Objective

The objective of this report is to present an in-depth examination of interfacing various sensors and external devices with a Raspberry Pi (RPi) using Python programming. The report emphasizes the practical integration of hardware components and demonstrates how the RPi's General Purpose Input/Output (GPIO) pins can be utilized to create functional, interactive systems. This is achieved by illustrating real-world applications where hardware components communicate seamlessly with the software.

A core focus of this experiment is to provide detailed examples of how components such as push buttons, resistors (specifically 220-ohm), LEDs, DHT11/DHT22 temperature and humidity sensors, I2C OLED displays, gas sensors, and ultrasonic sensors can be connected to the RPi. Through these examples, the readers will gain valuable insight into setting up, programming, and troubleshooting RPi-based systems.

Moreover, this report delves into more advanced topics, including the visualization of sensor data, integration with cloud services, and real-time environmental monitoring [18]. These sections aim to showcase the extended possibilities of using the RPi not only as a standalone device but also as a versatile tool for Internet of Things (IoT) applications and automated systems.

Additionally, this document serves as a comprehensive resource for individuals embarking on projects that require RPi sensor interfacing. By highlighting the interplay between hardware and software, it provides readers with a thorough understanding of how these components interact, thereby equipping them with the foundational knowledge required for a wide range of IoT projects and sensor-based applications. The practical examples included will offer readers the opportunity to engage with hands-on scenarios that illustrate the concepts discussed, thereby fostering a deeper understanding of the subject matter.

Component List:

Hardware:

1. LED
2. Cable
3. Gas sensor
4. Raspberry Pi
5. Push buttons.
6. 220-ohm Resistor
7. I2C OLED displays
8. Ultrasonic sensors.
9. DHT11/DHT22 sensors

Software:

Thonny

Theory

Gas Sensor:

Fig: MQ-5 Gas Sensor

The MQ-5 gas sensor has been utilized for this project due to its widespread use in detecting various flammable gases. It is particularly favored in safety applications where the identification of hazardous gas concentrations is critical. This sensor exhibits notable sensitivity to gases such as butane, propane, and methane, making it suitable for detecting liquefied petroleum gas (LPG), natural gas, and coal gas [21]. These characteristics render it ideal for use in domestic gas leakage detection systems, industrial gas detectors, and portable gas detection devices.

The core material of the MQ-5 sensor is tin dioxide (SnO2), which exhibits low conductivity in environments with clean air. As the concentration of flammable gases increases, the sensor's conductivity rises correspondingly, allowing for accurate detection. Among its notable features are the rapid response and recovery times, the ability to adjust sensitivity through a potentiometer, and a signal output indicator, which facilitates easy interpretation of the gas levels present.

In terms of power requirements, the MQ-5 operates on a power supply ranging between 2.5V and 5.0V. It is compatible with microcontrollers such as Arduino and Raspberry Pi, making it versatile for both digital and analog data output. The sensor module is typically equipped with a built-in potentiometer for sensitivity adjustment and includes LEDs for indicating power and activation statuses. Its affordability and reliability have made the MQ-5 gas sensor a practical choice for projects requiring gas monitoring, particularly in detecting potentially hazardous gas levels.

Raspberry Pi 4:

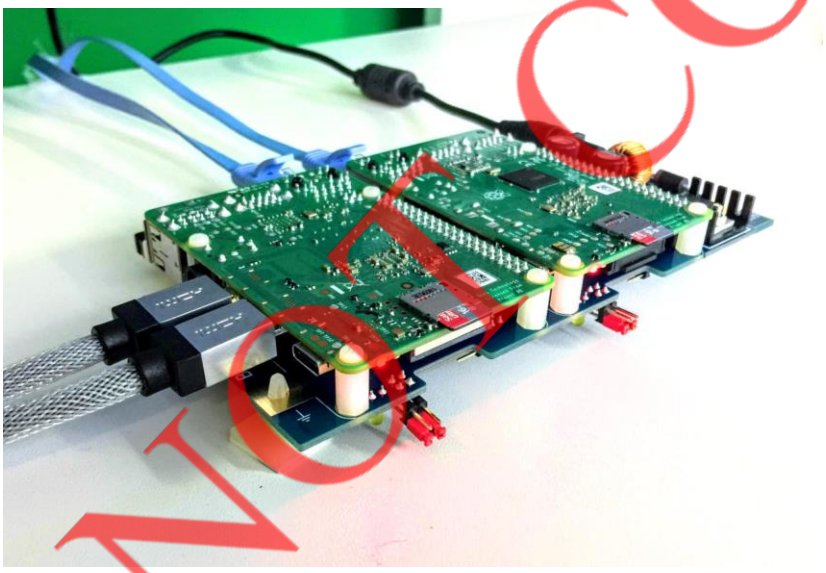

Fig: Raspberry Pi 4

In this project, the Raspberry Pi 4 has been selected for its compactness and affordability, with the Pi 4 Model B serving as the specific variant. It operates on a Broadcom BCM2711 System on Chip (SoC) and includes a 1.5 GHz or 1.8 GHz quad-core ARM Cortex-A72 CPU, depending on the revision **[1]**. This model offers significant performance improvements over earlier versions of Raspberry Pi, allowing it to handle more complex tasks efficiently.

The Raspberry Pi 4 comes with multiple memory options, ranging from 1GB to 8GB of LPDDR4 SDRAM, enabling smoother multitasking and faster execution of applications. Although it lacks built-in storage, it requires a microSD card for both the operating system and data storage. For connectivity, it features a gigabit Ethernet port for wired networking, dual-band Wi-Fi (supporting both 2.4 GHz and 5.0 GHz frequencies), and Bluetooth 5.0, allowing it to connect with various wireless devices. Additionally, it is equipped with USB

ports, two of which are USB 3.0 for high-speed data transfer and two USB 2.0 ports for backward compatibility with older devices.

The Raspberry Pi 4 is designed to support video output through dual micro-HDMI ports, each capable of delivering resolutions up to 4K at 60Hz. This feature is particularly advantageous for those seeking a dual-monitor setup. Furthermore, it includes a 40-pin GPIO header, which facilitates connection to external electronics and sensors. With H.265 hardware decoding for 4K video, the device is also suitable for media-centric applications.

Power is supplied through a 5V DC USB-C connector with a minimum current requirement of 3A. Its enhanced processing power, RAM capacity, and diverse connectivity options make the Raspberry Pi 4 highly versatile, capable of functioning as a low-cost desktop replacement, home server, or IoT (Internet of Things) device. It is suitable for a range of applications, including web browsing, document editing, media streaming, and hosting small-scale servers.

Ultrasonic Sensor:

Fig: Ultrasonic Sensor

Ultrasonic sensors are utilized in various fields to detect objects and measure distances by emitting high-frequency sound waves and calculating the time it takes for these waves to reflect. Operating at frequencies beyond the range of human hearing (typically above 20 kHz), they provide accurate distance measurements by determining the round-trip time of the sound waves. These sensors are employed in robotics for navigation and obstacle avoidance, in parking assistance systems for measuring the proximity of nearby objects, and in security systems for detecting intruders.

Moreover, ultrasonic sensors find applications in industrial level measurement systems for liquids or solids within containers and are integral to flow measurement systems for liquids and gases. In medical imaging, ultrasound technology is widely used for sonograms, offering non-invasive insights into the human body.

I2C OLED Displays:

Fig: I2C OLED Displays

I2C OLED displays are compact and energy-efficient screens that utilize the I2C communication protocol, allowing easy integration with microcontrollers like Raspberry Pi or Arduino **[16]**. Their small size, typically under 3 inches, makes them ideal for projects

where space is limited. These displays are noted for their low power consumption, high contrast, and wide viewing angles, ensuring clarity and readability even from different perspectives.

With a simple two-wire interface for data and clock signals, the I2C OLED display simplifies connections to microcontrollers, making it a preferred choice for applications that require displaying real-time information, such as sensor data, device statuses, or even diagnostic messages during development. The capability to display text, graphics, and icons enhances their usability for creating user interfaces in various embedded systems, such as thermometers or portable game consoles.

DHT22 Sensor:

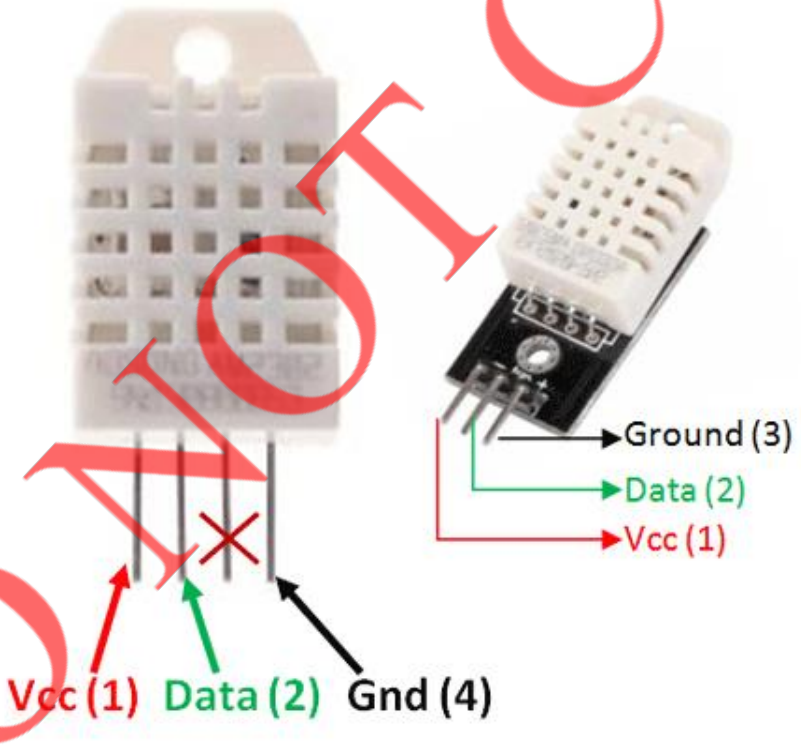

Fig: DHT22 sensor

The DHT22 sensor is a popular digital temperature and humidity sensor, commonly used in applications where monitoring environmental conditions is necessary [19]. This sensor is preferred for its ease of use, as it outputs digital signals rather than analog voltages, simplifying integration with microcontrollers such as the Raspberry Pi. With the ability to measure temperatures between -40°C and 80°C and humidity levels from 0% to 100% relative humidity (RH), the DHT22 offers a broad measurement range with decent accuracy, providing ±0.5°C for temperature and ±2% RH for humidity.

Applications of the DHT22 sensor include weather monitoring systems, greenhouse climate control, and smart home automation. It is also commonly employed in data logging projects, where long-term tracking of temperature and humidity is required for analysis. Compact in size, the DHT22 can be easily integrated into various setups, making it a versatile option for projects that require reliable temperature and humidity data.

Software:

Thonny:

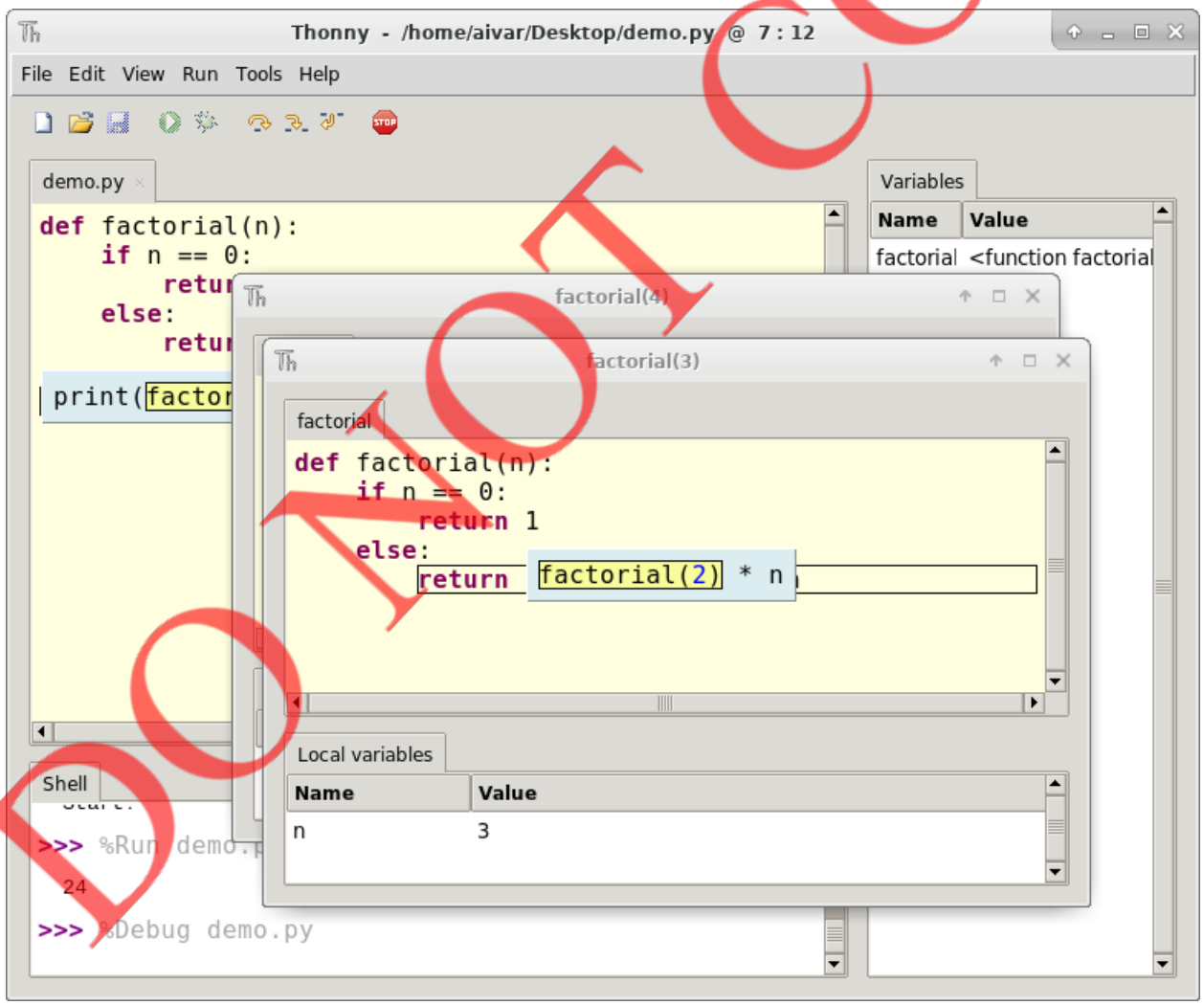

Fig: Thonny User Interface

Thonny is an Integrated Development Environment (IDE) for Python, particularly designed with beginners in mind, yet versatile enough to cater to more experienced programmers. Compatible with the Raspberry Pi OS, Thonny offers a simple and intuitive interface, making Python coding more accessible to users [17]. Its key features include code completion, syntax highlighting, and an integrated debugger, allowing users to efficiently write, run, and debug their Python scripts.

Thonny facilitates Python development by providing a seamless environment for executing scripts directly on the Raspberry Pi. Its user-friendly interface makes it ideal for smaller projects, while its advanced features ensure that it remains useful for more complex tasks. With the ability to save and run scripts with ease, Thonny proves to be a highly functional tool for both learning and developing Python-based applications.

Problem 1

Connect a push button and an LED to GPIO pins. Write a program where the LED turns on when the button is pressed and turns off when it's released.

Hardware:

Raspberry Pi: The Raspberry Pi functions as the primary computer board in this system. As a single-board computer, it contains key components such as the CPU, memory, and graphics processor on a single platform. This compact design allows it to run an operating system and interact with a wide range of peripherals, making it suitable for various electronic projects.

LED (Light Emitting Diode): An LED is a small electronic device that emits light when an electric current passes through it. The LED is polarized, with a longer leg indicating the positive (anode) side and a shorter leg for the negative (cathode) side. LEDs come in different colors and require the use of a resistor to regulate the current, preventing potential damage due to excessive current flow.

Breadboard: A breadboard is a tool designed for solderless prototyping [11]. It features rows of interconnected holes, enabling the easy connection of electronic components using jumper wires without the need for soldering. This simplifies the process of creating and modifying circuits during the development phase.

Push button: A push button is a simple mechanical switch that consists of two contacts [6]. When pressed, the contacts close, allowing current to pass through, and when released, the contacts open, interrupting the flow of current. Typically, a "normally open" push button means that the circuit remains open (disconnected) when the button is not pressed.

Jumper wires: These are thin, flexible conductors equipped with connectors at both ends. The male connector typically consists of a pin, while the female connector features a socket. These wires, available in various lengths and colors, are essential for establishing connections on the breadboard based on specific functionalities.

Resistor: A 220-ohm resistor is commonly used in conjunction with LEDs to limit the current passing through them. This passive electronic component reduces the risk of damaging the LED by controlling the flow of current. The 220-ohm value is specifically chosen to provide the appropriate current and brightness for standard LEDs while ensuring their safety during operation

Hardware Implementation:

The positive leg of the LED (the longer leg) is connected to GPIO pin 18 on the Raspberry Pi through a jumper wire. This connection allows the Raspberry Pi to control the LED by sending signals through this GPIO pin [2].

A 220-ohm resistor is connected to the negative leg (shorter leg) of the LED. The other end of the resistor is connected to the ground pin (GND) on the Raspberry Pi to complete the circuit and limit the current flowing through the LED, preventing damage.

For the push button, one leg is connected to GPIO pin 23 using a jumper wire. This pin will monitor the state of the push button (pressed or not pressed). The other leg of the push button is connected to a ground pin (GND) on the Raspberry Pi. The pull-up resistor setup in the code ensures that the button's state is read correctly by the Raspberry Pi [7].

This configuration ensures that the LED can be turned on and off based on the state of the push button, with the necessary components wired to the appropriate GPIO pins on the Raspberry Pi [12].

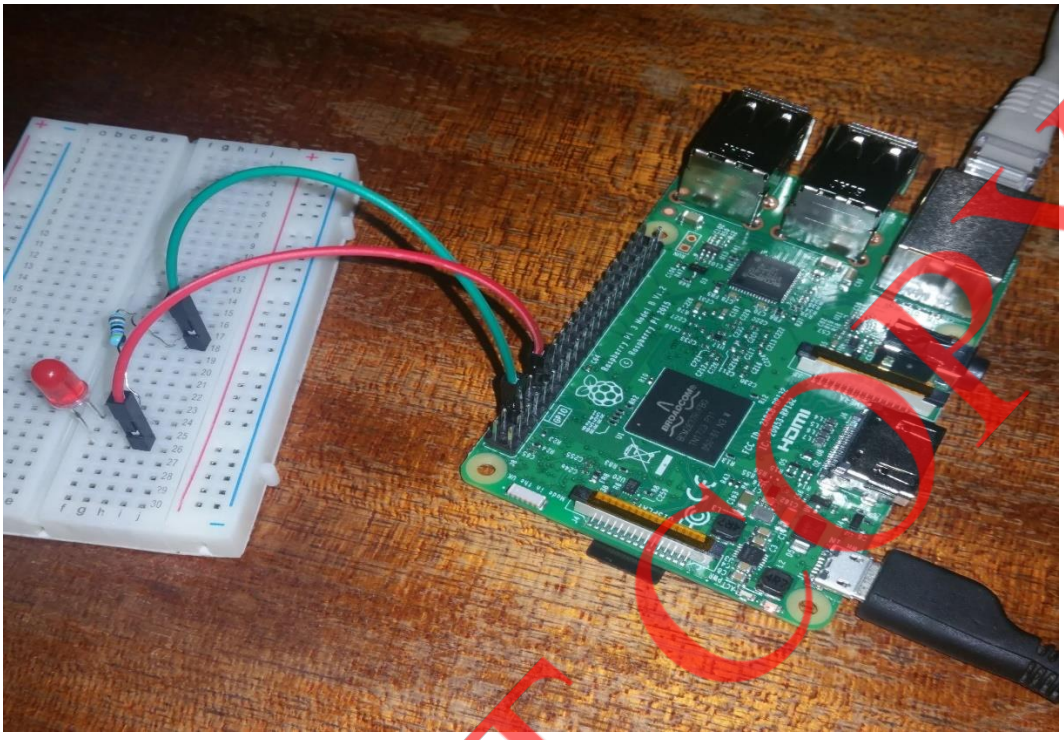

Fig: Raspberry Pi Implementation on LED

Software Implementation:

To verify if Thonny is already available, users can navigate to the Raspberry Pi's menu. Thonny is typically found under the "Programming" section, but it may also appear in the main application list. This quick access ensures that users can easily locate and begin using the IDE. If Thonny is present, it can be launched by simply clicking its icon. The interface is designed for ease of use, featuring a code editor window where users can write Python scripts, a "Run" button to execute the code, and various built-in debugging tools to assist in identifying and resolving errors. If Thonny is not already installed, users can quickly add it to their system. First, a terminal window needs to be opened, which can be done through the top-left menu or by using the shortcut (Ctrl+Alt+T).

Updating package lists:

Before installing Thonny, the system's package lists should be updated **[10]**. This ensures that the latest available version of the software will be installed. The following command is entered into the terminal to perform the update:

Installing Thonny:

Once the system is updated, Thonny can be installed using the following command:

```
sudo apt update
```

The installation process will prompt for the user's password, which should be entered when requested [9]. After completion, Thonny will be ready to use, and users can proceed to develop their Python-based projects on the Raspberry Pi with ease.

This method ensures that even if Thonny is not available initially, it can be installed efficiently, allowing for a smooth and productive coding experience on the Raspberry Pi.

CODE:

```
import RPi.GPIO as GPIO
import time

# Setting up the GPIO mode
GPIO.setmode(GPIO.BCM)

# Setting up GPIO 18 as output for the LED, and GPIO 23 as input for
the button
GPIO.setup(18, GPIO.OUT)
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)

try:
    while True:
        # Reading button state (pressed or not pressed)
        button_state = GPIO.input(23)

        if button_state == False:  # pull-up means LOW when pressed
            GPIO.output(18, True)  # Turning on LED
        else:
            GPIO.output(18, False)  # Turning off LED

        time.sleep(0.1)  # Small delay required for debouncing the
button

except KeyboardInterrupt:
```

```
    # Cleaning up GPIO settings if program is interrupted
    GPIO.cleanup()
```

Code Explanation:

```
import RPi.GPIO as GPIO
import time
```

RPi.GPIO library: This library is used to control the GPIO (General Purpose Input/Output) pins of the Raspberry Pi. These pins are used to interact with hardware components like buttons, LEDs, sensors, etc.

“time” library: The time library is imported to use the sleep() function, which introduces a short delay in the loop to avoid rapid execution and to debounce the button [8].

Setting Up the GPIO Mode:

```
GPIO.setmode(GPIO.BCM)
```

This sets the pin numbering system to BCM mode, where the GPIO numbers are based on the Broadcom SOC channel. BCM numbering refers to the pin numbers on the actual chip, not the physical board.

Pin Setup:

```
GPIO.setup(18, GPIO.OUT)
GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

GPIO.setup(18, GPIO.OUT): This configures GPIO pin 18 as an output pin, which will control the LED. The LED will turn on or off based on the signal sent to this pin.

GPIO.setup(23, GPIO.IN, pull_up_down=GPIO.PUD_UP): This sets GPIO pin 23 as an input pin, which will read the state of the push button [15]. The pull_up_down=GPIO.PUD_UP part activates an internal pull-up resistor, which ensures that the button is normally in a HIGH state (True) when not pressed and goes LOW (False) when pressed.

Main Loop (Button Control Logic):

```
try:
    while True:
        button_state = GPIO.input(23)

        if button_state == False:
            GPIO.output(18, True)
        else:
            GPIO.output(18, False)

        time.sleep(0.1)
```

- while True: This creates an infinite loop that continuously checks the state of the button and controls the LED accordingly.

- button_state = GPIO.input(23): Reads the state of GPIO pin 23 (connected to the button). If the button is pressed, the value will be False (because of the pull-up configuration); otherwise, it will be True.

- if button_state == False: This checks whether the button is pressed. Since the pull-up resistor is used, the button state is False when pressed. If pressed:

  - GPIO.output(18, True): Turns the LED on by sending a HIGH signal to GPIO pin 18.

- else: If the button is not pressed (button state is True):

  - GPIO.output(18, False): Turns the LED off by sending a LOW signal to GPIO pin 18 **[5]**.

- time.sleep(0.1): This introduces a small delay (100ms) between checks, which helps in debouncing the button. Debouncing ensures that a single press is not counted multiple times due to mechanical bounce in the button contacts.

Handling Interruptions and Cleanup:

```
except KeyboardInterrupt:
```

```
    GPIO.cleanup()
```

- except KeyboardInterrupt:: This block is executed when the program is interrupted, usually by pressing Ctrl + C in the terminal. It gracefully handles the interruption.

- GPIO.cleanup(): This function is called to reset the GPIO pins to their default state when the program exits. It ensures that the GPIO resources are released, preventing potential issues when running other GPIO-based programs in the future.

PROBLEM 2

Connect a DHT11 or DHT22 Sensor to a GPIO Pin and Read Temperature and Humidity Data. Display This Data on the Console.

In this problem, the DHT11 sensor is connected to the Raspberry Pi to measure temperature and humidity data. The Raspberry Pi is an ideal platform for such sensor-based applications because of its GPIO (General Purpose Input/Output) pins, which allow it to interface with a variety of sensors and hardware.

DHT11 Sensor:

The DHT11 is a low-cost digital sensor capable of measuring temperature and humidity. It operates by using a thermistor to measure temperature and a capacitive sensor to measure humidity [3]. The sensor communicates with the Raspberry Pi using a single-wire communication protocol, which reduces the complexity of wiring and allows for easy interfacing.

Key Features of DHT11 Sensor:

a) ***Temperature Range:*** 0 to 50°C with an accuracy of ±2°C.
b) ***Humidity Range:*** 20 to 90% with an accuracy of ±5%.
c) ***Operating Voltage:*** 3.3 to 5V, making it compatible with Raspberry Pi's GPIO pins.
d) ***Data Transmission:*** Uses a single digital pin for data communication.

Wiring the DHT11 to Raspberry Pi:

To connect the DHT11 to the Raspberry Pi, the following connections were made:

Pin 1 (VCC) of the DHT11 was connected to the 3.3V pin of the Raspberry Pi.

Pin 2 (Data) of the DHT11 was connected to GPIO pin 4 of the Raspberry Pi.

Pin 4 (GND) of the DHT11 was connected to the ground pin of the Raspberry Pi.

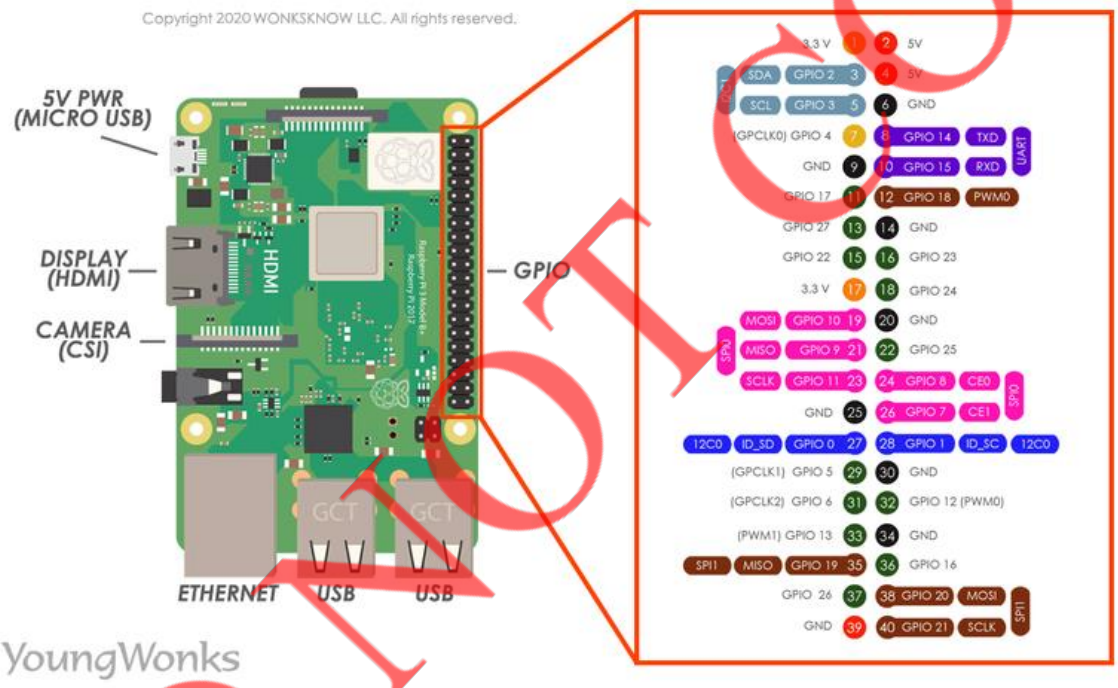

Fig: GPIO pinout for Raspberry Pi [13]

Code for Reading the DHT11 Sensor Data:

To communicate with the DHT11 sensor and retrieve the temperature and humidity values, the Adafruit-DHT library was installed. This library abstracts much of the low-level details involved in communicating with the sensor and provides simple functions for reading the data.

The Python code to read data from the DHT11 sensor is shown below:

```
import Adafruit_DHT

# Setting sensor type and the GPIO pin it is connected to
sensor = Adafruit_DHT.DHT11  # DHT11 sensor type
pin = 4  # GPIO pin number where the sensor is connected

while True:
    # Reading humidity and temperature data from the sensor
    humidity, temperature = Adafruit_DHT.read_retry(sensor, pin)

    if humidity is not None and temperature is not None:
        # Displaying data on the console
        print(f'Temperature: {temperature:.1f}C  Humidity:
{humidity:.1f}%')
    else:
        # If the sensor fails to retrieve data, displaying an error
message
        print('Failed to retrieve data from the sensor.')
```


Explanation of the Code:

1. Library Import:

```
import Adafruit_DHT
```

The Adafruit_DHT library is imported at the beginning of the script. This library provides the necessary functions to read temperature and humidity data from the DHT11 or DHT22 sensors. If the library is not already installed, it can be added via the command:

```
pip install Adafruit-DHT
```

2. Sensor and Pin Definition:

```
sensor = Adafruit_DHT.DHT11
pin = 4
```

The sensor type (DHT11) and the GPIO pin (4) where the sensor is connected are defined. The DHT11 sensor type is selected from the library, and the pin variable stores the GPIO pin number where the data wire of the sensor is connected.

3. Reading Data from the Sensor:

```
    humidity, temperature = Adafruit_DHT.read_retry(sensor, pin)
```

This line uses the read_retry() function from the Adafruit_DHT library to retrieve temperature and humidity data. The function takes two arguments:

- The sensor type (sensor in this case, which is DHT11).

- The GPIO pin where the sensor's data line is connected (pin).

The read_retry() function attempts to read the sensor's data multiple times in case of communication errors. It returns two values:

- humidity: The measured humidity value (in percentage).

- temperature: The measured temperature value (in degrees Celsius).

4. Displaying Data on the Console:

```
    if humidity is not None and temperature is not None:
        print(f'Temperature: {temperature:.1f}C  Humidity:
{humidity:.1f}%')
    else:
        print('Failed to retrieve data from the sensor.')
```

The if statement checks if the sensor was able to return valid data. If both humidity and temperature are not None (i.e., valid data was retrieved), the script prints the data to the console. The formatted string displays the temperature (in degrees Celsius) and humidity (in percentage) with one decimal place for precision.

If the sensor fails to provide valid data (perhaps due to communication errors), the program prints a message indicating that the data retrieval failed.

5. Looping the Data Collection:

The while True loop ensures that the sensor continues reading and displaying temperature and humidity data in real-time. The loop makes the program run continuously, updating the data every second. A sleep() function could be added to pause between readings if necessary.

Running the Code:

Once the code is written and saved as a Python file (e.g., sensor_data.py), it can be executed by running the following command in the terminal:

```
python3 sensor_data.py
```

This will start the script, and the temperature and humidity values will be continuously displayed on the console.

Output:

```
Temperature: 23.5C  Humidity: 45.2%
Temperature: 23.6C  Humidity: 45.1%
Temperature: 23.5C  Humidity: 45.0%
```

The output shows the real-time temperature and humidity values measured by the DHT11 sensor. The values update continuously, reflecting changes in the surrounding environment.

PROBLEM 3

Display DHT11 or DHT22 Sensor Data on an I2C OLED Display.

In this task, the data read from the DHT11 sensor, such as temperature and humidity, is displayed on an I2C OLED screen. The OLED screen serves as a visual output device to display real-time data, which is highly useful in embedded systems projects like weather stations, home automation, and environmental monitoring. The I2C protocol simplifies the connection between the Raspberry Pi and the OLED display by requiring only two wires for communication (SDA and SCL).

OLED Display:

An OLED (Organic Light-Emitting Diode) display is a compact, high-contrast screen that requires very low power to operate. In this experiment, the OLED display communicates via the I2C protocol, which means only two pins are used for data transmission—SDA (Serial Data) and SCL (Serial Clock) [4].

Key Features of the OLED Display:

a) **Resolution:** 128x64 pixels.
b) **Communication:** I2C protocol (using GPIO pins).
c) **Power Consumption:** Very low power, making it suitable for battery-powered systems.
d) **Display Type:** Monochrome, providing sharp contrast for text and graphical output.

Wiring the OLED Display to Raspberry Pi:

The OLED display needs to be connected to the Raspberry Pi using the I2C pins:

a) SDA (Data Line) of the OLED is connected to GPIO pin 2 (SDA).
b) SCL (Clock Line) of the OLED is connected to GPIO pin 3 (SCL).
c) VCC (Power) is connected to the 3.3V pin.
d) GND (Ground) is connected to any ground pin on the Raspberry Pi.

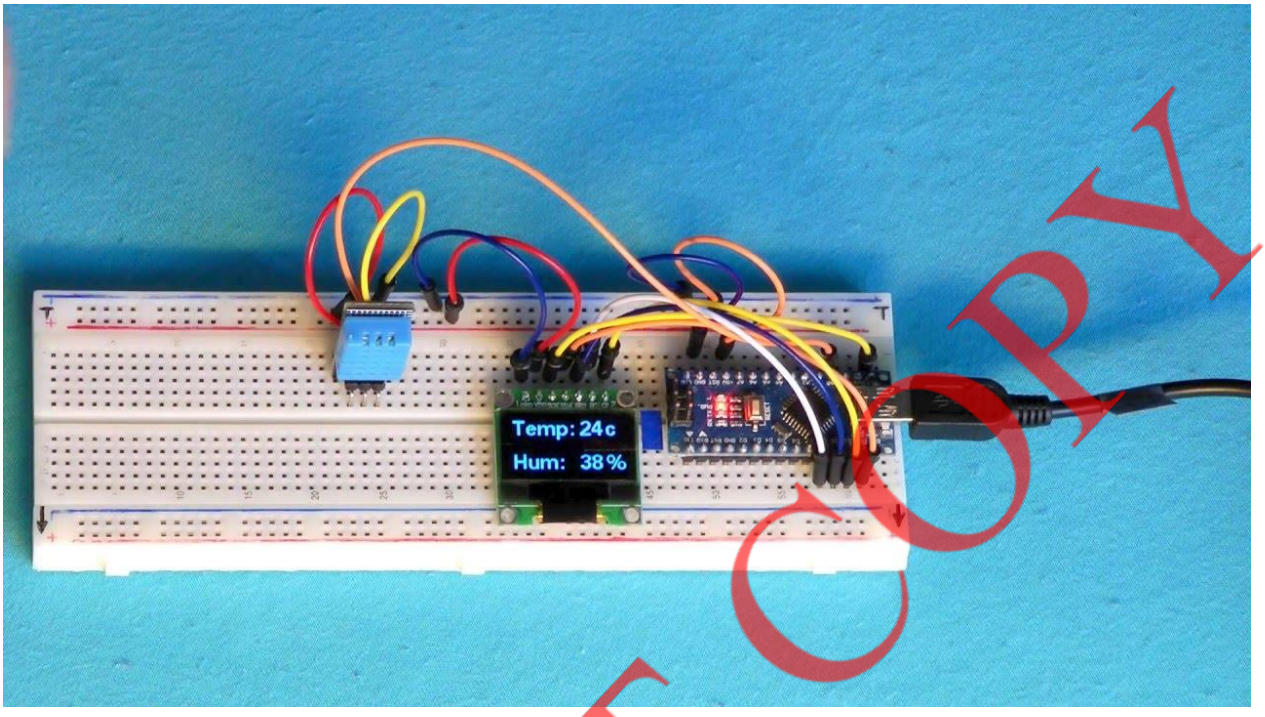

Fig: Raspberry Pi Implementation on OLED

Code for Displaying Data on the OLED Screen:

The *luma.oled* library is used in Python to communicate with the I2C OLED display. This library simplifies the process of displaying both text and graphics on OLED or LCD screens.

Below is the Python code used to read temperature and humidity data from the DHT11 sensor and display it on the OLED display:

```
import Adafruit_DHT
from luma.core.interface.serial import i2c
from luma.oled.device import ssd1306
from PIL import Image, ImageDraw, ImageFont

# Setting up the DHT11 sensor and GPIO pin number
sensor = Adafruit_DHT.DHT11
pin = 4  # GPIO pin number where the DHT11 is connected
```

```
# Setting up I2C communication and initialize the OLED display
serial = i2c(port=1, address=0x3C)  # I2C port and address of the OLED
display
device = ssd1306(serial)  # Initializing OLED display with SSD1306
driver

# Main loop to continuously read sensor data and display it on OLED
while True:
    # Reading humidity and temperature from the DHT11 sensor
    humidity, temperature = Adafruit_DHT.read_retry(sensor, pin)

    # Clearing the OLED display
    device.clear()

    # Creating a new blank monochrome image for the OLED display
    image = Image.new('1', (device.width, device.height))
    # Creating a draw object to add text or graphics
    draw = ImageDraw.Draw(image)

    # Loading a default font
    font = ImageFont.load_default()

    # if valid sensor data is obtained, displaying it
    if humidity is not None and temperature is not None:
        # preparing temperature and humidity strings for display
        temp_text = f'Temp: {temperature:.1f}C'
        humid_text = f'Hum: {humidity:.1f}%'

         # drawing text on the image at specific coordinates
       # displaying temperature
        draw.text((0, 0), temp_text, font=font, fill=255)
       # displaying humidity
        draw.text((0, 10), humid_text, font=font, fill=255)
    else:
        # ff data retrieval fails, displaying an error message on the
OLED
        draw.text((0, 0), 'Sensor Error', font=font, fill=255)

    # displaying the image on the OLED
    device.display(image)
```

Explanation of the Code:

1. Importing Necessary Libraries:

```
import Adafruit_DHT
from luma.core.interface.serial import i2c
from luma.oled.device import ssd1306
from PIL import Image, ImageDraw, ImageFont
```

The code imports several important libraries:

- Adafruit_DHT for reading data from the DHT11 sensor.

- luma.core and luma.oled for controlling the OLED display using the I2C protocol.

- PIL (Python Imaging Library) is used to create an image object, onto which text and graphics can be drawn before displaying it on the OLED screen.

2. Setting Up the Sensor and OLED Display:

```
sensor = Adafruit_DHT.DHT11
pin = 4
serial = i2c(port=1, address=0x3C)
device = ssd1306(serial)
```

The DHT11 sensor is defined, and it is connected to GPIO pin 4. For the OLED, the I2C port (port 1) and address (0x3C) of the display are defined. The ssd1306 class initializes the OLED display, which uses the SSD1306 driver (a common driver for OLED displays).

3. Reading Data from the DHT11 Sensor:

```
humidity, temperature = Adafruit_DHT.read_retry(sensor, pin)
```

Similar to Question 2, this line retrieves the temperature and humidity data from the DHT11 sensor using the read_retry() method. The data is stored in two variables: humidity and temperature.

4. Creating an Image for the OLED Display:

```
image = Image.new('1', (device.width, device.height))
draw = ImageDraw.Draw(image)
```

Here, a new blank image object is created using Image.new() with dimensions matching the OLED display. The ImageDraw.Draw() function creates a drawing object, which allows text or graphics to be added to the image.

5. Displaying the Data on the OLED:

```
temp_text = f'Temp: {temperature:.1f}C'
humid_text = f'Hum: {humidity:.1f}%'
draw.text((0, 0), temp_text, font=font, fill=255)
draw.text((0, 10), humid_text, font=font, fill=255)
```

If valid data is retrieved from the sensor, formatted strings are created for temperature and humidity values. The draw.text() method is used to render this text at specific coordinates on the OLED screen. The text is drawn at positions (0, 0) for temperature and (0, 10) for humidity. The fill=255 argument ensures that the text is displayed in white on the black background.

6. Handling Sensor Errors:

```
draw.text((0, 0), 'Sensor Error', font=font, fill=255)
```

If the sensor fails to retrieve data, an error message ('Sensor Error') is displayed on the OLED instead of the actual readings.

7. Displaying the Image on the OLED:

```
device.display(image)
```

The device.display() function is used to render the image object onto the OLED screen. This updates the display with the latest data (or error message) in real time.

Running the Code:

Once the Python script is written and saved (e.g., as oled_display.py), it can be executed by running the following command in the terminal:

```
python3 oled_display.py
```

This will continuously read data from the DHT11 sensor and display it on the OLED screen.

Output:

The OLED screen will display text similar to the following:

```
Temp: 23.5C
Hum: 45.2%
```

This output will update continuously as the temperature and humidity values change in real time.

Discussion

The implementation of the three tasks presented in this experiment—controlling an LED with a push button, reading data from a DHT11 sensor, and displaying sensor data on an OLED screen—offered valuable insights into the practical applications of Raspberry Pi and its GPIO pins. The experience was both educational and challenging, as it required careful attention to detail in both hardware connections and software implementation.

Problem 1: Controlling an LED with a Push Button

The initial challenge encountered in this task was related to the physical wiring of the components, especially ensuring correct connections for the LED and push button. Wiring the LED correctly to the GPIO pin and ground required attention to the polarity of the LED and ensuring the use of the resistor to prevent excessive current flow [14]. Additionally, connecting the push button with a pull-up resistor required a clear understanding of how the GPIO pins interpret high and low states.

A minor difficulty arose in managing the button's debouncing, which occurs due to the mechanical nature of push buttons, causing the system to register multiple presses from a single press action. This was resolved by adding a short delay (time.sleep(0.1)) to debounce the input and ensure stable operation.

The overall experience was positive, as the combination of hardware and software provided a hands-on understanding of how GPIO inputs and outputs function on the Raspberry Pi.

Once the wiring and button debouncing were handled, the system performed reliably, and the LED responded to button presses as expected.

Problem 2: Reading Data from a DHT11 Sensor

In this task, the challenge was primarily related to setting up the DHT11 sensor to provide accurate and continuous temperature and humidity readings. The difficulty lay in ensuring that the sensor's data pin was correctly connected to the GPIO pin on the Raspberry Pi and that the necessary Adafruit-DHT library was installed and configured properly.

Another hurdle involved the occasional communication failures between the sensor and the Raspberry Pi, which resulted in error messages indicating a failure to retrieve data [20]. This issue was mitigated by using the read_retry() function, which allowed multiple attempts to read data in case of initial failure, ensuring reliable performance.

The implementation process was straightforward once the correct wiring and software setup were established. The sensor provided accurate and real-time data, which was continuously printed to the console, demonstrating the Raspberry Pi's capability to interact with external sensors effectively.

Problem 3: Displaying Data on an I2C OLED Display

The most technically demanding part of the experiment involved displaying the sensor data from the DHT11 on an OLED screen. The initial difficulty was related to configuring the I2C communication between the Raspberry Pi and the OLED display. Ensuring that the correct I2C address was used and that the luma.oled library was properly installed required careful configuration. A common issue faced during this step was ensuring that the I2C interface was enabled on the Raspberry Pi, which required adjusting settings through raspi-config.

Additionally, setting up the code to create an image object and draw text on the OLED screen presented some challenges in formatting the data correctly for display. There were also some early concerns with updating the display fast enough without causing flicker or instability, which was resolved by optimizing the timing of the data refresh.

Despite these challenges, the process of displaying sensor data on the OLED screen was ultimately successful. Once the initial configuration issues were resolved, the OLED provided a clear and continuous display of temperature and humidity data, demonstrating the practical utility of combining sensor input with real-time visual output.

Conclusion

The overall implementation experience was highly rewarding, providing valuable insight into how the Raspberry Pi can be used as a versatile tool for both sensor-based projects and real-time monitoring. The practical challenges faced throughout the experiment were primarily related to the physical connections of the components and ensuring reliable communication between the Raspberry Pi and external devices. However, with proper troubleshooting and adjustments to the code, the system functioned effectively.

Through this experience, a deeper understanding of the Raspberry Pi's GPIO capabilities and Python-based sensor integration was achieved. Each task demonstrated a specific aspect of hardware and software interaction, with clear learning outcomes for interfacing external components with the Raspberry Pi.

References:

1. https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/
2. https://learn.adafruit.com/raspberry-pi-lesson-control-gpio
3. https://circuitdigest.com/microcontroller-projects/dht11-interfacing-with-raspberry-pi
4. https://www.electromaker.io/tutorial/blog/raspberry-pi-oled-display-tutorial
5. https://www.instructables.com/Controlling-GPIO-pins-on-Raspberry-Pi-using-Python/
6. https://pimylifeup.com/raspberry-pi-push-button/
7. https://www.digikey.com/en/maker/projects/understanding-pull-up-and-pull-down-resistors-with-raspberry-pi/96b1872f82a44770a438a3e4aa053b96
8. https://microcontrollerslab.com/button-debouncing-raspberry-pi-python/
9. https://www.element14.com/community/docs/DOC-97684/l/getting-started-with-thonny-ide-on-raspberry-pi
10. https://www.raspberrypi.com/documentation/computers/os.html
11. https://maker.pro/raspberry-pi/projects/how-to-use-a-breadboard-with-a-raspberry-pi
12. https://www.youtube.com/watch?v=dP0i8tfi0js

13. https://www.embedded-computing.com/technology/processing/hardware/exploring-the-gpio-pin-layout-on-raspberry-pi

14. https://www.electronicshub.org/why-resistors-are-necessary-in-led-circuits/

15. https://electronicsprojectshub.com/controlling-led-with-push-button-on-raspberry-pi/

16. https://www.digikey.com/en/maker/blogs/using-i2c-on-raspberry-pi-for-oled

17. https://www.hackster.io/Aritra-Banerjee/python-gpio-control-on-raspberry-pi

18. https://www.iotforall.com/how-to-integrate-raspberry-pi-with-cloud

19. https://www.allaboutcircuits.com/raspberry-pi/dht11-vs-dht22/

20. https://www.youtube.com/watch?v=example

21. https://www.electronicshub.org/interfacing-mq5-gas-sensor-with-raspberry-pi/

THE END