Unit 5
Interrupts and Timers

Unit 5: Syllabus

Interrupts and Timers:

Types of interrupts, Nested vector interrupt controller (NVIC) in Cortex-M cores, Interrupt vectors, Priorities, Programming interrupts.

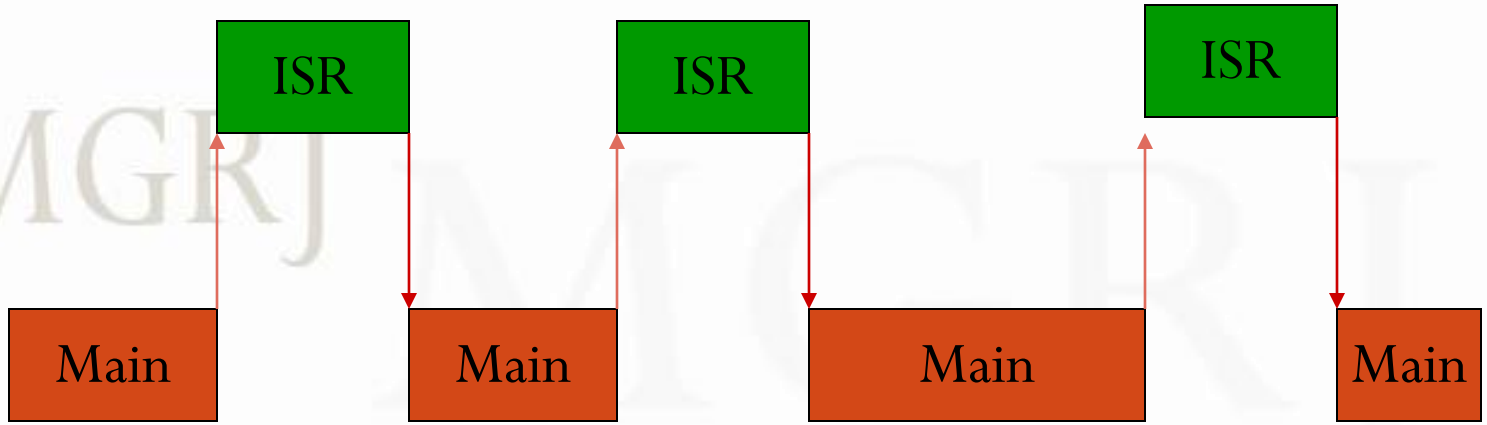
Timers, Controlling the operation, Programming with timers, Pulse width modulators, Programming modulators to generate PWM wave for given specifications.

Basics of Interrupts/Exceptions

- Interrupts are a common feature available in almost all microcontrollers.
- Interrupts are events typically generated by hardware (e.g., peripherals or external input pins) that cause changes in program flow control outside a normal programmed sequence (e.g., to provide service to a peripheral).
- When interrupt happens, the processor suspends the current executing task and executes a part of the program called the exception handler or Interrupt service routine(ISR).
- In Cortex-M processors, there are a number of exception sources.

- Interrupt causes main function to be suspended & the interrupt service routine(ISR) to run

Interrupt level execution (Back ground function)



Base-level Execution (Fore ground function)

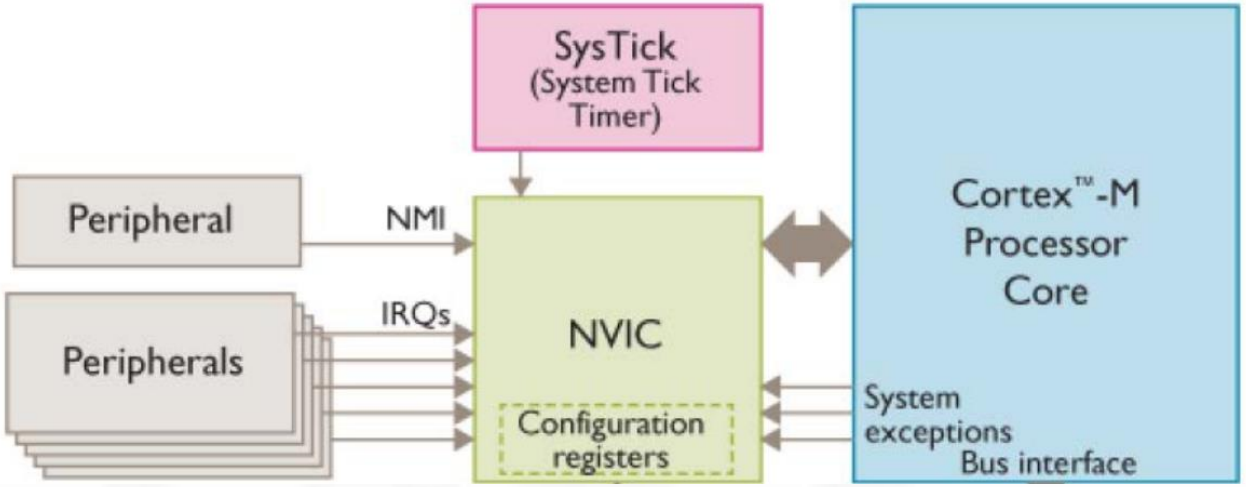
Interrupt/Exception (occurs asynchronously/Synchronously)

Types of interrupts

- **Hardware interrupt:** These occurs by the interrupt request signal from peripheral circuits. All hardware interrupts are asynchronous w.r.t of code execution because arrival is not known.
- **Software interrupt:** These occurs by executing a dedicated instruction. Software interrupts are synchronous as the occurrence these are dependent on instruction execution.

Nested Vectored Interrupt Controller(NVIC)

- An interrupt controller called NVIC supporting up to 240 interrupt requests and from 8 to 256 interrupt priority levels.
- Exceptions are processed by the NVIC. The NVIC can handle a number of Interrupt Requests (IRQs) and a Non-Maskable Interrupt (NMI) request.

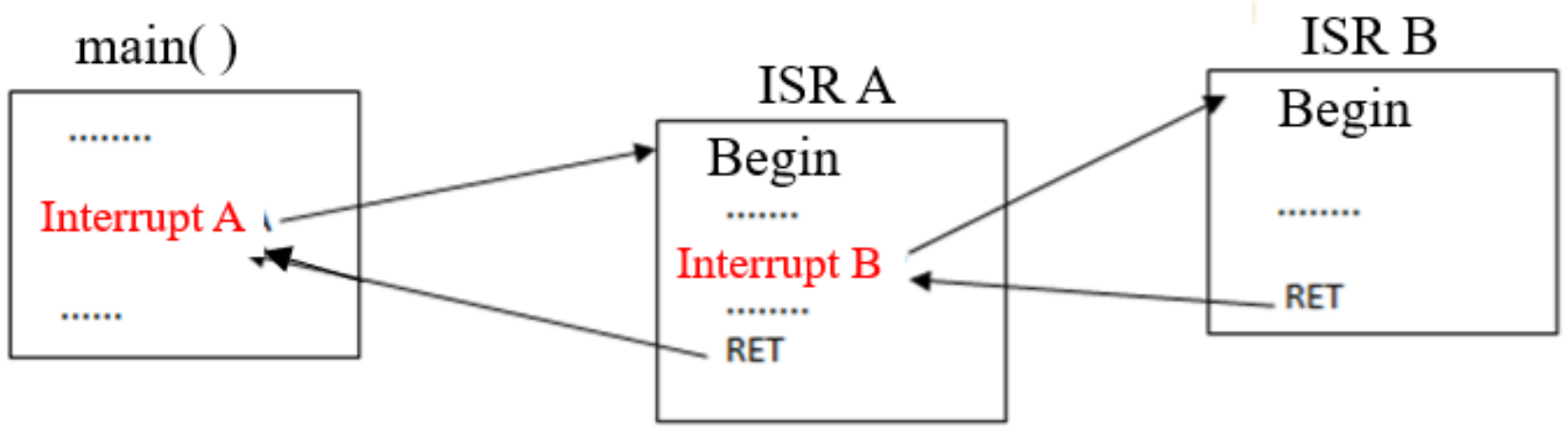


- Usually IRQs are generated by on-chip peripherals or from external interrupt inputs though I/O ports.
- Inside the processor there is also a timer called SysTick, which can generate a periodic timer interrupt request.

- Each exception source has an exception number. Exception numbers 1 to 15 are classified as system exceptions, and exceptions 16 and above are for interrupts.
- The design of the NVIC in the Cortex-M4 processors can support up to 240 interrupt inputs. However, in practice the number of interrupt inputs implemented in the design is far less, typically in the range of 16 to 100.
- The exception number is reflected in various registers, including the IPSR, and it is used to determine the exception vector addresses. Exception vectors are stored in a vector table.
- Reset is a special kind of exception. When the processor exits from a reset, it executes the reset handler in Thread mode.

Nesting of interrupts in NVIC

- NVIC supports nesting of interrupts. The working is shown in figure below. However, the context of ISR A must be saved on stack before passing the control to ISR B by user.

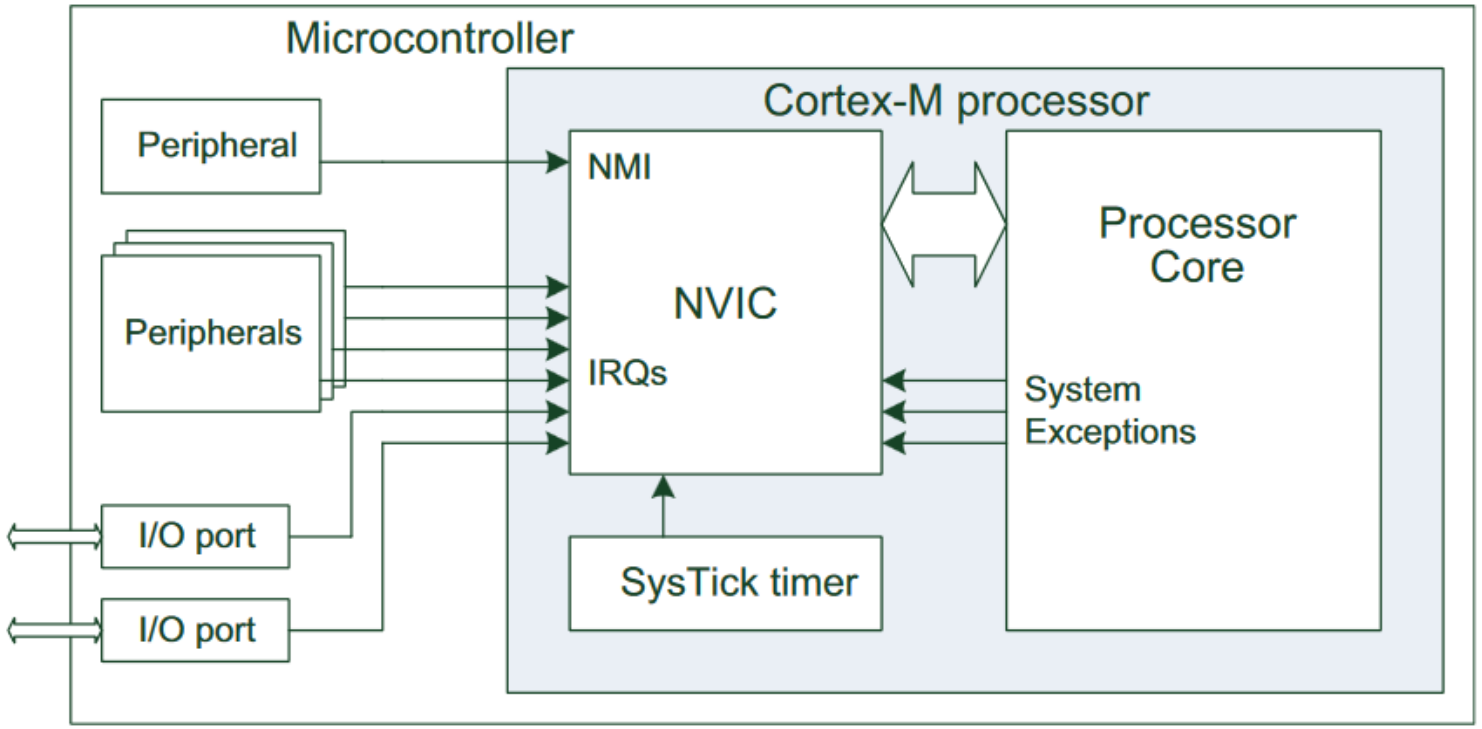


Interrupt B is higher or equal priority of interrupt A

Interrupt sources in Cortex M4 based MCUs

- Exceptions are numbered 1 to 15 for system exceptions and 16 and above for interrupt inputs (inputs to the processor, but not necessarily accessible on the I/O pins of the package).
- Most of the exceptions, including all interrupts, have programmable priorities, and a few system exceptions have fixed priority.
- The Cortex-M4 NVIC supports up to 240 IRQs (Interrupt Requests), a Non-Maskable Interrupt (NMI), a SysTick (System Tick) timer interrupt, and a number of system exceptions. Most of the IRQs are generated by peripherals such as timers, I/O ports, and communication interfaces (e.g., UART, I2C).
- The NMI is usually generated from peripherals like a watchdog timer or Brown-Out Detector (BOD). The rest of the exceptions are from the processor core. Interrupts can also be generated using software.

Interrupt sources in Cortex M4 based MCUs…



IRQs,NMI: These interrupt requests.

Exceptions: These are similar to interrupts but generated because of execution of an instruction or fault occurring while program execution.

Interrupt sources in Cortex M4 based MCUs…

- Exception types 1 to 15 are system exceptions (there is no exception type 0).

Exception type	Priority	Exception number	Descriptions
Reset	-3(Highest)	1	Reset
NMI	-2	2	Non-Maskable Interrupt (NMI), can be generated from on chip peripherals or from external sources.
Hard Fault	-1	3	All fault conditions, if the corresponding fault handler is not enabled
MemManage Fault	Programmable	4	Memory management fault
Bus Fault	Programmable	5	Bus error; usually occurs when AHB interface receives an error response from a bus slave (also called prefetch abort if it is an instruction fetch or data abort.
Usage Fault	Programmable	6	Exceptions due to program error or trying to access co-processor
Reserved	---	7-10	
SVC	Programmable	11	SuperVisor Call
Debug Monitor	Programmable	12	Debug monitor
Reserved	--	13	
PensSV	Programmable	14	Pendable service call;
SYSTICK	Programmable	15	System Tick Timer

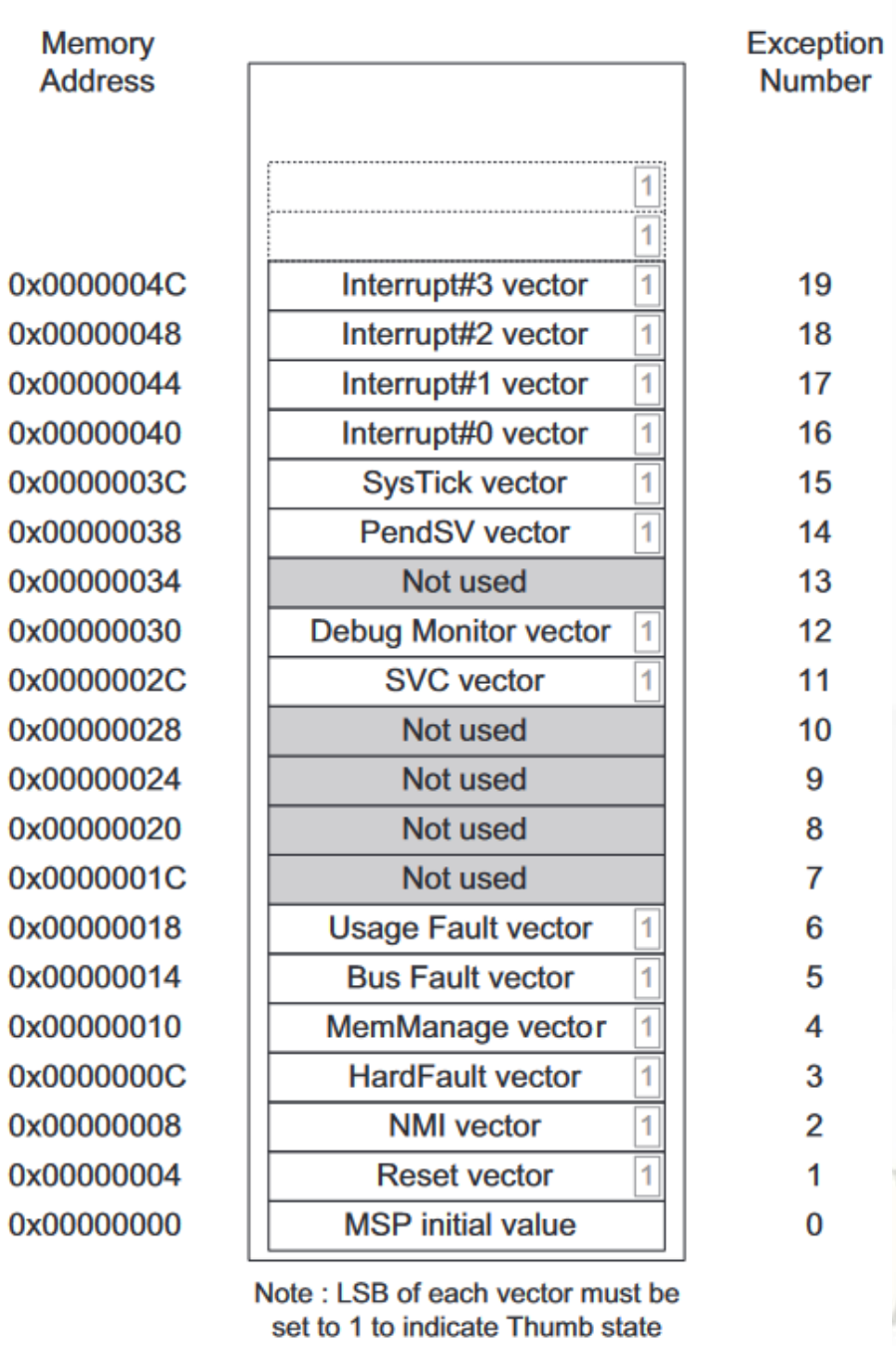
Interrupt sources in Cortex M4 based MCUs…

- Exceptions of type 16 or above are external interrupt inputs.

Exception type	Priority	Exception number	Descriptions
Interrupt #0	Programmable	16	It can be generated from on chip peripherals or from external sources.
Interrupt #1	Programmable	17	
----	---	--	
Interrupt #239	Programmable	255	

Vector Table

- When the Cortex-M processor accepts an exception request, the processor needs to determine the starting address of the exception handler (or ISR if the exception is an interrupt).
- By default, the vector table starts at memory address 0, and the vector address is arranged ac- cording to the exception number times four.



Vector Table of STM32F407VG

- The vector table is normally defined in the start up codes provided by the microcontroller vendors.

```
__Vectors       DCD     __initial_sp
                DCD     Reset_Handler
                DCD     NMI_Handler
                DCD     HardFault_Handler
                DCD     MemManage_Handler
                DCD     BusFault_Handler
                DCD     UsageFault_Handler
                DCD     0
                DCD     0
                DCD     0
                DCD     0
                DCD     SVC_Handler
                DCD     DebugMon_Handler
                DCD     0
                DCD     PendSV_Handler
                DCD     SysTick_Handler
```

```
; External Interrupts
DCD     WWDG_IRQHandler
DCD     PVD_IRQHandler
DCD     TAMP_STAMP_IRQHandler
DCD     RTC_WKUP_IRQHandler
DCD     FLASH_IRQHandler
DCD     RCC_IRQHandler
DCD     EXTI0_IRQHandler
DCD     EXTI1_IRQHandler
DCD     EXTI2_IRQHandler
DCD     EXTI3_IRQHandler
DCD     EXTI4_IRQHandler
DCD     DMA1_Stream0_IRQHandler
DCD     DMA1_Stream1_IRQHandler
DCD     DMA1_Stream2_IRQHandler
DCD     DMA1_Stream3_IRQHandler
DCD     DMA1_Stream4_IRQHandler
DCD     DMA1_Stream5_IRQHandler
DCD     DMA1_Stream6_IRQHandler
DCD     ADC_IRQHandler
DCD     CAN1_TX_IRQHandler
DCD     CAN1_RX0_IRQHandler
DCD     CAN1_RX1_IRQHandler
DCD     CAN1_SCE_IRQHandler
```

```
DCD     EXTI9_5_IRQHandler
DCD     TIM1_BRK_TIM9_IRQHandler
DCD     TIM1_UP_TIM10_IRQHandler
DCD     TIM1_TRG_COM_TIM11_IRQHandler
DCD     TIM1_CC_IRQHandler
DCD     TIM2_IRQHandler
DCD     TIM3_IRQHandler
DCD     TIM4_IRQHandler
DCD     I2C1_EV_IRQHandler
DCD     I2C1_ER_IRQHandler
DCD     I2C2_EV_IRQHandler
DCD     I2C2_ER_IRQHandler
DCD     SPI1_IRQHandler
DCD     SPI2_IRQHandler
DCD     USART1_IRQHandler
DCD     USART2_IRQHandler
DCD     USART3_IRQHandler
DCD     EXTI15_10_IRQHandler
DCD     RTC_Alarm_IRQHandler
DCD     OTG_FS_WKUP_IRQHandler
DCD     TIM8_BRK_TIM12_IRQHandler
DCD     TIM8_UP_TIM13_IRQHandler
DCD     TIM8_TRG_COM_TIM14_IRQHandler
DCD     TIM8_CC_IRQHandler
DCD     DMA1_Stream7_IRQHandler
DCD     FMC_IRQHandler
DCD     SDIO_IRQHandler
DCD     TIM5_IRQHandler
```

Vector Table of STM32F407VG…

```
DCD     SPI3_IRQHandler
DCD     UART4_IRQHandler
DCD     UART5_IRQHandler
DCD     TIM6_DAC_IRQHandler
DCD     TIM7_IRQHandler
DCD     DMA2_Stream0_IRQHandler
DCD     DMA2_Stream1_IRQHandler
DCD     DMA2_Stream2_IRQHandler
DCD     DMA2_Stream3_IRQHandler
DCD     DMA2_Stream4_IRQHandler
DCD     ETH_IRQHandler
DCD     ETH_WKUP_IRQHandler
DCD     CAN2_TX_IRQHandler
DCD     CAN2_RX0_IRQHandler
DCD     CAN2_RX1_IRQHandler
DCD     CAN2_SCE_IRQHandler
DCD     OTG_FS_IRQHandler
DCD     DMA2_Stream5_IRQHandler
DCD     DMA2_Stream6_IRQHandler
DCD     DMA2_Stream7_IRQHandler
DCD     USART6_IRQHandler
DCD     I2C3_EV_IRQHandler
DCD     I2C3_ER_IRQHandler
DCD     OTG_HS_EP1_OUT_IRQHandler
DCD     OTG_HS_EP1_IN_IRQHandler
DCD     OTG_HS_WKUP_IRQHandler
DCD     OTG_HS_IRQHandler
DCD     DCMI_IRQHandler
DCD     0
DCD     HASH_RNG_IRQHandler
DCD     FPU_IRQHandler
```

Vector table is defined in startup_stm32f407xx.s file

Exception Sequence

- When an exception occurs and is accepted by the processor, the core performs sequence of operations before passing control to ISR. The process is called exception sequence and is as follows:

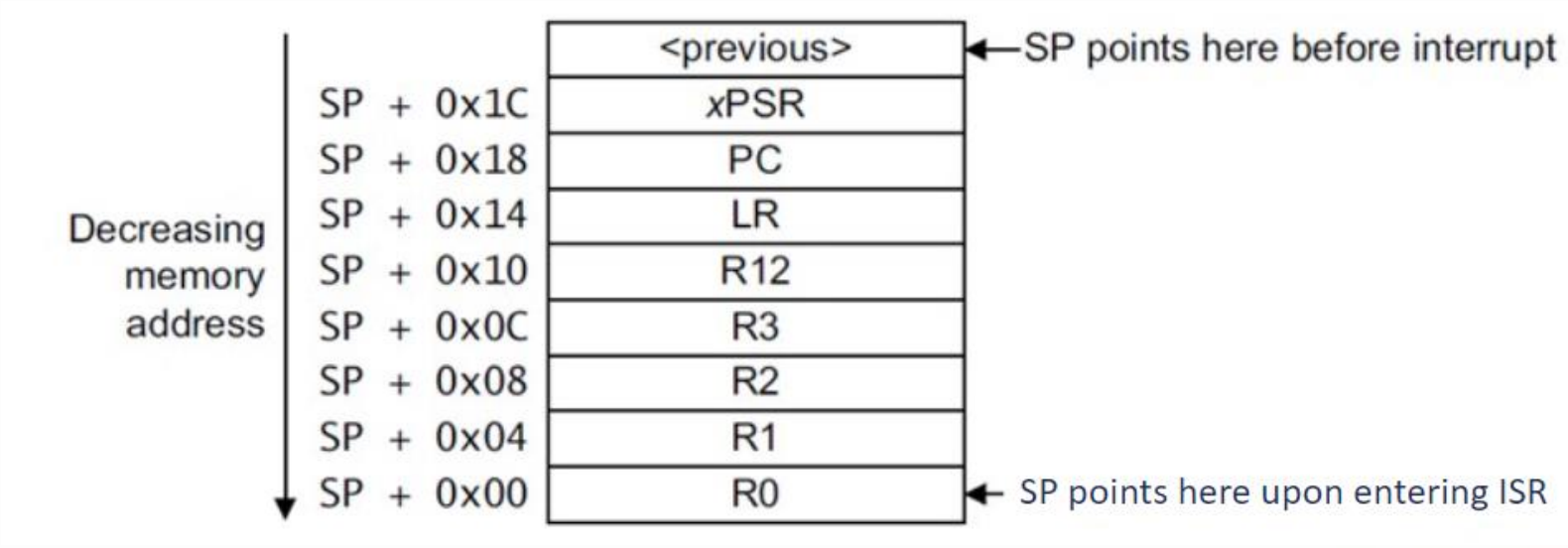
  - Finishes current instructions (except for lengthy instructions).
  - Push 8 32 bit words(xPSR, Return address, LR, R12,R3,R2,R1,R0) on to stack.
  - Switch to handler privileged mode and use MSP.
  - Load LR with EXC_RETRUN code.
  - Load IPSR with exception number.
  - Start execution of interrupt service routine.

EXC_RETURN: The value of this code is used to trigger the exception return mechanism when it is loaded into the Program Counter (PC) using BX, POP, or memory load instructions (LDR or LDM).

- **EXC_RETURN Example:**

  Value 0xFFFFFFF9 indicate Return to Thread mode and use the Main Stack for return.

- **The figure below shows storing of context on stack.**



Interrupt management

- The Cortex-M processors have a number of programmable registers for managing interrupts and exceptions.
- Most of these registers are inside the NVIC and System Control Block (SCB).
- There are also special registers inside the processor core for interrupt masking (e.g., PRIMASK, FAULTMASK, and BASEPRI).
- To make it easier to manage interrupts and exceptions, the CMSIS-Core provides a number of access functions.
- For general application programming, the best practice is to use the CMSIS-Core access functions.

Interrupt management…

- **Interrupt Making registers:**

-The **PRIMASK** register is used to disable all exceptions except NMI and HardFault.

- The **FAULTMASK** register is used to disable all exceptions except NMI.

- The **PRIMASK** register is used to disable all exceptions of priority equal to or less than certain value.

Interrupt management…

- In C programming, we can use the functions provided in CMSIS-Core for accessing masking registers.

```
void __enable_irq(); // Clear PRIMASK
void __disable_irq(); // Set PRIMASK
void __enable_fault_irq(void); // Clear FAULTMASK
void __disable_fault_irq(void); // Set FAULTMASK to disable interrupts
__set_BASEPRI(0x60); // Disable interrupts with priority
                     // 0x60-0xFF using CMSIS-Core function
```

Priorities

- In microcontrollers, whether and when an exception can be accepted by the processor and get its handler executed can be dependent on the priority of the exception.
- A higher-priority (smaller number in priority level) exception can pre-empt a lower-priority (larger number in priority level) exception; this is the nested exception/interrupt scenario.
- ***Reset, NMI, and HardFault*** have fixed priority levels and their priority levels are represented with negative numbers to indicate that they are of higher priority than other exceptions.
- Other exceptions have programmable priority levels, which range from ***0 to 255***.
- In C programming, we can use function provided in CMSIS-Core for accessing masking registers:

void NVIC_SetPriority(IRQn_Type IRQn, unit32_t Priority);

Priorities…

- IRQn can specify any device specific interrupt, or processor exception number.
- The number assignment in different interrupts in STM32F4xx is as follows(available in stm32f407xx.h file):

```
/******  Cortex-M4 Processor Exceptions Numbers ***********************************
  NonMaskableInt_IRQn          = -14,     /*!< 2 Non Maskable Interrupt
  MemoryManagement_IRQn        = -12,     /*!< 4 Cortex-M4 Memory Management Interrupt
  BusFault_IRQn                = -11,     /*!< 5 Cortex-M4 Bus Fault Interrupt
  UsageFault_IRQn              = -10,     /*!< 6 Cortex-M4 Usage Fault Interrupt
  SVCall_IRQn                  = -5,      /*!< 11 Cortex-M4 SV Call Interrupt
  DebugMonitor_IRQn            = -4,      /*!< 12 Cortex-M4 Debug Monitor Interrupt
  PendSV_IRQn                  = -2,      /*!< 14 Cortex-M4 Pend SV Interrupt
  SysTick_IRQn                 = -1,      /*!< 15 Cortex-M4 System Tick Interrupt
```

```
/******  STM32 specific Interrupt Numbers *****************************************************
  WWDG_IRQn                    = 0,       /*!< Window WatchDog Interrupt
  PVD_IRQn                     = 1,       /*!< PVD through EXTI Line detection Interrupt
  TAMP_STAMP_IRQn              = 2,       /*!< Tamper and TimeStamp interrupts through the EXTI line
  RTC_WKUP_IRQn                = 3,       /*!< RTC Wakeup interrupt through the EXTI line
  FLASH_IRQn                   = 4,       /*!< FLASH global Interrupt
  RCC_IRQn                     = 5,       /*!< RCC global Interrupt
  EXTI0_IRQn                   = 6,       /*!< EXTI Line0 Interrupt
  EXTI1_IRQn                   = 7,       /*!< EXTI Line1 Interrupt
  EXTI2_IRQn                   = 8,       /*!< EXTI Line2 Interrupt
  EXTI3_IRQn                   = 9,       /*!< EXTI Line3 Interrupt
  EXTI4_IRQn                   = 10,      /*!< EXTI Line4 Interrupt
  DMA1_Stream0_IRQn            = 11,      /*!< DMA1 Stream 0 global Interrupt
  DMA1_Stream1_IRQn            = 12,      /*!< DMA1 Stream 1 global Interrupt
  DMA1_Stream2_IRQn            = 13,      /*!< DMA1 Stream 2 global Interrupt
  DMA1_Stream3_IRQn            = 14,      /*!< DMA1 Stream 3 global Interrupt
```

Programming example in STM32CubeMX for using ADC interrupt

- Use STM32CubeMX to configure ADC1 of STM32F407VG MCU to following parameters:
- Channel: IN1(PA1), PCLK2:60 MHz, ADCCLK:30MHz, Sampling time:3 ADCCLKs, Resolution:12 bits, Triggering: S/W controlled, Analog input range(dynamic range):0 V to 3.3 V(DC)

Write application code in Keil IDE to use ADC to analog to digital conversion and display equivalent digital value in debug window and configure ADC in interrupt mode.

Note: This is 6(a) experiment of ADC configured to read digital value in polling mode. Now, with a few modifications, code can be made to read value in interrupt mode.

• HAL APIs of ADC used in experiment 6(a)

HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc)

This Enables ADC and starts conversion of the regular channels in polling mode.

Parameters: hadc pointer to a ADC_HandleTypeDef structure that contains the configuration information for the specified ADC.

Return type: HAL status

HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc, uint32_t Timeout)

Poll for regular conversion complete. ADC conversion flags EOS (end of sequence) and EOC (end of conversion) are cleared by this function.

Parameters: hadc pointer to a ADC_HandleTypeDef structure that contains the configuration information for the specified ADC.

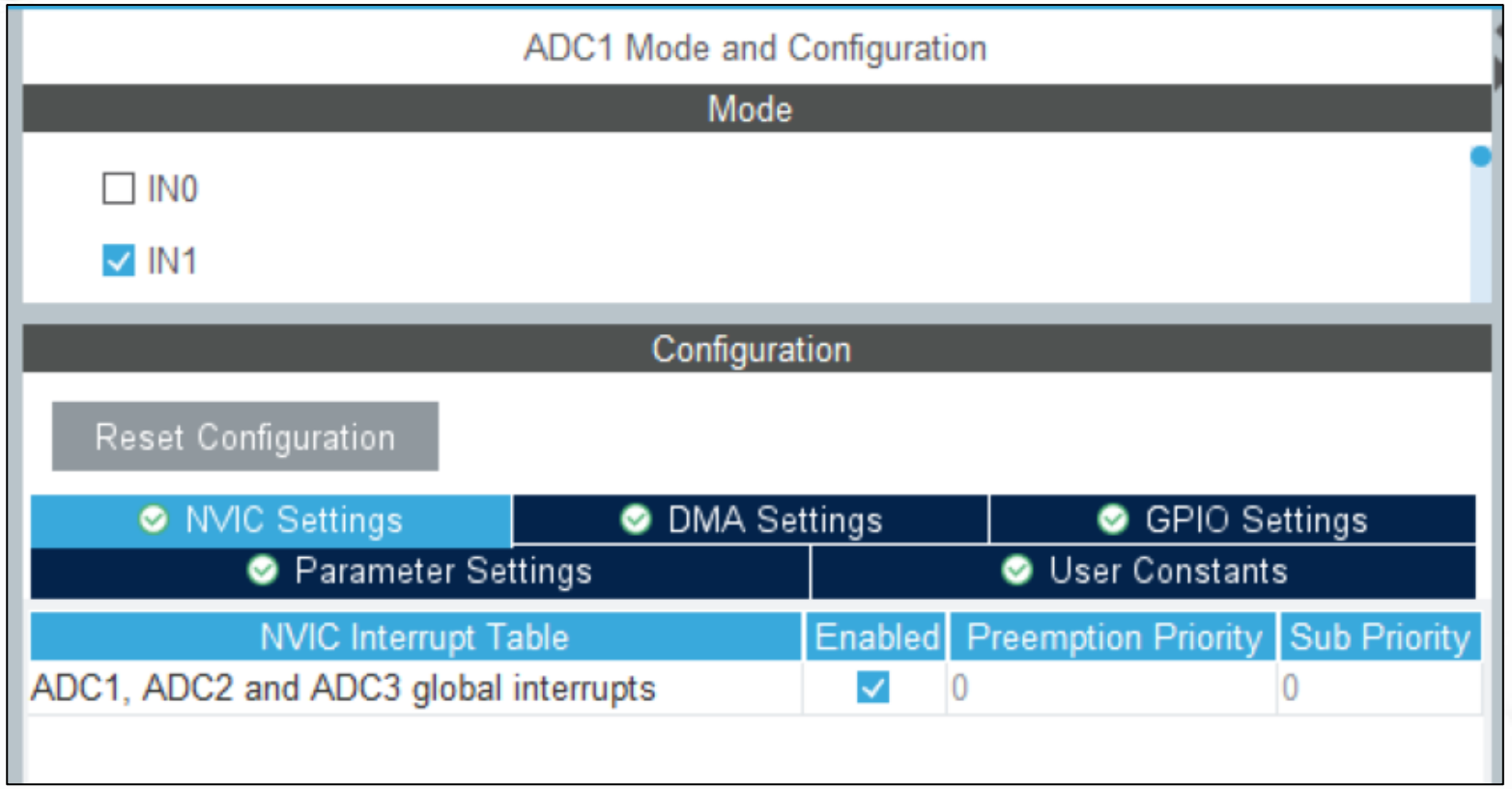
Timeout Timeout value in millisecond.

Return type: HAL status

uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc)

Gets the converted value from data register of regular channel.

- In STM32CubeMX, enable ADC1,ADC2 and ADC3 global interrupts under NVIC settings in ADC 1 configuration. Retain all the settings of experiment 6(a).



Program:

```
#include "main.h"
ADC_HandleTypeDef hadc1; // handler for ADC
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
uint32_t analogValue;    // 32- bit variable for ADC value
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
        analogValue=HAL_ADC_GetValue(&hadc1);
}
int main(void)
{
        HAL_Init();
        SystemClock_Config();
        MX_GPIO_Init();
        MX_ADC1_Init();
        HAL_ADC_Start_IT(&hadc1);
        while (1)
        {
        }
}
```

APIs generated:

HAL_StatusTypeDef HAL_ADC_Start_IT(ADC_HandleTypeDef* hadc)

Enables the interrupt and starts ADC conversion of regular channels.

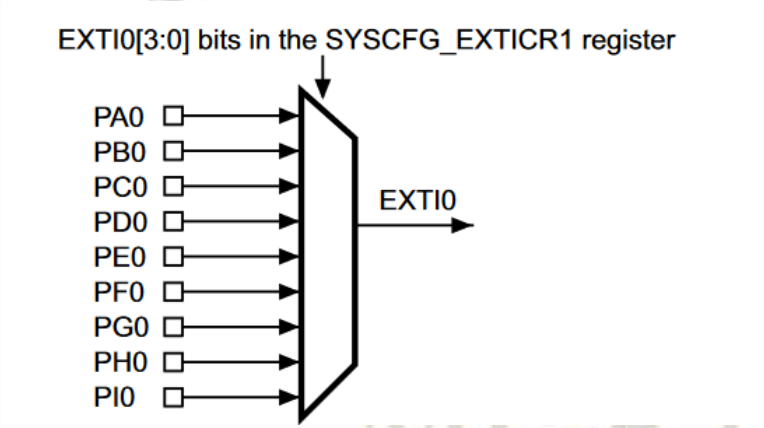
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)

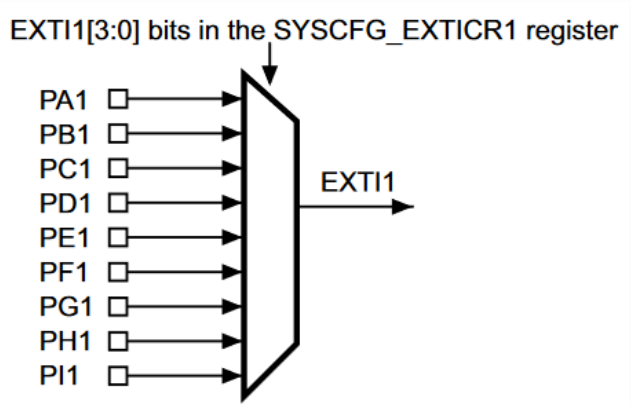
This function is called from HAL_ADC_IRQHandler( ) upon analog to digital conversion.

Note that, after initialization main function executes endless loop and statement in to read ADC value is executed upon interrupt generation.

Interrupts on GPIO pins(external interrupts)

- Up to 140 GPIOs of STM32F407xx are connected to the 16 external interrupt/event lines in the following manner:





- Similarly, interrupts on other GPIOs are used to generate EXTI2,EXTI3,…EXTI15..
- The external interrupt/event controller(EXTI) module in MCU is used to control these interrupts.
- The system configuration controller(SYSCFG) is used to manage the external interrupt line connection to the GPIOs

Registers for external interrupt control

SYSCFG external interrupt configuration register 1 (SYSCFG_EXTICR1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

Bits 15:0 **EXTIx[3:0]**: EXTI x configuration (x = 0 to 3)

These bits are written by software to select the source input for the EXTIx external interrupt.

0000: PA[x] pin
0001: PB[x] pin
0010: PC[x] pin
0011: PD[x] pin
0100: PE[x] pin
0101: PF[x] pin
0110: PG[x] pin
0111: PH[x] pin
1000: PI[x] pin
1001: PJ[x] pin

Interrupt mask register (EXTI_IMR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **MRx:** Interrupt mask on line x

0: Interrupt request from line x is masked

1: Interrupt request from line x is not masked

Rising trigger selection register (EXTI_RTSR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **TRx:** Rising trigger event configuration bit of line x

0: Rising trigger disabled (for Event and Interrupt) for input line

1: Rising trigger enabled (for Event and Interrupt) for input line

Falling trigger selection register (EXTI_FTSR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									TR22	TR21	TR20	TR19	TR18	TR17	TR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 22:0 **TRx:** Falling trigger event configuration bit of line x

0: Falling trigger disabled (for Event and Interrupt) for input line

1: Falling trigger enabled (for Event and Interrupt) for input line.

- For programming example, refer experiment 9 in lab

Timers

- Timer/counter are part of micro-controller internal hardware. Timer/ counter are same as per hardware point of view. Timer is used to generate delay and counter is for counting external events.
- When we want to count events, we connect external event (e.g. event : rising edge or falling edge) source to input pin of timer/counter module. When external event occurs the value of counter increments. Value of counter represents no. of external events.
- During timer operation, timers runs on clock signal applied to CPU where as counter run on external clock signal applied.

SysTick Timer

- The Cortex-M processors have a integrated timer called the SysTick (System Tick) timer. It is integrated as a part of the NVIC and can generate the SysTick exception (exception type #15).
- The SysTick timer is a simple decrement 24-bit timer, and can run on processor clock frequency or from a reference clock frequency (normally an on-chip clock source).
- It counts from an initial value down to 0. When it reaches 0, in the next clock, it underflows and it raises a flag called COUNT and reloads the initial value and starts over again. We can set the initial value to a value between 0x000000 to 0xFFFFFF.

SysTick Timer Registers

- When the counter is enabled by setting bit 0 of the **Control and Status register**, the **current value register** decrements at every processor clock cycle or every rising edge of the reference clock. If it reaches zero, it will then load the value from the reload **value register** and continue.
- An additional register called SysTick Calibration Register is available to allow the on-chip hardware to provide calibration information for the software.

Address	CMSIS-Core Symbol	Register
0xE000E010	SysTick->CTRL	SysTick Control and Status Register
0xE000E014	SysTick->LOAD	SysTick Reload Value Register
0xE000E018	SysTick->VAL	SysTick Current Value Register
0xE000E01C	SysTick->CALIB	SysTick Calibration Register

SysTick Timer Registers …

- If we only want to generate a periodic SysTick interrupt, the easiest way is to use a CMSIS-Core function called "SysTick_Config":

uint32_t SysTick_Config(uint32_t ticks);

- This function sets the SysTick interrupt interval to "ticks," enables the counter using the processor clock, and enables the SysTick exception with the lowest exception priority.
- For example, if we have a clock frequency of 30MHz and we want to trigger a SysTick exception of 1KHz, we can use:

SysTick_Config(30000);

Time to reach count 0=30000x 1/30 MHZ = 1 ms

so, exception rate= 1KHz

Timers in MCU

- In addition to SysTick timers, STM32Fxx Arm comes with a large number of timers. The timers are called TIMx where x=0, 1, 2, 3, 4,5,..,14.
- They fall into three categories:

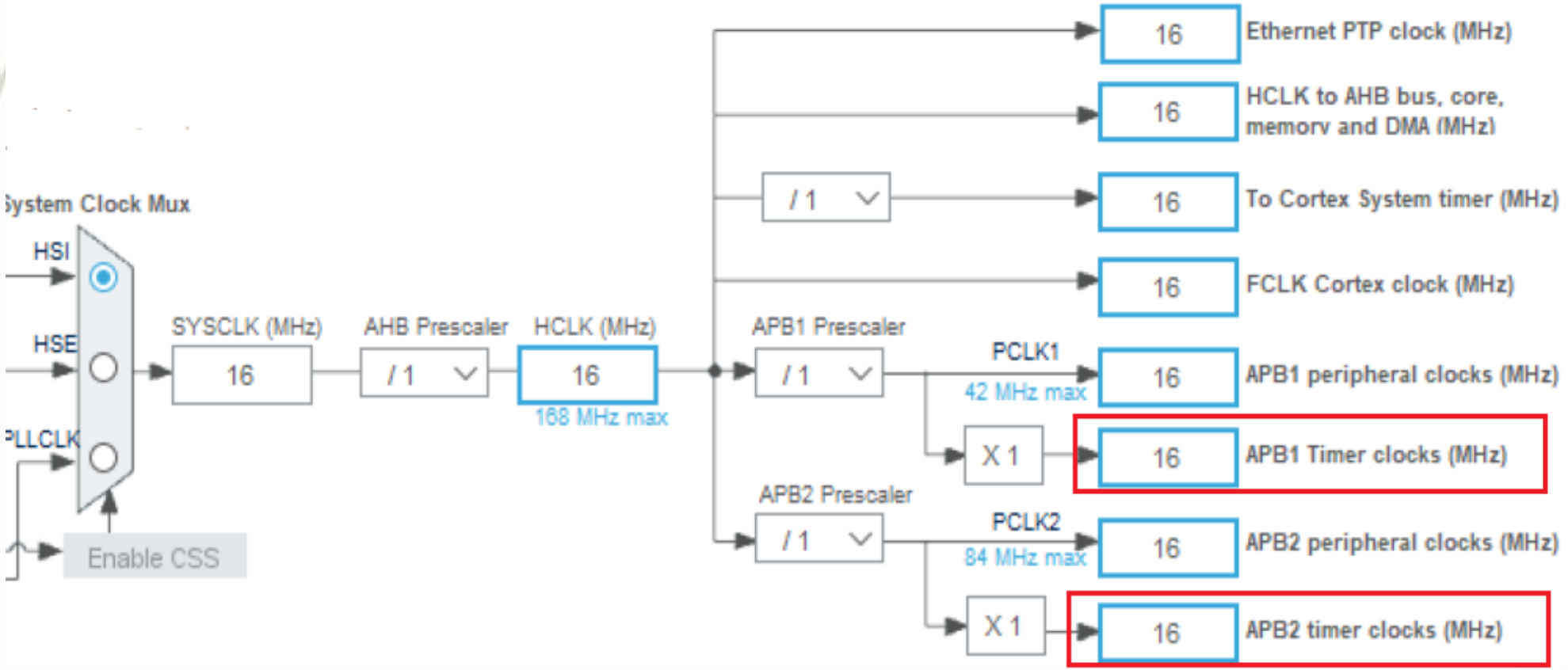
  a) Advanced Control Timer

  b) General Purpose Timer

  c) Basic Timer
- Capture mode – retrieves/store a timer value based on a signal event.
- Compare mode - constantly monitors a timer counter value and compares it to a value set in the application. Compare mode will trigger an event when the values match.

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	Capture/ compare channels
Advanced-control	TIM1, TIM8	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	4
General purpose	TIM2, TIM5	32-bit	Up, Down, Up/down	Any integer between 1 and 65536	4
	TIM3, TIM4	16-bit	Up, Down, Up/down	Any integer between 1 and 65536	4
	TIM9	16-bit	Up	Any integer between 1 and 65536	2
	TIM10, TIM11	16-bit	Up	Any integer between 1 and 65536	1
	TIM12	16-bit	Up	Any integer between 1 and 65536	2
	TIM13, TIM14	16-bit	Up	Any integer between 1 and 65536	1
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	0

Timers' clock



- Timer clocks can be reduced further by programming TIMx_PSC register to 1,2,3,4,..65536
- If APB1 and APB2 timer clock is 16 MHz, then clock signal to timers is 1 KHz if TIMx_PSC is set to 16000(16MHz/16000).

Registers of Timer(TIM2 to TIM9)

TIMx control register 1 (TIMx_CR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CEN (Counter Enable, D0): enables or disables the counter. When the CEN bit is set, the timer starts to count. It counts up or down depending on the DIR bit.

DIR (Direction, D4): This bit configures the Timer/Counter as an up or down counter. If the DIR bit is 0, the timer counts up. If the DIR bit is 1, the counter counts down.

Registers of Timer(TIM2 to TIM9)….

TIMx status register (TIMx_SR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

UIF (D0): This is like overflow flag in other microcontrollers. When the timer counts down from a starting value and reaches 0, the UIF is set high. In the case of up counter, the UIF goes HIGH when it reaches the top value and wraps around to zero. The UIF flag remains high until it is cleared by software.

Registers of Timer(TIM2 to TIM9)….

TIMx counter (TIMx_CNT)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNT[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

In the case of a 16-bit imer, TIMx_CNT is a 16-bit counter and can take values between 0 to 0xFFFF. For the 32-bit timers such as TIM2, the TIM2_CNT is 32 bit wide.

TIMx auto-reload register (TIMx_ARR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ARR[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- The counter is an up counter by default. It counts up from 0 to the auto- reload value. Notice that when the counter reaches the TIM_ARR value, the UIF flag in TIM_SR (status) register is set HIGH, the counter value is wrapped around to zero, then continues to count up.
- When the DIR bit of CR1 (Control 1) register is set (DIR=1), the counter counts from the autoreload value (content of the TIMx_ARR register) down to 0, then restarts from the auto-reload value and generates a counter underflow and the UIF flag is set.

TIMx prescaler (TIMx_PSC)

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

The prescaler slows down the counting speed of the timer by dividing the input clock of the timer.

Program(Register level Programming)

- Design STM32F407VG based system to toggle LED connected to PA5 at the rate of 10 Hz. Use Timer 2 to generate suitable delay.

Assume: APB1/APB2 Timer clock=16 MHz

If timer pre scale is set to 16000,then clock frequency =16M/16000=1 KHz

Timer clock period is 1 ms.

Rate of toggle=10Hz=> Delay=100 ms

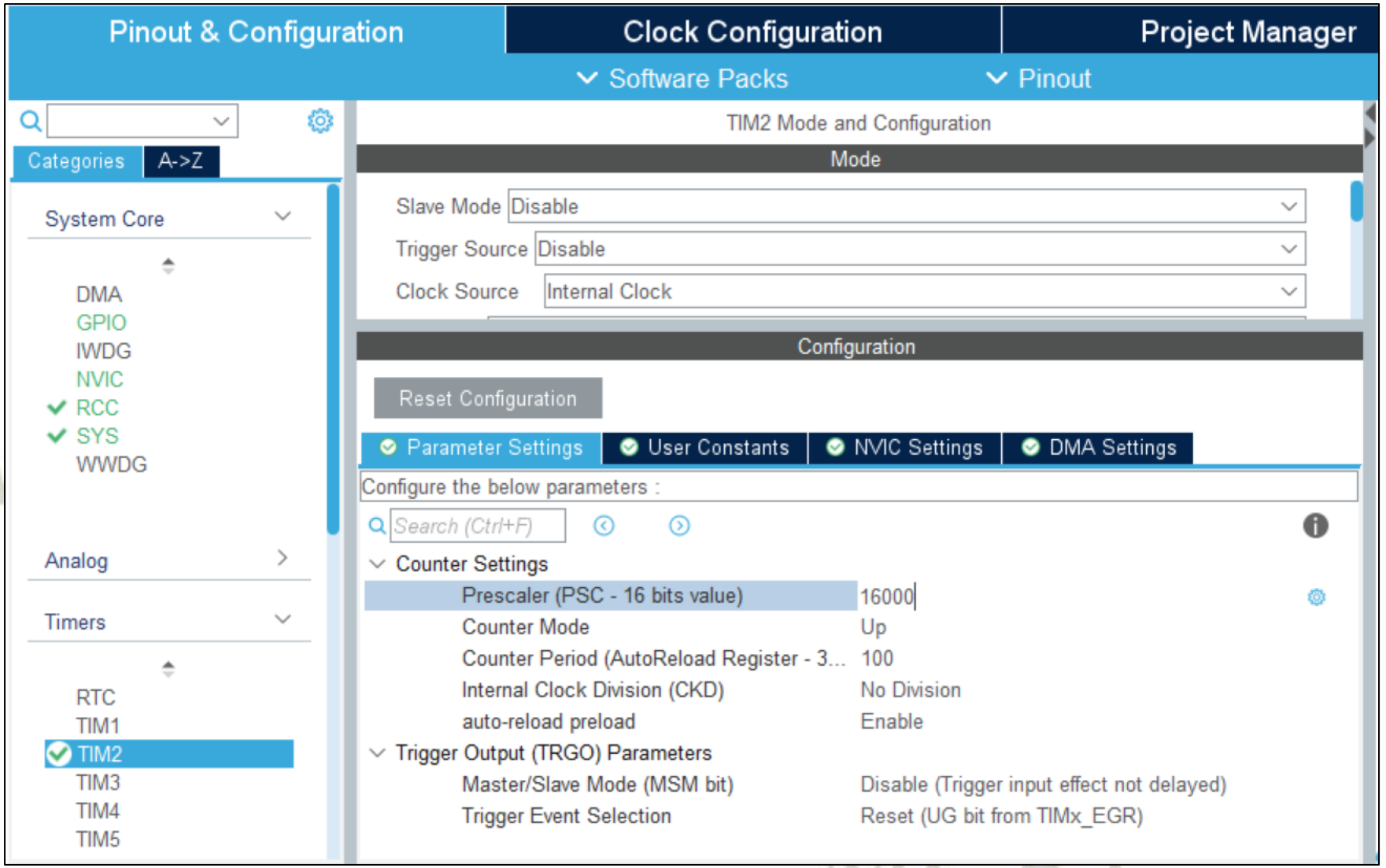
So, totally 100 states are required.

Program…

```
#include "stm32f4xx.h“
void main( ){
        RCC->AHB1ENR |= 1;                      //Enable GPIO clock
        GPIOA->MODER |= 0x00000400;             // set pin to output mode
        RCC->APB1ENR |= 1;                      //enable TIM2 clock
        TIM2->PSC = 16000;                      //divided by 16000
        TIM2->ARR = 100;
        TIM2->CNT = 0;                          //clear timer counter
        TIM2->CR1 = 1;                          //enable timer 2
        While(1){
                while(TIM2->SR & 0x01==0) ;             //wait till overflow flag is set
               TIM2->SR &= ~1;                          // Clear overflow flag
                GPIOA->ODR ^= 0x00000020;               // Toggle LED
                   }
          }
```

Redesign of Program using Interrupts in STM32CubeMx

- Configure for timer 2 in STM32CubeMx as shown in figure.
- Set PA5 as GPIO output.



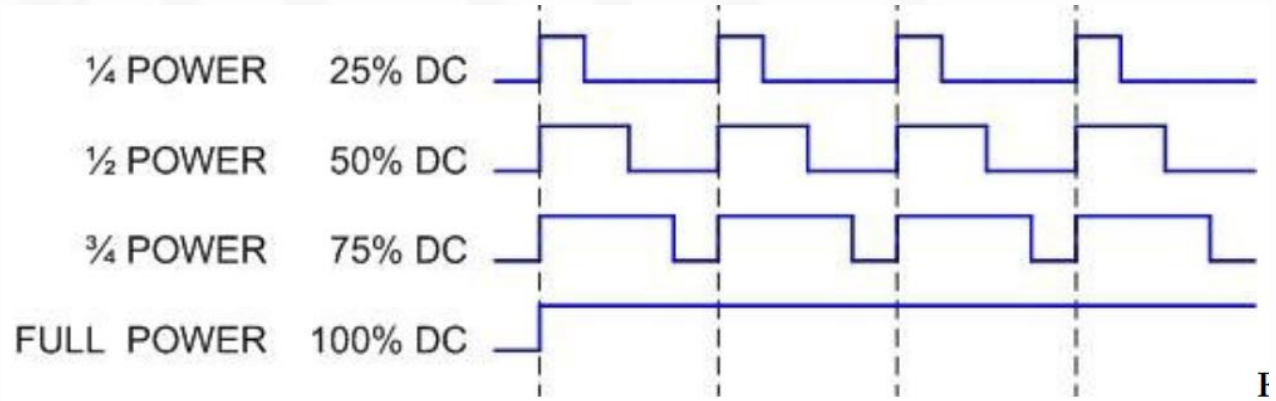
Program…

```
#include "main.h"
TIM_HandleTypeDef htim2;
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        }

int main(void)
{
SystemClock_Config();
  MX_GPIO_Init();
  MX_TIM2_Init();
HAL_TIM_Base_Start_IT(&htim2);
  while (1)
  {
  }
}
```

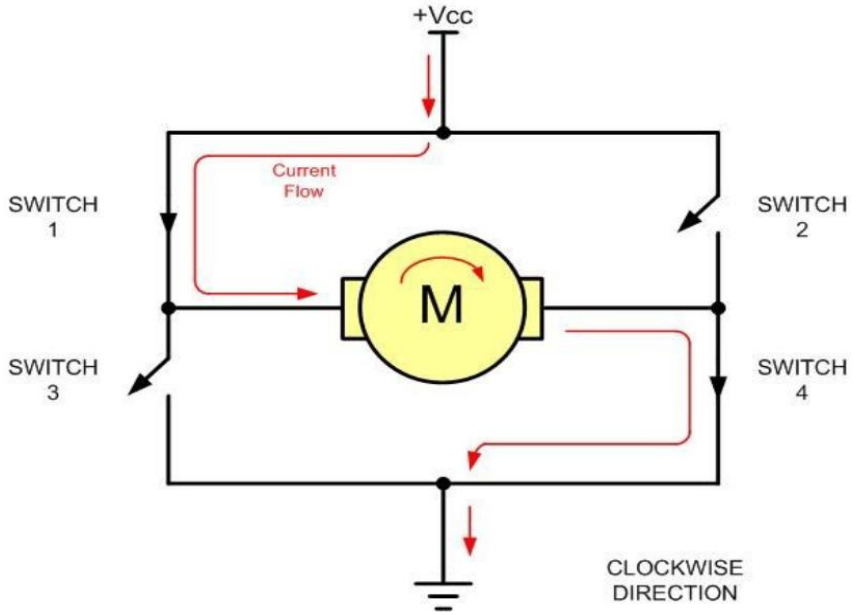
Pulse Width Modulators

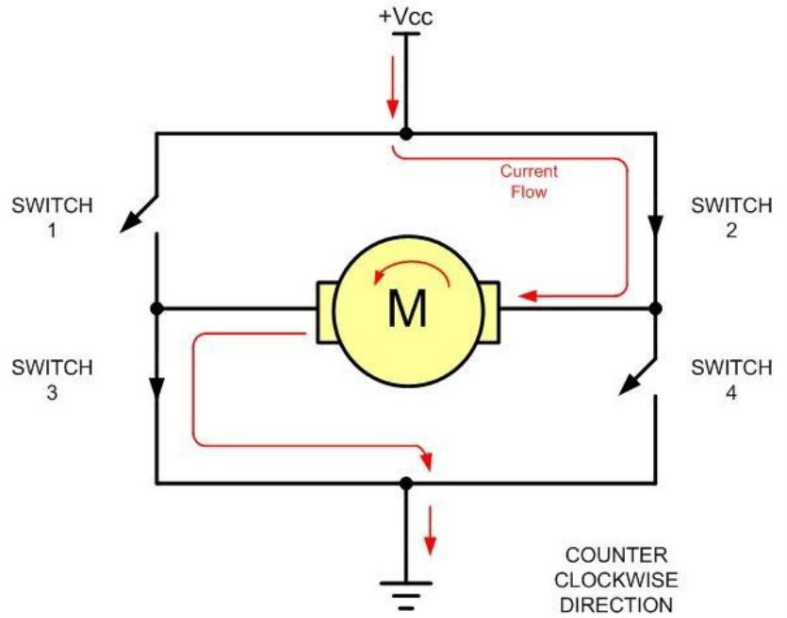
- Pulse Width Modulation (PWM) is a widely used technique in electronics and microcontroller-based systems for controlling analog devices using digital signals.
- It's especially useful for tasks like controlling the speed of motors, regulating the intensity of LEDs, and generating analog-like waveforms.
- In microcontrollers, PWM is achieved by rapidly toggling a digital signal on and off. The ratio of time the signal is ON (high) to the time it's OFF (low) is called the duty cycle. By changing the duty cycle, you can effectively control the average power delivered to the load.
- The figure shows comparison of PWM waves of different duty cycle.



Pulse Width Modulators…

- PWM is used to control the speed of DC motors. By changing the duty cycle, we change the average voltage applied to the motor.
- A direct current (DC) motor is a widely used device that translates electrical current into mechanical movement.
- In the DC motor, we have only + and – leads. Connecting them to a DC voltage source moves the motor in one direction. By reversing the polarity, the DC motor will rotate in the opposite direction.
- The figures shows direction control of DC motor.
- Under no load conditions, motor gives maximum RPM and as load increases speed decreases unless voltage or current applied to motor is increased.





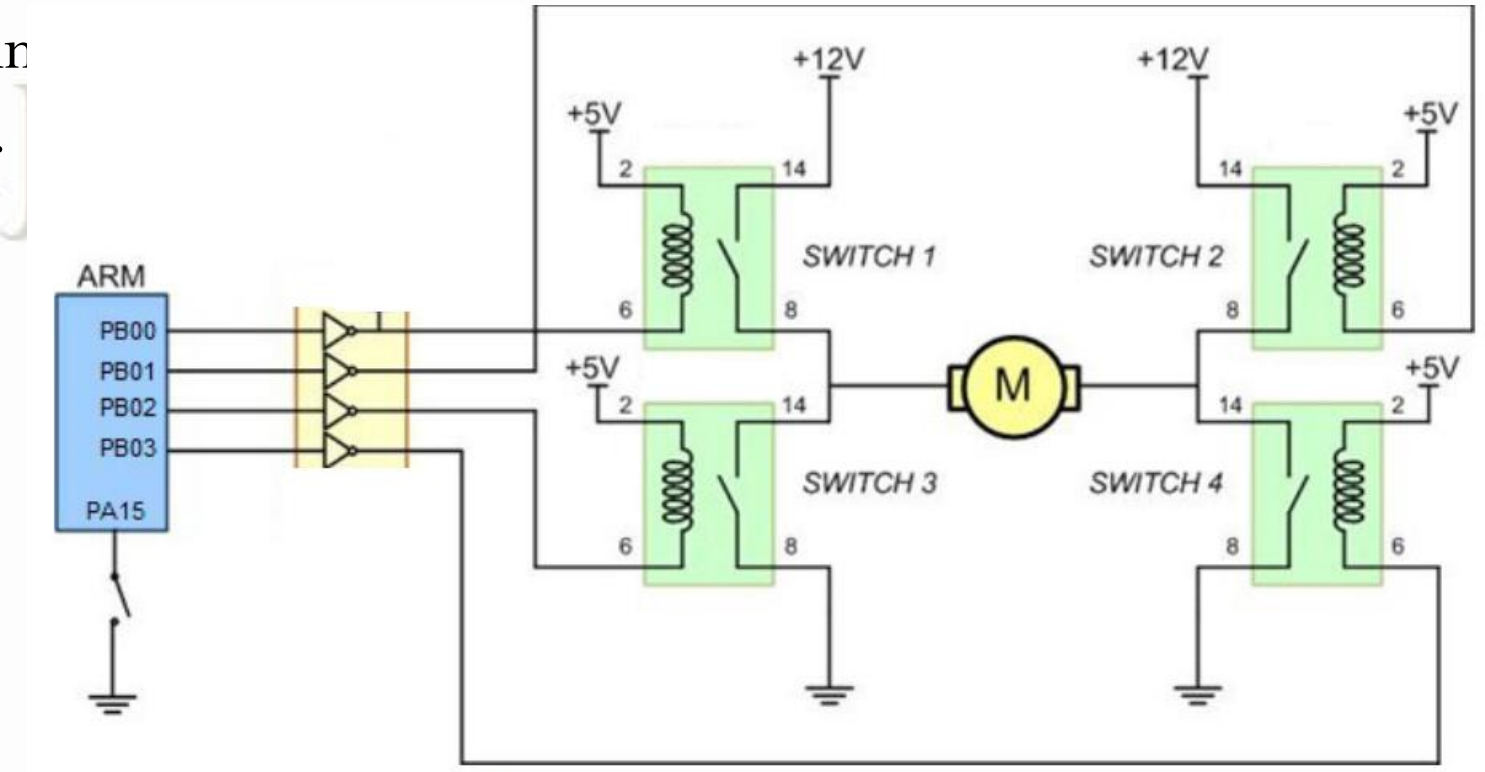
Program to control direction of motor at full speed

- A switch is connected to input pin PA15. Using relay H-Bridge and write the proper program to control the motor direction by the switch:

  - If PA15 = 1, the DC motor moves clockwise.

  - If PA15 = 0, the DC motor moves counterclockwise

The following setup can be used. Assur
STM32CubeMx is used generate HAL.



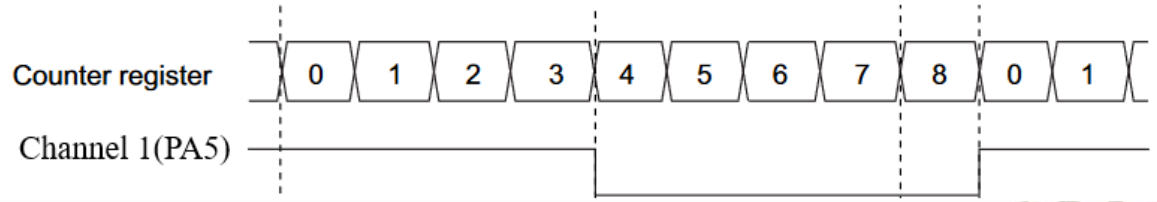
Program:

```
#include "main.h"
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
int main(void){
        HAL_Init();
        SystemClock_Config();
        MX_GPIO_Init();
       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, GPIO_PIN_SET);
        while (1){
           direction = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_15);
                  if(direction == GPIO_PIN_SET) {
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_3, GPIO_PIN_RESET);
                     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1|GPIO_PIN_1, GPIO_PIN_SET);
                      }
                else{
                     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0|GPIO_PIN_3, GPIO_PIN_SET);
                     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1|GPIO_PIN_1, GPIO_PIN_RESET);
                     }

            }
        }
```

PWM module in STM32F407

- Timers can be used to generate PWM wave on port pins. These pins are called channels.
- Pulse width modulation mode allows generating a signal with a frequency determined by the value of the TIMx_ARR register and a duty cycle determined by the value of the TIMx_CCRx register.
- When the channel 1 of timer 2 is configured as PWM mode, the output of Channel 1 is turned on when the counter starts counting from 0. When the counter matches the content of TIM2_CCR1, Ch1 output is turned off. When the counter matches TIM2_ARR, the counter is cleared to 0 and the output is turned on and the counter starts counting up again.
- Example: If TIM2_CCR1=4,TIM2_ARR=8 then waveform generated on channel 1 is as follows.



Program:

- Design STM32F407VG based system to generate PWM wave of 33%duty cycle on channel 1.

  Assume: APB1/APB2 Timer clock=16 MHz.

If timer pre scale is set to 16000,then clock frequency =16M/16000=1 KHz

Timer clock period is 1 ms.

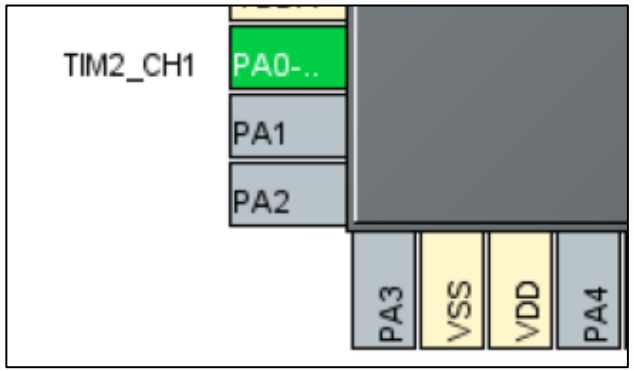
Assume PWM period as 1 sec, Duty cycle=33%

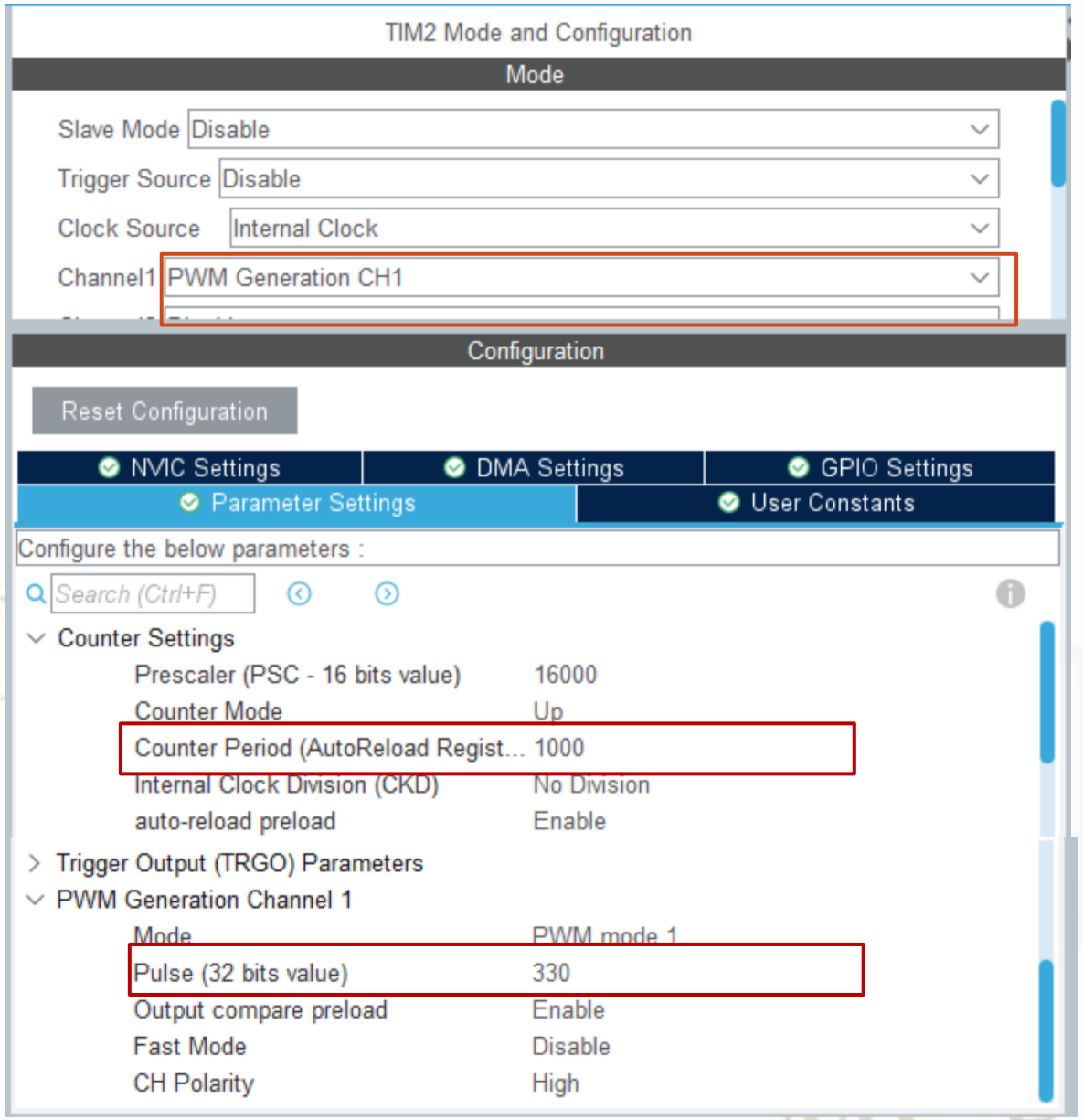
T_{ON}=330 ms T_{OFF}=670ms

TIM2_CCR1=330, TIM2_ARR=1000

Program…

- Configure for timer 2 in STM32CubeMx as shown in figure.
- In pinout view,PA0 is shown as TIM2_CH1 for PWM generation.





Program…

```
#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)
{
        SystemClock_Config();
        MX_GPIO_Init();
        MX_TIM2_Init();
        HAL_TIM_Base_Start(&htim2);
        while (1)
        {
        }
}
```

Note: On PA0 PWM is generated with duty cycle of 33% and period of 1 sec.