

Lab Manual: Digital Systems Laboratory (EEP3020)

Based on STMicroelectronic Kits (STM 32F4 Series)

Sudhanshu Gaurhar (P20EE016)-TA, Binod Kumar-Instructor
Department of Electrical Engineering, IIT Jodhpur

January-May 2024



॥ त्वं ज्ञानमयो विज्ञानमयोऽसि ॥

Important Notice:

1. Sharing this document to any one outside IITJ community is NOT allowed and prior written permission of authors must be taken.
2. Pasting this document or some parts of it on any online platform/website is strictly prohibited under all circumstances.

Contents

1	Introduction	5
1.1	Instruction to use Keil	6
1.2	Pin Configuration	9
2	Assembly language programming	14
3	Mixing C and assembly language	16
4	General Purpose Input Output (GPIO)	17
4.1	GPIO Input Modes: Pull Up and Pull Down	17
4.2	GPIO Output Mode: Push-Pull	17
4.3	GPIO Output Mode: Open-Drain	18
4.4	Registers Required to Programme	19
4.4.1	RCC— >AHB1 peripheral clock enable register	19
4.4.2	GPIO port mode register	20
4.4.3	GPIO port output data register	20
4.4.4	GPIO port input data register	21
5	System Tick Timer	22
5.1	System Tick Registers	23
5.2	Programming Example	23
5.3	Programming Exercise	23
6	General Purpose Timer	24
6.1	General Purpose Timer Registers	24
6.1.1	RCC APB1 peripheral clock enable register (RCC_APB1ENR)	24
6.1.2	TIMx control register 1 (TIMx_CR1)	24
6.1.3	TIMx prescaler (TIMx_PSC)	26
6.1.4	TIMx auto-reload register (TIMx_ARR)	26
6.1.5	TIMx counter (TIMx_CNT)	27
6.1.6	Programming Example	27
6.2	Programming Exercises	27
7	Analog To Digital Converter	28
7.1	ADC Registers	28
7.1.1	ADC Control Register (ADC_CR2)	28
7.1.2	ADC Control Register (ADC_CR1)	29
7.1.3	ADC status register (ADC_SR)	30
7.2	Programming Example Using Potentiometer	31
7.3	Programming Exercise	32
8	Universal Asynchronous Receiver and Transmitter (UART)	33
8.1	USART registers	33
8.1.1	Status register (USART_SR)	33
8.1.2	Data register (USART_DR)	35
8.1.3	Baud rate register (USART_BRR)	35
8.1.4	Control register 1 (USART_CR1)	36
8.1.5	Control register 2 (USART_CR2)	38
8.2	Algorithm for Configuration of UART1	39
8.3	Algorithm for Sending a character	39

8.4	Algorithm for Receiving a character	39
8.5	Interfacing of ultrasonic sensor	40
8.5.1	Mathematical Formulation	40
8.6	Programming Exercise	41
9	Seven Segment Interfacing	42
9.1	Common Cathode Configuration	42
9.2	Common Anode Configuration	42
9.3	Programming Exercise	42
10	External Interrupts	43
10.1	Configuring External Interrupts	43
10.2	External Interrupts mapping with GPIOs	44
10.3	Registers	45
10.3.1	Interrupt Mask Register (EXT1_IMR)	45
10.3.2	Rising Trigger selection register (EXT1_RTSTR)	45
10.3.3	Pending register (EXT1_PR)	46
10.3.4	SYSCFG external interrupt configuration register 4 (SYSCFG_EXTIC4)	46
10.4	Example Program	47
10.5	Programming excercise	47

List of Figures

1	Fig 1: Pull Up and Pull Down Configuration.	17
2	Fig. 2 If the digital output is 0, then the GPIO output pin is pulled down to the ground in a push-pull setting.	18
3	Fig 3. If the digital output is 1, then the GPIO output pin is pulled up to the Vcc in a push-pull setting.	18
4	Fig. 4 If the digital output is 0, then the output pin is pushed to the ground in an open-drain setting (the scenario of drain).	18
5	Fig 5. If the digital output is 1, then the output pin is floating in an open-drain setting (the scenario of open).	18
6	Fig. 6:RCC AHB1 Register.	19
7	Fig.7:Mode Register.	20
8	Fig.8:Output Data Register.	20
9	Fig.9:Input Data Register.	21
10	Fig.10:System Tick Working	22
11	Fig.11:System Tick Registers	23
12	Fig.12:APB1ENR Register	24
13	Fig.13:Timer Control register 1	24
14	Fig.14:Prescaler Register	26
15	Fig.15:Auto reload Register	27
16	Fig.16: Counter Register	27
17	Fig.17:ADC Control Register-2	28
18	Fig.18:ADC Control Register-1	29
19	Fig.19:ADC status register	30
20	Fig.20:Potentiometer	31
21	Fig.21:USART status register	33
22	Fig.22:USART Baud Rate Register	35
23	Fig.23:USART Control Register 1	36
24	Fig.24:USART Control Register2	38
25	Fig.25:Ultrasonic Sensor Working	40
26	Fig.26:Common Cathode and common anode Seven Segment	42
27	Fig.27:External Interrupt Bus	43
28	Fig.28:External Interrupt Mapping	44
29	Fig.29:Interrupt Mask Register	45
30	Fig.30:Rising Trigger selection register	45
31	Fig.31:Pending register	46
32	Fig.32:SYSCFG external interrupt configuration register 4	46
33	Registers in ARM architecture	48

1 Introduction

Digital systems can be very closely found in almost all real-life applications in the modern age. Such systems comprise of a hardware running some piece of software on it. In other words, these designs can also be referred to as embedded systems. This laboratory course aims to impart skills to students in the areas of familiarity with embedded systems (here considered as kit/board, for example- STM32 Nucleo-144 board) and embedded application development for applications ranging from LED blinking to timer usage for delays/motor control etc. This laboratory manual exclusively caters to application development on the widely popular STM32 Nucleo-144 board.

The STM32 Nucleo-144 board provides an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 microcontroller. For the compatible boards, the internal or external SMPS significantly reduces power consumption in Run mode. The ST Zio connector, which extends the ARDUINO® Uno V3 connectivity, and the ST morpho headers provide an easy means of expanding the functionality of the Nucleo open development platform with a wide choice of specialized shields. **The STM32 Nucleo-144 board does not require any separate probe as it integrates the ST-LINK debugger/programmer.**

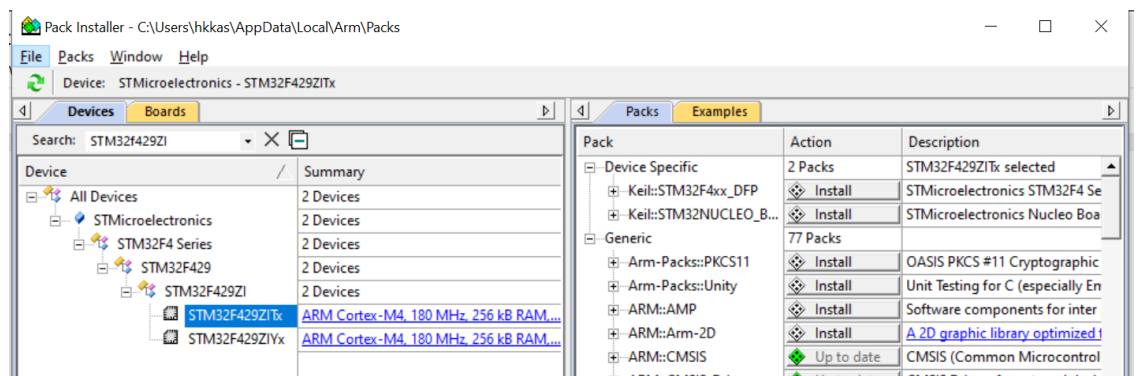
- STM32 microcontroller in an LQFP144 package
- 3 user LEDs
- 2 user and reset push-buttons
- 32.768 kHz crystal oscillator
- Board connectors:
 - SWD
 - ST Zio expansion connector including ARDUINO® Uno V3
 - ST morpho expansion connector
- Flexible power-supply options: ST-LINK USB VBUS, USB connector, or external sources
- On-board ST-LINK debugger/programmer with USB re-enumeration capability: mass storage, Virtual COM port, and debug port
- Comprehensive free software libraries and examples available with the STM32Cube MCU Package
- Support of a wide choice of Integrated Development Environments (IDEs) including IAR Embedded Workbench®, MDK-ARM, and STM32CubeIDE
- Board-specific features
 - External or internal SMPS to generate Vcore logic supply
 - Ethernet compliant with IEEE-802.3-2002
 - USB OTG full speed or SNK/UFP (full-speed or high-speed mode), depending on the USB connector type
 - Board connectors:
 - USB with Micro-AB or USB Type-C®
 - Ethernet RJ45

Keil is an open-source simulator that can be utilized for running ARM assembly programs or C applications developed for STM32 Nucleo-144 board.

1.1 Instruction to use Keil

Follow the following instructions for installing the Keil MDK software on Windows.

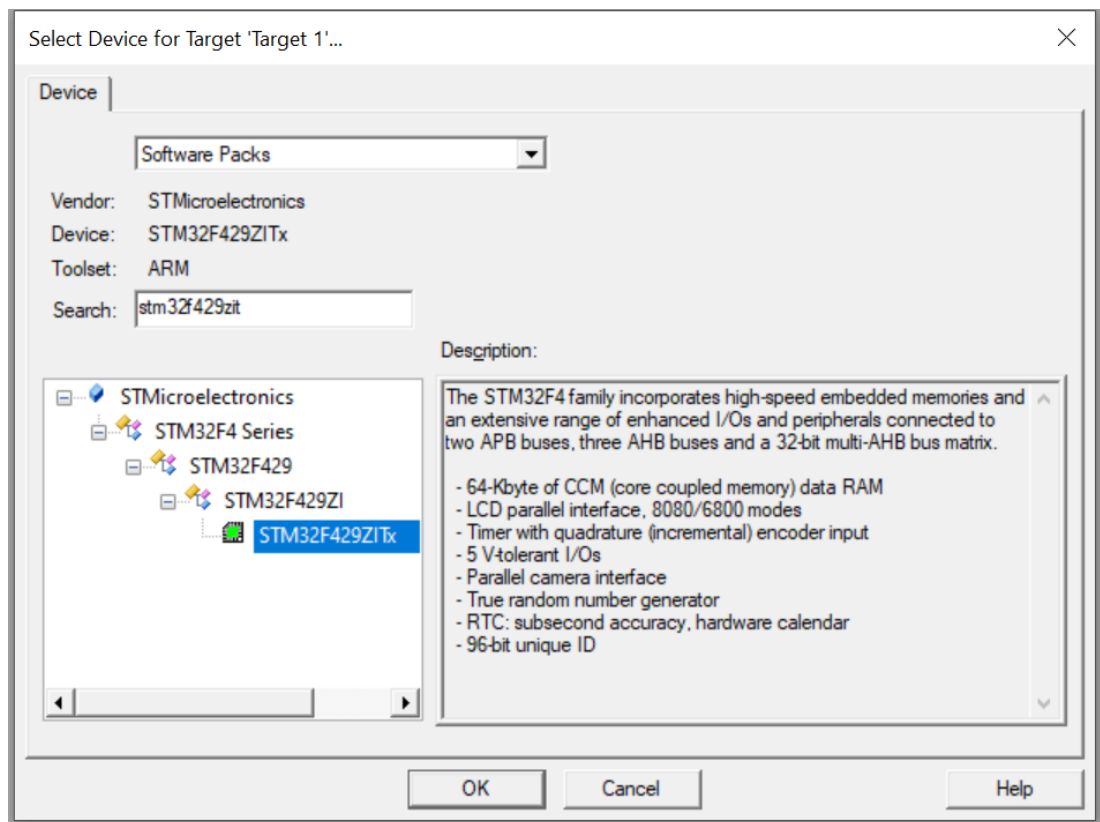
1. Install the software named MDK538a.exe
2. Once the installation is complete, the package installer will open; you need to download the STM32429xx package. Following is the screenshot of the package installer.



3. Search for **STM32F429ZITx**, now select Tx version and on the right panel, click on install for **Device Specific** options.
 - a. Keil: STM32F4xx_DFP
 - b. Keil: STM32Nucleo_Board
4. Once they are downloaded and installed itself, they will show **Up to date**.
5. Open Keil µvision software.

Once the installation is complete, and all the required packages are installed, you can work on projects by creating new projects. Following are the instructions for that. **You need to follow these steps every time you create new projects.**

1. On the menu bar, select Project -> New µvision Project. A window will pop up. For this tutorial, give the name **Tutorial**, and a new folder will be created in the selected folder.
2. Now the following window will pop up. You need to select the device STM32F429ZITx, you can search for it; it will appear you need to select it and click **OK**.



3. Now, a new window will pop up. You need to select the following options.
 - a. CMSIS -> Core
 - b. Device -> Startup
 - c. Click OK

Now, you will have 2 options depending on the experiment, whether you need the device for a particular experiment or not. If no device is needed, for eg. the second lab experiment won't require the device; you need to turn on the simulator. Follow the following instructions for turning on the simulator.

1. Click on the **Options for Target** on the toolbar. Below is the highlighted button in blue.



2. Once the pop up opens, go to the **Debug** option and select
 - a. Use simulator
 - b. Click OK

Now you can run projects.

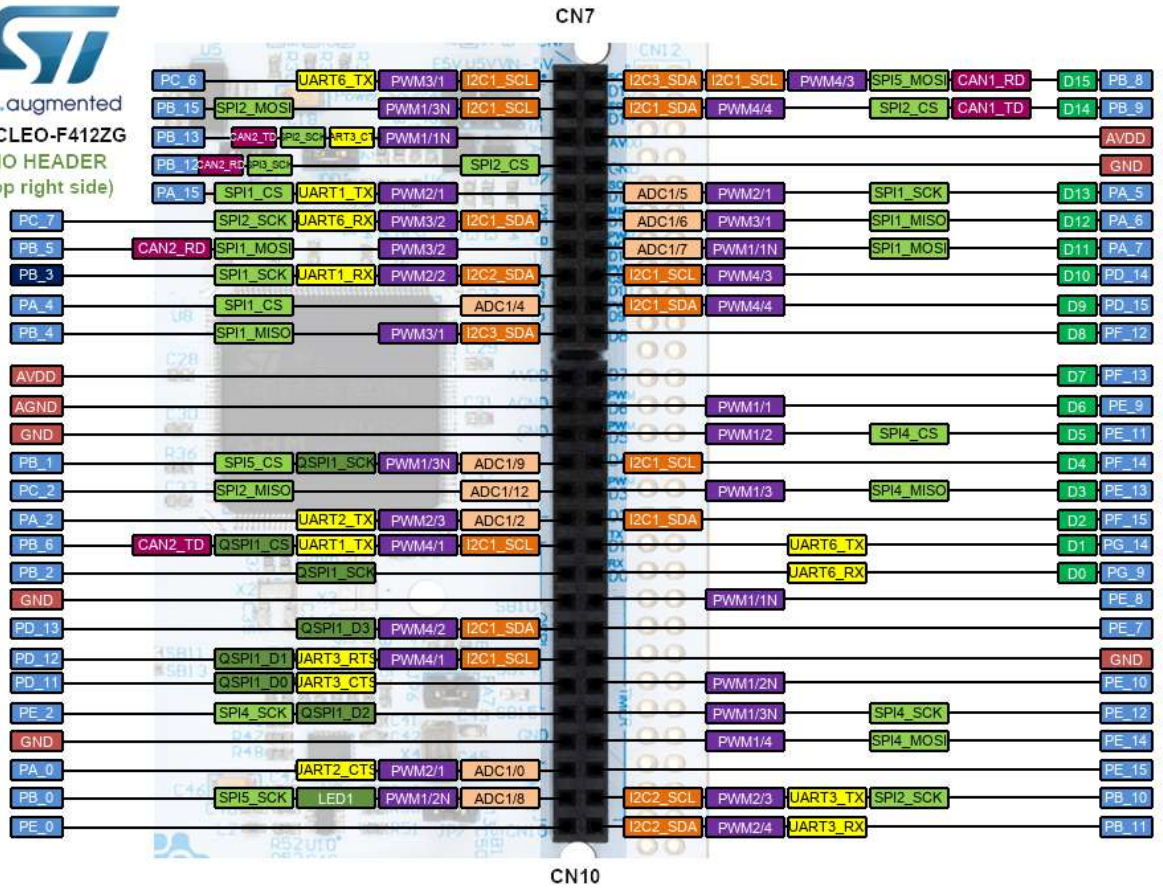
1.2 Pin Configuration

The STM32 F4-series stands as a pioneering group of microcontrollers, marking the advent of the ARM Cortex-M4F core in the STM32 lineup. Distinguished by its cutting-edge features, the F4-series introduces DSP (Digital Signal Processing) and floating-point instructions, setting new standards in microcontroller capabilities. This series not only maintains pin-to-pin compatibility with the STM32 F2-series but also incorporates enhancements such as higher clock speeds, 64 KB CCM static RAM, full-duplex I2S protocol, an improved real-time clock, and faster ADCs.

Key specifications of the STM32 F4-series include:

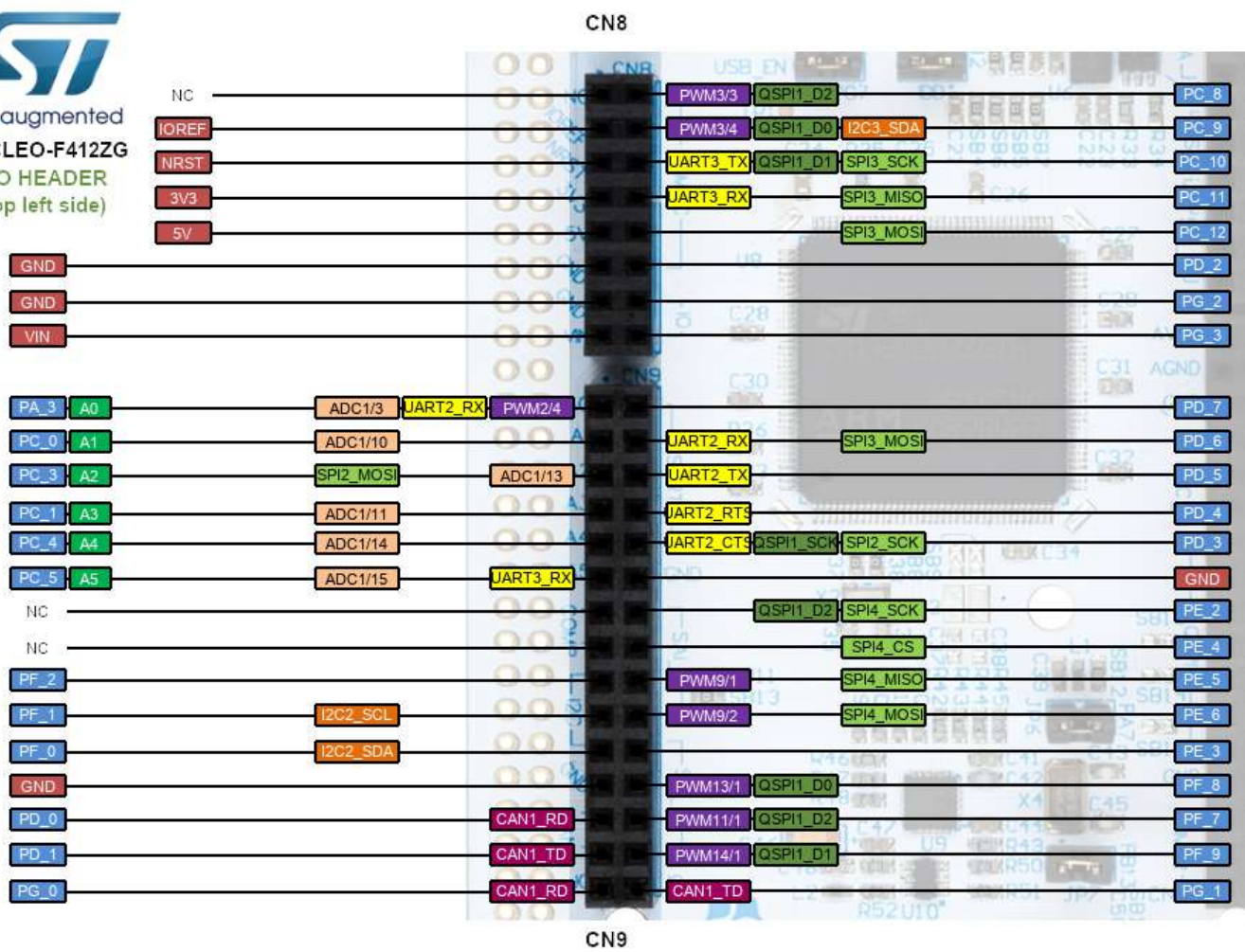
- Core:
 - ARM Cortex-M4F core with a maximum clock rate ranging from 84 to 180 MHz.
 - Memory:
 - Static RAM, offering up to 192 KB of general-purpose memory, 64 KB of core-coupled memory (CCM), and 4 KB of battery-backed memory.
 - Flash memory ranging from 512 KB to 2048 KB for general-purpose use, along with additional segments for system boot and one-time programmable (OTP) memory.
 - Unique device identifier number: Each chip features a factory-programmed 96-bit identifier.
 - Peripherals:
 - USB 2.0 OTG HS and FS, two CAN 2.0B, SPI, I2S, I2C, USART, UART, SDIO for SD/MMC cards, timers, watchdog timers, temperature sensor, ADCs, DACs, GPIOs, DMA, real-time clock (RTC), CRC engine, and RNG engine.
 - Digital filter for sigma-delta modulators (DFSDM) interface in specific models.
 - Additional features in specific models include Ethernet MAC, camera interface, cryptographic processor, hash processor, and LCD-TFT controller.
 - Oscillators: Internal oscillators at 16 MHz and 32 kHz, with optional external oscillators in the range of 4 to 26 MHz and 32.768 to 1000 kHz.
 - IC Packages: Available in various packages such as WLCSP64, LQFP64, LQFP100, LQFP144, LQFP176, and UFBGA176. The STM32F429/439 models also offer LQFP208 and UFBGA216.
 - Operating Voltage: Operating voltage range spans from 1.8 to 3.6 volts.
- In summary, the STM32 F4-series is a versatile and advanced microcontroller series, combining the power of the ARM Cortex-M4F core with a rich array of peripherals and memory options, making it suitable for a wide range of embedded applications.

In the upcoming 4 pages, the four different views of this particular kit are shown. The special functionality corresponding to each pin is also highlighted.



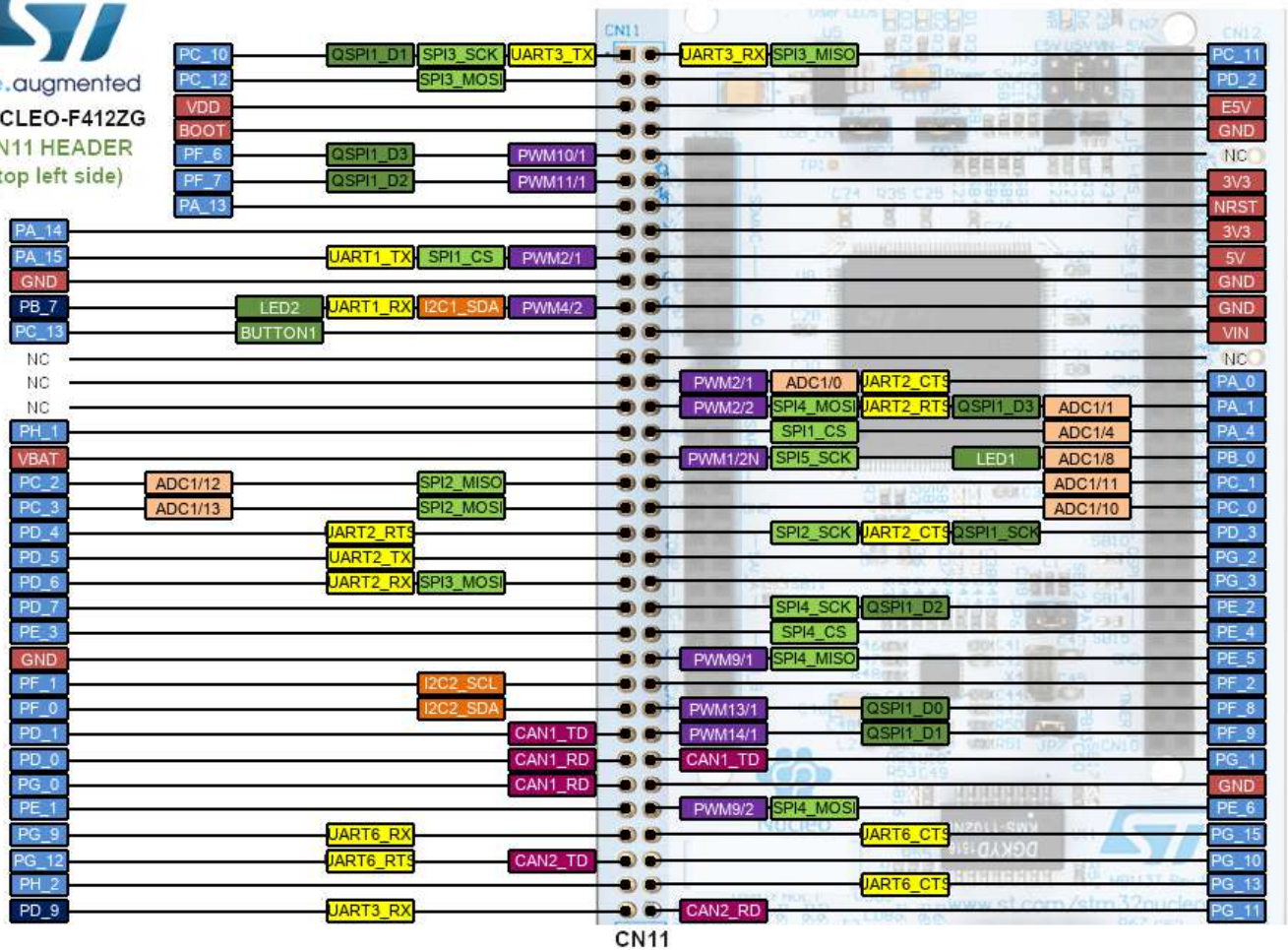


life.augmented
NUCLEO-F412ZG
ZIO HEADER
(top left side)



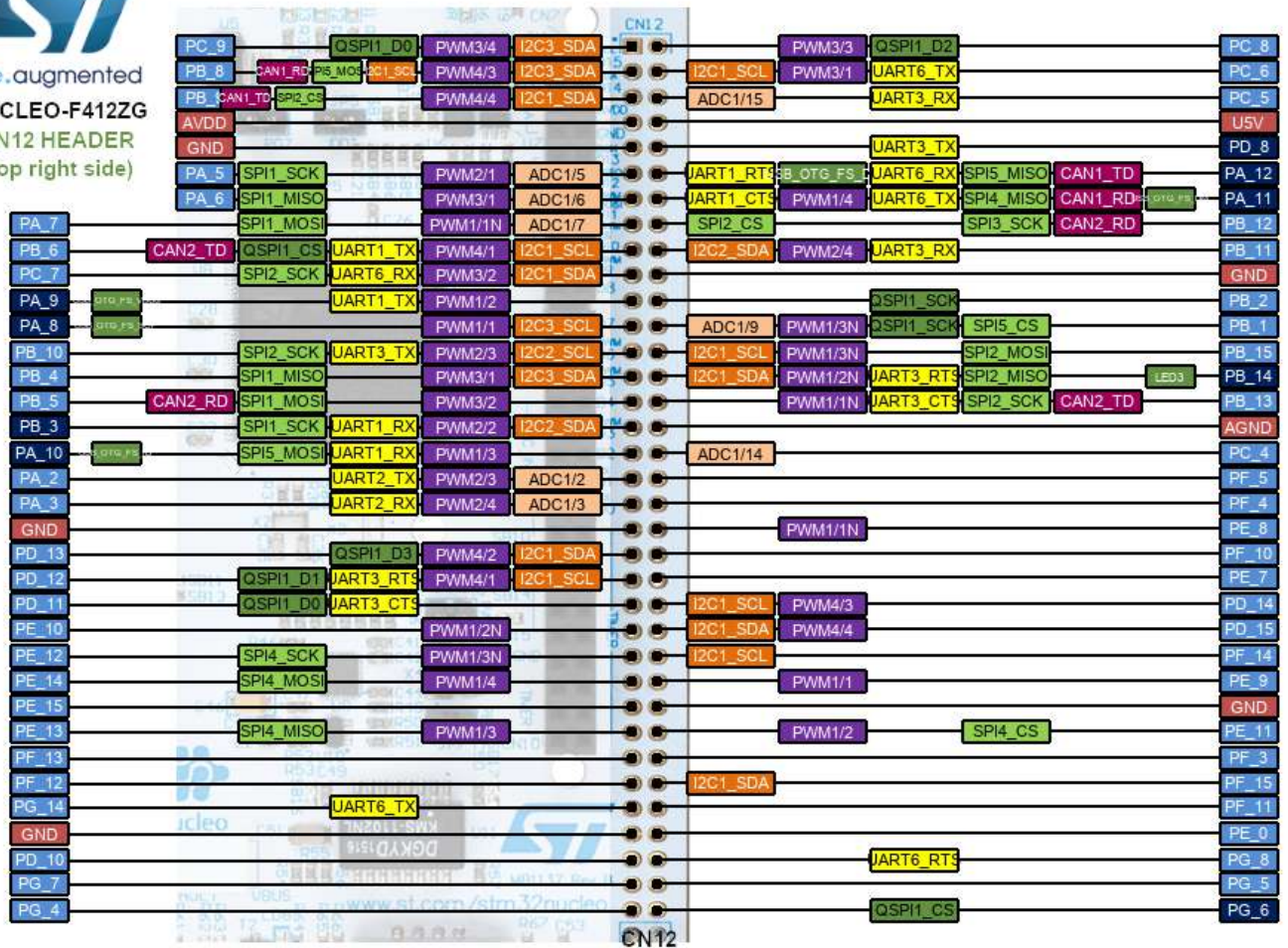


life.augmented
NUCLEO-F412ZG
CN11 HEADER
(top left side)





life.augmented
NUCLEO-F412ZG
CN12 HEADER
(top right side)



2 Assembly language programming

Fundamentally, the most basic instructions executed by a computer are binary codes, consisting of ones and zeros. Those codes are directly translated into the “on” and “off” states of the electricity moving through the computer’s physical circuits. In essence, these simple codes form the basis of “machine language”, the most fundamental variety of programming language. Of course, no human would be able to construct modern software programs by explicitly programming ones and zeros. Instead, human programmers must rely on various layers of abstraction that can allow themselves to articulate their commands in a format that is more intuitive to humans. However, these high-level commands need to be translated into machine language. Rather than doing so manually, programmers rely on assembly languages whose purpose is to automatically translate between these high-level and low-level languages. Programmers need to write in assembly languages, such as when the demand for performance is especially high, or when the hardware in question is incompatible with any current high-level languages. **Please refer to Appendix-A for a compendium of ARM assembly instructions.**

Under the “Source Group” tab right click for “Add new item to group”, in which you can add an assembly file (i.e. ‘.s’ file). Once created, you can write the code in it.

```
AREA MYCODE, CODE
ENTRY
EXPORT __main
__main
Your code and logic here
END
```

Once code is completed, you need to build the code by the tab provided or by using ‘F7’.

If it’s with no error, You can start/ stop debugging by the debug tab. Then you will enter debugging mode and will be able to visualize values of all the registers.

Here you can run the whole code, or step by step, can add breakpoint to stop in between and many more.

1.Find the factorial of a number stored in one of the registers

```
AREA factorial , CODE
ENTRY
EXPORT __main
__main
    LDR r0 , =0x00000006
    MOV r1 , #1
    MOV r2 , r0
    MOV r3 , #0
LOOP
    CMP r3 , r0
    BGE DONE
    MUL r1 , r2 , r1
    SUB r2 , r2 , #1
    ADD r3 , r3 , #1
    B LOOP
DONE
    STOP
B
    STOP
END
```

2.Write an assembly language program to transfer an array of data from one memory block to another.

```

        AREA MYCODE, CODE
        ENTRY
EXPORT  __main
__main
        LDR R0, =source
        LDR R1, =destination
        MOV R2, #10
loop
        LDR R3, [R0], #4
        STR R3, [R1], #4
        SUB R2, R2, #1
        CMP R2, #0
        BNE loop
stop
        B stop
source
        DCD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
destination
        DCD 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
ALIGN
END

```

Programming Exercises :

Exercise 2 : Find the sum of first n natural numbers, where n is stored in one of the registers. Find the sum of first n natural numbers, where n is stored in one of the registers.

Exercise 2 : Write an assembly language program to find the transpose of a 3x3 matrix and store the result in continuous memory locations. The elements of the matrix are stored in continuous memory locations.

3 Mixing C and assembly language

Note that some embedded systems are coded purely in assembly language, but most are coded in C with assembly language used only for time-critical processing, if at all. This is because the code development process is much faster (and hence less expensive) when writing in C when compared to assembly language. Writing assembly code as functions that can be called from C code as C functions result in modular programs, which gives us the best of both worlds: the fast, modular development of C and the high performance of assembly code. It is also possible to add inline assembly code to C code. Details of this process are mentioned in Appendix A. First, we will create the main C function. This function contains two variables (a and b) with character arrays. This function calls two other functions, which can be later defined.

Task 1. Write an assembly function to copy a string from one variable to another, and the calling of the function should be in c program (int main).

Note 1: Please correct the indentation if required.

Note 1: Save the file with the .c extension.

There is a minor change in the compiler setting you need to make to run this code.

Steps:

1. Open options for the target.
2. Go to the Target tab, and on the right-hand side, you will find the ARM compiler. Ensure that "Use default compiler version 5" is selected.

```
--asm void your_strcpy(const char *src , char *dst)

{
loop
    LDRB r2 , [r0]    // Load byte into r2 from memory pointed to
                      // by r0 (src pointer)
    ADDS r0 , #1      // Increment src pointer

    STRB r2 , [r1]    // Store byte in r2 into memory pointed to
                      // by dst pointer
    ADDS r1 , #1      // Increment dst pointer
    CMP r2 , #0       // Was the byte 0?
    BNE loop          // If not , repeat the loop
    BX lr             // Else return from subroutine
}

int main (void)
{
const char a[] = "Hello_world!";
char b[20];
your_strcpy(a , b);
while(1);
}
```

Programming Exercises

Exercises:1 Write the function (in ARM assembly) to capitalize a given string. Include this in the program and use the above main function.

Exercises:2 Write a function (in ARM assembly) to reverse a given string. Include this in the program and use the above main function.

4 General Purpose Input Output (GPIO)

In this lab, we will utilize the STM32 Nucleo-144 board. The schematic of the board has been provided. The board is equipped with two LEDs connected to PB.7 (BLUE) and PB.14 (RED). Please connect the provided MicroUSB cable to the board. Create a project and choose the device as STM32F412ZG. Opt for ST-LINK Debugger in the debug options.

Software can program a GPIO pin to serve one of the following four different functions:

1. Digital input that detects whether an external voltage signal is higher or lower than a predetermined threshold.
2. Digital output that controls the voltage on the pin.
3. Analog functions that perform digital-to-analog or analog-to-digital conversion.
4. Other complex functions such as PWM output, LCD driver, timer-based input capture, external interrupt, and interfaces like USART, SPI, I2C, PC, and USB communication.

4.1 GPIO Input Modes: Pull Up and Pull Down

When a GPIO pin is used as digital input, the pin has three states: high voltage, low voltage, or high impedance (also called floating or tri-stated). Pull-up and pull-down are used to ensure the input pin has a valid high (logic 1) or a valid low (logic 0) when the external circuit does not drive the pin. When software configures a pin as pull-up, the pin is internally connected to the power supply via a resistor, as shown in Figure 1. The pin is always read as high (logic 1) unless the external circuit drives this pin low. Similarly, when a pin is configured as pull-down, the pin is then internally connected to the ground via a resistor, as shown in Figure 1. The pin is always read as low (logic 0) unless the external circuit drives this pin high.

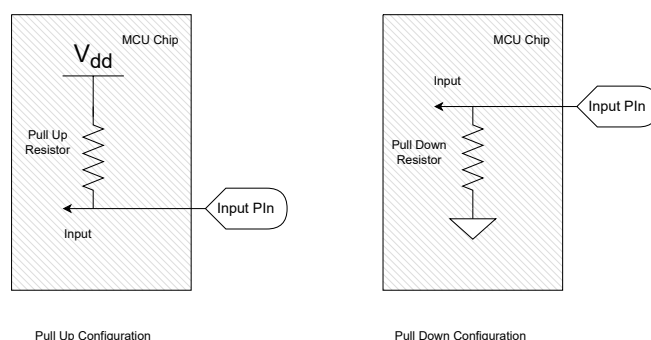


Fig 1: Pull Up and Pull Down Configuration.

4.2 GPIO Output Mode: Push-Pull

- When logic 0 is outputted, the transistor connected to the ground is turned on to sink an electric current from the external circuit, as shown in Figure 2.
- When the pin outputs logic 1, the transistor connected to the power supply is turned on, and it provides an electric current to the external circuit connected to the output pin, as shown in Figure 3.

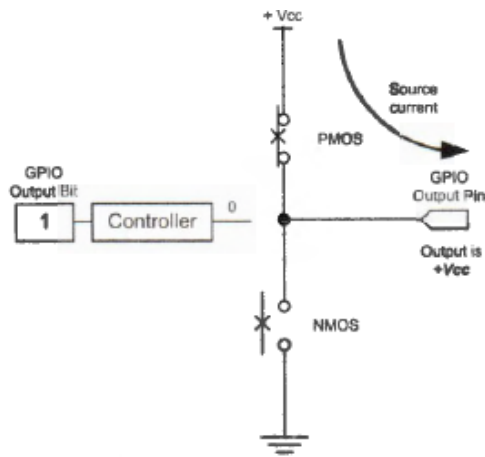


Fig. 2 If the digital output is 0, then the GPIO output pin is pulled down to the ground in a push-pull setting.

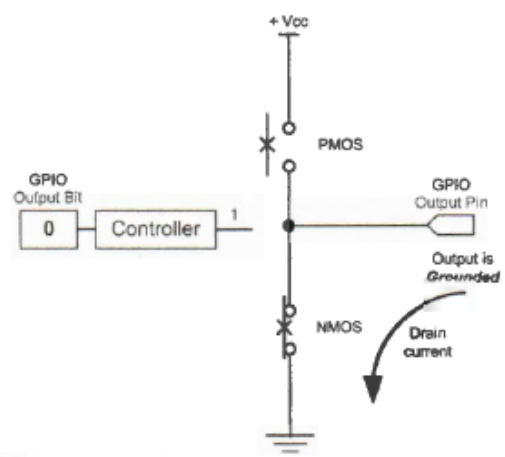


Fig 3. If the digital output is 1, then the GPIO output pin is pulled up to the Vcc in a push-pull setting.

4.3 GPIO Output Mode: Open-Drain

- When software outputs a logic 0, the open-drain circuit can sink an electric current from the external load connected to the GPIO pin.
- However, when software outputs a logic 1, it cannot supply any electric current to the external load because the output pin is floating, connected to neither the power supply nor the ground.

An open-drain output has only two states: low voltage (logic 0), and high impedance (logic 1). It often has an external pull-up resistor.

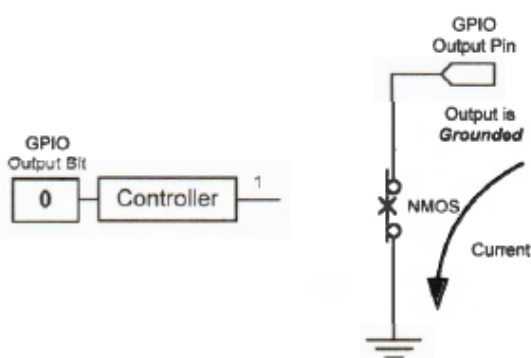


Fig. 4 If the digital output is 0, then the output pin is pushed to the ground in an open-drain setting (the scenario of drain).

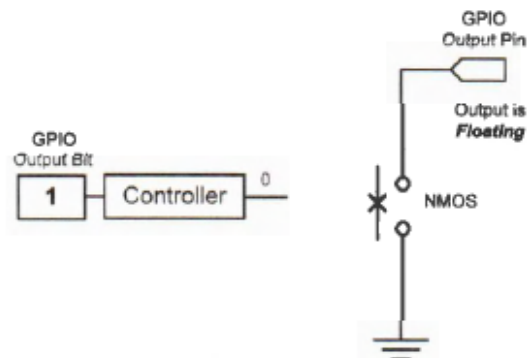


Fig 5. If the digital output is 1, then the output pin is floating in an open-drain setting (the scenario of open).

4.4 Registers Required to Programme

4.4.1 RCC – >AHB1 peripheral clock enable register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DMA2 RST	DMA1 RST	Res.	Res.	Res.	Res.	Res.
									rw	rw					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CRCRST	Res.	Res.	Res.	Res.	GPIOH RST	GPIOG RST	GPIOF RST	GPIOE RST	GIOD RST	GPIOC RST	GPIOB RST	GPIOA RST
			rw					rw	rw	rw	rw	rw	rw	rw	rw

Fig. 6:RCC AHB1 Register.

Clock Enable Register Bit Description

1: CRC clock enabled

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

Bits 11:8 Reserved, must be kept at reset value.

Bit 22 DMA2EN: DMA2 clock enable

Set and cleared by software.

0: DMA2 clock disabled

1: DMA2 clock enabled

Bit 7 GPIOHEN: IO port H clock enable

Set and reset by software.

0: IO port H clock disabled

1: IO port H clock enabled

Bit 21 DMA1EN: DMA1 clock enable

Set and cleared by software.

0: DMA1 clock disabled

1: DMA1 clock enabled

Bit 6 GPIOGEN: IO port G clock enable

Set and cleared by software.

0: IO port G clock disabled

1: IO port G clock enabled

Bits 20:13 Reserved, must be kept at reset value.

Bit 5 GPIOFEN: IO port F clock enable

Set and cleared by software.

0: IO port F clock disabled

1: IO port F clock enabled

Bit 12 CRCEN: CRC clock enable

Set and cleared by software.

0: CRC clock disabled

0: IO port C clock disabled

1: IO port C clock enabled

Bit 4 GPIOEEN: IO port E clock enable

Set and cleared by software.

0: IO port E clock disabled

1: IO port E clock enabled

Bit 1 GPIOBEN: IO port B clock enable

Set and cleared by software.

0: IO port B clock disabled

1: IO port B clock enabled

Bit 3 GPIODEN: IO port D clock enable

Set and cleared by software.

0: IO port D clock disabled

1: IO port D clock enabled

Bit 0 GPIOAEN: IO port A clock enable

Set and cleared by software.

0: IO port A clock disabled

1: IO port A clock enabled

Bit 2 GPIOCEN: IO port C clock enable

Set and cleared by software.

4.4.2 GPIO port mode register

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

Fig.7:Mode Register.

MODERy[1:0]: Port x Configuration Bits

These bits (where $y = 2y, 2y + 1, y = 0..15$) are written by software to configure the I/O direction mode.

- **00**: Input (reset state)
- **01**: General purpose output mode
- **10**: Alternate function mode
- **11**: Analog mode

4.4.3 GPIO port output data register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

Fig.8:Output Data Register.

Port Output Data and Reserved Bits

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

Bits 15:0 ODRy: Port output data (where $y = 0..15$). These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (where $x = A...H$).

4.4.4 GPIO port input data register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Fig.9:Input Data Register.

Port Input Data and Reserved Bits

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

Bits 15:0 IDRy: Port input data (where $y = 0..15$). These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

The following program will blink the LED connected to PB7.

```
#include "stm32f412Zx.h"
void delay(int dd);
int main() {
    RCC->AHB1ENR |= 0x02;           // Enables Clock
    GPIOB->MODER |= 0x10004000;      // Mode Register for PB7
    GPIOB->ODR = 0x40;               // Output Data Register for PB7

    while (1) {
        GPIOB->ODR = 0x80;
        delay(20000);
        GPIOB->ODR = 0x00;
        delay(10000);
    }
}

void delay(int dd) {
    int i;
    for (; dd > 0; dd--) {
        delay(10000);
    }
}
```

Programming Exercise

Write the program and verify on Hardware to blink LED connected on PB14.

5 System Tick Timer

The Use of system timer(SysTick) to produce delays This is a 24-bit down-counter. The system timer is a standard hardware component built into ARM Cortex processors. Almost all ARM Cortex processors have the system timer component. If enabled, the system timer can periodically generate SysTick interrupt requests. The Nested Vectored Interrupt Controller (NVIC) monitors and handles all interrupt requests based on their priority levels. For SysTick interrupts, NVIC forces the core to execute the interrupt service routine named SysTick Handler. The system timer is a 24-bit, down counter. The counter decrements, from the reload value to zero. After the counter reaches zero, the system timer copies, the reload value, stored in the reload value register. Then, the system timer starts to count down again. The operation of the SysTick controlled by Control and status register. First of all, the clock

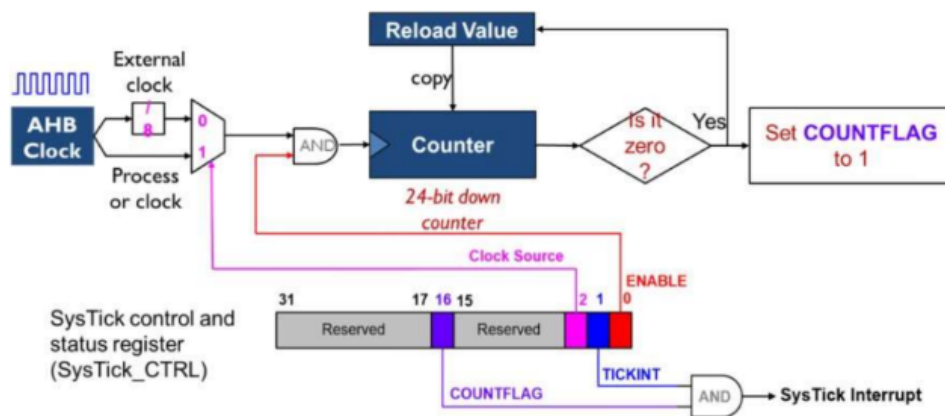


Fig.10: System Tick Working

source bit, selects the clock source, for the counter. If the clock source bit is 1, the processor clock is selected. If the clock source bit is 0, the external clock is selected. For STM32 the processor clock is the AHB clock. The external clock is the AHB clock slowed down by a frequency divider, 8. Software can set or clear Enable bit (bit 0) to enable or disable the system timer. Specifically, the enable bit can enable or disable the clock signal, by using this and gate. If the enable bit is one, the system timer is enabled, because the signal of the clock source, can pass through the and gate. The interrupt enable bit (Tick INT) enables the interrupt. A SysTick interrupt request is generated, if, both the interrupt enable bit, and the Count flag bit, are 1. In other words, if the interrupt enable bit is 1, a SysTick interrupt request is generated, every time the time counter decrements from 1 to 0.

5.1 System Tick Registers

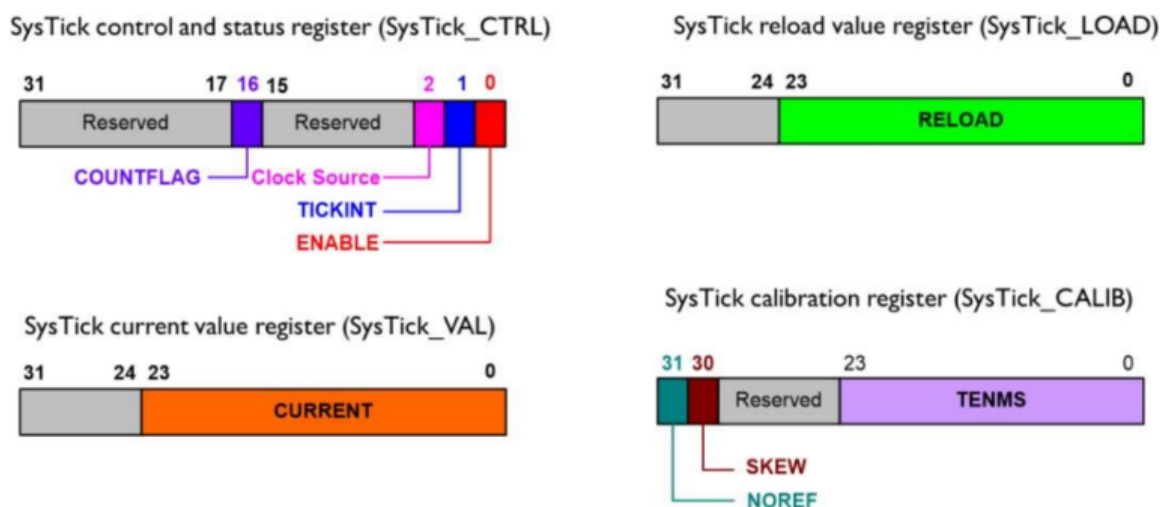


Fig.11: System Tick Registers

The system timer is controlled by four registers (shown in Fig. 11, including the control and status register, the reload value register (SysTick Load), the current-value register (SysTick VAL), and the calibration register (SysTick CALIB).

5.2 Programming Example

Write a program to toggle LED using Sys Tick Timer.

```
#include "stm32f4xx.h" // Device header
int main()
{
    RCC->AHB1ENR |= 1; //enable clock
    GPIOA->MODER |= 0x5000; //mode register
    GPIOA->ODR = 0xc0; //data register
    SysTick->LOAD = 3600000; // Time period of one clock cycle*Load value
    SysTick->VAL = 0;
    SysTick->CTRL = 0x5;
    while(1)
    {
        if (SysTick->CTRL & 0x10000)
        {
            GPIOA->ODR ^= 0xc0;
        }
    }

    return 1;
}
```

5.3 Programming Exercise

Write a program to toggle the user Led (PB7 and PB14). Provide 10 sec delay between successive toggling.

6 General Purpose Timer

The general-purpose timers consist of a 16-bit or 32-bit auto-reload counter driven by a programmable prescaler. They may be used for a variety of purposes, including measuring the pulse lengths of input signals (input capture) or generating output waveforms (output compare and PWM). Pulse lengths and waveform periods can be modulated from a few microseconds to several milliseconds using the timer prescaler and the RCC clock controller prescalers. The timers are completely independent, and do not share any resources.

6.1 General Purpose Timer Registers

6.1.1 RCC APB1 peripheral clock enable register (RCC_APB1ENR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	PWR EN	Res.	CAN2 EN	CAN1 EN	I2CFMP1 EN	I2C3 EN	I2C2 EN	I2C1 EN	Res.	Res.	USART3 EN	USART2 EN	Res.
			r/w		r/w	r/w	r/w	r/w	r/w	r/w			r/w	r/w	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Res.	Res.	WWDG EN	RTCAPB	Res.	TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
r/w	r/w			r/w	r/w		r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Fig.12:APB1ENR Register

Clock Enable Configuration

Bit 0 TIM2EN: TIM2 clock enable

Set and cleared by software.

0: TIM2 clock disabled

1: TIM2 clock enabled

6.1.2 TIMx control register 1 (TIMx_CR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

Fig.13:Timer Control register 1

TIMx Control Register (TIMx_CR1)

Bits 15:10 Reserved, must be kept at reset value.

Bits 9:8 CKD: Clock division

- 00: $t_{DTS} = t_{CK_INT}$
- 01: $t_{DTS} = 2 \times t_{CK_INT}$
- 10: $t_{DTS} = 4 \times t_{CK_INT}$

-
- 11: Reserved

Bit 7 ARPE: Auto-reload preload enable

- 0: TIMx_ARR register is not buffered
- 1: TIMx_ARR register is buffered

Bits 6:5 CMS: Center-aligned mode selection

- 00: Edge-aligned mode. The counter counts up or down depending on the direction bit (DIR).
- 01: Center-aligned mode 1. The counter counts up and down alternatively. Output compare interrupt flags of channels configured in output (CCxS=00 in TIMx_CCMRx register) are set only when the counter is counting down.
- 10: Center-aligned mode 2. The counter counts up and down alternatively. Output compare interrupt flags of channels configured in output (CCxS=00 in TIMx_CCMRx register) are set only when the counter is counting up.
- 11: Center-aligned mode 3. The counter counts up and down alternatively. Output compare interrupt flags of channels configured in output (CCxS=00 in TIMx_CCMRx register) are set both when the counter is counting up or down.

Note: It is not allowed to switch from edge-aligned mode to center-aligned mode as long as the counter is enabled (CEN=1)

Bit 4 DIR: Direction

- 0: Counter used as upcounter
- 1: Counter used as downcounter

Note: This bit is read-only when the timer is configured in Center-aligned mode or Encoder mode.

Bit 3 OPM: One-pulse mode

- 0: Counter is not stopped at update event
- 1: Counter stops counting at the next update event (clearing the bit CEN)

Bit 2 URS: Update request source

- 0: Any of the specified events generate an update interrupt or DMA request if enabled. These events can be:
 - Counter overflow/underflow
 - Setting the UG bit
 - Update generation through the slave mode controller
- 1: Only counter overflow/underflow generates an update interrupt or DMA request if enabled.

Bit 1 UDIS: Update disable

- 0: UEV enabled. The Update (UEV) event is generated by one of the specified events.
 - Counter overflow/underflow
 - Setting the UG bit
 - Update generation through the slave mode controller

Buffered registers are then loaded with their preload values.

- 1: UEV disabled. The Update event is not generated, shadow registers keep their value (ARR, PSC, CCRx). However, the counter and the prescaler are reinitialized if the UG bit is set or if a hardware reset is received from the slave mode controller.

Bit 0 CEN: Counter enable

- 0: Counter disabled
- 1: Counter enabled

Note: External clock, gated mode, and encoder mode can work only if the CEN bit has been previously set by software. However, trigger mode can set the CEN bit automatically by hardware. CEN is cleared automatically in one-pulse mode when an update event occurs.

Follow below calculation formula for determining timer period:

- Let
 - PSC = Prescaler (TIMx->PSC)
 - ARR = Auto-Reload Register (TIMx->ARR)
- Then

$$freq_{timer} = \frac{freq_{clock}}{(PSC + 1) \times (ARR + 1)}$$

or

$$period_{timer} = period_{clock} \times (PSC + 1) \times (ARR + 1)$$

6.1.3 TIMx prescaler (TIMx_PSC)

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

Fig.14:Prescaler Register

Prescaler Configuration

Bits 15:0 PSC[15:0]: Prescaler value

The counter clock frequency CK_CNT is equal to $\frac{f_{CK_PSC}}{PSC[15:0]+1}$.

PSC contains the value to be loaded in the active prescaler register at each update event.

6.1.4 TIMx auto-reload register (TIMx_ARR)

Auto-reload Value Configuration

Bits 15:0 ARR[15:0]: Auto-reload value

ARR is the value to be loaded in the actual auto-reload register.

The counter is blocked while the auto-reload value is null.

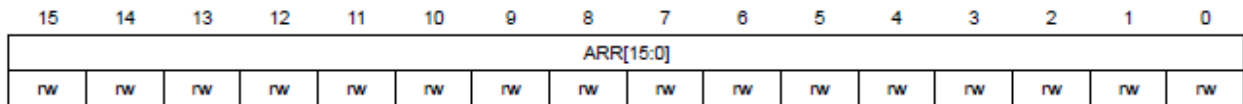


Fig.15:Auto reload Register

6.1.5 TIMx counter (TIMx_CNT)

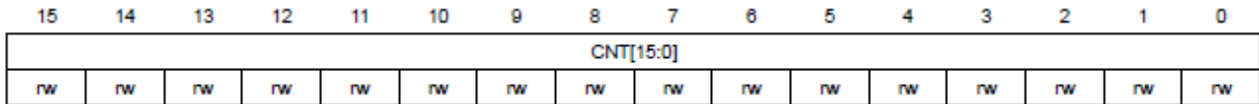


Fig.16: Counter Register

Counter Value Configuration

Bits 15:0 CNT[15:0]: Counter value

6.1.6 Programming Example

```

void timer2_initialize ()
{
    RCC->AHB1ENR |= 0x01; //enable clock
    RCC->AHB1ENR |= 0x02; //enable clock

    //GPIOA->MODER |=0x5000; //mode register
    GPIOB->MODER |= 0x10004000; //mode register to configure PB14 & PB7 as output
    //GPIOA->ODR =0xC0; //data register

    GPIOB->ODR = 0x0000; //data register Make PB7 & PB14 zero

    RCC->APB1ENR |= 0x02; //enable clock for APB1 where TIM2 is located
    TIM2->PSC = calculate; //prescaler Previous Value
    TIM2->ARR = calculate; //auto-Reload Resistor Value

    TIM2->CNT = 0; //clear Counter

    TIM2->CR1 = 1; // Start Timer 2

    GPIOB->ODR = 0x4080;
}

```

Note: This is not a complete code.

6.2 Programming Exercises

Write a program to toggle LED every 5 sec using General Purpose Timer (Timer 2). What changes need to be done for a Down Counting Mode?

7 Analog To Digital Converter

One of the most common peripherals on many modern microcontrollers is the analog-to-digital converter (ADC). The processor reads an analog voltage (usually somewhere between 0 V and the given reference voltage) and reports it as a binary value. The STM32F412ZGT6 processor is equipped with a 12-bit Successive Approximation based ADC having up to 16 external channels and 3 internal channels.

The main features include:

- ADC can work in different modes such as: single conversion, continuous conversion, scanning, etc.
- ADC can work on different individual sampling rates and resolutions, i.e., 6, 8, 10, or 12 bits.
- Interrupt generation feature at end of conversion, overrun events, etc.
- The analog watchdog feature allows the application to detect if the input voltage goes beyond the user-defined higher or lower thresholds.
- Channel selection can be done in any number and sequence by changing some dedicated registers (ADC_SQRx).

We shall be starting with a single channel to be configured in single or continuous conversion mode. A general procedure in doing so may be given by:

- Enable required ADC and GPIO clocks. (Need to use the corresponding registers in RCC module)
- Set resolution using CR1 register.
- Set End of conversion (EOC), data-alignment (Left / Right), and continuous conversion bits in CR2 register.
- Set channel sequence length using ADC_SQR1 register (as an example only 1 channel).
- Set respective GPIO pins in Analog mode (11 in for respective bit in MODER).
- Enable ADC by setting ADON bit.
- Set channel sequence using ADC_SQRx registers.
- Clear status register and start conversion using SWSTART in CR2.

7.1 ADC Registers

7.1.1 ADC Control Register (ADC_CR2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	SWSTART	EXTEN			EXTSEL[3:0]				Res.	JSWSTART	JEXTEN		JEXTSEL[3:0]		
	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
Res.	Res.	Res.	Res.	ALIGN	EOCS	DDS	DMA	Res.	Res.	Res.	Res.	Res.	Res.	CONT	ADON
				rw	rw	rw	rw							rw	rw

Fig.17:ADC Control Register-2

Bit 30 SWSTART: Start conversion of regular channels.

- This bit is set by software to start conversion and cleared by hardware as soon as the conversion starts.
- 0: Reset state
- 1: Starts conversion of regular channels
- Note: This bit can be set only when ADON = 1 otherwise no conversion is launched.

Bit 11 ALIGN: Data alignment

- This bit is set and cleared by software.
- 0: Right alignment
- 1: Left alignment

Bit 10 EOCS: End of conversion selection.

- The EOC bit is set at the end of each regular conversion.

Bit 1 CONT: Continuous conversion.

- This bit is set and cleared by software. If it is set, conversion takes place continuously until it is cleared.
- 0: Single conversion mode
- 1: Continuous conversion mode

Bit 0 ADON: A/D Converter ON / OFF

- This bit is set and cleared by software.
- 0: Disable ADC conversion and go to power down mode
- 1: Enable ADC

7.1.2 ADC Control Register (ADC_CR1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	OVRIE	RES	RES	AWDEN	JAWDEN	Res.	Res.	Res.	Res.	Res.	Res.
					rw	rw	rw	rw	rw						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Fig.18:ADC Control Register-1

Bits 25:24 RES[1:0]: Resolution

These bits are written by software to select the resolution of the conversion.

- 00: 12-bit (15 ADCCLK cycles)
- 01: 10-bit (13 ADCCLK cycles)
- 10: 8-bit (11 ADCCLK cycles)
- 11: 6-bit (9 ADCCLK cycles)

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
Reserved										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_wd	rc_wd	rc_wd	rc_wd	rc_wd	rc_wd

Fig.19:ADC status register

7.1.3 ADC status register (ADC_SR)

Bits 31:6 Reserved, must be kept at reset value.

Bit 5 OVR: Overrun

- This bit is set by hardware when data is lost (either in single mode or in dual/triple mode). It is cleared by software. Overrun detection is enabled only when DMA = 1 or EOCS = 1.
- 0: No overrun occurred
- 1: Overrun has occurred

Bit 4 STRT: Regular channel start flag

- This bit is set by hardware when regular channel conversion starts. It is cleared by software.
- 0: No regular channel conversion started
- 1: Regular channel conversion has started

Bit 3 JSTRT: Injected channel start flag

- This bit is set by hardware when injected group conversion starts. It is cleared by software.
- 0: No injected group conversion started
- 1: Injected group conversion has started

Bit 2 JEOC: Injected channel end of conversion

- This bit is set by hardware at the end of the conversion of all injected channels in the group. It is cleared by software.
- 0: Conversion is not complete
- 1: Conversion complete

Bit 1 EOC: Regular channel end of conversion

- This bit is set by hardware at the end of the conversion of a regular group of channels. It is cleared by software or by reading the ADC_DR register.
- 0: Conversion not complete (EOCS=0), or sequence of conversions not complete (EOCS=1)
- 1: Conversion complete (EOCS=0), or sequence of conversions complete (EOCS=1)

Bit 0 AWD: Analog watchdog flag

- This bit is set by hardware when the converted voltage crosses the values programmed in the ADC_LTR and ADC_HTR registers. It is cleared by software.

- 0: No analog watchdog event occurred
- 1: Analog watchdog event occurred

ADC regular sequence register (ADC_SQRx):

Used for pre-deciding the number of channels used and their sequence. (Refer datasheet for elaborated explanation)

ADC regular data register (ADC_DR):

Bits 15:0 → These bits are read-only. They contain the conversion result from the regular channels, i.e., the current equivalent digital value of the input analog signal can be observed using this register.

ADC1 internal connections:

- Channel IN_0 to IN_7 → PA0 to PA7
- Channel IN_8 to IN_9 → PB0 to PB1
- Channel IN_10 to IN_15 → PC0 to PC5

7.2 Programming Example Using Potentiometer

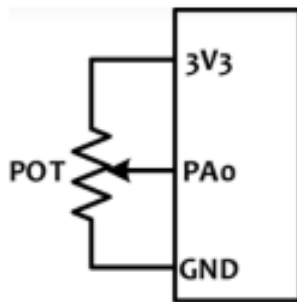


Fig.20:Potentiometer

Problem: Configure ADC1 Channel IN_0 to read data at its respective GPIO pin.

```
#include "stm32f412Zx.h" // Device header

int adc_value; // Defining variable to watch ADC values

int main() {
    RCC->APB2ENR |= 0x100; // Enable clock for ADC1
    RCC->AHB1ENR |= 0x1;
    // Enable clock for GPIOA- PA0 is internally connected to IN_0)

    ADC1->CR2 |= 0x2; // Enable continuous conversion mode
    ADC1->CR2 |= 0x400; // EOC after each conversion
    ADC1->CR2 |= 0x1; // ADON =1 enable ADC1
    ADC1->SQR3 |= 0; // Conversion in regular sequence

    GPIOA->MODER |= 0x3; // Analog mode for PA0

    while(1) {
        ADC1->SR = 0; // Clear the status register
        ADC1->CR2 |= (0x40000000); // Starting conversion by SWSTART
```

```
        while (ADC1->SR & (0x2)) {    // Check until conversion completes
            adc_value = ADC1->DR;    // Update ADC values to variable defined
        }
    }
```

NOTE: If a potentiometer is available, then one can connect its two fixed ends at Vcc and GND and its wiper end to PA0. By doing so, you can observe the DR value (i.e., variable `adc_value`) in ADC1 using debugging at different wiper positions. Otherwise, you can directly connect PA0 to extreme points, i.e., Vcc and GND, and observe DR values one by one on both.

7.3 Programming Exercise

Write a program to blink LED (any LED available on the board can be used) when analog input to PA0 pin drops below 1.78 volts. (Use potentiometer or voltage divider network to provide this voltage to pin PA0.)

8 Universal Asynchronous Receiver and Transmitter (UART)

The universal synchronous asynchronous receiver transmitter (USART) offers a flexible means of full-duplex data exchange with external equipment requiring an industry standard NRZ asynchronous serial data format. The USART offers a very wide range of baud rates using a fractional baud rate generator. 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 multibuffer configuration.

8.1 USART registers

8.1.1 Status register (USART_SR)

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
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

Fig.21:USART status register

Bit 31:10 Reserved, must be kept at reset value.

Bit 9 CTS: CTS flag

- 0: No change occurred on the CTS status line
- 1: A change occurred on the CTS status line

Note: This bit is not available for UART4 & UART5.

Bit 8 LBD: LIN break detection flag

- 0: LIN Break not detected
- 1: LIN break detected

Note: An interrupt is generated when LBD=1 if LBDIE=1.

Bit 7 TXE: Transmit data register empty

- 0: Data is not transferred to the shift register
- 1: Data is transferred to the shift register

Note: This bit is used during single buffer transmission.

Bit 6 TC: Transmission complete

- 0: Transmission is not complete
- 1: Transmission is complete

Bit 5 RXNE: Read data register not empty

- 0: Data is not received

-
- 1: Received data is ready to be read

Bit 4 IDLE: IDLE line detected

- 0: No Idle Line is detected
- 1: Idle Line is detected

Note: The IDLE bit will not be set again until the RXNE bit has been set itself (i.e., a new idle line occurs). **Bit 3 ORE:** Overrun error

- 0: No Overrun error
- 1: Overrun error is detected

Note: When this bit is set, the RDR register content will not be lost, but the shift register will be overwritten. An interrupt is generated on the ORE flag in case of Multi Buffer communication if the EIE bit is set.

Bit 2 NF: Noise detected flag

- 0: No noise is detected
- 1: Noise is detected

Note: This bit does not generate an interrupt as it appears at the same time as the RXNE bit, which itself generates an interrupt. An interrupt is generated on the NF flag in case of Multi Buffer communication if the EIE bit is set.

Note: When the line is noise-free, the NF flag can be disabled by programming the ONEBIT bit to 1 to increase the USART tolerance to deviations (Refer to Section 30.3.5: USART receiver tolerance to clock deviation on page 988).

Bit 1 FE: Framing error

- 0: No Framing error is detected
- 1: Framing error or break character is detected

Note: This bit does not generate an interrupt as it appears at the same time as the RXNE bit, which itself generates an interrupt. If the word currently being transferred causes both frame error and overrun error, it will be transferred, and only the ORE bit will be set. An interrupt is generated on the FE flag in case of Multi Buffer communication if the EIE bit is set.

Bit 0 PE: Parity error

- 0: No parity error
- 1: Parity error

This bit is set by hardware when a parity error occurs in receiver mode. It is cleared by a software sequence (a read from the status register followed by a read or write access to the USART_DR data register). The software must wait for the RXNE flag to be set before clearing the PE bit. An interrupt is generated if PEIE = 1 in the USART_CR1 register.

8.1.2 Data register (USART_DR)

Bits 31:9 Reserved, must be kept at reset value.

Bits 8:0 DR[8:0]: Data value

Contains the Received or Transmitted data character, depending on whether it is read from or written to. The Data register performs a double function (read and write) since it is composed of two registers, one for transmission (TDR) and one for reception (RDR). The TDR register provides the parallel interface between the internal bus and the output shift register. The RDR register provides the parallel interface between the input shift register and the internal bus. When transmitting with the parity enabled (PCE bit set to 1 in the USART_CR1 register), the value written in the MSB (bit 7 or bit 8, depending on the data length) has no effect because it is replaced by the parity. When receiving with parity enabled, the value read in the MSB bit is the received parity bit.

8.1.3 Baud rate register (USART_BRR)

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
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Fig.22:USART Baud Rate Register

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

Bits 15:4 DIV_Mantissa[11:0]: Mantissa of USARTDIV

These 12 bits define the mantissa of the USART Divider (USARTDIV).

Bits 3:0 DIV_Fraction[3:0]: Fraction of USARTDIV

These 4 bits define the fraction of the USART Divider (USARTDIV). When OVER8=1, the DIV_Fraction3 bit is not considered and must be kept cleared.

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

OVER8 = 1, if Oversampling by 8 is used
OVER8 = 0, if Oversampling by 16 is used

If OVER8=1, that means we are using an oversampling by 8, and the receiver block of the peripheral takes 8 samples to understand a bit.

Consider reference clock (F_{ck}) as 42 MHz and desired Baud rate as 115200 bps. The contents of the above register can be computed as per below illustration:

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
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

DIV_Mantissa is the integer part = 22 = 0x16

DIV_Fraction = 0.79 * 16 = 12.64 round off to 13 = 0xD = 1101

$$\text{Baudrate} = \frac{f_{CK}}{16 * \text{USARTDIV}} = \frac{f_{CK}}{16 * (22 + 0.79)} = \frac{f_{CK}}{16 * 22 + 16 * 0.79}$$

8.1.4 Control register 1 (USART_CR1)

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
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Fig.23:USART Control Register 1

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

Bit 15 OVER8: Oversampling mode

- 0: Oversampling by 16
- 1: Oversampling by 8

Note: Oversampling by 8 is not available in the Smartcard, IrDA, and LIN modes: when SCEN=1, IREN=1, or LINEN=1, then OVER8 is forced to '0 by hardware.

Bit 14 Reserved, must be kept at reset value.

Bit 13 UE: USART enable

- 0: USART prescaler and outputs disabled
- 1: USART enabled

Bit 12 M: Word length

- 0: 1 Start bit, 8 Data bits, n Stop bit
- 1: 1 Start bit, 9 Data bits, n Stop bit

Note: The M bit must not be modified during a data transfer (both transmission and reception).

Bit 11 WAKE: Wakeup method

- 0: Idle Line
- 1: Address Mark

Bit 10 PCE: Parity control enable

- 0: Parity control disabled
- 1: Parity control enabled

Bit 9 PS: Parity selection

- 0: Even parity
- 1: Odd parity

Bit 8 PEIE: PE interrupt enable

- 0: Interrupt is inhibited
- 1: An USART interrupt is generated whenever PE=1 in the USART_SR register

Bit 7 TXEIE: TXE interrupt enable

- 0: Interrupt is inhibited
- 1: An USART interrupt is generated whenever TXE=1 in the USART_SR register

Bit 6 TCIE: Transmission complete interrupt enable

- 0: Interrupt is inhibited
- 1: An USART interrupt is generated whenever TC=1 in the USART_SR register

Bit 5 RXNEIE: RXNE interrupt enable

- 0: Interrupt is inhibited
- 1: An USART interrupt is generated whenever ORE=1 or RXNE=1 in the USART_SR register

Bit 4 IDLEIE: IDLE interrupt enable

- 0: Interrupt is inhibited
- 1: An USART interrupt is generated whenever IDLE=1 in the USART_SR register

Bit 3 TE: Transmitter enable

- 0: Transmitter is disabled
- 1: Transmitter is enabled

Note: During transmission, a "0" pulse on the TE bit ("0" followed by "1") sends a preamble (idle line) after the current word, except in smartcard mode. When TE is set, there is a 1 bit-time delay before the transmission starts.

Bit 2 RE: Receiver enable

- 0: Receiver is disabled
- 1: Receiver is enabled and begins searching for a start bit

Bit 1 RWU: Receiver wakeup

- 0: Receiver in active mode
- 1: Receiver in mute mode

Note: Before selecting Mute mode (by setting the RWU bit), the USART must first receive a data byte; otherwise, it cannot function in Mute mode with wakeup by Idle line detection. In Address Mark Detection wakeup configuration (WAKE bit=1), the RWU bit cannot be modified by software while the RXNE bit is set.

Bit 0 SBK: Send break

- 0: No break character is transmitted
- 1: Break character will be transmitted

8.1.5 Control register 2 (USART_CR2)

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
Res.	LINEN	STOP[1:0]		CLKEN	CPOL	CPHA	LBCL	Res.	LBDIE	LBDL	Res.	ADD[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

Fig.24:USART Control Register2

Bits 31:15 Reserved, must be kept at reset value.

Bit 14 LINEN: LIN mode enable

- 0: LIN mode disabled
- 1: LIN mode enabled

The LIN mode enables the capability to send LIN Synch Breaks (13 low bits) using the SBK bit in the USART_CR1 register and to detect LIN Sync breaks.

Bits 13:12 STOP: STOP bits

- 00: 1 Stop bit
- 01: 0.5 Stop bit
- 10: 2 Stop bits
- 11: 1.5 Stop bit

Note: The 0.5 Stop bit and 1.5 Stop bit are not available for UART4 & UART5.

Bit 11 CLKEN: Clock enable

- 0: CK pin disabled
- 1: CK pin enabled

This bit is not available for UART4 & UART5.

Bit 10 CPOL: Clock polarity

- 0: Steady low value on CK pin outside the transmission window
- 1: Steady high value on CK pin outside the transmission window

This bit is not available for UART4 & UART5.

Bit 9 CPHA: Clock phase

- 0: The first clock transition is the first data capture edge
- 1: The second clock transition is the first data capture edge

Note: This bit is not available for UART4 & UART5. **Bit 8 LBCL:** Last bit clock pulse

- 0: The clock pulse of the last data bit is not output to the CK pin
- 1: The clock pulse of the last data bit is output to the CK pin

Note: 1: The last bit is the 8th or 9th data bit transmitted depending on the 8 or 9 bit format selected by the M bit in the USART_CR1 register. 2: This bit is not available for UART4 & UART5.

Bit 7 Reserved, must be kept at reset value.

Bit 6 LBDIE: LIN break detection interrupt enable

- 0: Interrupt is inhibited
- 1: An interrupt is generated whenever LBD=1 in the USART_SR register

Bit 5 LBDL: LIN break detection length

- 0: 10-bit break detection
- 1: 11-bit break detection

Bit 4 Reserved, must be kept at reset value.

Bits 3:0 ADD[3:0]: Address of the USART node

- This bit-field gives the address of the USART node. This is used in multiprocessor communication during mute mode, for wake up with address mark detection.

8.2 Algorithm for Configuration of UART1

1. Enable the UART Clock (RCC— >APB1ENR) and GPIO Clock.
2. Configure the UART Pins for Alternate Functions.
3. Enable the USART by setting the UE bit in USART_CR1 register to 1.
4. Program the M bit in USART_CR1 to define the word length.
5. Select the desired baud rate using the USART_BRR register.
6. Enable the Transmitter/Receiver by setting the TE and RE bits in USART_CR1 Register.

8.3 Algorithm for Sending a character

1. Write the data to send in the USART_DR register. This action clears the TXE (Transmit Data Register Empty) bit. Repeat this step for each data to be transmitted, especially in the case of a single buffer transmission.
2. After writing the last data into the USART_DR register, wait until TC (Transmission Complete) bit is set to 1. This indicates that the transmission of the last frame is complete. This step is crucial, especially when the USART is disabled or enters the Halt mode, to avoid corrupting the last transmission.

8.4 Algorithm for Receiving a character

1. Wait for the RXNE (Read Data Register Not Empty) bit to set. It indicates that the data has been received and can be read.
2. Read the data from the USART_DR (Data Register) Register. This action also clears the RXNE bit, allowing the reception of the next data.

8.5 Interfacing of ultrasonic sensor

The ultrasonic sensor module is equipped with four pins: VCC, GND, Trig (Trigger Pin), and Echo. Its primary function is to measure the distance between the sensor and objects positioned in front of it. This measurement process involves emitting eight bursts of 40KHz signals from the transmitter. Following the transmission of these signals, a timer is initiated. When these signals encounter an object, they bounce back and are picked up by the receiver. The time it takes for the signals to return to the sensor aids in calculating the distance from the module to the object. The ultrasonic sensor primarily remains in an idle state, awaiting activation through a deliberate trigger. To initiate the sensor's operation, a short pulse is sent to the TRIGGER pin, with a duration greater than 2 microseconds. This pulse can even last for a few milliseconds. Once triggered, the ultrasonic sensor releases eight pulses of sound waves, operating at an ultrasonic frequency of 40KHz. These sound waves travel along an almost direct path until they encounter an object, upon which they reflect back to the sensor module.

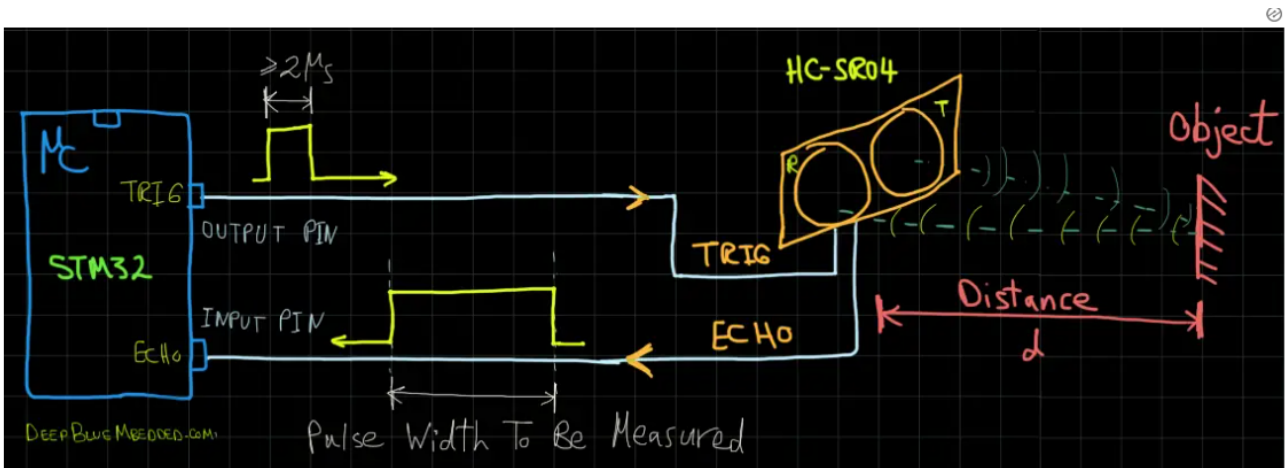


Fig.25:Ultrasonic Sensor Working

The sensor module, in turn, generates a digital pulse on the echo pin. The width of this pulse corresponds to the time taken for the sound to make a round trip between the module and the detected object. As system designers, our task is to capture the incoming echo pulse from the sensor and measure its duration. With this information in hand, calculating the distance between our sensor and the object ahead becomes a straightforward task. Given that sound maintains a constant speed in the air, we can accurately determine the distance based on the measured echo pulse width.

8.5.1 Mathematical Formulation

The ultrasonic sensor emits 8 acoustic bursts that travel through the air at a speed of 340 m/s. To calculate the distance, we can use the basic formula:

$$\text{Distance} = \text{Speed} \times \text{Time}$$

In this formula, the time taken for the receiver to receive the signal back from the object must be divided by 2, as it accounts for the round trip travel time of the signal.

The clock frequency of the STM32F412 microcontroller is 16 MHz by default. We will use this frequency to calculate the time taken for the signal to return. Each clock cycle of the microcontroller represents 0.0625 microseconds, which is equivalent to $\frac{1}{16 \text{ MHz}}$. Therefore, the number of clock cycles multiplied by 0.0625 microseconds gives us the time taken by the receiver to receive the signal.

Finally, the distance in centimeters (cm) can be calculated as follows:

$$\text{Time} = \frac{\text{Clock Cycles} \times 0.0625 \times 0.000001}{2}$$

$$\text{Distance} = (340 \times 100) \times \text{Time}$$

Here, we've converted time into seconds and distance into centimeters to ensure consistent units.

8.6 Programming Exercise

Write a program to interface Ultrasonic sensor and display the distance to Putty software using Serial communication.

Hint: Use timer to generate 10us pulse.

9 Seven Segment Interfacing

A seven-segment display is a form of electronic display device used to represent decimal numbers through the use of seven individually addressable segments. Each segment is a light-emitting diode (LED) or other display technology that can be independently turned on or off. The seven segments are arranged in a rectangular fashion to form the digit representation of numbers from 0 to 9. The basic structure of a seven-segment display includes seven LEDs, arranged in the shape of the digit "8."

9.1 Common Cathode Configuration

In the common cathode display, all the cathode connections of the LED segments are joined together to logic "0" or ground. The individual segments are illuminated by application of a "HIGH", or logic "1" signal via a current limiting resistor to forward bias the individual Anode terminals (a-g).

9.2 Common Anode Configuration

In the common anode display, all the anode connections of the LED segments are joined together to logic "1". The individual segments are illuminated by applying a ground, logic "0" or "LOW" signal via a suitable current limiting resistor to the Cathode of the particular segment (a-g).

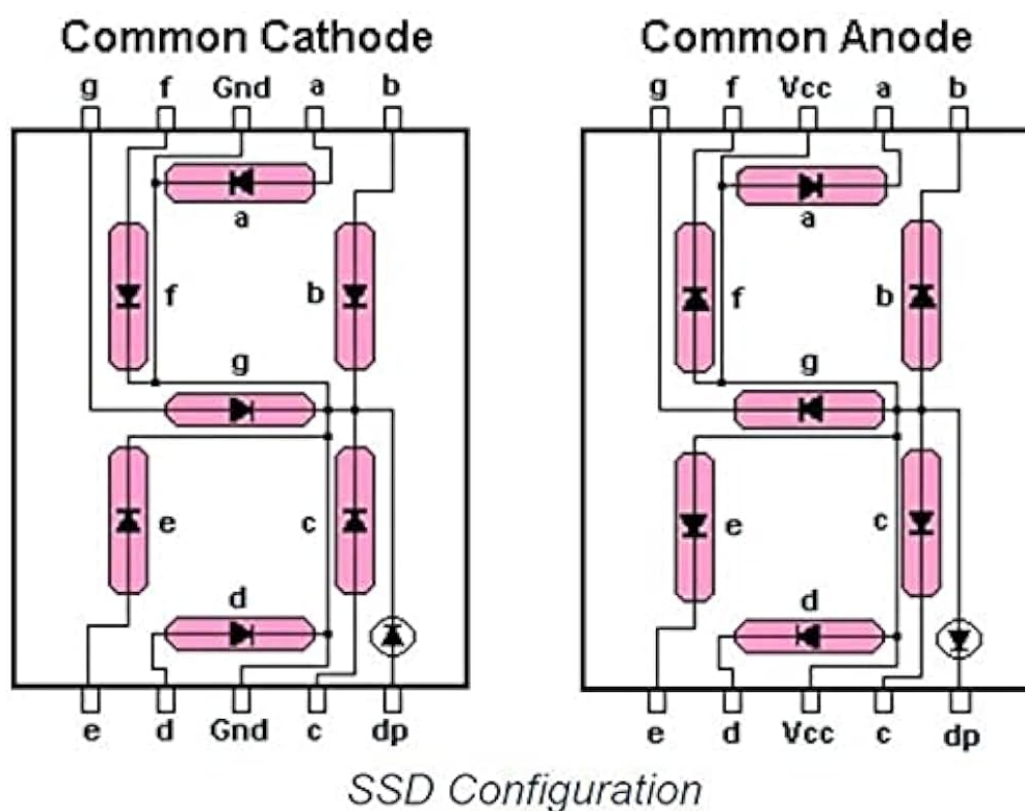


Fig.26:Common Cathode and common anode Seven Segment

9.3 Programming Exercise

Write a program to interface a seven-segment display with any available ports on the Nucleo board, displaying a count from 0 to 9.

10 External Interrupts

There are 23 different external interrupts multiplexed with GPIOs that can be used for different purposes. The Nucleo 144 board has a user push-button switch connected to PC13 (please see the schematic of the board). The external interrupt/event controller consists of up to 23 edge detectors for generating event/interrupt requests. Each input line can be independently configured to select the type (interrupt or event) and the corresponding trigger event (rising, falling, or both). Each line can also be masked independently. A brief structure of the external interrupts is shown below:

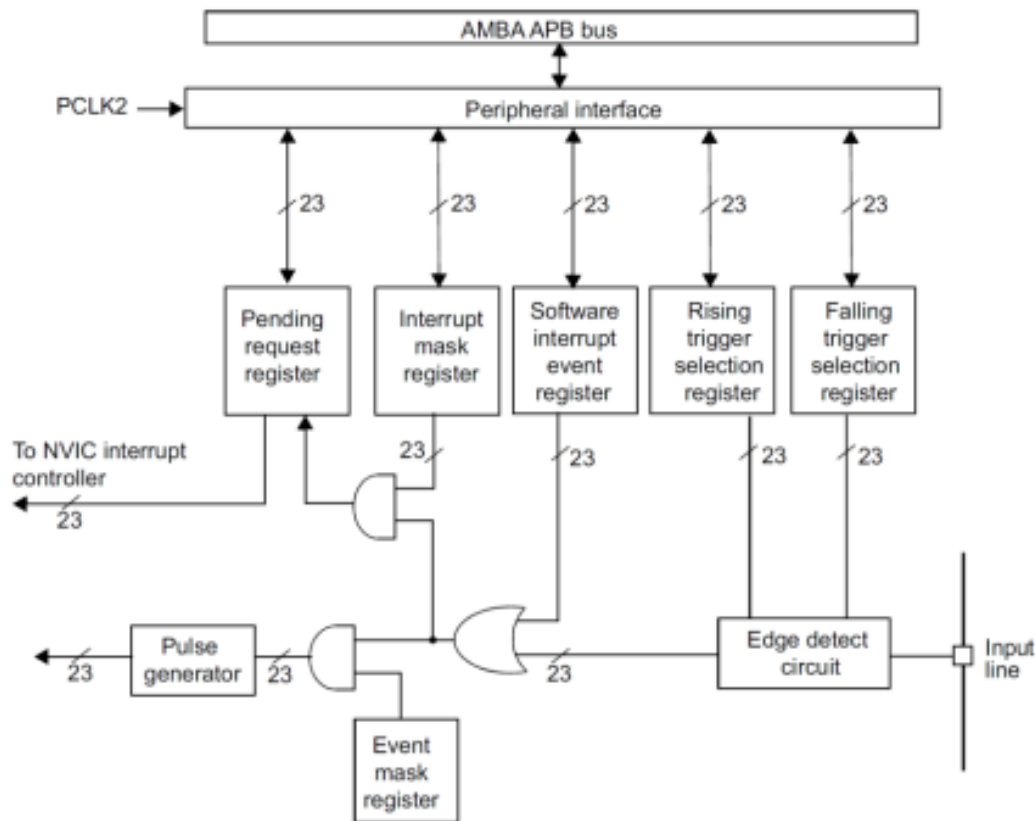


Fig.27:External Interrupt Bus

10.1 Configuring External Interrupts

To configure the 23 lines as interrupt sources, use the following procedure:

- Configure the mask bits of the 23 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 23 lines can be correctly acknowledged.

10.2 External Interrupts mapping with GPIOs

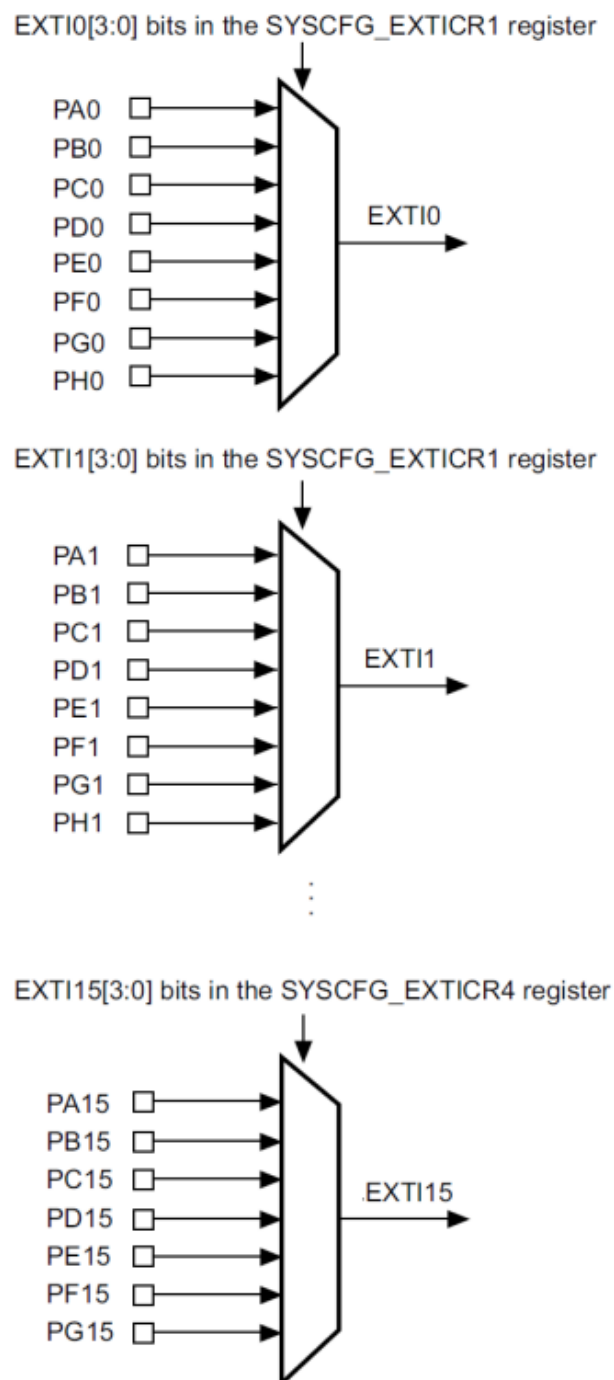


Fig.28:External Interrupt Mapping

10.3 Registers

10.3.1 Interrupt Mask Register (EXT1_IMR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	MR22	MR21	Res.	Res.	MR18	MR17	MR16
									r/w	r/w			r/w	r/w	r/w
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
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

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

Bits 22:21 **MR[22:21]**: Interrupt mask on line x

0: Interrupt request from line x is masked

1: Interrupt request from line x is not masked

Bits 20:19 Reserved, must be kept at reset value.

Bits 18:0 **MR[18:0]**: Interrupt mask on line x

0: Interrupt request from line x is masked

1: Interrupt request from line x is not masked

Fig.29:Interrupt Mask Register

10.3.2 Rising Trigger selection register (EXT1_RTSTR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	Res.	Res.	TR18	TR17	TR16
									r/w	r/w			r/w	r/w	r/w
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
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

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

Bits 22:21 **TR[22:21]**: 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

Bits 20:19 Reserved, must be kept at reset value.

Bits 18:0 **TR[18:0]**: 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

Fig.30:Rising Trigger selection register

10.3.3 Pending register (EXT1_PR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PR22	PR21	Res.	Res.	PR18	PR17	PR16
									rc_w1	rc_w1			rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

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

Bits 22:21 **PR[22:21]**: Pending bit

0: No trigger request occurred

1: selected trigger request occurred

This bit is set when the selected edge event arrives on the external interrupt line.

This bit is cleared by programming it to '1'.

Bits 20:19 Reserved, must be kept at reset value.

Bits 18:0 **PR[18:0]**: Pending bit

0: No trigger request occurred

1: selected trigger request occurred

This bit is set when the selected edge event arrives on the external interrupt line.

This bit is cleared by programming it to '1'.

Fig.31:Pending register

10.3.4 SYSCFG external interrupt configuration register 4 (SYSCFG_EXTIC4)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI15[3:0]				EXTI14[3:0]				EXTI13[3:0]				EXTI12[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 = 12 to 15)

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

Fig.32:SYSCFG external interrupt configuration register 4

10.4 Example Program

Observe the below program and also run the same program in your board to verify the output. You will observe that user LED will glow once you press the button. Here interrupt is used to glow the LED. Answer the exercise problems given to you. (Note: In this lab we will use STM32 Nucleo-144 board. The schematic of the board is provided to you. There are two LEDs connected to PB.7 (BLUE) and PB.14 (RED). Connect the MicroUSB cable provided with the board. Create project and select the device as STM32f412ZG. Select ST-LINK Debugger in debug option.)

```
#include "stm32f412Zx.h"

void delay(int dd);

int main() {
    __disable_irq();
    RCC->AHB1ENR |= 0x02; // Enable Clock Port B
    RCC->AHB1ENR |= 0x4;  // Enable Clock Port C PB connected to PC13
    RCC->APB2ENR |= 0X4000; // Enable clock to SYSCFG
    GPIOB->MODER |= 0x00004000; // Mode Register for PB7
    SYSCFG->EXTICR[3] = 0X0020; // Port C for EXT 13 INTERRUPT
    EXTI->IMR |= 0x2000; // Unmask interrupt
    EXTI->RTSR |= 0x2000; // Rising edge
    NVIC_EnableIRQ(EXTI15_10_IRQn);
    __enable_irq();
    while(1) {}
}

void EXTI15_10_IRQHandler(void) {
    GPIOB->ODR = 0X80; // Turn on LED
    EXTI->PR = 0X2000;
}
```

10.5 Programming exercise

Exercise 1: Blink LED Twice on Button Press

Modify the given program so that the LED connected to PB7 blinks twice when the button is pressed. Introduce a delay between each blink to allow for observation of the output. The LED blinks in response to the button press, and the program is designed to capture the button press using an interrupt mechanism.

Exercise 2: Use Timer for Delay

In Exercise 1, enhance the program to use a timer to provide a delay between successive blinks of the LED. By incorporating a timer for the delay, the program gains more control over timing intervals and can achieve more precise delays. This modification enhances the flexibility and accuracy of the LED blinking pattern.

Appendix-A: ARM INSTRUCTION SET

Similar to high level languages, ARM supports operations on different datatypes. The data types we can load (or store) can be signed and unsigned words, halfwords, or bytes. The extensions for these data types are: -h or -sh for halfwords, -b or -sb for bytes, and no extension for words. The difference between signed and unsigned data types is:

Signed data types can hold both positive and negative values and are therefore lower in range. Unsigned data types can hold large positive values (including 'Zero') but cannot hold negative values and are therefore wider in range.

#	Alias	Purpose
R0	-	General purpose
R1	-	General purpose
R2	-	General purpose
R3	-	General purpose
R4	-	General purpose
R5	-	General purpose
R6	-	General purpose
R7	-	Holds Syscall Number
R8	-	General purpose
R9	-	General purpose
R10	-	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter
CPSR	-	Current Program Status Register

Registers in ARM architecture

Depending upon the context, registers r13 and r14 can also be used as general-purpose registers, which can be particularly useful since these registers are banked during a processor mode change. While the CPU is running any type of operating system, it is risky to utilize r13 as a general register since operating systems frequently believe that r13 always refers to a valid stack frame. The registers r0 to r13 are orthogonal in ARM state, which means that any instruction that can be applied to r0 may also be applied to any of the other registers. There are, however, instructions that treat r14 and r15 differently. There are two program status registers: cpsr and spsr, in addition to the 16 data registers (the current and saved program status registers, respectively). The register file includes all of the registers that a programmer has access to. The current mode of the computer determines which registers are visible to the programmer.

Memory access instructions

```

LDR    Rd, [Rn]           ; load 32-bit number at [Rn] to Rd
LDR    Rd, [Rn,#off]      ; load 32-bit number at [Rn+off] to Rd
LDR    Rd, =value         ; set Rd equal to any 32-bit value (PC rel)
LDRH   Rd, [Rn]           ; load unsigned 16-bit at [Rn] to Rd
LDRH   Rd, [Rn,#off]      ; load unsigned 16-bit at [Rn+off] to Rd
LDRSH   Rd, [Rn]          ; load signed 16-bit at [Rn] to Rd
LDRSH   Rd, [Rn,#off]     ; load signed 16-bit at [Rn+off] to Rd
LDRB   Rd, [Rn]           ; load unsigned 8-bit at [Rn] to Rd
LDRB   Rd, [Rn,#off]      ; load unsigned 8-bit at [Rn+off] to Rd
LDRSB   Rd, [Rn]          ; load signed 8-bit at [Rn] to Rd
LDRSB   Rd, [Rn,#off]     ; load signed 8-bit at [Rn+off] to Rd
STR     Rt, [Rn]           ; store 32-bit Rt to [Rn]
STR     Rt, [Rn,#off]      ; store 32-bit Rt to [Rn+off]
STRH    Rt, [Rn]           ; store least sig. 16-bit Rt to [Rn]
STRH    Rt, [Rn,#off]     ; store least sig. 16-bit Rt to [Rn+off]
STRB    Rt, [Rn]           ; store least sig. 8-bit Rt to [Rn]
STRB    Rt, [Rn,#off]     ; store least sig. 8-bit Rt to [Rn+off]
PUSH    {Rt}              ; push 32-bit Rt onto stack
POP     {Rd}              ; pop 32-bit number from stack into Rd
ADR     Rd, label         ; set Rd equal to the address at label
MOV{S}  Rd, <op2>         ; set Rd equal to op2
MOV     Rd, #im16         ; set Rd equal to im16, im16 is 0 to 65535
MVN{S}  Rd, <op2>         ; set Rd equal to -op2

```

Branch instructions

```

B       label             ; branch to label      Always
BEQ     label             ; branch if Z == 1      Equal
BNE     label             ; branch if Z == 0      Not equal
BCS     label             ; branch if C == 1      Higher or same, unsigned ≥
BHS     label             ; branch if C == 1      Higher or same, unsigned ≥
BCC     label             ; branch if C == 0      Lower, unsigned <
BLO     label             ; branch if C == 0      Lower, unsigned <
BMI     label             ; branch if N == 1      Negative
BPL     label             ; branch if N == 0      Positive or zero
BVS     label             ; branch if V == 1      Overflow
BVC     label             ; branch if V == 0      No overflow
BHI     label             ; branch if C==1 and Z==0 Higher, unsigned >
BLS     label             ; branch if C==0 or Z==1 Lower or same, unsigned ≤
BGE     label             ; branch if N == V      Greater than or equal, signed ≥
BLT     label             ; branch if N != V      Less than, signed <
BGT     label             ; branch if Z==0 and N==V Greater than, signed >
BLE     label             ; branch if Z==1 or N!=V Less than or equal, signed ≤
BX      Rm                ; branch indirect to location specified by Rm
BL      label             ; branch to subroutine at label, return address in LR
BLX     Rm                ; branch to subroutine indirect specified by Rm

```

Interrupt instructions

```

CPSIE   I                 ; enable interrupts (I=0)
CPSID   I                 ; disable interrupts (I=1)

```

Logical instructions

```

AND{S}  {Rd}, {Rn}, <op2> ; Rd=Rn&op2      (op2 is 32 bits)
ORR{S}  {Rd}, {Rn}, <op2> ; Rd=Rn|op2      (op2 is 32 bits)
EOR{S}  {Rd}, {Rn}, <op2> ; Rd=Rn^op2      (op2 is 32 bits)
BIC{S}  {Rd}, {Rn}, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S}  {Rd}, {Rn}, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S}  Rd, Rm, Rs        ; logical shift right Rd=Rm>>Rs (unsigned)
LSR{S}  Rd, Rm, #n        ; logical shift right Rd=Rm>>n (unsigned)

```

```

ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n (signed, unsigned)

```

Arithmetic instructions

```

ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12 ; Rd = im12 - Rn
CMP   Rn, <op2>        ; Rn - op2      sets the NZVC bits
CMN   Rn, <op2>        ; Rn - (-op2)   sets the NZVC bits
MUL{S} {Rd,} Rn, Rm     ; Rd = Rn * Rm   signed or unsigned
MLA   Rd, Rn, Rm, Ra    ; Rd = Ra + Rn*Rm signed or unsigned
MLS   Rd, Rn, Rm, Ra    ; Rd = Ra - Rn*Rm signed or unsigned
UDIV  {Rd,} Rn, Rm      ; Rd = Rn/Rm     unsigned
SDIV  {Rd,} Rn, Rm      ; Rd = Rn/Rm     signed

```

Notes Ra Rd Rm Rn Rt represent 32-bit registers

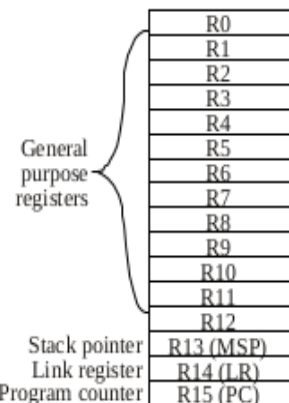
value any 32-bit value: signed, unsigned, or address
 {S} if S is present, instruction will set condition codes
 #im12 any value from 0 to 4095
 #im16 any value from 0 to 65535
 {Rd,} if Rd is present Rd is destination, otherwise Rn
 #n any value from 0 to 31
 #off any value from -255 to 4095
 label any address within the ROM of the microcontroller
 op2 the value generated by <op2>

Examples of flexible operand <op2> creating the 32-bit number. E.g., Rd = Rn+op2

```

ADD Rd, Rn, Rm          ; op2 = Rm
ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned
ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned
ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed
ADD Rd, Rn, #constant  ; op2 = constant, where X and Y are hexadecimal digits:
    • produced by shifting an 8-bit unsigned value left by any number of bits
    • in the form 0x00XY00XY
    • in the form 0xXY00XY00
    • in the form 0xXYXYXYXY

```



Condition code bits

N negative
 Z zero
 V signed overflow
 C carry or unsigned overflow

