1) What is an interrupt vector table?

Ans:- An Interrupt Vector Table (IVT) is a data structure in computer systems that maps interrupt or exception numbers to corresponding handler routines, allowing the processor to efficiently respond to external events or software requests.

2)What are the three steps required to complete the configuration of an external interrupt?

Configuring an external interrupt typically involves the following three steps:

1.Hardware Setup
2Interrupt Configuration
3.Interrupt Service Routine (ISR):

3.What are the different types of interrupts supported by STM32 microcontrollers?

ANS:-

External Interrupts (EXTI),Timers and Timers' Interrupts,UART/USART Interrupts:,SPI (Serial Peripheral Interface) Interrupts,

I2C (Inter-Integrated Circuit) Interrupts:,DMA (Direct Memory Access) Interrupts,ADC (Analog-to-Digital Converter) Interrupts,USB (Universal Serial Bus) Interrupts:,CAN (Controller Area Network) Interrupts:,NVIC (Nested Vectored Interrupt Controller) Interrupts.

4. What is the difference between a blocking and a non-blocking function?

ANS:-

Blocking Function:

Synchronous: A blocking function is synchronous, meaning that when you call it, your program will typically wait until the function completes its task before moving on to the next line of code.

Non-blocking Function:

Asynchronous: A non-blocking function is asynchronous, meaning that when you call it, your program doesn't wait for the function to finish. Instead, it continues to execute the next lines of code.

5.What are the two main types of memory used in a microcontroller?

Microcontrollers typically use two main types of memo

Program Memory (Flash Memory):

Data Memory (RAM - Random Access Memory):

6.What type of signals are generated from discrete devices using PWM controlling technique?

ANS:-

Digital signals with varying duty cycles are generated from discrete devices using PWM (Pulse Width Modulation) control techniques.

7. What is the difference between a compiler and an assembler?

ANS:-

Compiler:

- High-Level Language: A compiler is a software tool that translates high-level programming languages (like C, C++, Java) into machine code or lower-level assembly language.
- Abstraction: It operates at a high level of abstraction, allowing programmers to write code in a more human-readable and abstract form.
- Translation: The compiler analyzes the entire source code and generates an executable binary or machine code file. The output is often independent of a specific microprocessor or target architecture.

2. Assembler:

- Low-Level Language: An assembler, on the other hand, is a software tool that translates assembly language code (mnemonics) into machine code.
- Abstraction: It operates at a lower level of abstraction, dealing with symbolic representations of machine instructions.
- Translation: The assembler translates assembly code line by line into machine code specific to the target microprocessor or architecture. It produces a binary file that is closely tied to the underlying hardware.

8.What are the different types of interrupts used in microcontrollers?

ANS:-

External Interrupts (EXTI),Timers and Timers' Interrupts,UART/USART Interrupts:,SPI (Serial Peripheral Interface) Interrupts, I2C (Inter-Integrated Circuit) Interrupts:,DMA (Direct Memory Access) Interrupts,ADC (Analog-to-Digital Converter) Interrupts,USB (Universal Serial Bus) Interrupts:,CAN (Controller Area Network) Interrupts:,NVIC (Nested Vectored Interrupt Controller) Interrupts,Watchdog Timer (WDT) Interrupts,External Bus Interface (EBI) Interrupts,NVIC (Nested Vectored Interrupt Controller) Interrupts,Real-Time Clock (RTC) Interrupts.

9.What is the difference between a synchronous serial interface and an asynchronous serial interface?

ANS:-

Synchronous Serial Interface:

1. **Clock Signal:** In a synchronous serial interface, data transmission is synchronized by a shared clock signal that both the sender and receiver use to coordinate the timing of data bits.
2. **Fixed Data Rate:** Synchronous communication operates at a fixed data rate, which means that data bits are sent and received at a predetermined rate based on the clock signal. This rate is typically constant throughout the communication.
3. **Start and Stop Bits:** Synchronous interfaces do not use start and stop bits as found in asynchronous communication. Instead, data is sent as a continuous stream of bits.
4. **Precise Timing:** Synchronous communication is more precise in terms of timing because both devices are synchronized to the same clock signal, ensuring accurate bit transfer.

Asynchronous Serial Interface:

1. **No Shared Clock Signal:** In an asynchronous serial interface, there is no shared clock signal. Instead, devices rely on the sender and receiver independently agreeing on the data rate (baud rate) in advance.
2. **Variable Data Rate:** Asynchronous communication allows for variable data rates, meaning that the sender and receiver must agree on the baud rate, and both devices must configure their clocks accordingly.
3. **Start and Stop Bits:** Asynchronous interfaces use start and stop bits to frame each data byte. Start and stop bits are used to signal the beginning and end of a data byte and provide synchronization between sender and receiver.

4. **Less Precise Timing:** Asynchronous communication is less precise in terms of timing compared to synchronous communication. Variations in clock accuracy between devices can introduce timing discrepancies.

10.What are the different types of timers used in microcontrollers?

ANS:-

types of timers used in microcontrollers:

General-Purpose Timers,Watchdog Timers (WDT),

Real-Time Clocks (RTC),Pulse Width Modulation (PWM) Timers.

11.What are the different types of serial communication protocols supported by STM32 microcontrollers?

ANS:-

UART (Universal Asynchronous Receiver-Transmitter)

1. USART (Universal Synchronous/Asynchronous Receiver-Transmitter)
2. SPI (Serial Peripheral Interface)
3. I2C (Inter-Integrated Circuit)
4. CAN (Controller Area Network)
5. USB (Universal Serial Bus)
6. Ethernet
7. Modbus

12.What is the purpose of the DMA controller in STM32 microcontrollers?

ANS:-

The purpose of the DMA (Direct Memory Access) controller in STM32 microcontrollers is to enable efficient and high-speed data transfers between memory and peripheral devices without involving the CPU, thereby reducing CPU load and improving system performance.

13.What is the purpose of the FreeRTOS operating system?

ANS:-

The purpose of the FreeRTOS operating system is to provide a real-time multitasking kernel for embedded systems, facilitating task management and scheduling to ensure timely execution of tasks and efficient resource utilization.

14.What is the difference between a polling loop and a callback function?

ANS:-

Polling Loop:

- **Active Waiting:** In a polling loop, the program actively checks (polls) the status or condition of an event or resource in a repetitive loop.
- **Synchronous:** It is a synchronous approach where the program execution is blocked or busy-waiting until the event or condition being polled becomes true or is satisfied.
- **Resource-Intensive:** Polling loops can consume CPU resources continuously, even when no meaningful work is being done, as the loop keeps running.
- **Simple to Implement:** Polling loops are relatively straightforward to implement and are suitable for simple scenarios where waiting for events is brief and doesn't affect overall system performance.

2. **Callback Function:**

- **Event-Driven:** Callback functions are used in an event-driven or asynchronous programming model, where the program registers a callback to be executed when a specific event or condition occurs.
- **Non-Blocking:** The program doesn't wait for the event but continues executing other tasks until the event triggers the associated callback function.
- **Efficient:** Callbacks are more resource-efficient as they allow the CPU to perform other tasks while waiting for events. This makes them suitable for systems with multiple concurrent tasks.
- **Complexity:** Implementing callback-based systems can be more complex than polling loops, as it requires managing

15.What is the use of watchdog in STM32?

ANS:-

The watchdog in STM32 serves as a hardware safety mechanism to monitor and reset the microcontroller in case of software or system failures, ensuring the system's reliability and fault tolerance.

16.SPI is Synchronous or Asynchronous Communication?

ANS:-

SPI (Serial Peripheral Interface) is a synchronous communication protocol.

17.What are the 4 logic signals specified by the SPI bus?

ANS:-

The SPI (Serial Peripheral Interface) bus typically uses four logic signals:

1. MOSI (Master Out Slave In):
2. MISO (Master In Slave Out)
3. SCLK (Serial Clock)
4. SS(Slave Select)

18.What is I2C communication?

ANS:-

I2C (Inter-Integrated Circuit) is a synchronous serial communication protocol that allows multiple devices to communicate with each other over a two-wire bus, consisting of a data line (SDA) and a clock line (SCL), in a bidirectional manner.

19.What is the communication mode of UART?

ANS:-

Communication 3 modes:

•Simplex (Transmit in one direction only).

•Half duplex (Either transmit or receive)

•Full duplex (both devices send and receive at the same time).

20.What is the essential element of the UART Frame?

ANS:-

The essential elements of a UART (Universal Asynchronous Receiver-Transmitter) frame include:

Start Bit:

Data Bits:

Parity Bit

Stop Bit.

Part-B 10*3=30

1. How do you use the SysTick timer to generate a delay?

Ans:-

Initialization:

- Configure the SysTick timer with the desired time interval for the delay.
- Set the SysTick Control and Status Register (SysTick_CTRL) to enable the timer and select the clock source (typically the CPU clock).

2. **Start the Timer:**
   - Enable the SysTick timer, which initiates the countdown.
3. **Wait for the Timer to Expire:**
   - Enter a loop that continuously checks the status of the SysTick timer.
   - The timer status is typically stored in the SysTick Control and Status Register (SysTick_CTRL).
4. **Delay Completion:**
   - Exit the loop when the timer status indicates that the SysTick timer has counted down to zero, signifying that the desired delay has elapsed.
5. **Optional Interrupt (if needed):**
   - If the microcontroller supports SysTick interrupts, you can configure an interrupt handler to execute when the SysTick timer reaches zero. This can be useful for multitasking or handling other tasks during the delay.

2. Difference between i2c and spi ?

ANS:-

Number of Wires:

- I2C uses a two-wire bus (SDA and SCL) for communication, making it simpler in terms of wiring and connections.
- SPI typically requires at least four wires (MISO, MOSI, SCLK, and SS/CS), though it can involve more lines for additional devices, making it more complex in terms of wiring.

2. **Topology:**
    - I2C supports a multi-master, multi-slave topology, allowing multiple master devices to communicate with multiple slave devices on the same bus.
    - SPI typically uses a master-slave topology, where one master device communicates with one or more slave devices, but direct communication between slave devices is less common.
3. **Clocking Mechanism:**
    - I2C is a synchronous protocol but uses a clock-stretching mechanism, allowing a slave to slow down the clock if it needs more time to process data.
    - SPI is a fully synchronous protocol, where the clock signal (SCLK) is shared among devices, ensuring precise and consistent timing for data transfer.

3.what is ADC?and explain flow its works.
ANS:-
ADC stands for Analog-to-Digital Converter. It is a crucial component in microcontrollers and other electronic devices that allows them to convert analog signals, such as voltage or current, into digital values that can be processed and manipulated by the digital circuits.

explanation of how ADC works in three steps:

1. **Sampling:**
    - The ADC begins by sampling an analog signal at a specific point in time.
    - It captures the instantaneous value of the analog signal, which may represent a continuous waveform or a voltage level from a sensor, for example.
2. **Quantization:**
    - After sampling, the ADC quantizes the sampled analog voltage into a discrete digital value.
    - This quantization process divides the analog voltage range into discrete "bins" or steps.
    - The ADC assigns a digital value (typically binary) to the sampled voltage based on which bin or step it falls into.

3. **Conversion:**
    - The quantized digital value is then output by the ADC as a binary number, typically represented in binary code, such as a binary or hexadecimal number.
    - The digital value can now be processed, stored, or transmitted by the microcontroller or digital circuitry for various applications, including data analysis, control systems, or display.

4.How do you communicate with a slave device using I2C?
ANS:-
To communicate with a slave device using the I2C (Inter-Integrated Circuit) protocol, you typically follow these three steps:

1. **Address the Slave:**
    - Start the I2C communication by sending the 7-bit address of the slave device you want to communicate with. The address should match the slave's configured I2C address.
    - Specify whether you want to read from or write to the slave by setting the R/W (Read/Write) bit.
2. **Data Transfer:**
    - For writing data to the slave, send the data bytes that you want to transmit.
    - For reading data from the slave, initiate the read operation and receive the data bytes transmitted by the slave.
3. **End the Communication:**
    - To conclude the communication, send a stop condition to release the I2C bus.
    - The slave device acknowledges the communication as it receives data bytes.

5.How do you send a string of characters over the UART.
ANS:-
To send a string of characters over UART (Universal Asynchronous Receiver-Transmitter), you can follow these three steps:

1. **Initialize the UART:** Configure the UART module of your microcontroller with the desired settings, including the baud rate, data bits, stop bits, and parity settings. This initialization is typically done in your code's setup or initialization section.
2. **Transmit the String:** Use a function or loop to transmit each character of the string one by one. This can be achieved using a loop that iterates through the characters in the string and sends them individually to the UART transmit buffer.
3. **End of Transmission:** After sending all the characters in the string, you can optionally send a special character, like a newline ('\n') or carriage return ('\r'), to signify the end of the string. This step is particularly useful if the receiving end expects a specific delimiter to identify the end of a message.

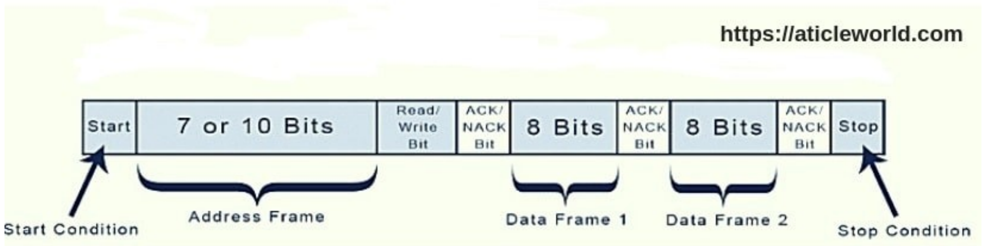
6 .Explain the operation and frame of I2C protocol.
ANS:-
I2C is a chip to chip communication protocol. In I2C, communication is always started by the master. When the master wants to communicate with slave then he asserts a start bit followed by the slave address with read/write bit.
After asserting the start bit, all slave comes in the attentive mode. If the transmitted address match with any of the slave on the bus then an ACKNOWLEDGEMENT (ACK) bit is sent by the slave to the master.

After getting the ACK bit, master starts the communication. If there is no slave whose address match with the transmitted address then master received a NOT-ACKNOWLEDGEMENT (NACK) bit, in that situation either master assert the stop bit to stop the communication or assert a repeated start bit on the line for new communication.

When we send or receive the bytes in i2c, we always get a NACK bit or ACK bit after each byte of the data is transferred during the communication.

In I2C, one bit is always transmitted on every clock. A byte which is transmitted in I2C could be an address of the device, the address of register or data which is written to or read from the slave.

In I2C, SDA line is always stable during the high clock phase except for the start condition, stop condition and repeated start condition. The SDA line only changes their state during the low clock phase.



I2C FRAME

7.Explain the purpose of a start bit and a stop bit in UART communication?

Ans:-

Start Bit (1st Mark):

- **Synchronization:** The start bit signals the beginning of a data byte transmission and serves as a synchronization point between the sender (transmitter) and receiver.
- **Wake-Up Signal:** It wakes up the receiver from an idle state, allowing it to prepare for data reception.
- **Frame Timing:** The duration of the start bit establishes the timing for the entire data frame, helping the receiver sample subsequent bits at the correct intervals.

2. **Stop Bit(s) (1 or More Marks):**

- **Signal End of Data:** The stop bit(s) indicate the end of the data byte and frame. They notify the receiver that there are no more data bits to be received.
- **Idle State:** After the stop bit(s), the UART line returns to an idle state, where it remains until the next start bit arrives. This idle state helps maintain the line's stability.
- **Error Detection:** Stop bits can help in detecting framing errors. If the receiver expects a specific number of stop bits but receives a different number, it can detect a potential error in the received data frame.

8.Explain the purpose of an address byte in I2C communication?

Ans:-

Device Selection:

- The address byte is used to select a specific slave device on the I2C bus. Each slave device connected to the bus is assigned a unique 7-bit or 10-bit address.
- When a master device initiates communication, it sends the address byte to specify the target slave device with which it wants to communicate.

2. **Multiple Device Support:**

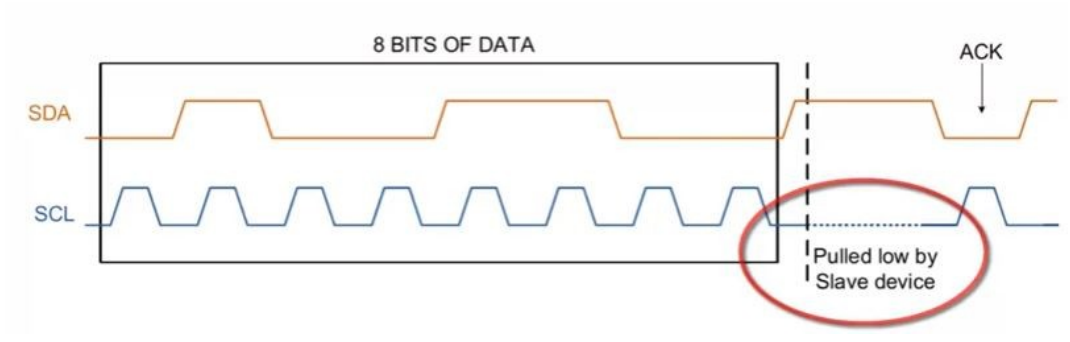
- I2C supports multiple slave devices on the same bus, and the address byte allows the master to communicate with any of these devices individually.
- The master can send a sequence of address bytes to access different slave devices on the bus during the same communication session.

3. **Read or Write Indication:**
    - In addition to selecting the slave device, the address byte also indicates whether the master intends to read from or write to the selected slave device.
    - The least significant bit (LSB) of the address byte is used to signal the direction of data transfer, with '0' indicating a write operation and '1' indicating a read operation.

9.Explain I2C clock stretching?

ANS:-

In I2c, communication can be paused by the clock stretching to holding the SCL line low and it cannot continue until the SCL line released high again.



In I2C, slave able to receive a byte of data on the fast rate but sometimes slave takes more time in processing the received bytes in that situation slave pull the SCL line to pause the transaction and after the processing of the received bytes, it again released the SCL line high again to resume the communication.

10.Explain the role of the shift register in Master and Slave devices in SPI?

Ans:-

In SPI communication, shift registers are employed for data transfer. When the master device writes data to the slave (using the MOSI bus), it simultaneously receives dummy data on the MISO bus to maintain synchronization. Conversely, during reads from the slave, the master sends dummy data on MOSI while receiving actual data on MISO, ensuring consistent clocking and bidirectional communication. This "dummy read" and "dummy write" approach is crucial for data integrity and synchronization in SPI.

Part-C 5*10=50

1.Write a Program using 1 led and 1 switch

conditions:

if we press the switch 1st time led will toggle 1 time

if we press the switch 2nd time led will toggle 2 times.

if we press the switch 3rd time led will toggle 3 times.

and this should be in a loop.

also print the number of times switch pressed.

Take onboard led and switch.

ANS:-

```
while (1)
  {
          uint8_t count=0;
          while(count<4)
          {
                  if(!(HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)))
                  {
                          count++;
                          if(count==1)
                          {
                                  printf("Switch pressed %d times\r\n",count);
                                  led_toggle(count);


                          }
                          else if(count==2)
                          {
                                  printf("Switch pressed %d times\r\n",count);
                                  led_toggle(count);


                          }
                          else if(count==3)
```

```
                {
                        printf("Switch pressed %d times\r\n",count);
                        led_toggle(count);


                }
        }
        else
        {
                printf("Switch not pressed\r\n");
                HAL_GPIO_WritePin(GPIOA,GPIO_PIN_5,GPIO_PIN_RESET);
                HAL_Delay(100);
        }
}
```

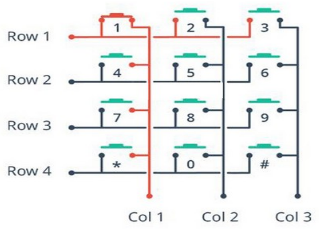
To create a program using an onboard LED and switch on an STM32 microcontroller that toggles the LED a certain number of times based on the number of times the switch is pressed, you can follow these initialization steps:

1. **Configure GPIO Pins:**
    - Initialize and configure the GPIO pins for the onboard LED and switch. You need to set the pin modes, pull-up or pull-down resistors, and other necessary settings.
2. **Configure External Interrupt (for the Switch):**
    - Enable the external interrupt for the GPIO pin connected to the switch.
    - Configure the interrupt trigger edge (e.g., rising edge, falling edge, or both) to detect the switch press.
3. **Initialize a Variable:**
    - Declare a variable (e.g., `pressCount`) to keep track of the number of times the switch is pressed. Initialize it to 0.
4. **Main Loop:**
    - In the main loop, monitor the value of `pressCount` to determine how many times the LED should toggle.
    - Toggle the LED accordingly.
5. **Interrupt Handler:**
    - Implement an interrupt handler function that gets executed when the switch is pressed.
    - In the interrupt handler, increment the `pressCount` variable.

2.Write a Program for Keypad Matrix 3x4

when I pressed any key in keypad it just print on

which line that key is present.



Initialize GPIO Pins:

- Configure the GPIO pins for the keypad rows and columns. Set the row pins as input pins with pull-up or pull-down resistors, and the column pins as output pins.

2. **Configure EXTI (External Interrupt) for Row Pins:**
   - Since the keypad rows are inputs, configure external interrupts on these pins. You should configure interrupt trigger edges (e.g., rising or falling edge) and enable the EXTI interrupt for each row.
3. **Enable GPIO Clocks:**
   - Enable the clocks for the GPIO ports used for the keypad rows and columns.
4. **Initialize NVIC (Nested Vector Interrupt Controller):**
   - Set the NVIC priority group and enable the EXTI interrupt in the NVIC.
5. **Main Loop:**
   - In the main loop, you'll continuously scan the keypad to detect key presses.
   - When a key press is detected, print the corresponding row or line to indicate where the key is pressed.

3.Suppose you have 2 stm32 board then write a program to send a string from one board and received it on another board.

Ans:-

Transmitter Board (STM32 A):

1. **Initialize STM32 A:**
   - Set up STM32 A using your development environment (e.g., STM32CubeIDE) for bare-metal programming.
   - Create a new project and configure the necessary settings for your specific board.
2. **Initialize UART for Transmitting:**
   - Configure the UART peripheral to transmit data.

- Set the UART communication parameters, including the baud rate, data bits, stop bits, and parity.
- Enable the UART transmitter.

3. **Send a String:**
    - Prepare the string you want to send.
    - Use UART transmission functions or write directly to the UART data register to send the string over UART to STM32 B.

Receiver Board (STM32 B):

1. **Initialize STM32 B:**
    - Set up STM32 B using your development environment similar to STM32 A for bare-metal programming.
    - Create a new project and configure the necessary settings for your specific board.
2. **Initialize UART for Receiving:**
    - Configure the UART peripheral to receive data.
    - Set the UART communication parameters (baud rate, data bits, stop bits, parity) to match STM32 A's settings.
3. **Receive and Process the String:**
    - Create a buffer to store the received string.
    - Use UART reception functions or read from the UART data register to receive data from STM32 A and store it in the buffer.
    - Process or display the received string as needed.

4.write a program to generate timer base interrupt when I pressed an switch then after each 500 ms led should toggle and after pressing the switch again led should stop toggling.

ANS:-

```
 /*Enable clock access to GPIOA*/
RCC->AHB1ENR |=GPIOAEN;


/*Set PA5 mode to alternate function*/
GPIOA->MODER &=~(1U<<10);
GPIOA->MODER |=(1U<<11);
```

```
/*Set PA5 alternate function type to TIM2_CH1 (AF01)*/
GPIOA->AFR[0] |=AFR5_TIM;

/*Enable clock access to tim2*/
RCC->APB1ENR |=TIM2EN;

/*Set prescaler value*/
TIM2->PSC =  1600 - 1 ;  //  16 000 000 / 1 600 = 10 000
/*Set auto-reload value*/
TIM2->ARR =  10000 - 1;  // 10 000 / 10 000 = 1

/*Set output compare toggle mode ccmr1*/
 TIM2->CCMR1 |=0x30;
/*Enable tim2 ch1 in compare mode ccer*/
 TIM2->CCER |=CCER_CC1E;



/*Clear counter*/
TIM2->CNT = 0;
/*Enable timer*/
TIM2->CR1 = CR1_CEN;
```

1.Configure GPIO:

Configure a GPIO pin for the LED and another for the switch. Ensure you configure the GPIO pin for the switch as an input with a pull-up or pull-down resistor.

2. Initialize TIM (Timer):

Initialize a timer to generate interrupts at a 500 ms interval. You can use TIM2, TIM3, or another suitable timer depending on your specific STM32 model.

Configure the timer to generate an interrupt when the timer's value reaches a certain value that corresponds to 500 ms.

3. Initialize NVIC (Nested Vector Interrupt Controller):

Set up the NVIC to handle timer interrupts. Enable the specific timer interrupt in the NVIC and set its priority.

4. Create a State Variable:

Create a state variable to track whether the LED should toggle or not. Initialize this variable to indicate that the LED should not toggle initially.

5. Timer Interrupt Handler:

Write an interrupt handler for the timer that toggles the LED based on the state variable. If the state variable indicates that the LED should toggle, then toggle the LED; otherwise, keep it turned off.

6. Switch Press Detection:

Poll the switch in your main loop or use external interrupts to detect when the switch is pressed. When the switch is pressed, change the state variable to indicate that the LED should toggle.

5.Write a program to show how to interface any ADC based module with stm32 Board.

ANS:-

```
/**Configure the ADC GPIO pin **/

/*Enable clock access to GPIOA*/

RCC->AHB1ENR |= GPIOAEN;

        //Set the mode of PA1 to analog/

        GPIOA->MODER |= (3U<<2);


        /**Configure the ADC module**/

        /*Enable clock access to ADC */

        RCC->APB2ENR |=ADC1EN;

        //Conversion sequence start/

        ADC1->SQR3 |=ADC_CH1;

        //Conversion sequence length/

        ADC1->SQR1 |= ADC_SEQ_LEN_1;


        //Enable ADC module/

        ADC1->CR2 |= CR2_AD0N;

}
```

```
void start_converstion(void)
{
        //Continuous conversion//
        ADC1->CR2 |=CR2_COUNT;
        //Start adc conversion/
        ADC1->CR2 |= CR2_SWSTART;
}
int adc_read(void)
{
        //Wait for conversion to be complete/
        while(!(ADC1->SR & SR_EOC)){}

        //Read converted result/
        return (ADC1->DR);
}
```

1. Hardware Setup:

- Connect the ADC module to your STM32 board. Ensure you connect power (VCC and GND), analog input signals, and any required control signals (e.g., clock, chip select).

2. Initialize STM32 Peripherals:

- Set up your STM32 microcontroller project in your development environment (e.g., STM32CubeIDE).
- Initialize and configure the GPIO pins for the ADC module's connections.

3. Configure ADC Peripheral:

- Initialize and configure the ADC peripheral on the STM32.
- Set the ADC resolution, sampling time, and other settings based on your module's specifications.

4. Start ADC Conversion:

- Start the ADC conversion using appropriate library functions or register settings.
- You can trigger conversions manually or configure the ADC to trigger conversions automatically based on a timer or other events.

5. Read ADC Values:

- After starting the conversion, read the ADC values from the ADC data register.

6. Process the ADC Data:

- Depending on your application, you may need to process the ADC data. This could involve scaling, filtering, or converting it to engineering units.

7. Display or Use the Data:

- Use the processed ADC data in your application. You can display it on an LCD, transmit it via UART, or use it for control purposes.

8. Error Handling:

- Implement error handling routines to handle situations like overflows, underflows, or other potential issues with the ADC.

9. Calibration (Optional):

- If necessary, perform ADC calibration to improve the accuracy of your ADC measurements.

6.Write all the steps to interface any SPI based Module with stm32 MCU.

ANS:-

```
void SPI_Init(void)
{
	#define AF5 0x05
	RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN; //enable clock forn gpio a
	RCC->APB2ENR|=RCC_APB2ENR_SPI1EN; //enable clock for spi1
	GPIOA->MODER|=GPIO_MODER_MODE5_1|GPIO_MODER_MODE6_1|
GPIO_MODER_MODE7_1;
	GPIOA->MODER&=~(GPIO_MODER_MODE5_0|GPIO_MODER_MODE6_0|
GPIO_MODER_MODE7_0);
	GPIOA->OSPEEDR|=GPIO_OSPEEDER_OSPEEDR5|
GPIO_OSPEEDER_OSPEEDR6|GPIO_OSPEEDER_OSPEEDR7;


	GPIOA->AFR[0]|=(AF5<<20)|(AF5<<24)|(AF5<<28);
	SPI1->CR2=0;
	SPI1->CR1=SPI_CR1_SSM|SPI_CR1_MSTR|SPI_CR1_BR_2|SPI_CR1_SSI|
SPI_CR1_SPE;
}



int8_t SPI_Transmit(uint8_t *data, uint32_t size)
{



	uint32_t i		=0;
	uint8_t  temp	=0;
	uint32_t start=millis();
	temp =SPI1->DR;
	temp=SPI1->SR;
	while(i<size)
		{
		while(!((SPI1->SR)&SPI_SR_TXE)){if(millis()-start>1000){
			printf("TXE timed out\r\n");
			return -1;}} // wait to transmision buffer to be emplty
		SPI1->DR= data[i];
		while(!(SPI1->SR&SPI_SR_BSY)){if(millis()-start>1000){printf("BSY timed out\r\n");return -1;}}
		i++;
		}
while(!((SPI1->SR)&SPI_SR_TXE)){if(millis()-start>1000){printf("TXE2 timedout\r\n");return -1;}}
while((SPI1->SR)&SPI_SR_BSY){if(millis()-start>1000){printf("BSY2 timed out\r\n"); return -1;}}
temp =SPI1->DR;
temp=SPI1->SR;
return 0;
}
```

```
int8_t SPI_Receive(uint8_t *data, uint32_t size)
{
while(size)
            {
      uint32_t start=millis();
            SPI1->DR=0;
            while(!(SPI1->SR&SPI_SR_RXNE)){if(millis()-start>200){return -1;}}
            *data++=(SPI1->DR);
                  size--;
            }
return 0;
}
```

1. Hardware Setup:

- Connect the SPI-based module to your STM32 board. Ensure you connect power (VCC and GND), SPI data lines (SCK, MISO, MOSI), and any required control signals (e.g., chip select or slave select).
- Ensure that the voltage levels of the module match the STM32 board's voltage levels (e.g., 3.3V or 5V).

2. Initialize STM32 Peripherals:

- Set up your STM32 microcontroller project in your development environment (e.g., STM32CubeIDE).
- Initialize and configure the GPIO pins for the SPI communication lines (SCK, MISO, MOSI).

3. Configure SPI Peripheral:

- Initialize and configure the SPI peripheral on the STM32.
- Set the SPI communication parameters, including clock polarity, clock phase, data order (MSB or LSB first), and data frame format.
- Configure the SPI communication speed (baud rate).
- Enable the SPI peripheral.

4. Chip/Slave Select (CS/SS):

- Configure the GPIO pin for chip select (CS or SS) based on your module's requirements.
- Implement logic to control the CS/SS line to select or deselect the module before and after communication.

5. Data Transfer:

- To send data to the module, write data to the SPI data register.
- To receive data from the module, read data from the SPI data register.

6. Error Handling:

- Implement error handling routines to handle potential issues with SPI communication, such as data overrun or underrun.

7.Write all the steps to interface any I2C based Module with STM32 MCU.

ANS:-

1. Hardware Setup:

- Connect the I2C-based module to your STM32 board. Ensure you connect power (VCC and GND), the I2C data lines (SDA and SCL), and any required control signals.
- Ensure that the voltage levels of the module match the STM32 board's voltage levels (e.g., 3.3V or 5V).

2. Initialize STM32 Peripherals:

- Set up your STM32 microcontroller project in your development environment (e.g., STM32CubeIDE).
- Initialize and configure the GPIO pins for the I2C data lines (SDA and SCL).

3. Configure I2C Peripheral:

- Initialize and configure the I2C peripheral on the STM32.
- Set the I2C communication parameters, including the I2C clock speed, addressing mode (7-bit or 10-bit), and whether to enable stretching of the clock signal.
- Enable the I2C peripheral.

4. Device Addressing:

- Determine the 7-bit or 10-bit address of the I2C device/module you want to communicate with. Refer to the module's datasheet for this information.

5. Communication:

- To communicate with the I2C device, use library functions or write to/read from the I2C data register.
- Implement logic to initiate read or write operations based on the device's data sheet and communication protocol.