with normal priority, and the entry function is Task2_init

- Step4:Also one important thing about using RTOS is that, we can't use systick as the time base. So go to sys, and choose some other timebase as shown below:



- Step5: you are using UART2 for transmitting data, and pins PA0, and PA1 as output to blink led.

1.2 Writting Code

_To overcome this problem, we will use the RTOS. So, we have created 2 tasks, and if you scroll down the main.c file, you will see the entry functions for the tasks are defined there. After you input code like this:

```
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_0);
      //send_deftask();
      osDelay(1000);
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_Task2_init */
/**
* @brief Function implementing the Task2 thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_Task2_init */
void Task2_init(void const * argument)
{
  /* USER CODE BEGIN Task2_init */
  /* Infinite loop */
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_1);
    //send_task2();
    osDelay(1000);
  }
  /* USER CODE END Task2_init */
}
```

_In order to create a new Task, we have to follow some set of steps, and they are as follows:-

- Define a ThreadID for the task. This variable will store the unique ID of the task, once created. Later, all the operations will require this ID.
  /* USER CODE BEGIN PV */
  osThreadId Task3Handle;
  /* USER CODE END PV */
- Define the entry function for the task. This is the main function of the task. Your program will be written inside it. Remember that the tasks in the Free RTOS, are not designed to handle any return value. So, the entry function should always have an infinite loop, inside which, your whole program should be written.
  /* USER CODE BEGIN PFP */
  void Task3_init(void const * argument);
  /* USER CODE END PFP */
  void Task3_init(void const * argument)
  {
  while(1)
  {
  osDelay(3000);
  }
  }
- Inside our main function, we need to define the task first and than create it.
  /* USER CODE BEGIN RTOS_THREADS */
  /* add threads, ... */
  //define threads
  osThreadDef(Task3,Task3_init,osPriorityNormal,0,128);
  Task3Handle = osThreadCreate(osThread (Task3), NULL);
  /* USER CODE END RTOS_THREADS */
  _osThreadDef takes the parameters as the name of the task, the entry function,

the priority, instance, and the stack size.
_After the task is defined, we can create it using osThreadCreate, and assign the ID to the Task3Handle.

_After you want to send data through USART in each Task.you can follow like this:

- Step1:You create function for sending data

```c
/* Private user code ---------------------------------------------
 /* USER CODE BEGIN 0 */
void send_deftask (void)
 {
     uint8_t data[]="Hello from deftask\n";
     HAL_UART_Transmit(&huart2, data, sizeof(data), 500);
 }
void send_task2 (void)
 {
     uint8_t data[]="Hello from Task2\n";
     HAL_UART_Transmit(&huart2, data, sizeof(data), 500);
 }
void send_task3 (void)
 {
     uint8_t data[]="Hello from Task3\n";
     HAL_UART_Transmit(&huart2, data, sizeof(data), 500);
 }
 /* USER CODE END 0 */
```

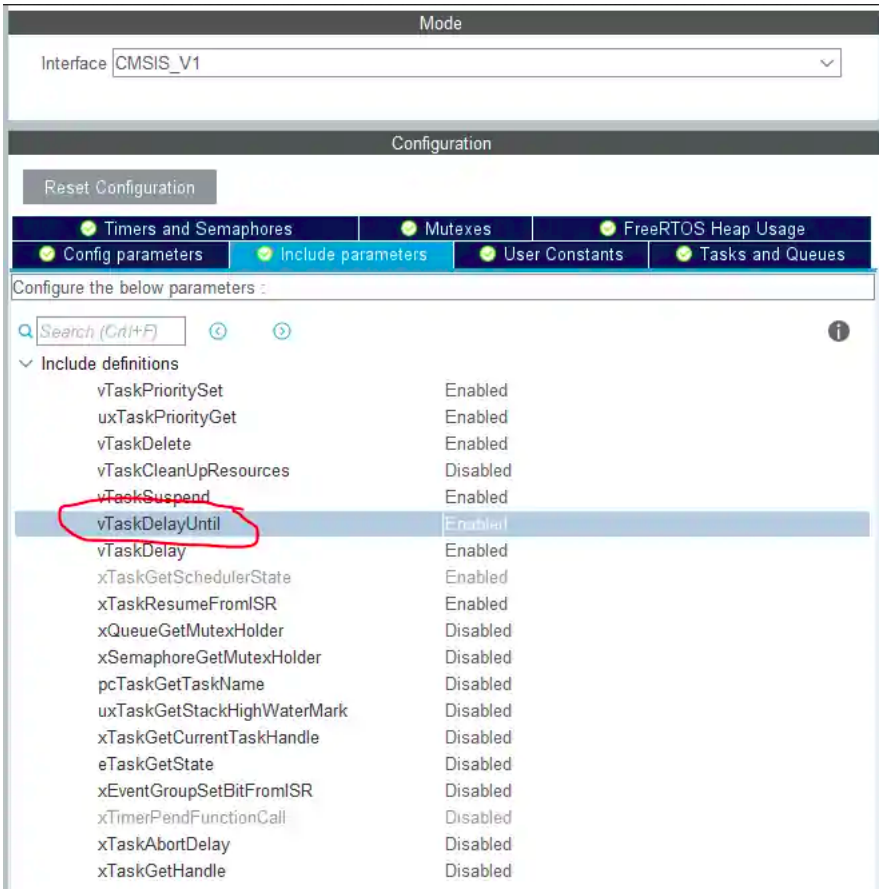
- Step2:You input function to each Task.

```c
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
      //HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_0);
      send_deftask();
      osDelay(1000);
  }
  /* USER CODE END 5 */
}


/* USER CODE END Header_Task2_init */
void Task2_init(void const * argument)
{
  /* USER CODE BEGIN Task2_init */
  /* Infinite loop */
  for(;;)
  {
    //HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_1);
    send_task2();
    osDelay(1000);
  }
  /* USER CODE END Task2_init */
}

void Task3_init(void const * argument)
{
    while(1)
    {
        send_task3();
        osDelay(3000);
    }
}
```

6. Example2:Task Operations

2.1 Setup

- Step1: Go to Middleware− >FreeRTOS after you choose CMSIS_V1 because it is supported by majority of STM32 devices.
- Step2: In the include parameters tab, include vTaskDelayUntill. I will show you how this works in a while.



2.2 Create TASK

Creating a new Task involves certain set of steps. They are as follows:

- Step1:define a task handler. This will also store the ID of the created task.
  /* USER CODE BEGIN PV */
  osThreadId Task2Handle;
  /* USER CODE END PV */
- Step2: inside the main function, define the thread using osThreadDef.
  /* USER CODE BEGIN RTOS_THREADS */
  /* add threads, ... */asdf
  osThreadDef(Task2,Task2_init,osPriorityNormal,0,128);
  Task3Handle = osThreadCreate(osThread (Task2), NULL);
  /* USER CODE END RTOS_THREADS */
  _Here Task2 is the name of the task to be created
  _task2_init is the entry function of the task.
  _osPriorityNormal is the Priority of the task
- Step3:we will write the code inside the entry function of the task, that we want the task to perform.
  /* USER CODE END PFP */
  void Task2_init(void const * argument)
  {
  while(1)
  {
  printf (”Task2 = %d”, indx++);
  osDelay(2000);
  }
  }

2.3 Suspend and Resume Tasks

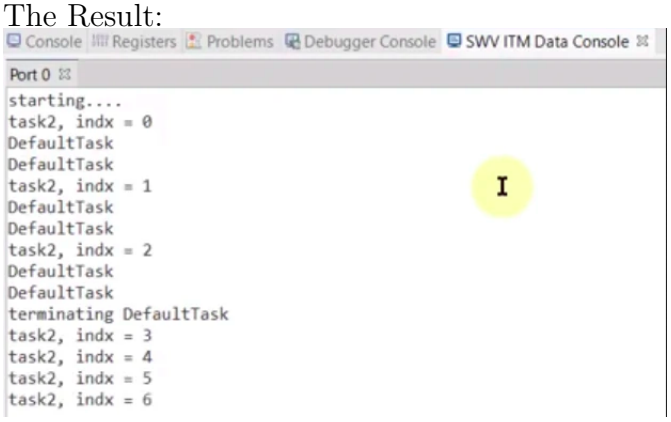
```
void task2_init (void const * argument)
{
    while (1)
    {
        printf ("Task2 = %d\n", indx++);
            osDelay(2000);

        if (indx==4)
        {
               printf ("suspending DefaultTask\n");
                   osThreadSuspend(defaultTaskHandle);

        }
        if (indx ==7)
        {
           printf ("Resuming DefaultTask\n");
           osThreadResume(defaultTaskHandle);
            }
    }
}
```

- I am using a particular condition to suspend the DefaultTask. Note that the Thread ID of the Default Task is passed to the osThreadSuspend.
- To resume this suspended Task, we will use another function called osThread-Resume. Again, the parameter to this function is also the Thread ID of the suspended Task.

2.4 Terminating the Task

```
void task2_init (void const * argument)
{
    while (1)
    {
        printf ("Task2 = %d\n", indx++);
        osDelay(2000);

        if (indx == 3)
        {
            printf ("Terminating DefaultTask\n");
            osThreadTerminate(defaultTaskHandle);
        }
    }
}
```

The Task is terminated, it can't be resumed. To terminate the task, we will use the osThreadTerminate function, whose parameter is the Thread ID of the thread, that you want to terminate.
The Result:



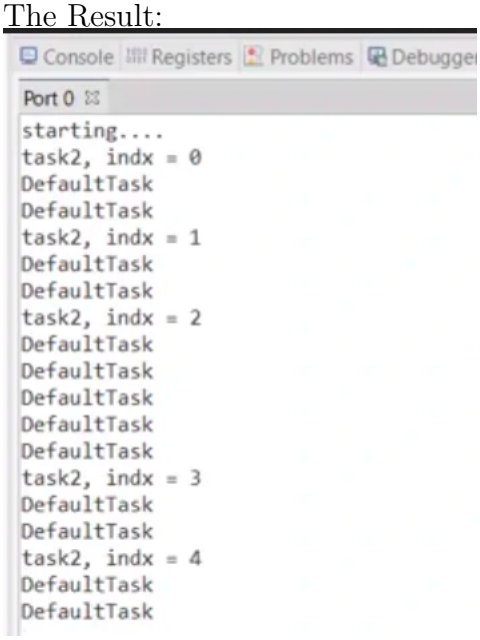
2.5 Block the task for some time

This is like, auto resuming of the task after some particular amount of time. We

will use the function osDelayUntil to do so. The arguments of this function as as follows:
osDelayUntil (uint32_t *PreviousWakeTime, uint32_t millisec)PreviousWakeTime Pointer to a variable that holds the time at which the task was last unblocked.

```
void task2_init (void const * argument)
{
    while (1)
    {
        printf ("Task2 = %d\n", indx++);
        osDelay(2000);

        if (indx == 3)
        {
          uint32_t PreviousWakeTime = osKernelSysTick();
          osDelayUntil(&PreviousWakeTime, 4000);
        }
    }
}
```

The Result:



7. **Example3: How to use Binary Semaphore**
Semaphores are basically used to synchronize tasks with other events in the system. In FreeRTOS, semaphores are implemented based on queue mechanism. There are 4 types of semaphores in FreeRTOS:

- Binary Semaphore
- Counting Semaphore
- Mutex
- Recursive

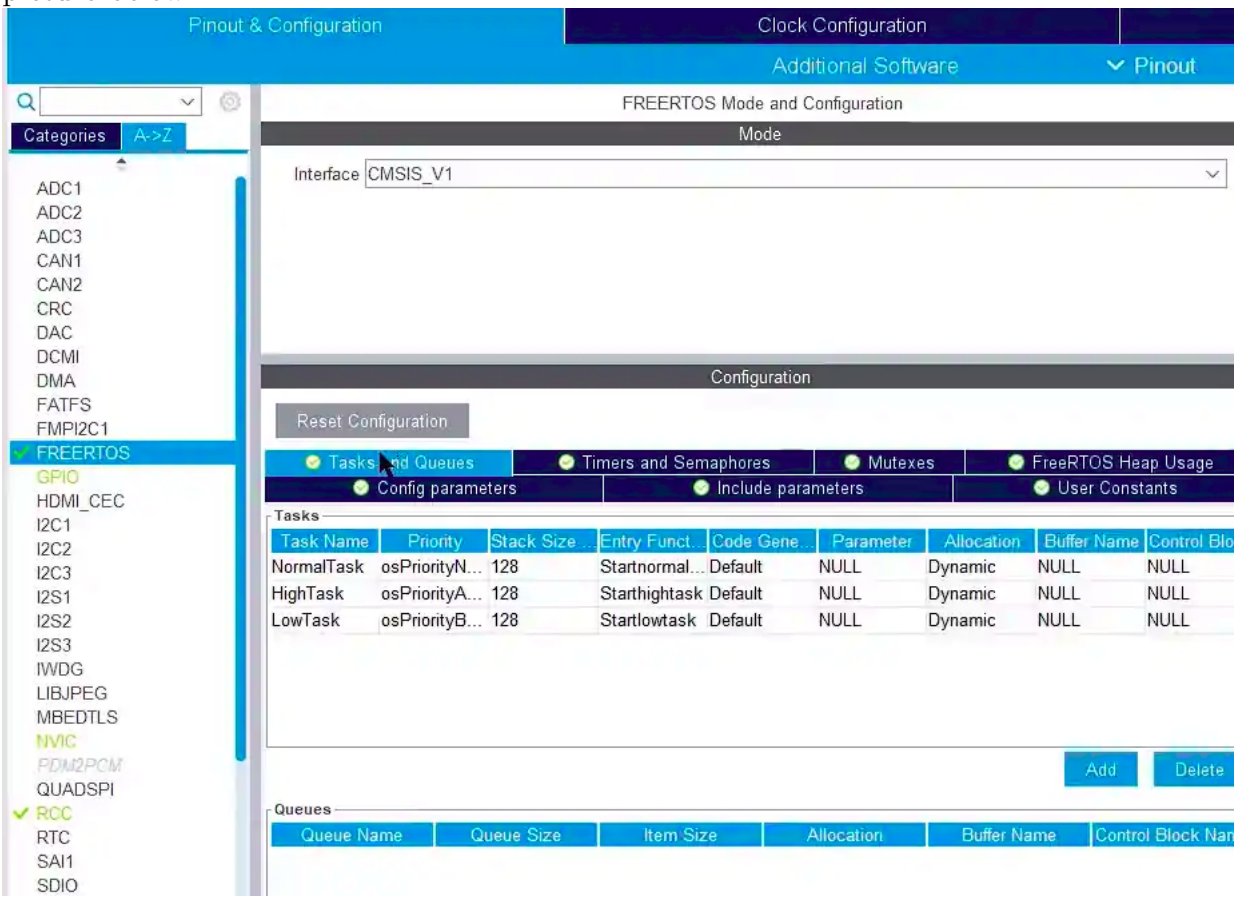
7.1 HOW DOES IT WORK

The working of Binary Semaphore is pretty straight forward. A Binary Semaphore is called Binary because either it is there (‘1’) or it is not (‘0’). There is no third condition in it. So, a Task either have the semaphore or it simply doesn’t. For a Task, we can create a condition that it must have the semaphore, in order to execute itself. Therefore, if the Task don’t have the semaphore, it have no other option but to wait for it to be released by the Task currently having the semaphore.

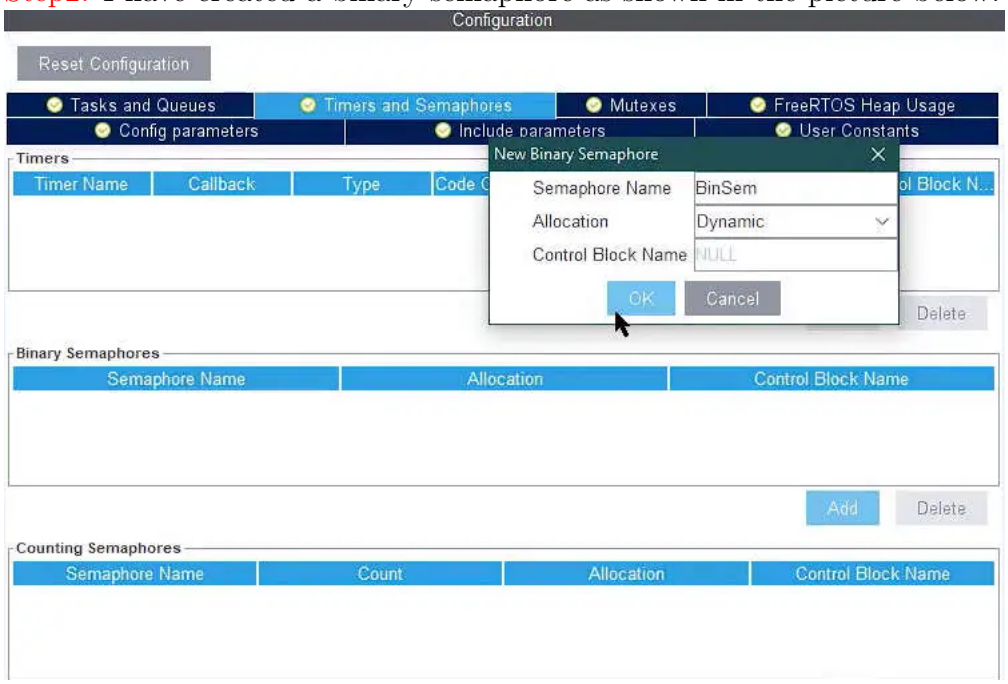
Let's say there is a LOW Priority Task running in the critical section. A HIGH Priority Task can preempt the LPT at any point in time. But if the LPT have the semaphore, and it doesn't release it until it comes out of the critical section, than HPT have no other option but to wait for the semaphore, and it can't preempt the LOW Task.

7.2 Setup

- Step1: I have created 3 different Tasks with different Priorities as shown in the picture below.



- Step2: I have created a binary semaphore as shown in the picture below:



7.3 Some Insight into the CODE

As mentioned above, I have created 3 different Tasks with different priorities, and now is the time to write those Tasks.

- This **Normal Task** is not waiting for any semaphore, so it can run independently, preempting the LOWER Task when needed.Below is the Medium Priority Task:

```
void Startnormaltask(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    char *str1 = "Entered MediumTask\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str1, strlen (str1), 100);


    char *str2 = "Leaving MediumTask\n\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str2, strlen (str2), 100);
    osDelay(500);
  }
  /* USER CODE END 5 */
}
```

  there is no semaphore needed to run the Medium Priority Task. I have created this task to show you guys that the semaphore is not compulsory for any task to run.

- The **High Task** waits for the semaphore before executing

High Priority Task

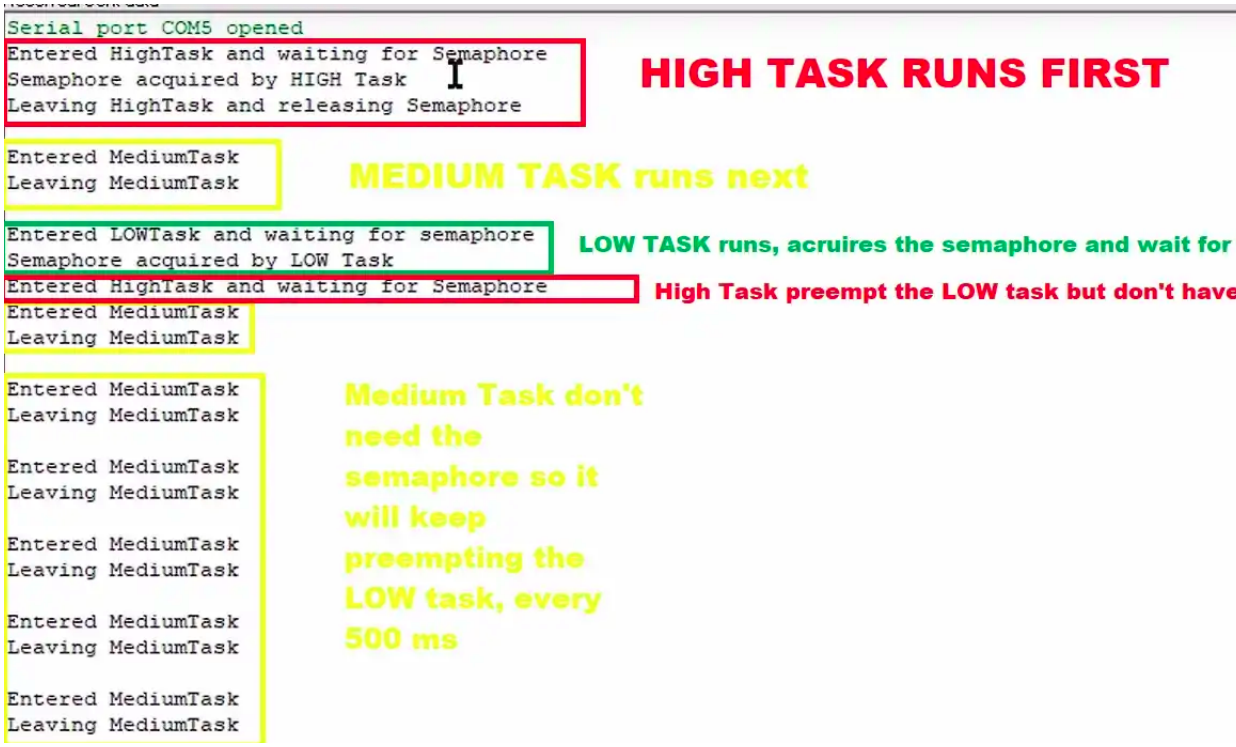
```
void Starthightask(void const * argument)
{
  for(;;)
  {
    char *str1 = "Entered HighTask and waiting for Semaphore\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str1, strlen (str1), 100);

    osSemaphoreWait(BinSemHandle, osWaitForever);

    char *str3 = "Semaphore acquired by HIGH Task\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str3, strlen (str3), 100);

    char *str2 = "Leaving HighTask and releasing Semaphore\n\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str2, strlen (str2), 100);

    osSemaphoreRelease(BinSemHandle);
    osDelay(500);
  }
}
```

- **LOW Task** also waits for the semaphore. Once it acquires it, than it waits for the button to be pressed. And after the button is pressed, the semaphore is released.

Low priority Task

```
void Startlowtask(void const * argument)
{
  for(;;)
  {
    char *str1 = "Entered LOWTask and waiting for semaphore\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str1, strlen (str1), 100);

    osSemaphoreWait(BinSemHandle, osWaitForever);
    char *str3 = "Semaphore acquired by LOW Task\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str3, strlen (str3), 100);

    while (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13));  // wait till the pin go low

    char *str2 = "Leaving LOWTask and releasing Semaphore\n\n";
    HAL_UART_Transmit(&huart2, (uint8_t *) str2, strlen (str2), 100);

    osSemaphoreRelease(BinSemHandle);
    osDelay(500);
  }
}
```

The Result:

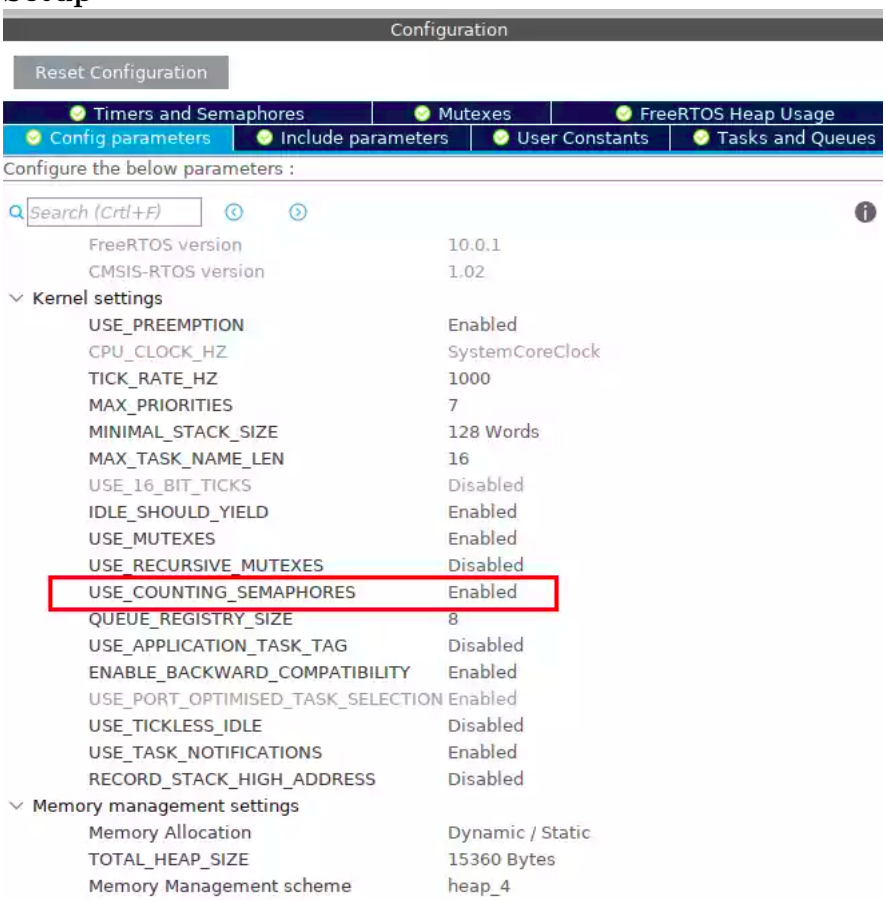


8. **Example4:How to use Counting Semaphore**

   From now onward, I am not going to use the CMSIS API anymore, and instead I will use the FreeRTOS functions directly. This will help you understand the process more clearly, and you can use the same functions across different microcontrollers, that supports FreeRTOS.

   Counting semaphore can be used to control the access to the resource. To obtain control of a resource, a task must first obtain a semaphore. Thus decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back and thus incrementing the semaphore count value.

4.1 Setup



4.2 Some Insight into the CODE

- **Step1:**we need to create handlers for the Tasks, and the semaphore.

Define tasks and semaphores

```
// create task defines
TaskHandle_t HPThandler;
void HPT_TASK (void *pvParameters);

TaskHandle_t MPThandler;
void MPT_TASK (void *pvParameters);

TaskHandle_t LPThandler;
void LPT_TASK (void *pvParameters);

TaskHandle_t VLPThandler;
void VLPT_TASK (void *pvParameters);

// semaphore related
SemaphoreHandle_t CountingSem;

// resource related
int resource[3] = {111,222,333};
int indx = 0;

// uart related
uint8_t rx_data = 0;
```

- **Step2:**CountingSem is the handler for counting semaphore
  CountingSem = xSemaphoreCreateCounting(3,0);
  if (CountingSem == NULL) HAL_UART_Transmit(&huart2, (uint8_t *) "Un-

able to Create Semaphore", 28, 100);
else HAL_UART_Transmit(huart2, (uint8_t *) "Counting Semaphore created successfully", 41, 1000);

- Inside the main function, first of all we need to create the counting semaphore.
- **xSemaphoreCreateCounting** takes 2 parameters, First is the maximum number of count, second is the initial value. I have created a semaphore with 3 tokens, and initial number of tokens available will be 0.
- If there is some error, and Semaphore couldn't create, it will return NULL, or else it will return some other value.

- **Step3:** create Tasks

create Tasks

```
// create TASKS

xTaskCreate(HPT_TASK, "HPT", 128, NULL, 3, &HPThandler);
xTaskCreate(MPT_TASK, "MPT", 128, NULL, 2, &MPThandler);
xTaskCreate(LPT_TASK, "LPT", 128, NULL, 1, &LPThandler);
xTaskCreate(VLPT_TASK, "VLPT", 128, NULL, 0, &VLPThandler);

vTaskStartScheduler();
```

- Here HPT_TASK is the function, where the task code is written
- "HPT" is just a name of this task. This name is not used anywhere in the program.
- 128 is the stack size
- NULL indicates that I am not passing any argument to the Task
- 3 is the Priority of the Task
- &HPThandler is the handler of the Task
- At the end, Start the Scheduler with vTaskStartScheduler()

- **Step4:** High priority Task

```
void HPT_TASK (void *pvParameters)
{
    char sresource[3];
    int semcount = 0;
    char ssemcount[2];

    // Give 3 semaphores at the beginning..
    xSemaphoreGive(CountingSem);
    xSemaphoreGive(CountingSem);
    xSemaphoreGive(CountingSem);

    while (1)
    {
        char str[150];
        strcpy(str, "Entered HPT Task\n About to ACQUIRE the Semaphore\n ");
        semcount = uxSemaphoreGetCount(CountingSem);
        itoa (semcount, ssemcount, 10);
        strcat (str, "Tokens available are: ");
        strcat (str, ssemcount);
        strcat (str, "\n\n");
        HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);

        xSemaphoreTake(CountingSem, portMAX_DELAY);
```

```
        xSemaphoreTake(CountingSem, portMAX_DELAY);

        itoa (resource[indx], sresource, 10);
        strcpy (str, "Leaving HPT Task\n Data ACCESSED is:: ");
        strcat (str, sresource);
        strcat (str, "\n Not releasing the Semaphore\n\n\n");
        HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);

        indx++;
        if (indx>2) indx=0;

        vTaskDelay(3000);
//      vTaskDelete(NULL);
    }
}
```

- **Step5:**: Interrupt callback
  I have also implemented UART to receive the data from the serial console. On receiving character 'r' the following will take place:-
  - First, we must define a **xHigherPriorityTaskWoken** and initialize it to the pdFALSE
  - Than, give the semaphore using **xSemaphoreGiveFromISR**. This takes the **xHigherPriorityTaskWoken** as the parameter
  - If a Higher Priority Task has preempted the Low Priority Task, from which we entered the ISR, than a context switch should be performed, and the interrupt safe API function will set ***pxHigherPriorityTaskWoken** to pdTRUE
  - **portEND_SWITCHING_ISR** will perform the context switching and the control will go to the High Priority Task.

    ```
        xSemaphoreTake(CountingSem, portMAX_DELAY);

        itoa (resource[indx], sresource, 10);
        strcpy (str, "Leaving HPT Task\n Data ACCESSED is:: ");
        strcat (str, sresource);
        strcat (str, "\n Not releasing the Semaphore\n\n\n");
        HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);

        indx++;
        if (indx>2) indx=0;

        vTaskDelay(3000);
//      vTaskDelete(NULL);
    }
}
    ```

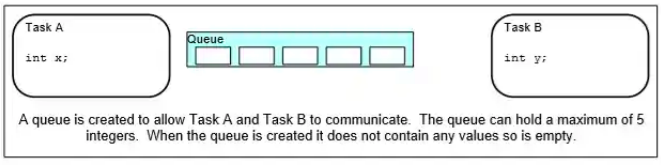
    The Result:

```
Counting Semaphore created successfully

Entered HPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 3

Leaving HPT Task
 Data ACCESSED is:: 111
 Not releasing the Semaphore


Entered MPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 2

Leaving MPT Task
 Data ACCESSED is:: 222
 Not releasing the Semaphore


Entered LPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 1

Leaving LPT Task
 Data ACCESSED is:: 333
 Not releasing the Semaphore


Entered VLPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 0

Entered LPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 0

Entered MPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 0

Entered HPT Task
 About to ACQUIRE the Semaphore
 Tokens available are: 0
```

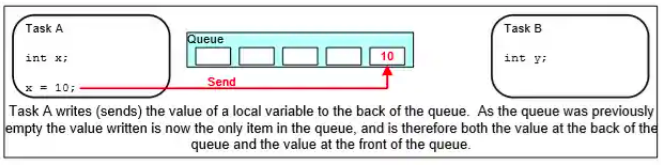
1 2 3 4 5 6 7

9. **Example5:Using Queue**

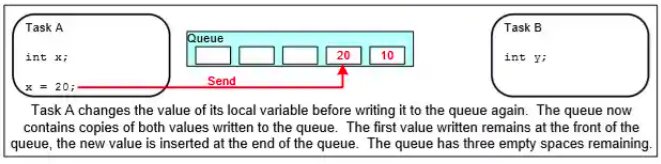
In this tutorial we are going to learn how to use Queue to communicate between the tasks.



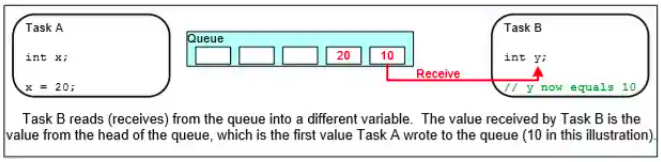
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



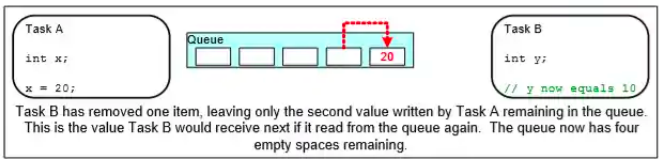
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

- Queue is the easiest way to send and receive data between the tasks.
- We are going to use the simple queue at first, where all the elements in the Queue are of same data types, and later we will use a structured Queue, where the data types can be different.

5.1 Simple Queue

In a simple Queue all the elements are of same type. For example a Queue can only hold 5 integers, or 6 characters, or 3 unsigned integers etc.

- **Step1:** A Queue is recognized by it's handler, so first of all we need to create a handler for the Queue

```
67 /* USER CODE BEGIN 0 */
68 /******************** TASK HANDLERS *****************/
69 xTaskHandle Sender_HPT_Handler;
70 xTaskHandle Sender_LPT_Handler;
71 xTaskHandle Receiver_Handler;
72
73 /******************* QUEUE HANDLER *****************/
74 xQueueHandle SimpleQueue;
75
76 /******************* TASK FUNCTIONS ***************/
77 void Sender_HPT_Task (void *argument);
78 void Sender_LPT_Task (void *argument);
79 void Receiver_Task (void *argument);
80
```

- **Step2:** Next, inside the main function, we will create a Queue, which can store 5 integers

```
113   /* USER CODE BEGIN 2 */
114   /******************* Create Integer Queue ***********/
115   SimpleQueue = xQueueCreate(5,sizeof (int));
116   if (SimpleQueue ==0)
117   {
118       char *str = "Unable to create Integer Queue\n\n";
119       HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen(str), HAL_MAX_DELAY);
120   }
121   else
122   {
123       char *str= "Integer Queue Create successfully\n\n";
124       HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen(str), HAL_MAX_DELAY);
125   }
126
127   /******************** TASK RELATED ***************/
128
129   xTaskCreate(Sender_HPT_Task, "HPT_SEND", 128, NULL, 3,&Sender_HPT_Handler);
130   xTaskCreate(Sender_LPT_Task, "LPT_SEND", 128, (void *)111, 2,&Sender_LPT_Handler);
131
132   xTaskCreate(Receiver_Task, "Receiver", 128, NULL, 1, &Receiver_Handler);
133
134   HAL_UART_Receive(&huart2, &Rx_data, sizeof(Rx_data), 1);
135   vTaskStartScheduler();
```

If there is an error while creating a Queue, like shortage of memory, than the **xQueueCreate** function return a '0'.

- **Step3:** Inside the sender task function, we can send the data to the Queue using the following

```
278
279 /* USER CODE BEGIN 4 */
280 void Sender_HPT_Task (void *argument)
281 {
282     int i=222;
283     uint32_t TickDelay = pdMS_TO_TICKS(2000);
284     while (1)
285     {
286         if (xQueueSend(SimpleQueue, &i, portMAX_DELAY) == pdPASS)
287         {
288             char *str2 = " Successfully sent the number to the queue\nLeaving SENDER_HPT Task\n\n\n";
289             HAL_UART_Transmit(&huart2, (uint8_t *)str2, strlen (str2), HAL_MAX_DELAY);
290         }
291         vTaskDelay(TickDelay);
292     }
293 }
294 void Sender_LPT_Task (void *argument)
295 {
296     int ToSend;
297     uint32_t TickDelay = pdMS_TO_TICKS(1000);
298     while (1)
299     {
300         ToSend = (int) argument;
301         char *str= "Entered SENDER_LPT Task\n about to SEND a number to the queue\n\n";
302         HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen(str), HAL_MAX_DELAY);
303
304         xQueueSend(SimpleQueue, &ToSend, portMAX_DELAY);
305         char *str2 = "Successfully sent the number to the queue\nLeaving SENDER_LPT Task\n\n\n";
306         HAL_UART_Transmit(&huart2, (uint8_t *)str2, strlen(str2), HAL_MAX_DELAY);
307
308         vTaskDelay(TickDelay);
309     }
310 }
311 void Receiver_Task (void *argument)
312 {
313     int received=0;
314     uint32_t TickDelay = pdMS_TO_TICKS(5000);
315
316     while (1)
317         {
318             char str[100];
319             strcpy (str, "Entered RECEIVER Task\n about to RECEIVE a number from the queue\n\n");
320             HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen(str), HAL_MAX_DELAY);
321             if (xQueueReceive(SimpleQueue, &received, portMAX_DELAY) != pdTRUE)
322             {
323                 HAL_UART_Transmit(&huart2, (uint8_t *)"Error in Receiving from Queue\n\n", 31, 1000);
324             }
325             else
326             {
327                 sprintf(str, " Successfully RECEIVED the number %d to the queue\nLeaving RECEIVER Task\n\n\n",r
328                 HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);
329             }
330             vTaskDelay(TickDelay);
331         }
332 }
```

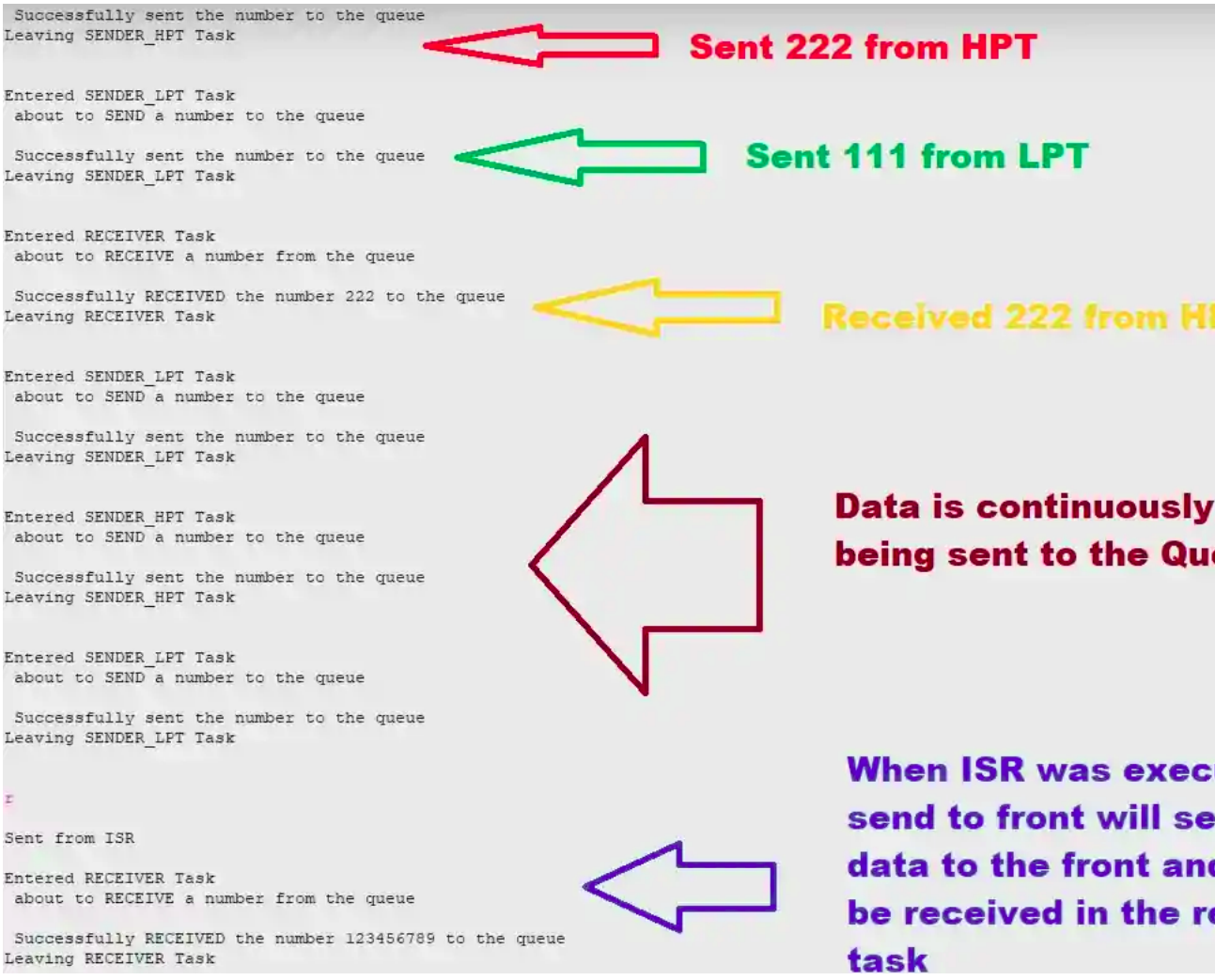
  - The parameters of **xQueuReceive** are handler to the Queue, the address where the data is to be stored, and the waiting time in case the Queue is Empty.
  - I have specified the waiting as **portMAX_DELAY**, that means the task is going to wait forever for the data to become available in the Queue.
  - If the data is received successfully, **xQueueReceive** will return **pdTRUE**, and we can display the data on the console.

- **Step4:** If we want to send the data to the Queue from an ISR, we have to use the interrupt safe version of these functions

```c
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Receive_IT(huart, &Rx_data, 1);
    int ToSend = 123456789;
    if (Rx_data == 'r')
    {
         /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE as
         it will get set to pdTRUE inside the interrupt safe API function if a
         context switch is required. */
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;

        if (xQueueSendToFrontFromISR(SimpleQueue, &ToSend, &xHigherPriorityTaskWoken) == pdPASS)
        {
            HAL_UART_Transmit(huart, (uint8_t *)"\n\nSent from ISR\n\n", 17, 500);
        }

        /* Pass the xHigherPriorityTaskWoken value into portEND_SWITCHING_ISR(). If
         xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR()
         then calling portEND_SWITCHING_ISR() will request a context switch. If
         xHigherPriorityTaskWoken is still pdFALSE then calling
         portEND_SWITCHING_ISR() will have no effect */

        portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
    }
}
/* USER CODE END 4 */
```

xQueueSendToFrontFromISR will send the data to the front of the Queue. All the data, which is already available in the queue, will shift back and next time if we read the Queue, we will get this particular data.

The Result:



Video teach:https://www.youtube.com/watch?v=J6J8EUcw6qU

5.2 Structured Queue

If we want to send the different data types, we have to use the Structured Queue.

- **Step1:** create the handler for this Queue and We need to create a structure which can store all the data types that we want to use. I will call it my_struct

```
/*************** QUEUE HANDLER ****************/
xQueueHandle St_Queue_Handler;

/*************** TASK  HANDLER ***************/
xTaskHandle Sender1_Task_Handler;
xTaskHandle Sender2_Task_Handler;
xTaskHandle Receiver_Task_Handler;

/*************** TASK FUNCTION ****************/
void Sender1_Task (void *argument);
void Sender2_Task (void *argument);
void Receiver_Task (void *argument);

/*************** STRUCTURE DEFINITION ***********/

typedef struct {
    char *str;
    int counter;
    uint16_t large_value;
}my_struct;

int indx1 = 0;
int indx2 = 0;
```

- **Step2:** Inside the main function, create the Queue

```
121   /****** create QUEUE ******/
122   St_Queue_Handler = xQueueCreate(2, sizeof(my_struct));
123   if (St_Queue_Handler ==0) //if there is some erroe while creating queue
124   {
125       char *str = "Unable to create STRUCTURE Queue\n\n";
126       HAL_UART_Transmit(&huart2,(uint8_t *)str, strlen(str), HAL_MAX_DELAY);
127   }
128   else
129   {
130       char *str = "STRUCTURE Queue Created successfully\n\n";
131       HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen(str), HAL_MAX_DELAY);
132   }
133
134   /************ CREATE TASKS ***********/
135   xTaskCreate(Sender1_Task, "SENDER1",128, NULL, 2, &Sender1_Task_Handler);
136   xTaskCreate(Sender2_Task, "SENDER2",128, NULL, 2, &Sender2_Task_Handler);
137   xTaskCreate(Receiver_Task, "RECEIVER",128, NULL, 1, &Receiver_Task_Handler);
138
139   /****** start the sheduler ********/
140   vTaskStartScheduler();
141   /* USER CODE END 2 */
```

I have created a Queue which can store 2 elements of **my_struct** data type. Again if there is some error while creating Queue, the **xQueueCreate** will return 0, or else it will return any other value.

- **Step3:** We will now send the data to this Queue from a sender Task

```c
void Sender1_Task (void *argument)
{
    my_struct *ptrtostruct;
    uint32_t TickDelay = pdMS_TO_TICKS(2000);
    while (1)
    {
        char *str = "Entered SENDER1 Task\n about to SEND to the queue\n\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);
        /****** ALOOCATE MEMORY TO THE PTR ********/
        ptrtostruct = pvPortMalloc(sizeof (my_struct));
        /********** LOAD THE DATA ***********/
        ptrtostruct->counter = 1+indx1;
        ptrtostruct->large_value = 1000 + indx1*100;
        ptrtostruct->str = "HELLO FROM SENDER 1 ";
        /***** send to the queue ****/
        if (xQueueSend(St_Queue_Handler, &ptrtostruct, portMAX_DELAY) == pdPASS)
        {
            char *str2 = " Successfully sent the to the queue\nLeaving SENDER1 Task\n\n\n";
            HAL_UART_Transmit(&huart2, (uint8_t *)str2, strlen (str2), HAL_MAX_DELAY);
        }

        indx1 = indx1+1;
        vTaskDelay(TickDelay);
    }
}
void Sender2_Task (void *argument)
{
    my_struct *ptrtostruct;
    uint32_t TickDelay = pdMS_TO_TICKS(2000);
    while (1)
    {
        char *str = "Entered SENDER2 Task\n about to SEND to the queue\n\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);
        /****** ALOOCATE MEMORY TO THE PTR ********/
        ptrtostruct = pvPortMalloc(sizeof (my_struct));
        /********** LOAD THE DATA ***********/
        ptrtostruct->counter = 1+indx2;
        ptrtostruct->large_value = 2000 + indx2*200;
        ptrtostruct->str = "Sender2 says Hii!! ";
        /***** send to the queue ****/
        if (xQueueSend(St_Queue_Handler, &ptrtostruct, portMAX_DELAY) == pdPASS)
        {
            char *str2 = " Successfully sent the to the queue\nLeaving SENDER2 Task\n\n\n";
            HAL_UART_Transmit(&huart2, (uint8_t *)str2, strlen (str2), HAL_MAX_DELAY);
        }
        indx2 = indx2+1;
```

Before sending the data to the Queue, we must allocate the memory to the structure. And to do that, we use the function **pvPortMALLOC**.
Before sending the data to the Queue, we must allocate the memory to the structure. And to do that, we use the function pvPortMALLOC.

- **Step4:** we will send this data to the Queue by passing it's address in the parameter.

```c
void Receiver_Task (void *argument)
{
    my_struct *Rptrtostruct;
    uint32_t TickDelay = pdMS_TO_TICKS(3000);
    char *ptr;

    while (1)
    {
        char *str = "Entered RECEIVER Task\n about to RECEIVE FROM the queue\n\n";
        HAL_UART_Transmit(&huart2, (uint8_t *)str, strlen (str), HAL_MAX_DELAY);

        /**** RECEIVE FROM QUEUE *****/
        if (xQueueReceive(St_Queue_Handler, &Rptrtostruct, portMAX_DELAY) == pdPASS)
        {
            ptr = pvPortMalloc(100 * sizeof (char)); // allocate memory for the string

            sprintf (ptr, "Received from QUEUE:\n COUNTER = %d\n LARGE VALUE = %u\n STRING = %s\n\n\n",Rptrtostruct->counter,Rptrtostruct->large_value, Rptrtostr
            HAL_UART_Transmit(&huart2, (uint8_t *)ptr, strlen(ptr), HAL_MAX_DELAY);

            vPortFree(ptr);  // free the string memory
        }

        vPortFree(Rptrtostruct);  // free the structure memory

        vTaskDelay(TickDelay);
    }
}
```

_We will create another pointer to struct in the receiver function to store the received data.
_After receiving the data, we will free the memory allocated by the sender Task using vPortFREE function.
The Result:

```
Entered SENDER1_Task
 about to SEND to the queue

 Successfully sent the to the queue
Leaving SENDER1_Task


Entered RECEIVER Task
 about to RECEIVE FROM the queue

 Successfully sent the to the queue
Leaving SENDER2_Task


Received from QUEUE:
 COUNTER = 1
 LARGE VALUE = 2000
 STRING = Sender2 says Hii!!!


Entered SENDER1_Task
 about to SEND to the queue

Entered SENDER2 Task
 about to SEND to the queue

Entered RECEIVER Task
 about to RECEIVE FROM the queue

 Successfully sent the to the queue
Leaving SENDER1_Task


Received from QUEUE:
 COUNTER = 2
 LARGE VALUE = 1100
 STRING = HELLO FROM SENDER 1
```

10. **Example6:Using MUTEX**

Mutex, which is short for Mutual Exclusion, does what's it name indicates. It prevents several tasks from accessing a resource mutually. It ensures that at one time, only one task have access to the resource.

In this tutorial, we will see how to use mutex. And I will also explain the difference between a mutex and a binary semaphore. Also we will learn about priority inversion and priority inheritance.

6.1 Simple Mutex Operation

- **Step1:** Create semaphore

```
61 /* USER CODE BEGIN 0 */
62 SemaphoreHandle_t SimpleMutex;
63
64 TaskHandle_t HPT_Handler;
65 TaskHandle_t MPT_Handler;
66 //TaskHandle_t LPT_Handler;
67
68 void HPT_Task(void);
69 void MPT_Task(void);
70 //void LPT_Task(void);
71 void Send_Uart (uint8_t *str)
72 {
73     xSemaphoreTake(SimpleMutex, portMAX_DELAY);
74     HAL_Delay(2000);
75     HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);
76     xSemaphoreGive(SimpleMutex);
77 }
78 /* USER CODE END 0 */
```

  - In the code above, I have created a **mutex handler (SimpleMutex)**, two task handlers, and defined the task functions.
  - Along with that, there is a function **(Send_Uart)**, which will acquire the mutex first, waits for 2 seconds, sends the data to the UART, and releases the mutex.

- **Step2:** write the Task Function

```
253 /* USER CODE BEGIN 4 */
254 void HPT_Task (void *argument)
255 {
256     uint8_t *strtosend = "IN HPT==========================\n";
257     while (1)
258     {
259         char *str = "Entered HPT and About to take MUTEX\n";
260         HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);
261 
262         Send_Uart(strtosend);
263 
264         char *str2 = "Leaving HPT\n\n";
265         HAL_UART_Transmit(&huart2, str2, strlen (str2), HAL_MAX_DELAY);
266 
267         vTaskDelay(2000);
268     }
269 }
270 
271 void MPT_Task (void *argument)
272 {
273     uint8_t *strtosend = "IN MPT.........................\n";
274     while (1)
275     {
276         char *str = "Entered MPT and About to take MUTEX\n";
277         HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);
278 
279         Send_Uart(strtosend);
280 
281         char *str2 = "Leaving MPT\n\n";
282         HAL_UART_Transmit(&huart2, str2, strlen (str2), HAL_MAX_DELAY);
283 
284         vTaskDelay(1000);
285     }
286 }
287 /* USER CODE END 4 */
```

  - The two Task function will make a call to the function, **Send_Uart**. Now we have to see, can the High Priority Task (HPT) preempt the Medium Priority Task (MPT) while the MPT holds the MUTEX.

- **Step3:** We will write the following into the Main Function

```
109   /* USER CODE BEGIN 2 */
110   SimpleMutex = xSemaphoreCreateMutex();
111 
112   if (SimpleMutex !=NULL)
113   {
114       HAL_UART_Transmit(&huart2, "Mutex Created\n\n", 15, 1000);
115 
116   }
117   // Create tasks
118   xTaskCreate(HPT_Task,"HPT",128,NULL,3,&HPT_Handler);
119   xTaskCreate(MPT_Task,"MPT",128,NULL,2,&HPT_Handler);
120   xTaskCreate(LPT_Task,"LPT",128,NULL,1,&HPT_Handler);
121 
122   vTaskStartScheduler();
```

The Result:

```
Mutex Created

Entered HPT and About to take MUTEX
IN HPT==========================
Leaving HPT

Entered MPT and About to take MUTEX
Entered HPT and About to take MUTEX
IN MPT.........................
IN HPT==========================
Leaving HPT

Leaving MPT
```

MUTEX was created successfully

HPT acquired the MUTEX, print the string, release the mutex, and goes into suspension for 2 seconds

MPT Runs, acquire the MUTEX, waits for 2 seconds to complete.

HPT will wake up, and preempts the MPT, It will try to acquire the mutex but can't. It will wait for the MPT to release the mutex.

MPT prints the string, release the mutex.

HPT acquire the mutex, prints it's string and goes into suspension

MPT also goes into suspension.

6.2 Priority Inversion in Semaphore

- **Step1:** I will make some changes in the code again.

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
SemaphoreHandle_t SimpleMutex;
SemaphoreHandle_t BinSemaphore;

TaskHandle_t HPT_Handler;
TaskHandle_t MPT_Handler;
TaskHandle_t LPT_Handler;

void HPT_Task(void);
void MPT_Task(void);
void LPT_Task(void);
void Send_Uart (uint8_t *str)
{
    xSemaphoreTake(BinSemaphore, portMAX_DELAY);
    HAL_Delay(2000);
    HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);
    xSemaphoreGive(BinSemaphore);
}
/* USER CODE END 0 */
```

  This time I have defined a handler for the binary semaphore also (BinSemaphore). Also all the three tasks are being used now.

- **Step2:** write the Task Function

```
  /* USER CODE BEGIN 2 */
  SimpleMutex = xSemaphoreCreateMutex();

  if (SimpleMutex !=NULL)
  {
      HAL_UART_Transmit(&huart2, "Mutex Created\n\n", 15, 1000);

  }
  BinSemaphore = xSemaphoreCreateBinary();
  if (BinSemaphore != NULL)
  {
      HAL_UART_Transmit(&huart2, "Semaphore Created\n\n", 19, 1000);
  }

  xSemaphoreGive(BinSemaphore);
  // Create tasks
  xTaskCreate(HPT_Task,"HPT",128,NULL,3,&HPT_Handler);
  xTaskCreate(MPT_Task,"MPT",128,NULL,2,&HPT_Handler);
  xTaskCreate(LPT_Task,"LPT",128,NULL,1,&HPT_Handler);

  vTaskStartScheduler();
  /* USER CODE END 2 */
```

- **Step3:** write the Main Function

```c
/* USER CODE BEGIN 4 */
void HPT_Task (void *argument)
{
    uint8_t *strtosend = "IN HPT==========================\n";
    while (1)
    {
        char *str = "Entered HPT and About to take MUTEX\n";
        HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);

        Send_Uart(strtosend);

        char *str2 = "Leaving HPT\n\n";
        HAL_UART_Transmit(&huart2, str2, strlen (str2), HAL_MAX_DELAY);

        vTaskDelay(2000);
    }
}

void MPT_Task (void *argument)
{
    while (1)
    {
        char *str = "IN MPT*****************************\n\n";
        HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);
        vTaskDelay(2000);
    }
}

void LPT_Task (void *argument)
{
    uint8_t *strtosend = "IN LPT.........................\n";
    while (1)
    {
        char *str = "Entered LPT and About to take Semaphore\n";
        HAL_UART_Transmit(&huart2, str, strlen (str), HAL_MAX_DELAY);

        Send_Uart(strtosend);

        char *str2 = "Leaving LPT\n\n";
        HAL_UART_Transmit(&huart2, str2, strlen (str2), HAL_MAX_DELAY);

        vTaskDelay(1000);
    }
}
/* USER CODE END 4 */
```

The Result:

```
Mutex Created

Semaphore Created

Entered HPT and About to take Semaphore
IN HPT==========================
Leaving HPT

IN MPT*****************************

Entered LPT and About to take Semaphore
Entered HPT and About to take Semaphore
IN MPT*****************************

IN MPT*****************************

IN LPT.........................
IN HPT==========================
Leaving HPT

IN MPT*****************************

Leaving LPT
```

Mutex and semaphore were created successfully

HPT runs, acquire the semaphore, prints the string, releases the semaphore, and goes into suspension for 750 ms.

MPT runs and goes into suspension for 2 seconds

PRIORITY INVERSION takes place. See another picture to understand it

11. **Example7: Software Timers**

A software timer allows a function to be executed at a set time in the future. The function executed by the timer is called the timer's callback function. The time between a timer

being started, and its callback function being executed, is called the timer's period. Put simply, the timer's callback function is executed when the timer's period expires.

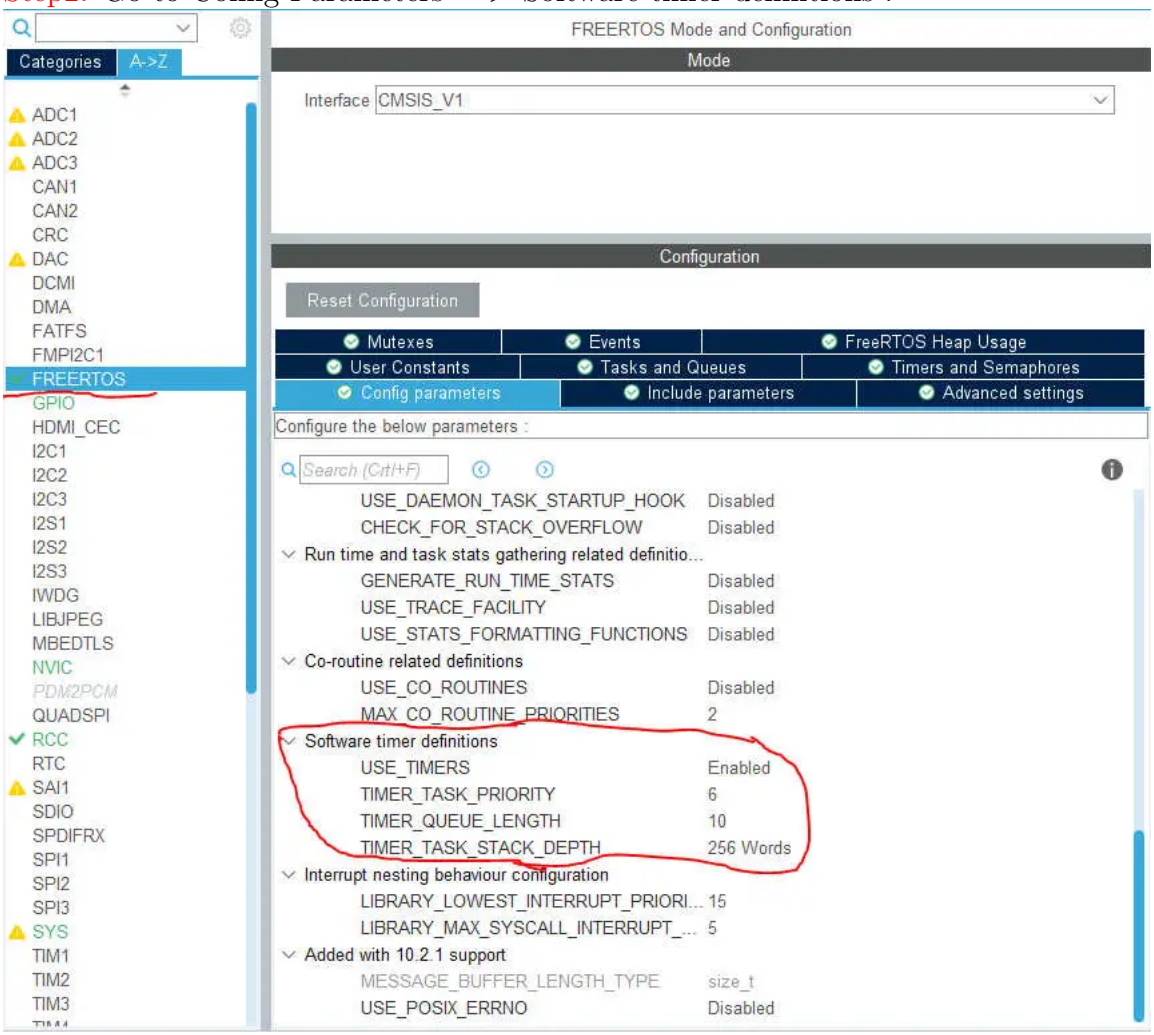
There are two type of timers:

- **one-shot timers**: An one-shot timer can execute its callback function only once. It can be manually re-started, but will not automatically re-start itself.
- **auto-reload timers**: An auto-reload timer will automatically re-start itself after each execution of its callback function, resulting in periodic callback execution.

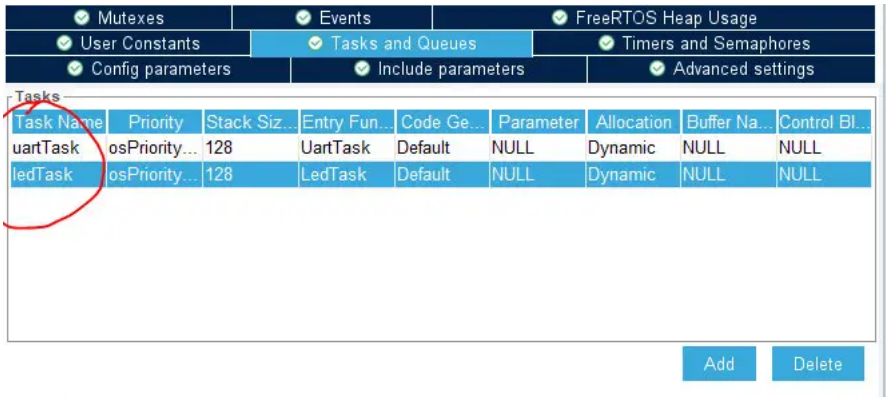
7.1 Using CMSIS

Setup

- Step1: Go to FreeRtos − > Interface(Select CMISS_V1)
- Step2: Go to Config Parameters − > Software timer definitions .

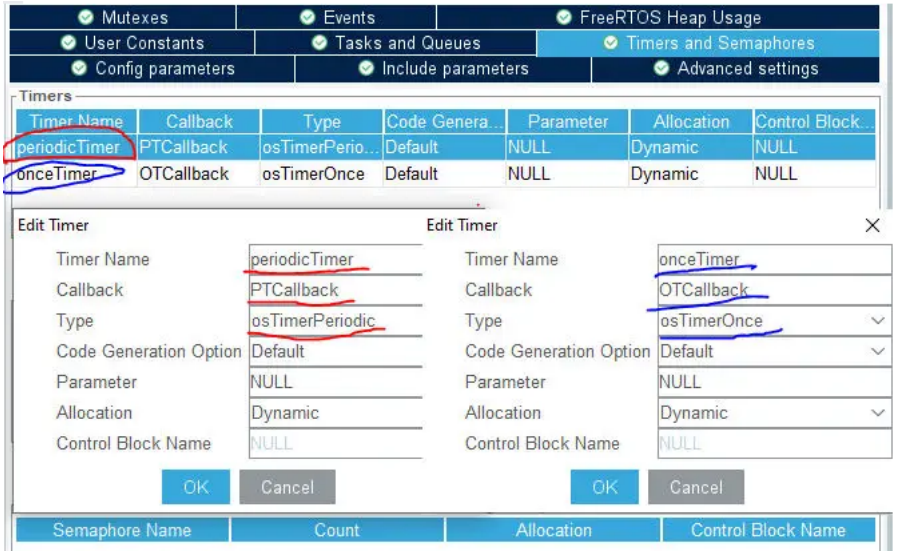


  - Enabled USE_TIMERS
  - Set the priority for the timers. I have set it to the maximum value i.e 6.
- Step3: Go to Task and Queues



I have set up 2 tasks i.e **uart task** and the **led task**. Both of them have the same priorities so that they don't preempt each other.

- Step4: Go to Timers and Semaphores



I have created 2 timers

- periodic Timer is the Periodic type (**osTimerPeriodic)**, and it's callback is declared as **PTCallback**
- once Timer is the One shot Timer (**osTimerOnce**), and it's callback function is **OTCallback**

- Step5: Select **UART2** and Setting PA5 as output(for on board LED) and PC13 as input(for on board user button).

Write Code in main.c

- Step1: Create function **UartTask**

```
void UartTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
    osTimerStart(periodicTimerHandle, 1000);
  /* Infinite loop */
  for(;;)
  {
    HAL_UART_Transmit(&huart2, "Sending from UART TASK\n", 23, 100);
    osDelay(2000);
  }
  /* USER CODE END 5 */
}
```

- **osTImerStart** will take the parameters as the **timer handle** that you want to start, and the **period** for the timer before it expires.
- inside the while loop, we will send the data to the uart and the task will keep running every 2 seconds

- Step2: Create function **LedTask**

```
void LedTask(void const * argument)
{
  /* USER CODE BEGIN LedTask */
  /* Infinite loop */
  for(;;)
  {
      if (!(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)))  // if the button is pressed
        {
          HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1);  // set the LED
          osTimerStart(onceTimerHandle, 4000);
        }
    osDelay(20);
  }
  /* USER CODE END LedTask */
}
```

  - If you push button So LED will High.
  - now start the one shot timer with a period of 4 seconds
  - and this task will run every 20 ms. I am choosing a very small suspension time because we don't want to miss any button input

- Step3: Create function **PTCallback**

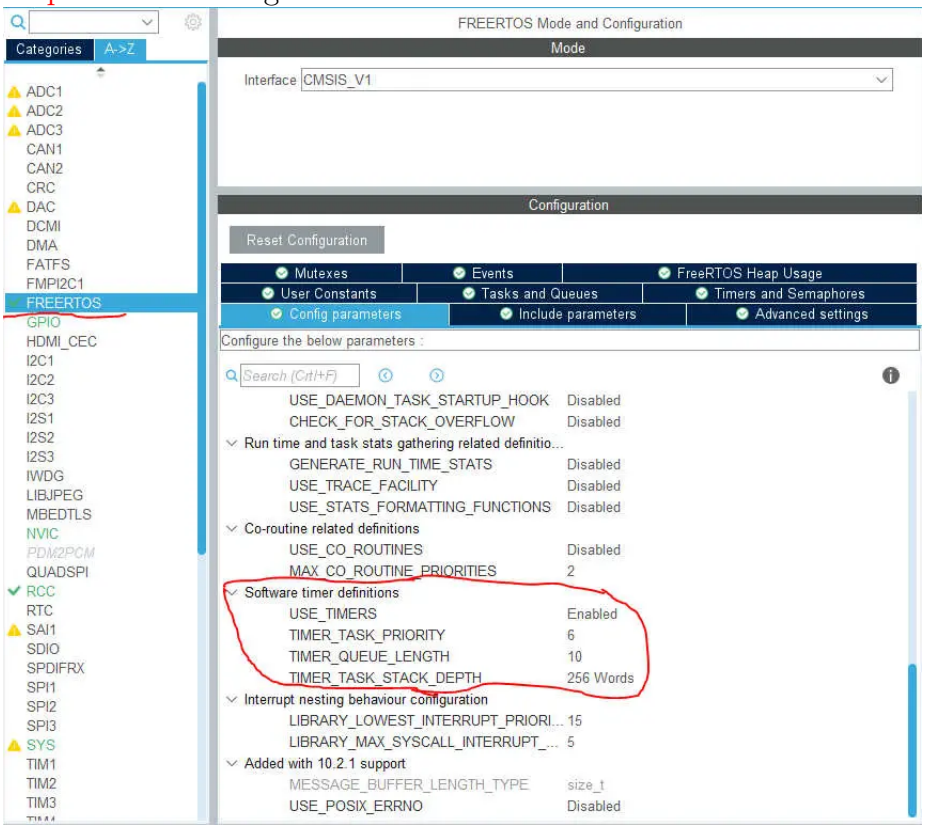
```
void PTCallback(void const * argument)
{
  /* USER CODE BEGIN PTCallback */
    HAL_UART_Transmit(&huart2, "Sending from PERIODIC TIMER\n", 28, 100);
  /* USER CODE END PTCallback */
}
```

- Step4: Create function **OTCallback**

7.2 Using FreeRtos Functions

Setup

- Step1: Go to FreeRtos $->$ Interface(Select CMISS_V1)
- Step2: Go to Config Parameters $->$ Software timer definitions .



  - Enabled USE_TIMERS

- Set the priority for the timers. I have set it to the maximum value i.e 6.

- Step3: Select **UART2** and Setting PA5 as output(for on board LED) and PC13 as input(for on board user button).

Write code in main.c

- Step1: we will define the timer handles and the task handles

```
67 /* Private user code ---------
68 /* USER CODE BEGIN 0 */
69 // Crate timer handles
70 xTimerHandle PTHandle;
71 xTimerHandle OTHandle;
72
73 // create task handles
74 xTaskHandle uartHandle;
75 xTaskHandle ledHandle;
76
```

- Step2: : The main function, we will create the timers

```
109   /* USER CODE BEGIN 2 */
110   // Create timers
111   PTHandle = xTimerCreate("timer1", pdMS_TO_TICKS(1000), pdTRUE,(void *) 1, TimerCallbac
112   OTHandle = xTimerCreate("timer2", pdMS_TO_TICKS(4000), pdFALSE, (void *)2, TimerCallba
113
114   // Create tasks
115   xTaskCreate(uartTask, "uart", 128, NULL, 1, &uartHandle);
116   xTaskCreate(ledTask, "led", 128, NULL, 1, &ledHandle);
117
118   /* USER CODE END 2 */
```

xTimerCreate takes the following parameters

  - **Name** of the timer − > you can give any name here, since it won't be used anywhere.
  - Time Period for the timer − > this is the period for which the timer will run in the background, before it expires
  - Timer Type − > Either the timer is Auto Reload (TRUE), or One Shot (FALSE)
  - Timer ID − > You can give any ID here to identify the timer
  - Timer Callback − > The callback function, which will be called once the timer is expired

- Step3: We will create the tasks

```
109   /* USER CODE BEGIN 2 */
110   // Create timers
111   PTHandle = xTimerCreate("timer1", pdMS_TO_TICKS(1000), pdTRUE,(void *) 1, TimerCallbac
112   OTHandle = xTimerCreate("timer2", pdMS_TO_TICKS(4000), pdFALSE, (void *)2, TimerCallba
113
114   // Create tasks
115   xTaskCreate(uartTask, "uart", 128, NULL, 1, &uartHandle);
116   xTaskCreate(ledTask, "led", 128, NULL, 1, &ledHandle);
117
118   /* USER CODE END 2 */
```

The arguments for the **xTaskCreate** are as follows

  - Task Function − > Here we will write the code related to the respective task
  - Task name − > you can give any name here, since it won't be used anywhere
  - Stack Size − > I am leaving this to 128
  - Parameters to the task − > We are not passing any parameters, so keep it NULL
  - Task Priority − > I am keeping the task priority to normal (1)
  - Task Handle − > The pointer to the handle for the task

- Step4: Create function **UartTask** and **LedTask**

```
279 /* USER CODE BEGIN 4 */
280 void uartTask (void *argument)
281 {
282     xTimerStart(PTHandle, 0);
283     while (1)
284     {
285         HAL_UART_Transmit(&huart2, (uint8_t *)"sending from UART TASK\n", 23, 100);
286         vTaskDelay(pdMS_TO_TICKS(2000));
287     }
288 }
289 void ledTask (void *argument)
290 {
291     while (1)
292     {
293         if (!(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)))  // if the button is pressed
294         {
295             HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 1);  // Turn On the LED
296             xTimerStart(OTHandle, 0);
297         }
298
299         vTaskDelay(pdMS_TO_TICKS(20));
300     }
301 }
```

  The Time to wait before starting the timer. I am keeping this as 0, since I want the timer to start immediately

- Step5: Create function TimerCallback

```
302
303 void TimerCallback (xTimerHandle xTimer)
304 {
305     if (xTimer == PTHandle)
306     {
307         HAL_UART_Transmit(&huart2, (uint8_t *) "sending from TIMER CALLBACK\n", 28, 100);
308     }
309
310     else if (xTimer == OTHandle)
311     {
312         HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, 0);
313     }
314 }
315
```

  - if the callback is made by the periodic timer, we will send the string the uart, indicating that the callback has been executed
  - and if the callback is called by the one shot timer, we will turn off the LED

Reslult:

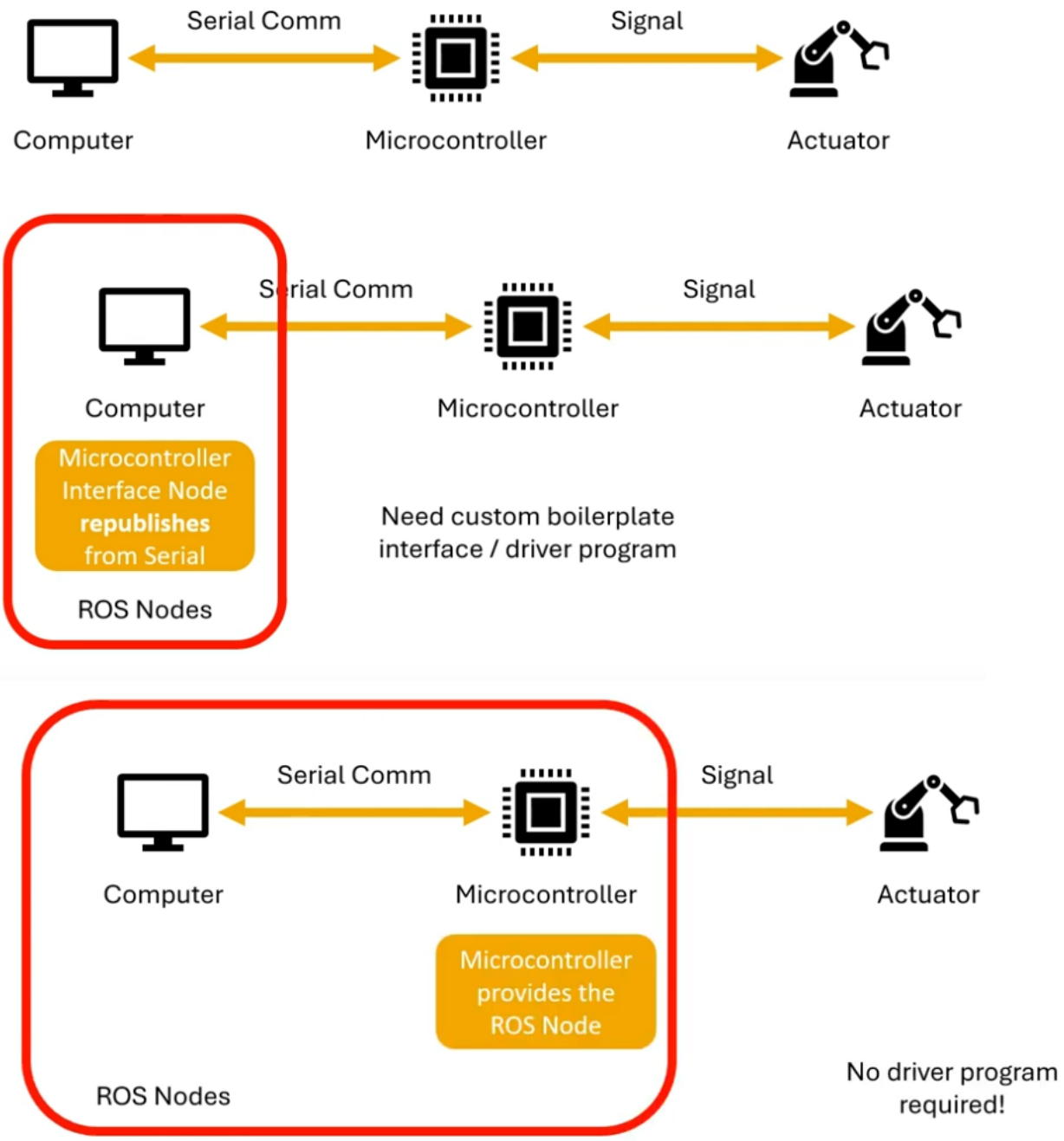
```
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
Sending from PERIODIC TIMER
Sending from PERIODIC TIMER
Sending from UART TASK
```

Periodic timer runs every 1 Second

while the UART Task runs every 2 Seconds

XXIX MicroROS

1. **What is MicroROS?**
   Pipeline with micro-ROS



Definition:

- Provides a ROS2 node using DDS() for microcontroller (Code is very similar to ROS2 C++)
- Does need micro-ROS agent active on host machine
- Low-level ROS2 integration active on microcontroller
- ROSSerial (for ROS1) is similar but more custom to ROS with a bridge converting ROSSerial to ROS.
- Does need higher-grade hardware than ROSSearial

Overview
The major concepts (publishers, subscriptions, services, timers, ...) are identical with ROS 2. They even rely on the same implementation, as the micro-ROS C API is based on the ROS 2 client support library (rcl), enriched with a set of convenience functions by the package rclc. That is, rclc does not add a new layer of types on top of rcl (like rclcpp and rclpy do) but only provides functions that ease the programming with the rcl types. New types are introduced only for concepts that are missing in rcl, such as the concept of an executor.

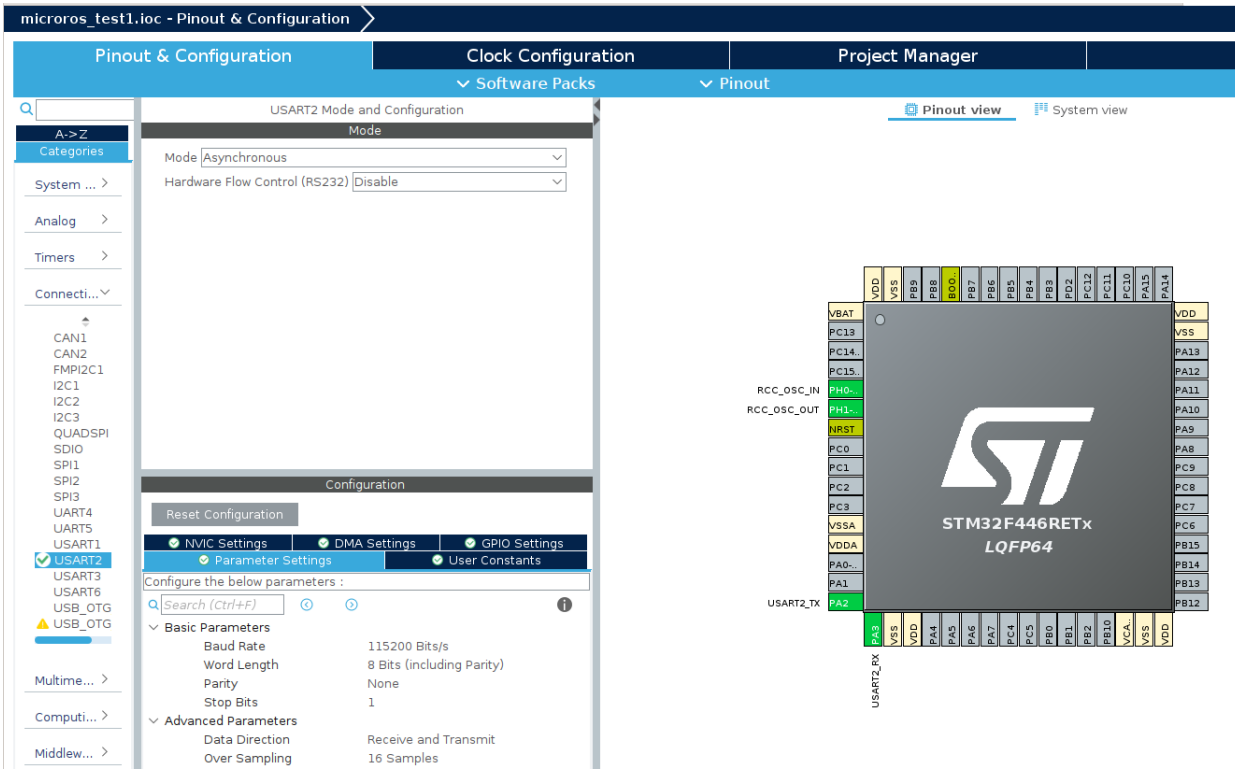
2. **How to Setup MicroRos**

- The first You must install ROS2(Foxy(when you use Ubuntu 20.04) or Humble(When you use Ubuntu 22.04) because STM32 Support ROS2 only. https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html#
- And Second you install MicroROS (https://micro.ros.org/docs/tutorials/core/first_application_linux/)
- After you Setup Micro_ROS in Stm32 https://github.com/micro-ROS/micro_ros_stm32cubemx_utils/tree/foxy
- And This Link have Setup MicroRos and **MicroRos_agent** to communication STM32 with ROS : https://github.com/lFatality/stm32_micro_ros_setup
- The Video Setup For Method1: Foxy(https://youtu.be/bn-P3fxtTF4?si=VdRCyf0hPooHztUX or Humble(https://youtu.be/rOlcAEvJsm4?si=PDYJPrkZkFg5W_K-)
- The Video setup For Methode2: https://youtu.be/xbWaHARjSmk?si=fXQ6dThjNRg4kduo

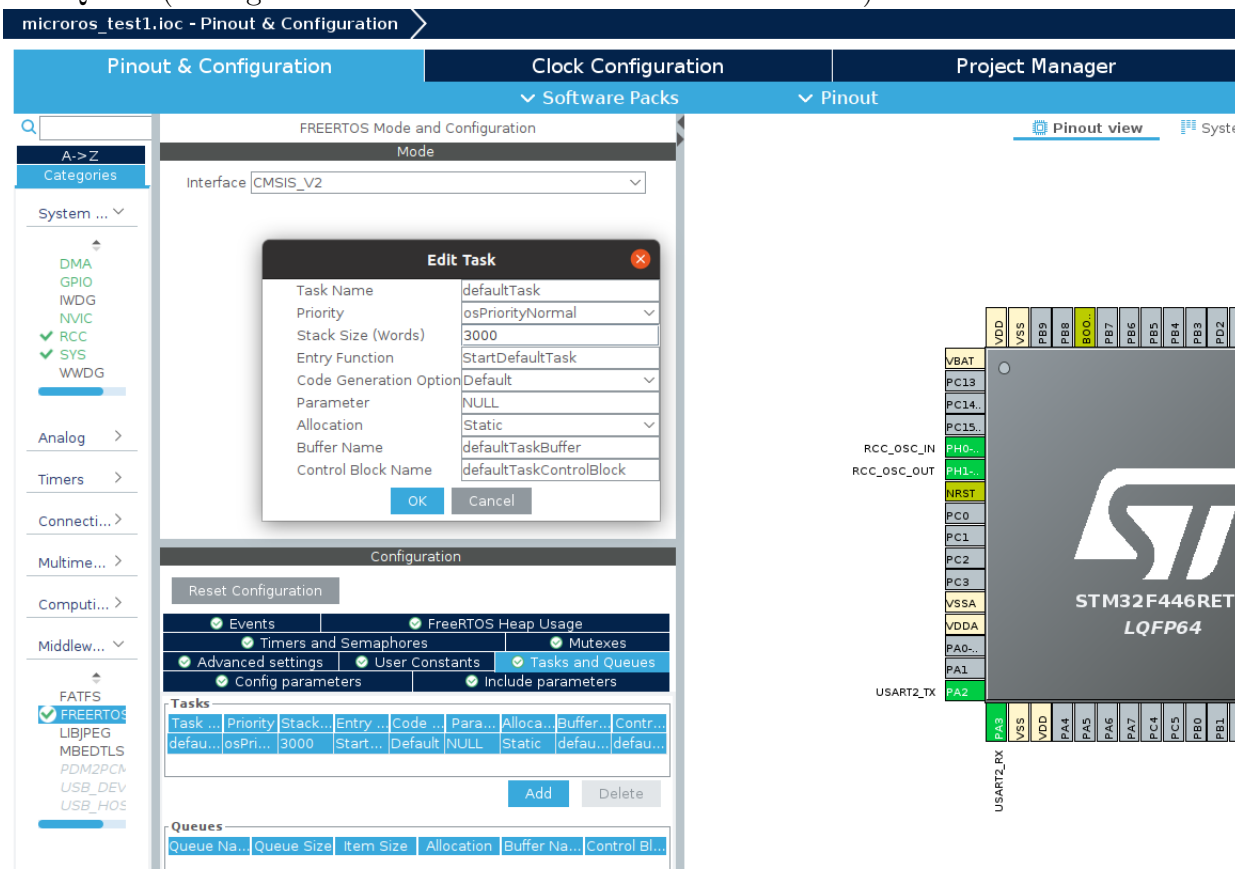
3. **Example1: Subscription and Publisher msg(data)**

1.1 **How to Configure STM32**

- **Step1:**Go to System − > TIMbase Source − > Click TIM1 and Go to RCC − > High Speed Clock − > click Crystal/ceramic Resonator.
- **Step2:** Go to USART2 − > click Asynchronous and NVIC Setting − > Click global Interrupt.



- **Step3:** Go to Middleware − > FreeRtos − > Click CMSIS_V2 and Go to Tasks and Queue (Change Stack Size to 3000 and allocation to static)



- **Step4:** Go to terminal and open file which create project file STM32 after git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.git src/micro_ros_setup
- **Step5:** login folder micro_ros_stm32cubemx_utils − > extra_sources copy file (microros_time.c,microros_allocators.c,custom_memory_manager.c, microros_transports/dm

enter to stm32 create new project − > core − > src .

- **Step6:** Click on project file − > properties − > C/C++ Builds − > Setting − > Build Step − > Pre-build steps add (docker pull microros/micro_ros_static_library_build && docker run –rm -v $workspace_loc : $ProjName : /project−−envMICROROS_LIBRA micro_ros_stm32cubemx_utils/microros_static_library_idemicroros/micro_ros_static_libr foxy)**Step7:***Addmicro − ROSincludedirectory.InProject-¿Settings-¿C/C + +Build-¿Settings-¿ToolSettingsTab-¿MCUGCCCompiler-¿Includepathsaddmicro_ros_s*
- **Step8:** Add the micro-ROS precompiled library. In Project − > Settings − > C/C++ Build − > Settings − > MCU GCC Linker − > Libraries
  - add ../micro_ros_stm32cubemx_utils/microros_static_library_ide/libmicroros in Library search path (-L)
  - add microros in Libraries (-l)

1.2 How to Setup Code in STM32

- **Step1:**Include library

```
/* Private includes ----------------------------------------------------------
/* USER CODE BEGIN Includes */
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <uxr/client/transport.h>
#include <rmw_microxrcedds_c/config.h>
#include <rmw_microros/rmw_microros.h>

#include <std_msgs/msg/int32.h>

#include "usart.h"
/* USER CODE END Includes */
```

- **Step2:**Input Function

```
/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN FunctionPrototypes */
bool cubemx_transport_open(struct uxrCustomTransport * transport);
bool cubemx_transport_close(struct uxrCustomTransport * transport);
size_t cubemx_transport_write(struct uxrCustomTransport* transport, const uint8_t * buf, size_t len, uint8_t * e
size_t cubemx_transport_read(struct uxrCustomTransport* transport, uint8_t* buf, size_t len, int timeout, uint8_

void * microros_allocate(size_t size, void * state);
void microros_deallocate(void * pointer, void * state);
void * microros_reallocate(void * pointer, size_t size, void * state);
void * microros_zero_allocate(size_t number_of_elements, size_t size_of_element, void * state);

/* USER CODE END FunctionPrototypes */
```

- **Step3:**Creat Task
  /* USER CODE BEGIN Init */
  defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);
  /* USER CODE END Init */
- **Step4:**Take this code enter to void StartDefaultTask(void *argument)

```c
/* Infinite loop */
    rmw_uros_set_custom_transport(
          true,
          (void *) &huart2,
          cubemx_transport_open,
          cubemx_transport_close,
          cubemx_transport_write,
          cubemx_transport_read);

        rcl_allocator_t freeRTOS_allocator = rcutils_get_zero_initialized_allocator();
        freeRTOS_allocator.allocate = microros_allocate;
        freeRTOS_allocator.deallocate = microros_deallocate;
        freeRTOS_allocator.reallocate = microros_reallocate;
        freeRTOS_allocator.zero_allocate =  microros_zero_allocate;

        if (!rcutils_set_default_allocator(&freeRTOS_allocator)) {
            printf("Error on default allocators (line %d)\n", __LINE__);
        }

        // micro-ROS app

        rcl_publisher_t publisher;
        std_msgs__msg__Int32 msg;
        rclc_support_t support;
        rcl_allocator_t allocator;
        rcl_node_t node;

        allocator = rcl_get_default_allocator();

        //create init_options
        rclc_support_init(&support, 0, NULL, &allocator);

        // create node
        rclc_node_init_default(&node, "cubemx_node", "", &support);

        // create publisher
        rclc_publisher_init_default(
          &publisher,
          &node,
          ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
          "cubemx_publisher");

        msg.data = 0;

        for(;;)
        {
          rcl_ret_t ret = rcl_publish(&publisher, &msg, NULL);
          if (ret != RCL_RET_OK)
          {
            printf("Error publishing (line %d)\n", __LINE__);
          }

          msg.data++;
          osDelay(10);
        }
```

1.3 Code Subscription in ROS2 foxy

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Int32

class MySubscriber(Node):
    def __init__(self):
        super().__init__('my_subscriber')
        self.subscription = self.create_subscription(
            Int32,
            'cubemx_publisher'
            ,self.callback
            ,10
         )
        self.subscriptions

    def callback(self,msg):
        self.get_logger().info('reveived: %d' % msg.data)


def main(args = None):
    rclpy.init(args=args)
    node = MySubscriber()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

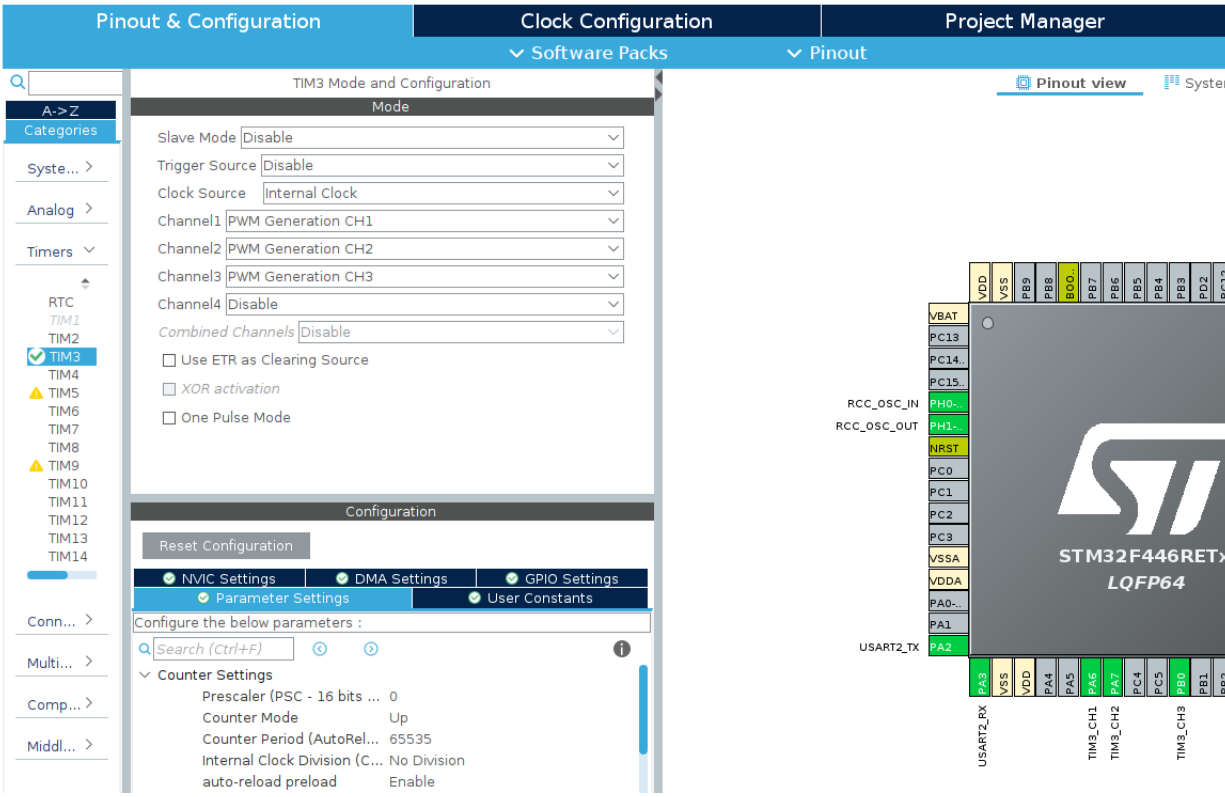
1.4 How to run code

- **Step1:** (sudo chmod 666 /var/run/docker.sock) to Debug code in Stm32 support.
- **Step2:** open workspace microros (cd microros_ws/) and run (source install/local_setup.bash and run (ros2 run micro_ros_agent micro_ros_agent serial -b 115200 –dev /dev/ttyACM0) create microros agent to the communication between your embedded controller and the rest of your ROS 2 software.
- **Step3:** open new terminal run (ros2 topic echo /cubemx_publisher to check data from stm32 which publisher to ROS2.
- **Step4:** RUN (ROS2 subscription)

4. **Example2:Publish twist from ROS2 and STM32 Subscription twist**

2.1 How to configure STM32 and Setup microro with Stm32

- **Step1:** Follow flow of Example1
- **Step2:** Configure Timer_PWM to Blink LED through PWM signal



2.2 How to Setup code in STM32

- **Step1:**Include library

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdbool.h>
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <uxr/client/transport.h>
#include <rmw_microxrcedds_c/config.h>
#include <rmw_microros/rmw_microros.h>

#include <std_msgs/msg/int32.h>
#include <std_msgs/msg/float32.h>
#include <std_msgs/msg/string.h>
#include "geometry_msgs/msg/twist.h"
#include <geometry_msgs/msg/vector3.h>

#include "usart.h"
#include "tim.h"
/* USER CODE END Includes */
```

- **Step2:**Input Function

```
/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN FunctionPrototypes */
bool cubemx_transport_open(struct uxrCustomTransport * transport);
bool cubemx_transport_close(struct uxrCustomTransport * transport);
size_t cubemx_transport_write(struct uxrCustomTransport* transport, const uint8_t * buf, size_t len, uint8_t * err);
size_t cubemx_transport_read(struct uxrCustomTransport* transport, uint8_t* buf, size_t len, int timeout, uint8_t* err);

void * microros_allocate(size_t size, void * state);
void microros_deallocate(void * pointer, void * state);
void * microros_reallocate(void * pointer, size_t size, void * state);
void * microros_zero_allocate(size_t number_of_elements, size_t size_of_element, void * state);

float map(float Input, float Min_Input , float Max_Input ,float Min_Output, float Max_Output){

    return (float) ((Input - Min_Input) * (Max_Output - Min_Output) / (Max_Input - Min_Input) + Min_Output);
}

void subscription_callback(const void *msgin) {
    const geometry_msgs__msg__Twist *twist_msg = (const geometry_msgs__msg__Twist *)msgin;
    led1= twist_msg->linear.x;
    led2= twist_msg->linear.y;
    led3= twist_msg->angular.z;

    Red=map(led1,0,255,0,65535);
    Green=map(led2,0,255,0,65535);
    Blue=map(led3,0,255,0,65535);

    TIM3->CCR1=Red;
    TIM3->CCR2=Green;
    TIM3->CCR3=Blue;

}
/* USER CODE END FunctionPrototypes */
```

- **Step3:**Creat Task to (This task is support)
  /* USER CODE BEGIN Init */
  defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);
  /* USER CODE END Init */
- **Step4:**Take this code enter to void StartDefaultTask(void *argument)

```
/* USER CODE BEGIN StartDefaultTask */
/* Infinite loop */
  rmw_uros_set_custom_transport(
              true,
              (void *) &huart2,
              cubemx_transport_open,
              cubemx_transport_close,
              cubemx_transport_write,
              cubemx_transport_read);
            rcl_allocator_t freeRTOS_allocator = rcutils_get_zero_initialized_allocator();
            freeRTOS_allocator.allocate = microros_allocate;
            freeRTOS_allocator.deallocate = microros_deallocate;
            freeRTOS_allocator.reallocate = microros_reallocate;
            freeRTOS_allocator.zero_allocate =  microros_zero_allocate;
            if (!rcutils_set_default_allocator(&freeRTOS_allocator)) {
                printf("Error on default allocators (line %d)\n", __LINE__);
            }
            // micro-ROS App //
            rclc_executor_t executor;
            // Initialize micro-ROS allocator
            rcl_allocator_t allocator;
            allocator = rcl_get_default_allocator();
            // Initialize support object
            rclc_support_t support;
            rclc_support_init(&support, 0, NULL, &allocator);
            // Create node object
            rcl_node_t node;
            rclc_node_init_default(&node, "stm32f446re_node", "", &support);
            //Create Subscription
            const char * sub_topic_name = "/cmd_vel";
            const rosidl_message_type_support_t * sub_type_support = ROSIDL_GET_MSG_TYPE_SUPPORT(geometry_msgs, msg, Twist);
            rclc_subscription_init_default(
                &subscriber,
                &node,
                sub_type_support,
                sub_topic_name);
             // create executor
            rclc_executor_init(&executor, &support.context, 1, &allocator);
            rclc_executor_add_subscription(&executor, &subscriber, &twist_msg, &subscription_callback, ON_NEW_DATA);
            // Spin executor to receive messages
            rclc_executor_prepare(&executor);
            rclc_executor_spin(&executor);
              for(;;)
              {
                  rcl_ret_t ret;
                ret = rcl_subscription_fini(&subscriber, &node);
                if (ret != RCL_RET_OK)
                {
                  printf("Error publishing (line %d)\n", __LINE__);
                }
                osDelay(10);
              }

/* USER CODE END StartDefaultTask */
```

2.3 Code Publisher in ROS2 foxy

```python
import sys
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class MyPublish(Node):
    def __init__(self):
        super().__init__('twist_publish')
        self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 1)
        timer_period = 0.5  # seconds
        self.i = 0.0
        self.timer_ = self.create_timer(timer_period, self.publish_message)

    def publish_message(self):
        message = Twist()
        message.linear.x = float(sys.argv[1])
        message.linear.y = float(sys.argv[2])
        message.angular.z = float(sys.argv[3])
        self.get_logger().info('Sending - Linear Velocity X : %f, Linear Velocity Y : %f, Angular Velocity :
                               % (message.linear.x, message.linear.y, message.angular.z))
        self.publisher_.publish(message)


def main(args=None):
    rclpy.init(args=args)
    minimal_publisher = MyPublish()
    rclpy.spin(minimal_publisher)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
if __name__ == '__main__':
    main()
```

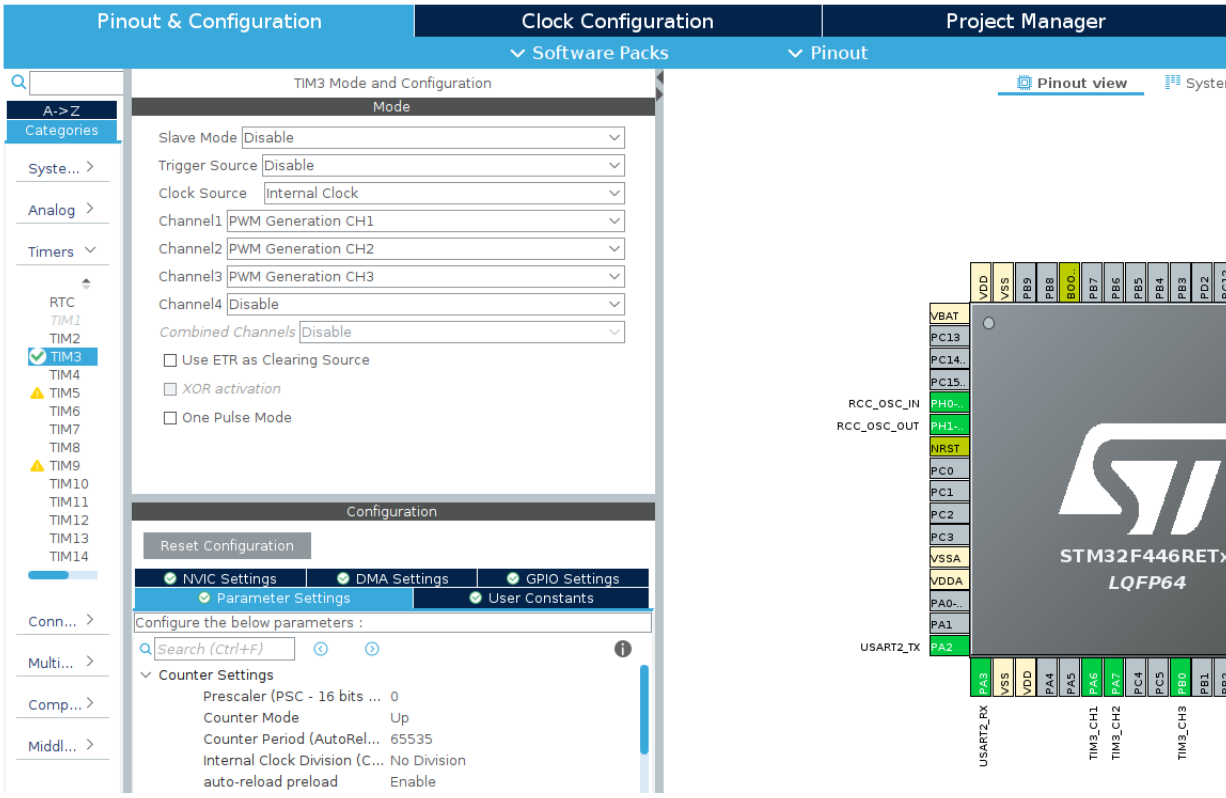
2.4 How to run code

- **Step1:** (sudo chmod 666 /var/run/docker.sock) to Debug code in Stm32 support.
- **Step2:** open workspace microros (cd microros_ws/) and run (source install/local_setup.bash and run (ros2 run micro_ros_agent micro_ros_agent serial -b 115200 –dev /dev/ttyACM0) create microros agent to the communication between your embedded controller and the rest of your ROS 2 software.
- **Step3:** open new terminal run (ros2 topic echo /cmd_vel to check data from ROS2 which publisher twist to STM32.
- **Step4:** RUN (ROS2 Publisher twist)

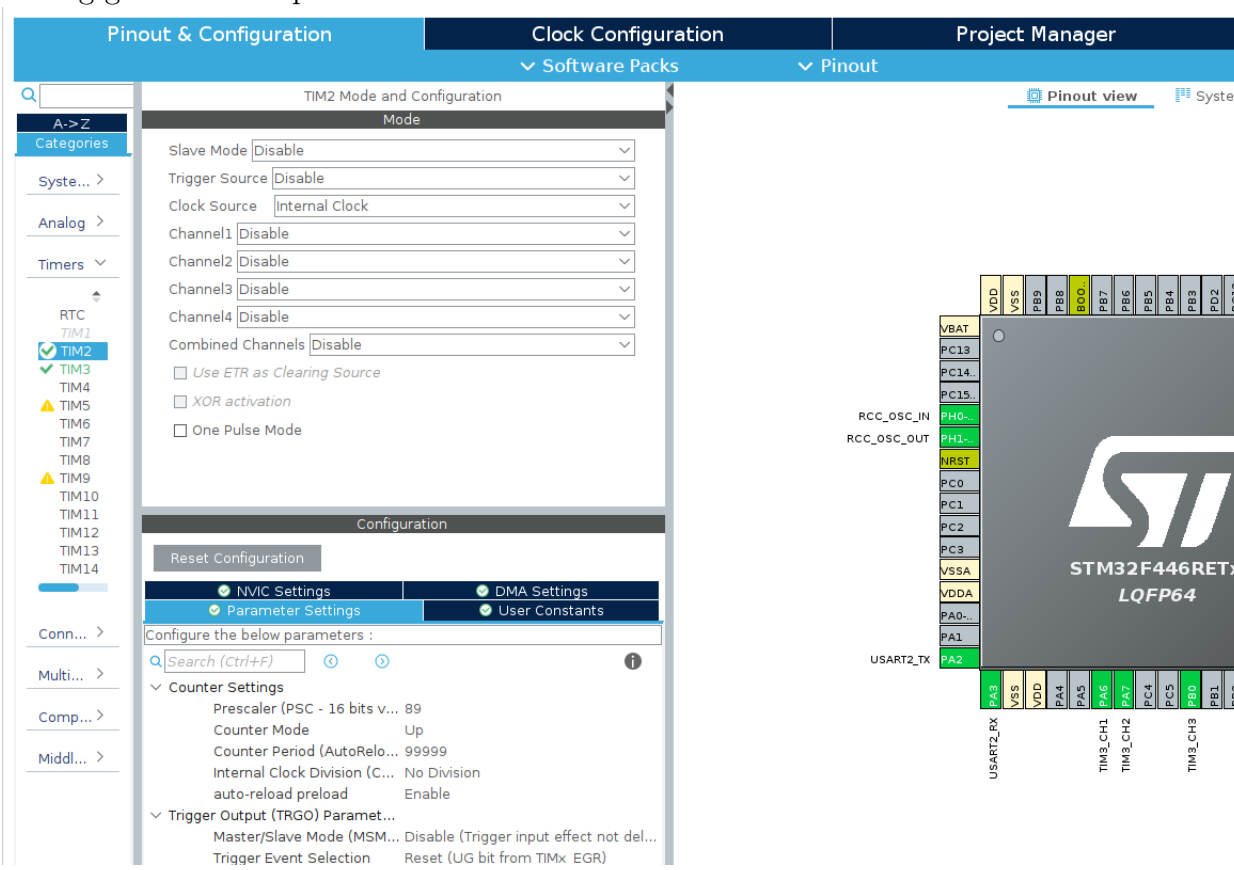
5. Example3:Client two_ints From ROS2 to STM32 (Service)

3.1 How to configure STM32 and Setup microros with Stm32

- **Step1:** Follow flow of Example1
- **Step2:** Configure Timer3_PWM to Blink LED through PWM signal



- **Step3:** Configure Timer2 Internal clock to make clock 0.01s=$\frac{90MHz}{90\times10000}$ and Nvic setting global Interrupt



3.2 How to Setup Code in STM32

- **Step1:**Include library

```c
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdbool.h>
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <uxr/client/transport.h>
#include <rmw_microxrcedds_c/config.h>
#include <rmw_microros/rmw_microros.h>

#include <std_msgs/msg/int32.h>
#include <std_msgs/msg/float32.h>
#include <std_msgs/msg/string.h>
#include "geometry_msgs/msg/twist.h"
#include <geometry_msgs/msg/vector3.h>
#include <example_interfaces/srv/add_two_ints.h>
#include <std_msgs/msg/int64.h>

#include "usart.h"
```

- **Step2:**Input Function

```c
/* USER CODE BEGIN Variables */
rcl_service_t service;
rcl_wait_set_t wait_set;
float led1;
float pwm_led;
example_interfaces__srv__AddTwoInts_Response res;
example_interfaces__srv__AddTwoInts_Request req;

/* USER CODE END Variables */
/* Definitions for defaultTask */
osThreadId_t defaultTaskHandle;
uint32_t defaultTaskBuffer[ 3000 ];
osStaticThreadDef_t defaultTaskControlBlock;
const osThreadAttr_t defaultTask_attributes = {
  .name = "defaultTask",
  .cb_mem = &defaultTaskControlBlock,
  .cb_size = sizeof(defaultTaskControlBlock),
  .stack_mem = &defaultTaskBuffer[0],
  .stack_size = sizeof(defaultTaskBuffer),
  .priority = (osPriority_t) osPriorityNormal,
};

/* Private function prototypes -----------------------------------------------*/
/* USER CODE BEGIN FunctionPrototypes */
bool cubemx_transport_open(struct uxrCustomTransport * transport);
bool cubemx_transport_close(struct uxrCustomTransport * transport);
size_t cubemx_transport_write(struct uxrCustomTransport* transport, const uint8_t * buf, size_t len, uint8_t * err);
size_t cubemx_transport_read(struct uxrCustomTransport* transport, uint8_t* buf, size_t len, int timeout, uint8_t* err);

void * microros_allocate(size_t size, void * state);
void microros_deallocate(void * pointer, void * state);
void * microros_reallocate(void * pointer, size_t size, void * state);
void * microros_zero_allocate(size_t number_of_elements, size_t size_of_element, void * state);
float map(float Input, float Min_Input , float Max_Input ,float Min_Output, float Max_Output){

    return (float) ((Input - Min_Input) * (Max_Output - Min_Output) / (Max_Input - Min_Input) + Min_Output);
}

void service_callback(const void * req, void * res)
{
    example_interfaces__srv__AddTwoInts_Request * req_in = (example_interfaces__srv__AddTwoInts_Request *) req;
    example_interfaces__srv__AddTwoInts_Response * res_in = (example_interfaces__srv__AddTwoInts_Response *) res;

    res_in->sum = req_in->a + req_in->b;
    led1=res_in->sum;
    pwm_led=map(led1, 0, 255, 0, 65535);

}
/* USER CODE END FunctionPrototypes */
```

- **Step3:**Creat Task
  /* USER CODE BEGIN Init */
  defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);
  /* USER CODE END Init */
- **Step4:**Take this code enter to void StartDefaultTask(void *argument)

```c
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void *argument)
{
  /* USER CODE BEGIN StartDefaultTask */
  /* Infinite loop */
    rmw_uros_set_custom_transport(
     true,
    (void *) &huart2,
    cubemx_transport_open,
    cubemx_transport_close,
    cubemx_transport_write,
    cubemx_transport_read);
    rcl_allocator_t freeRTOS_allocator = rcutils_get_zero_initialized_allocator();
    freeRTOS_allocator.allocate = microros_allocate;
    freeRTOS_allocator.deallocate = microros_deallocate;
    freeRTOS_allocator.reallocate = microros_reallocate;
    freeRTOS_allocator.zero_allocate =  microros_zero_allocate;

    if (!rcutils_set_default_allocator(&freeRTOS_allocator)) {
        printf("Error on default allocators (line %d)\n", __LINE__);
    }
    // micro-ROS App //
     rclc_executor_t executor;
    // Initialize micro-ROS allocator
    rcl_allocator_t allocator;
    allocator = rcl_get_default_allocator();
    // Initialize support object
     rclc_support_t support;
    rclc_support_init(&support, 0, NULL, &allocator);
    // Create node object
    rcl_node_t node;
    rclc_node_init_default(&node, "stm32f446re_node", "", &support);
    //Create service
    const char * service_name = "/add_two_ints";
    rclc_service_init_default(&service, &node,ROSIDL_GET_SRV_TYPE_SUPPORT(example_interfaces, srv, AddTwoInts),service_name);
    // Create executor
    rclc_executor_init(&executor, &support.context, 1, &allocator);
    rclc_executor_add_service(&executor, &service, &req, &res, service_callback);
    // Spin executor to receive messages
    rclc_executor_prepare(&executor);
    rclc_executor_spin(&executor);
  for(;;)
  {
      rcl_ret_t ret;
      ret = rcl_service_fini(&service, &node);
      if (ret != RCL_RET_OK)
      {
        printf("Error publishing (line %d)\n", __LINE__);
      }
      osDelay(10);
  }
  /* USER CODE END StartDefaultTask */
```

- **Step5:**Input code run pwm_timer in main.c

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM1) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */
  if (htim->Instance==TIM2)
    {

        TIM3->CCR1=pwm_led1;
        TIM3->CCR1=pwm_led2;
        TIM3->CCR1=pwm_led3;
    }

  /* USER CODE END Callback 1 */
}

```

3.3 Code client two_ints in ROS2 foxy

```python
import sys
from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node
class MinimalClientAsync(Node):
    def __init__(self):
        super().__init__('client_two_ints')
        self.cli = self.create_client(AddTwoInts, '/add_two_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddTwoInts.Request()
    def send_request(self, a, b):
        self.req.a = a
        self.req.b = b
        self.future = self.cli.call_async(self.req)
        rclpy.spin_until_future_complete(self, self.future)
        return self.future.result()
def main(args=None):
    rclpy.init(args=args)

    minimal_client = MinimalClientAsync()
    response = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    minimal_client.get_logger().info(
        'Result of add_two_ints: for %d + %d = %d' %
        (int(sys.argv[1]), int(sys.argv[2]), response.sum))

    minimal_client.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

3.4 How to run code

- **Step1:** (sudo chmod 666 /var/run/docker.sock) to Debug code in Stm32 support.
- **Step2:** open workspace microros (cd microros_ws/) and run (source install/local_setup.bash and run (ros2 run micro_ros_agent micro_ros_agent serial -b 115200 –dev /dev/ttyACM0) create microros agent to the communication between your embedded controller and the rest of your ROS 2 software.
- **Step3:**Debug and run STM32
- **Step4:** RUN (ROS2 Client two_ints)