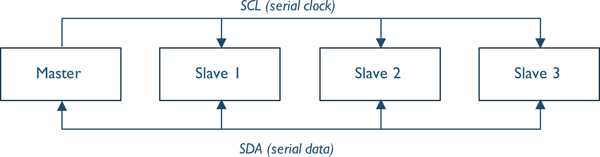
**I2C Communication based LCD Interface using an 8-bit I/O Expander**

**1. Aim:**

To display the string data using PCF8574A LCD and 8-bit I/O expander with help of 2-wire I2C interface.

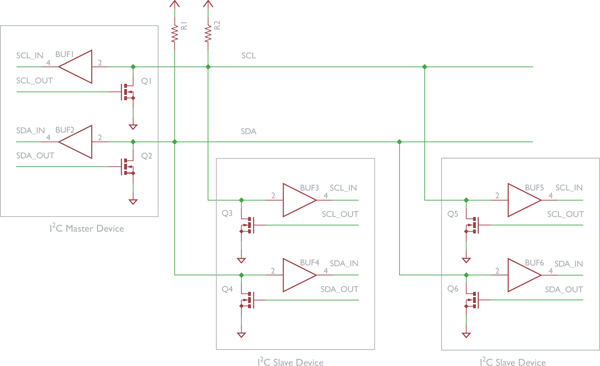
**2. Introduction:**

I2C is a synchronous protocol that uses a serial data (SDA) signal and serial clock (SCL) signal. I2C is a master/slave protocol. The master initiates all communications, and slave devices transmit only when the master allows them to. A major feature of the protocol is device addressing: each message includes device addressing information, and each device on the bus has a unique address. Only the addressed device will respond to a message. This allows a system to be built that shares the SDA/SCL bus as shown in the figure below without using additional control signals (such as slave selects for SPI).



The circuits driving the SDA and SCL signals are shown in Figure and are designed so that multiple devices can drive the signal simultaneously without damage. Each device’s drive circuit consists of a transistor connected between the I2C bus signal (SDA or SCL) and ground. A separate pull-up resistor is connected between the signal and VDD. For the master to send a 0 on SDA, the transistor Q2 is turned on, pulling SDA to ground. To send a one, the transistor Q2 is turned off, allowing the resistor R1 to pull SDA up to VDD. If two devices attempt to transmit different data simultaneously, the SDA signal will be a zero.

I2C supports a range of communication speeds: 100 kbit/s (standard), 400 kbit/s (full speed), 1 Mbit/s (fast), 3.2 Mbit/s (high speed). The maximum communication speed for a given system implementation is limited by how quickly the pull-up resistor can pull SCL or SDA up to VCC. This depends on the capacitance of the SCL or SDA signal, which is affected by the number of devices and the bus length. The maximum speed for a given device will be listed in its data sheet or reference manual.



**3.1 Message Format:**

There are several types of I2C message, with the basic format shown in Figure. Each message contains several fields:

•The start condition indicates the start of a message.

•The slave device address identifies the target of the communication.

•The read/write bit indicates whether the following data is to be read from the slave or written to it.

•The acknowledgment bit has two uses: to indicate if the addressed slave is present, and whether more data will be read. These are explained later in this section.

•One or more data bytes. For some I2C slave devices, the first data byte will be interpreted as a register address.

•A stop condition indicates the end of a message.

•An optional repeated start condition is used in some types of messages.

C:\Users\Admin\AppData\Local\Temp\fig8_28.jpg

Communication operations are structured as sequences of conditions (e.g. start, stop) and data transfers (one byte and one acknowledgment bit).

**3.2 Device Addressing:**

The master uses the device address to select a particular slave device on the bus. There are two addressing modes, one with 7-bit addresses and another with 10-bit addresses. For simplicity we will just discuss the 7-bit mode.

The first byte in the message has two parts, as shown in Figure. The device address is held in the upper seven bits, and the R/W bit is the LSB. This bit indicates whether the master will read from the slave (one) or write to it (zero). In practice, this byte is formed by shifting the slave address left by one bit and then adding the R/W bit.

1. **Sequence for a master to write two data bytes to a slave device**

Text indicates transmitting device. Blank indicates listening device.

C:\Users\Admin\AppData\Local\Temp\fig8_29.jpg

•The master first sends the start condition, the slave device address, and a write command (zero). If the addressed slave is present, it will assert the ACK bit. If the ACK bit is not asserted, then the master will terminate the message with a stop condition.

•The master sends the first byte of data.

•The slave sends an ACK to indicate it has been received.

•The master sends the second byte of data.

•The slave sends an ACK to indicate it has been received.

•The master sends the stop condition, indicating to all slaves that the message has completed.

1. **Sequence for a master to read two data bytes from a slave device**

**C:\Users\Admin\AppData\Local\Temp\fig8_30.jpg**

* The master first sends the start condition, the slave device address, and a read command (one). If the addressed slave is present, it will assert the ACK bit. If the ACK bit is not asserted, then the master will terminate the message with a stop condition.

•The master clocks the first byte of data out of the slave.

•The master sends an ACK to indicate it will read more data.

•The master clocks the second byte of data out of the slave.

•The master sends a NACK (one) to indicate that it does not want to read any more data in this message.

•The master sends the stop condition, indicating to all slaves that the message has completed.

**3.3 Register Addressing:**

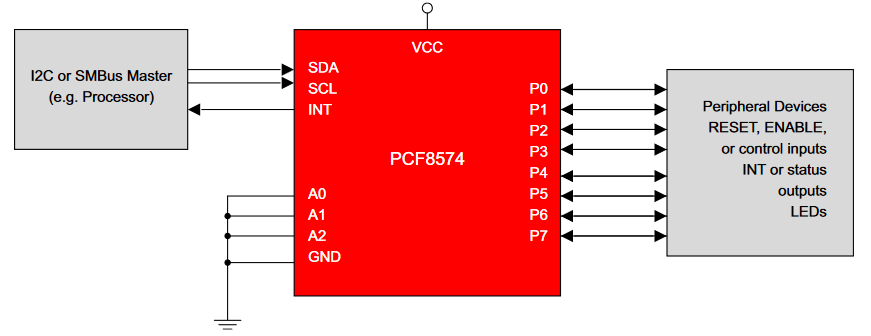
I2C also supports register addressing, in which each device is structured as a series of addressable registers that can be read or written. This standardizes information organization and simplifies system development. The first data byte is interpreted by the slave device as a register address.

**Sequences of writing and reading to a register in a device.**

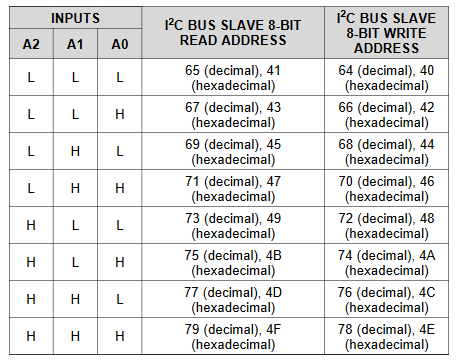
**C:\Users\Admin\AppData\Local\Temp\fig8_31.jpg**

**C:\Users\Admin\AppData\Local\Temp\fig8_32.jpg**

**3.4 I/O Expander Addressing:**

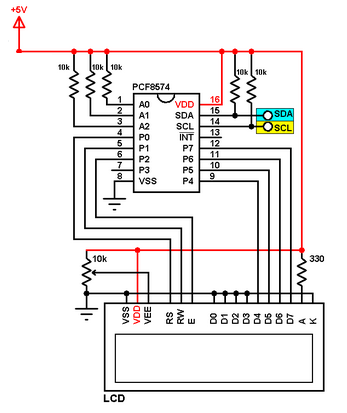


As shown in the figure below, based on the A0, A1, and A2 values device address is determined. In this experiment.



The address used for this experiment is 0x4E for Write, and 0x4F for Read. Since the LCD display is used in the experiment, only the write operation used throughout the experiment.

**4. Hardware Schematic Diagram:**



**5. Program to Send a String “VIT” from a Microcontroller to PCF8574A LCD and 8-bit I/O Expander:**

#include "stm32f4xx.h" // Include header file for the board

unsigned int temp; // Temporary variable to clear the bit

char Mes[10]="VIT";

// Pre initializing the functions

void configure\_GPIOB(void);

void LCD\_init(void);

void Transmit(char\*);

void msDelay(uint32\_t msTime);

void configure\_I2C(void);

void I2C\_start(void);

void I2C\_send(uint8\_t data);

void I2C\_address(void);

void I2C\_stop(void);

void I2C\_write4(uint8\_t data);

void I2C\_write8(uint8\_t data);

void I2C\_write4C(uint8\_t data);

void I2C\_write8C(uint8\_t data);

int main ()

{

configure\_GPIOB(); // To configure the GPIO pins PB6 and PB7

configure\_I2C(); // To configure the I2C1 register

msDelay(1);

LCD\_init();

//lower nibble

Transmit(Mes);

while (1)

{}

}

void configure\_GPIOB(void){ //to configure the GPIO pins PB6 and PB7

//SDA - PB7, SCL - PB6

RCC -> AHB1ENR |=(1UL<<1); //Enable clock for port B

GPIOB -> MODER |=(2UL<<(6\*2)); //PB6 to alternate function pin "10"

GPIOB -> MODER |=(2UL<<(7\*2)); //PB7 to alternate function pin "10"

GPIOB -> PUPDR |=(0x5UL<<(6\*2)); //set PB6 and PB7 as pull up pins "01"

GPIOB -> OTYPER |=(0x3UL<<6); //Set PB6 and PB7 as open drain "1"

GPIOB -> OSPEEDR |=(0xAUL<<(6\*2)); //Set PB6 and PB7 as high speed "10"

GPIOB -> AFR[0] |= (0x44<<(6\*4)); //Set PB6 and PB7 to alternate function 4 "0100"

}

void msDelay(uint32\_t msTime){ //function for software delay

// "For loop" takes 4 clock cycles to get executed. Clock frequency is 16MHz

// 16MHz/4=4MHz. If we want 1000ms (1second) delay, 4MHz/1000=4000, so we have to multiply by 4000 to get a delay of 1s

for (uint32\_t i=0;i<msTime\*4000;i++){

\_\_NOP();

}

}

void configure\_I2C(void){ // To configure the I2C registers

RCC -> APB1ENR |=(1UL<<21); // Enable I2C clock

I2C1 -> CR1 |= (1UL<<15); // Reset I2C

I2C1 -> CR1 &= ~(1UL<<15); // set I2C

I2C1 -> CR2 |=(16UL<<0); // Set peripheral clock at 16MHz

I2C1 -> OAR1 |=(1UL<<14); // Should be set high

I2C1 -> CCR |=(0x50UL<<0); // Set SCL as 100KHz

I2C1 -> TRISE |=(17UL<<0); // Configure maximum rise time

I2C1 -> CR1 |= (1UL<<0); // Enable I2C

}

void I2C\_start(void){ // To send the I2C start bit

I2C1 -> CR1 |= (1<<10); // Enable the ACK Bit

I2C1 -> CR1 |= (1<<8); // Send the start bit

while (!(I2C1->SR1 & (1<<0))); // Wait for SB bit to set

}

void I2C\_address(void){ // To send the I2C address

I2C1 -> DR = 0x4E; // Send the slave address // change according to the device

while (!(I2C1->SR1 & (1<<1))); // Wait for ADDR bit to set

temp = ((I2C1->SR1) | (I2C1->SR2)); // Read SR1 and SR2 to clear the ADDR bit

}

void I2C\_send(uint8\_t data){ // To send data to the I2C device

while (!(I2C1->SR1 & (1<<7))); // Wait till TX buffer is empty

I2C1 -> DR = data; // Write data to I2C slave

while (!(I2C1->SR1 & (1<<2))); // Wait till Byte transfer is completed

}

void I2C\_stop(void){ // To send the I2C stop bit

I2C1 -> CR1 |= (1<<9); // Stop I2C

}

void I2C\_write4(uint8\_t data){ // To send upper four bit for commands

uint8\_t d=0; // Reset value for d

d |= (data & 0xF0); // To select only the upper nibble

d |= 1<<2; // To set the enable bit

d |= 1<<3; // To set the backlight

I2C\_start(); // Send the start bit

I2C\_address(); // Send the I2C address

I2C\_send(d); // Send the data

msDelay(1); // Wait for 1ms

d &= ~(1<<2); // Set the enable bit low

I2C\_send(d); // Send the same data but without enable bit to latch the data

I2C\_stop(); // Send the stop bit

}

void I2C\_write8(uint8\_t data){ // To send lower four bit for commands

uint8\_t d=0;

d |= ((data & 0x0F)<<4); // To select only the lower nibble

d |= 1<<2;

d |= 1<<3;

I2C\_start();

I2C\_address();

I2C\_send(d);

msDelay(1);

d &= ~(1<<2);

I2C\_send(d);

I2C\_stop();

}

void I2C\_write4C(uint8\_t d){ // To send upper four bits for data

d |= 1<<2;

d |= 1<<0; // To select the data register not the command

d |= 1<<3;

I2C\_start();

I2C\_address();

I2C\_send(d);

msDelay(1);

d &= ~(1<<2);

I2C\_send(d);

I2C\_stop();

}

void LCD\_init(void)

{

// Initial commands to set up the LCD in 4 bit mode

I2C\_write4(0x30); //to clear the screen

msDelay(1);

I2C\_write4(0x30); //to clear the screen

msDelay(1);

I2C\_write4(0x30); //to clear the screen

msDelay(1);

I2C\_write4(0x20); //to turn on the screen

msDelay(1);

// Set LCD to 4 bit mode, 2 lines, 5x8 font

I2C\_write4(0x28); //upper nibble

msDelay(1);

I2C\_write8(0x28); //lower nibble

msDelay(1);

// Turn the display on with blinking cursor

I2C\_write4(0x0C); //upper nibble

msDelay(1);

I2C\_write8(0x0C); //lower nibble

msDelay(1);

// Clear the contents on the screen

I2C\_write4(0x01); //upper nibble

msDelay(37);

I2C\_write8(0x01); //lower nibble

msDelay(1);

// Set LCD to write from left to right

I2C\_write4(0x06); //upper nibble

msDelay(1);

I2C\_write8(0x06);

}

void Transmit(char \* add)

{

char data,d\_H=0,d\_L=0;

for(int i=0;i<3;i++)

{

data=add[i];

d\_H= (data & 0xF0);

I2C\_write4C(d\_H);

d\_L= ((data & 0x0F)<<4);

I2C\_write4C(d\_L); //lower nibble

msDelay(1);

}

}

**6. Conclusion:**

In conclusion, the I2C communication protocol has been properly used to send strings of words to the LCD display from the microcontroller.