

Module-4

IoT Physical Devices & End points

Physical Devices & End points: What is a IoT Device, Raspberry Pi, About the Board, Linux on Raspberry Pi, Raspberry Pi interfaces, Programming Raspberry Pi with Python, Case Studies illustrating IoT design – Home Automation, Cities, Agriculture.

4.1 What is an IoT Device?

IoT "Thing" and Devices

- A "Thing" in IoT is any object with a unique identifier that can send or receive data over a network (e.g., smart refrigerator, car).
- IoT devices are physical endpoints connected to the Internet that:
 - Send information (data sensed by connected sensors) about themselves or their surroundings to other devices or servers.
 - Allow actuation upon the physical environment remotely.
 - Examples: Home automation devices, industrial machines sending health data, wearable fitness trackers.

4.1.1 Basic Building Blocks of an IoT Device

An IoT device can be broken down into four core functional modules:

1. Sensing: Collecting data (e.g., temperature, humidity, light intensity) from on-board or attached sensors. The sensed information is communicated to other devices or cloud storage.
2. Actuation: Taking actions upon physical entities using attached actuators. For example, a relay switch can turn an appliance on or off based on remote commands.
3. Communication: Modules responsible for sending collected data and receiving data or commands from remote applications.

4. Analysis & Processing: Making sense of the collected data.

The IoT device used in this book is the Raspberry Pi, a small single-board computer. It is chosen because it is cheap, easy to find, and simple to use. There is also plenty of information about it on the Internet and in books. The same ideas explained in this book can also be used for other IoT devices. Before learning about Raspberry Pi, let's first understand the main parts of a single-board computer (SBC) used in IoT devices.

Figure 4.1 shows these parts, which include the CPU, GPU, RAM, storage, and different interfaces and peripherals.

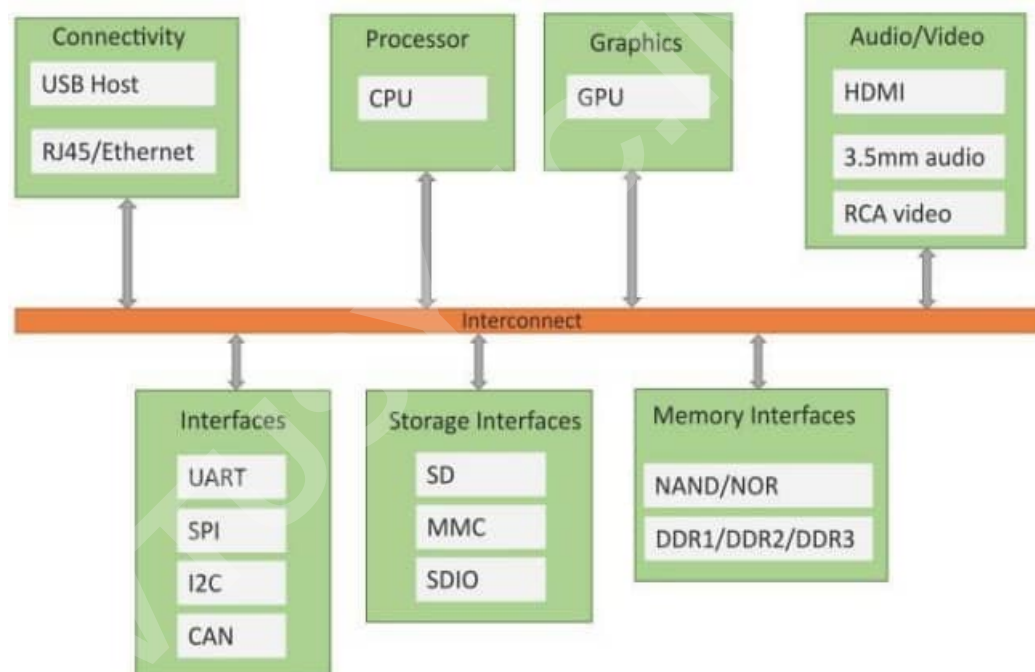


Figure 4.1: Block diagram of an IoT Device

4.2 Example Device: Raspberry Pi

The Raspberry Pi is a small and low-cost computer, about the size of a credit card. It can run different versions of the Linux operating system and can do most of the tasks a normal desktop computer can do.

It also has GPIO pins (General Purpose Input/Output) that allow us to connect sensors and actuators.

Since it runs on Linux, the Raspberry Pi can easily use the Python programming language.

4.3 About the Board

Figure 4.2 shows the Raspberry Pi board and its main components.

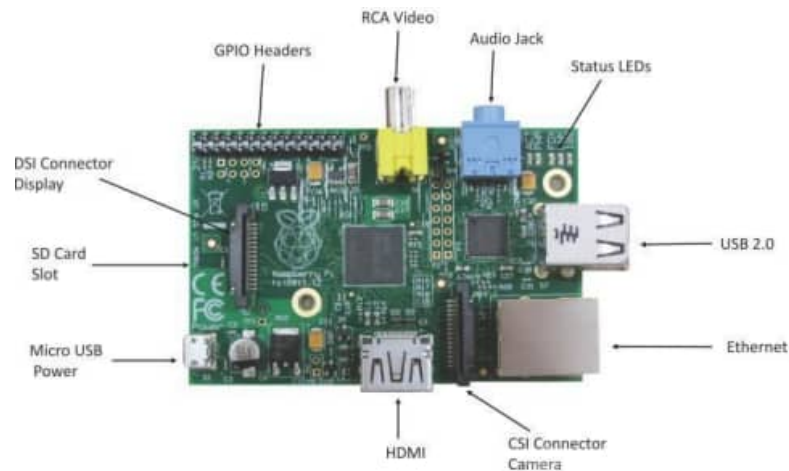


Figure4.2: Raspberry Pi board

1. Processor & RAM:

Raspberry Pi uses an ARM processor. The latest Model B (Revision 2) has a 700 MHz ARM1176JZ-F processor and 512 MB SDRAM.

2. USB Ports:

It has two USB 2.0 ports. Each port provides up to 100mA current. For devices needing more power, use an external powered USB hub.

3. Ethernet Port:

Includes a standard RJ45 Ethernet port for Internet connection. You can also use a USB Wi-Fi adapter.

4. HDMI Output:

Provides both video and audio. Connects to monitors using HDMI or HDMI-to-DVI adapter.

5. Composite Video Output:

Has an RCA jack for connecting old TVs that support PAL or NTSC.

6. Audio Output:

Includes a 3.5mm audio jack for sound output. The audio quality is lower than HDMI output.

7. GPIO Pins:

Comes with General Purpose Input/Output (GPIO) pins for connecting sensors and devices.

Types of pins: GPIO, I2C, SPI, and Serial (Rx/Tx).

8. Display Serial Interface (DSI):

Used to connect an LCD display.

9. Camera Serial Interface (CSI):

Used to connect a camera module.

10. Status LEDs:

Has five LEDs that show power, network, and SD card activity. Refer to Table 7.1 for their functions.

Status LED	Function
ACT	SD card access
PWR	3.3V Power is present
FDX	Full duplex LAN connected
LNK	Link/Network activity
100	100 Mbit LAN connected

Table 4.1: Raspberry Pi Status LEDs

11. SD Card Slot:

Stores the operating system. Requires at least an 8GB SD card with Linux or NOOBS installed.

12. Power Input:

Uses a micro-USB connector for power supply.

4.4 Linux on Raspberry Pi

Raspberry Pi supports different versions (flavors) of the Linux operating system.

- The main and recommended version is Raspbian, which is based on Debian Wheezy and specially optimized for Raspberry Pi. Appendix 1 explains how to install Raspbian on the device.

Other versions include:

- Arch Linux – A lightweight Linux system designed for ARM devices.
- Pidora – A Fedora-based Linux system made for Raspberry Pi.
- RaspBMC – A Linux distribution used as an XBMC media center.
- OpenELEC – A fast and user-friendly XBMC media-center system.
- RISC OS – A compact and high-speed operating system.

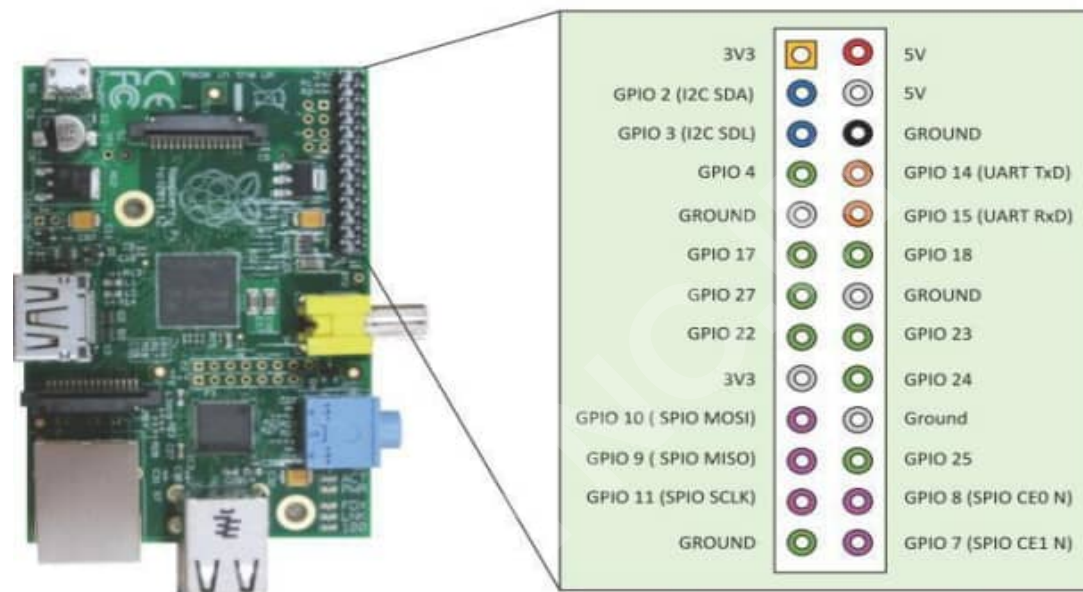


Figure 4.3: Raspberry Pi GPIO Headers

Figures 4.4 to 4.7 show different parts of the Raspbian system, including the desktop, file explorer, console, and web browser. Figure 4.4 shows the Raspbian desktop. Figure 4.5 shows the file explorer. Figure 4.6 shows the command console. Figure 4.7 shows the default web browser.

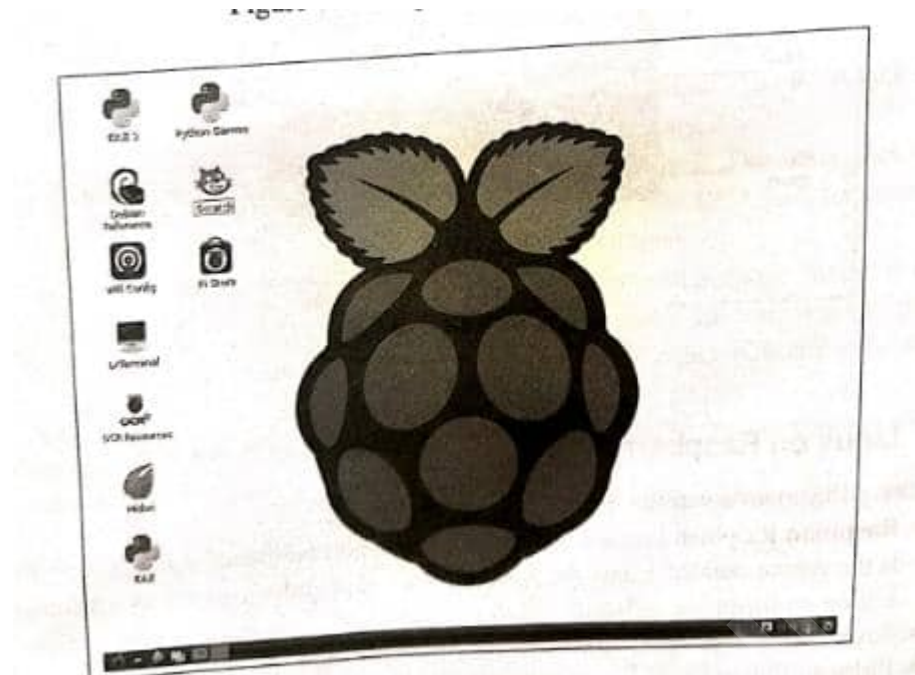


Figure 4.4: Raspbian Linux desktop

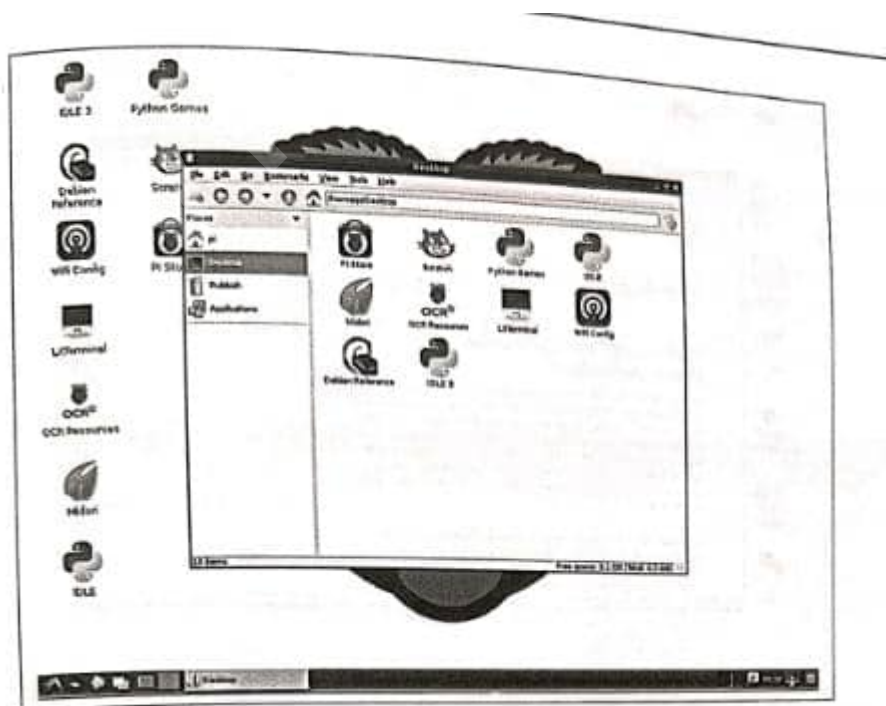


Figure 4.5: File Explorer on Raspberry Pi



Figure 4.6 Console on Raspberry Pi

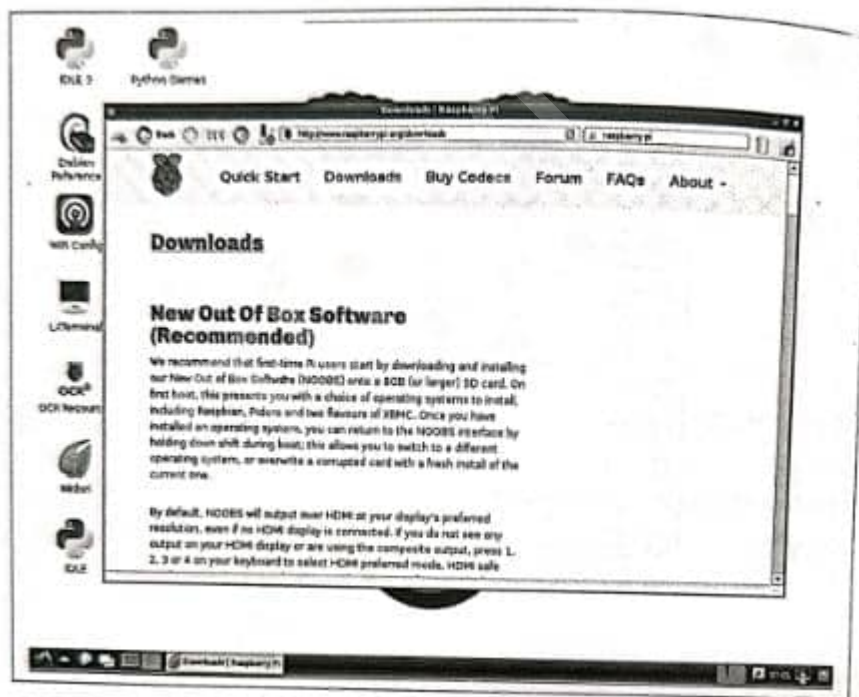


Figure 4.7: Browser on Raspberry Pi



Figure 4.8 Raspberry Pi Configuration tool

The raspi-config tool is used to configure Raspberry Pi. It can be opened from the command line by typing `$raspi-config` shown in figure 4.8.

With this tool, users can:

- Expand the root partition to use the full SD card,
- Set keyboard layout, locale, and timezone,
- Change password,
- Adjust memory split,
- Enable or disable SSH, and
- Change boot settings.

It is recommended to expand the root file system so that the entire SD card space can be used.

Accessing Raspberry Pi

Although Raspberry Pi has an HDMI output for connecting a monitor, it is often more convenient to use it through a VNC connection or SSH.

Using VNC or SSH allows you to control Raspberry Pi from your desktop or laptop without needing a separate display. Appendix A explains how to set up a VNC server on Raspberry Pi and how to connect using SSH.

Table 4.2 provides a list of frequently used Raspberry Pi commands.

Command	Function	Example
cd	Change directory	cd /home/pi
cat	Show file contents	cat file.txt
ls	List files and folders	ls /home/pi
locate	Search for a file	locate file.txt
lsusb	List USB devices	lsusb
pwd	Print name of present working directory	pwd
mkdir	Make directory	mkdir /home/pi/new
mv	Move (rename) file	mv sourceFile.txt destinationFile.txt
rm	Remove file	rm file.txt
reboot	Reboot device	sudo reboot
shutdown	Shutdown device	sudo shutdown -h now
grep	Print lines matching a pattern	grep -r "pi" /home/
df	Report file system disk space usage	df -Th
ifconfig	Configure a network interface	ifconfig
netstat	Print network connections, routing tables, interface statistics	netstat -lnpt
tar	Extract/create archive	tar -xzf foo.tar.gz
wget	Non-interactive network downloader	wget http://example.com/file.tar.gz

Table 4.2 Raspberry Pi Frequency used commands

4.5 Raspberry Pi Interfaces

Raspberry Pi provides several interfaces for data transfer, including Serial, SPI, and I2C, as shown in Figure 4.3.

4.5.1 Serial Interface

The serial interface allows communication with serial peripherals.

It has two main pins:

- Rx (Receive) – Receives data from peripherals.
- Tx (Transmit) – Sends data to peripherals.

4.5.2 SPI (Serial Peripheral Interface)

SPI is a synchronous serial protocol for communication with one or more peripheral devices.

One master device controls the communication, and one or more slave devices respond.

SPI pins on Raspberry Pi:

- MISO (Master In Slave Out) – Data sent from slave to master.
- MOSI (Master Out Slave In) – Data sent from master to slave.
- SCK (Serial Clock) – Clock signal generated by the master to synchronize data transfer.
- CE0 (Chip Enable 0) – Enables or disables a peripheral device.
- CE1 (Chip Enable 1) – Enables or disables another peripheral device.

4.5.3 I2C (Inter-Integrated Circuit)

I2C allows communication with hardware modules using only two pins:

- SDA (Data Line) – Transfers data.
- SCL (Clock Line) – Transfers the clock signal for synchronization.

It is also a synchronous data protocol, like SPI.

4.6 Programming Raspberry Pi with Python

Raspberry Pi runs Linux and supports Python by default, so you can run any Python program that works on a normal computer. Its **GPIO pins** provide general-purpose input/output capabilities, making it very useful for **Internet of Things (IoT)** projects. Using the GPIO pins along with **SPI, I2C, and Serial interfaces**, you can connect a variety of sensors and actuators. The input from sensors can be processed in Python, allowing you to perform actions such as sending data to a server, sending an email, or triggering a relay switch.

4.6.1 Controlling LED with Raspberry Pi

- Start with a basic example of controlling an LED using Raspberry Pi.
- The schematic diagram for connecting an LED is shown in Figure 4.9. In this example of BOX 4.1 shows the LED is connected to GPIO pin 18. You can connect the LED to any other GPIO pin as well. The program uses the RPi.GPIO module to control the GPIO on Raspberry Pi. In this program we set pin

18 direction to output and then write True/False alternatively after a delay of one second.

Box 4.1: Python program for blinking LED

```
import RPi.GPIO as GPIO
import time

GPIO.setmode (GPIO.BCM)
GPIO.setup(18, GPIO.OUT)

while True:
    GPIO.output (18, True)
    time.sleep(1)
    GPIO.output (18, False)
    time.sleep (1)
```

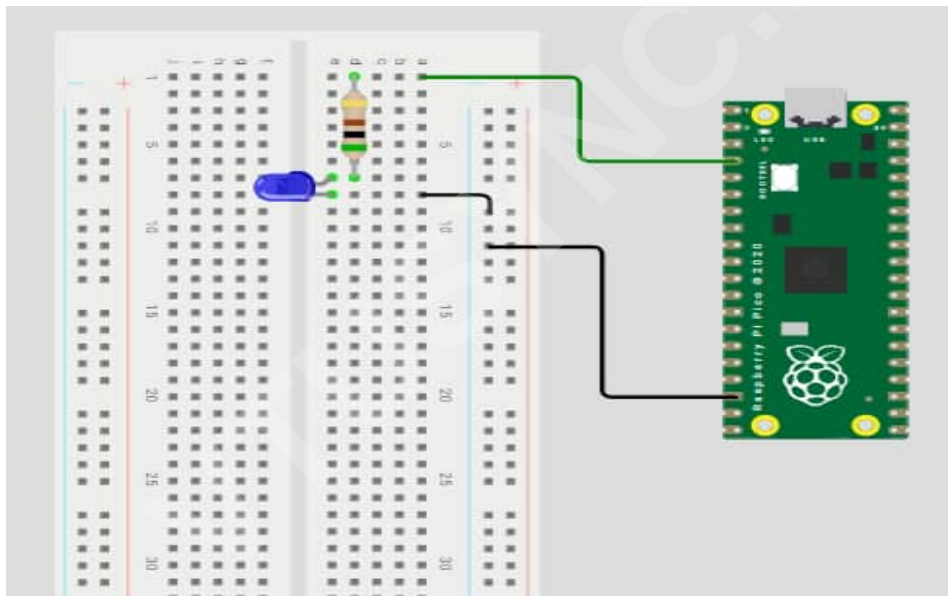


Figure 4.9: Controlling LED with Raspberry Pi

4.6.2 Interfacing an LED and Switch with Raspberry Pi:

- This example shows how to control an LED using a switch.
- Figure 4.10 presents the schematic diagram for connecting an LED and a switch to the Raspberry Pi.
- Box 4.2 provides a Python program where the LED is connected to GPIO pin 18 and the switch to pin 25. In an infinite loop, the program checks the switch

input and toggles the LED state when the switch is pressed, demonstrating how to read input from GPIO pins and act on it.

Box 4.2: Python program for controlling an LED with a switch

```
from time import sleep
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

#Switch Pin
GPIO.setup(25, GPIO.IN)

#LED Pin
GPIO.setup(18, GPIO.OUT)

state=False

def toggleLED(pin):
    state = not state
    GPIO.output(pin, state)

while True:
    try:
        if (GPIO.input(25) == True):
            toggleLED(pin)
            sleep(.01)
    except KeyboardInterrupt:
        exit()
```

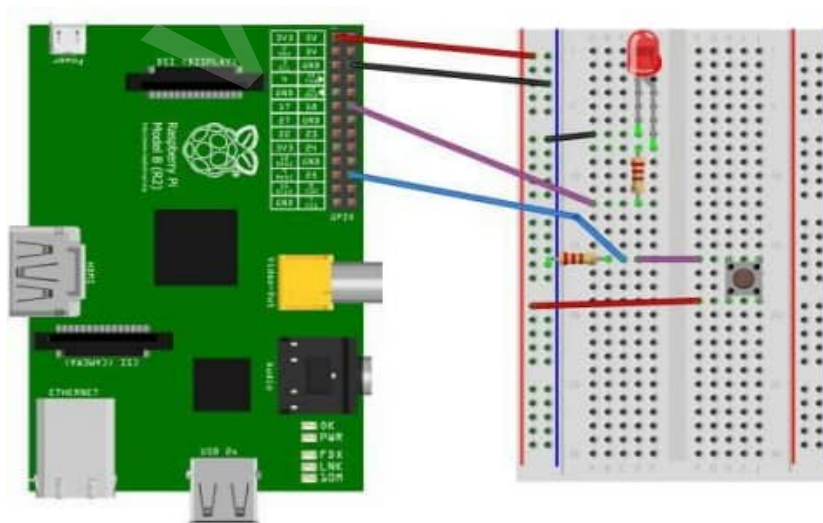


Figure 4.10: Interfacing LED and switch with Raspberry Pi

4.6.3 Interfacing a Light Sensor (LDR) with Raspberry Pi:

- This example demonstrates interfacing a Light Dependent Resistor (LDR) with Raspberry Pi to control an LED based on light intensity.
- Figure 4.11 shows the schematic: one side of the LDR is connected to 3.3V, the other side to a 1 μ F capacitor and GPIO pin 18. An LED is also connected to GPIO pin 18 and is controlled according to the sensed light level.
- Box 4.3 provides a Python program where the readLDR() function measures light intensity. The function sets the LDR pin to output low, then input, allowing the capacitor to charge through the LDR while a counter runs. When the input reads high (capacitor voltage > 1.4V), the counter stops.
- The counter value is proportional to the light level: more light decreases LDR resistance, increasing the capacitor charging time.

Box 4.3: Python program for switching LED/Light based on reading

```
import RPI.GPIO as GPIO
import time

GPIO.setmode (GPIO.BCM)
ldr_threshold = 1000
LDR_PIN =18
LIGHT_PIN = 25

def readLDR(PIN):
    reading=0
    GPIO.setup (LIGHT_PIN, GPIO.OUT)      GPIO.output (PIN, False)
    time.sleep(0.1}
    GPIO.setup (PIN, GPIO.IN)
    while (GPIO.input (PIN)==False):
        reading=reading+1
    return reading

def switchOnLight (PIN):
    GPIO.setup (PIN, GPIO.OUT)
    GPIO.output (PIN, True)

def switchoffLight (PIN):
    GPIO.setup (PIN, GPIO.OUT)
    GPIO.output (PIN, False)

while True:
```

```
ldr_reading = readLDR(LDR_PIN)
if ldr_reading < ldr_threshold:
    switchOnLight (LIGHT_PIN)
else:
    switchoffLight (LIGHT_PIN)
time.sleep(1)
```

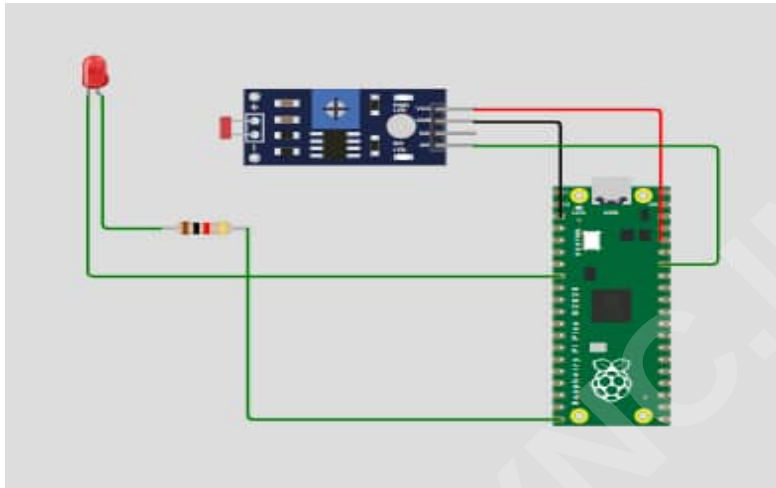


Figure 4.11: Interfacing LDR with Raspberry Pi

4.7 Home Automation:

(Case Studies illustrating IoT design of Home Automation:)

4.7.1 Smart Lighting

A smart home automation system enables users to remotely monitor and control electrical appliances such as lights, fans, and other devices through Internet-connected platforms. Using the Internet of Things (IoT) approach, these appliances can be made intelligent so that they react automatically to environmental conditions while still allowing manual human control. The design of a smart lighting system based on the IoT design methodology was first introduced conceptually in an earlier chapter. In this section, a concrete implementation of that design is described using the **Django web framework** and a **Raspberry Pi** controller.

The **purpose** of this home-automation system is to allow a user to switch household lights on or off remotely through a web application. The system operates in **two modes – automatic and manual**. In **automatic mode**, the intensity of ambient light in the room is measured through a light sensor (LDR). When the surrounding light level falls below a specified threshold, indicating darkness, the controller automatically switches on the light. When brightness increases, the light is turned off. In **manual mode**, the user can override automation and directly control the light state (ON or OFF) from the web interface using any connected device such as a phone or computer.

Figure 4.12 illustrates the deployment design of this home-automation IoT system. The design follows a layered structure that separates the application logic, data storage, and device control. The left portion of the diagram (labeled Local) represents the physical side of the system, where the devices and controller reside, while the right portion (labeled Cloud) represents the possibility of extending the system for cloud-based monitoring or analytics. Within the local segment, the App provides the user interface and communicates through RESTful web services with the database. These REST services interact with a Controller Service, which in turn communicates with the physical Resource and Device layers that include sensors and actuators. A Monitoring Node can analyze data or log historical information for further processing.

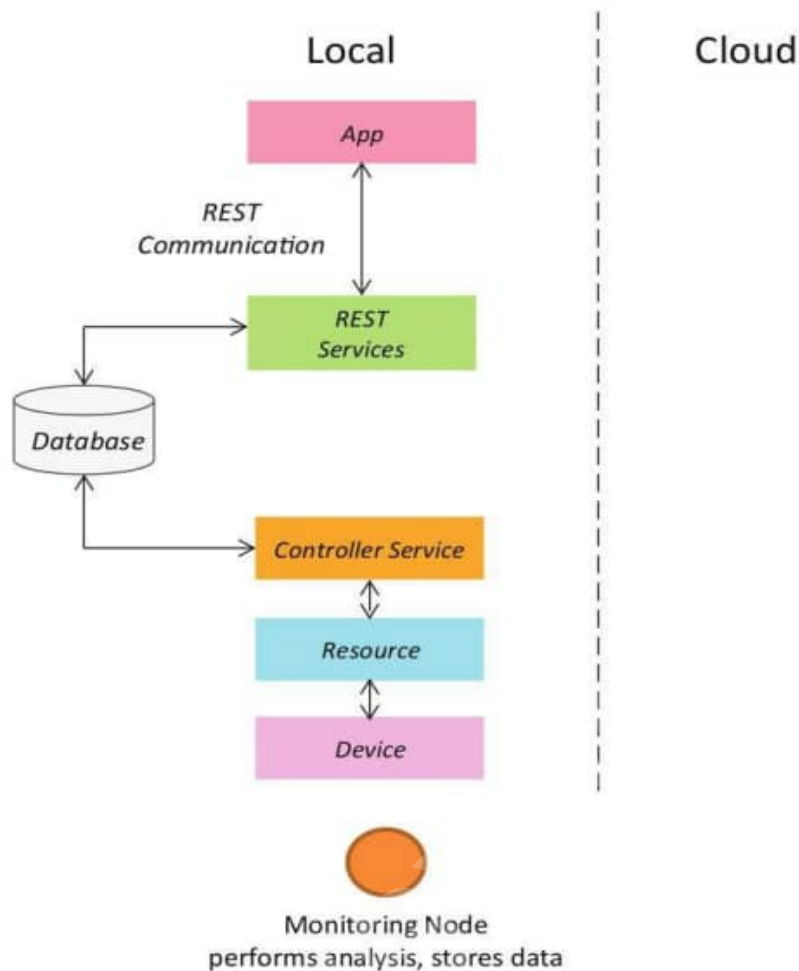


Figure 4.12: Deployment design of the home automation IoT system

The system design includes two REST services – Mode and State – along with a native controller service running on the Raspberry Pi. The Mode service manages whether the system is in automatic or manual operation, whereas the State service manages the current ON/OFF condition of the light. Each service supports two HTTP operations: a PUT request to update data (for example, changing the mode from manual to auto or switching the light on/off) and a GET request to retrieve the present value from the database. Both the mode and state values are stored persistently in a local status database, ensuring that the web application and the controller always share a consistent view of system status.

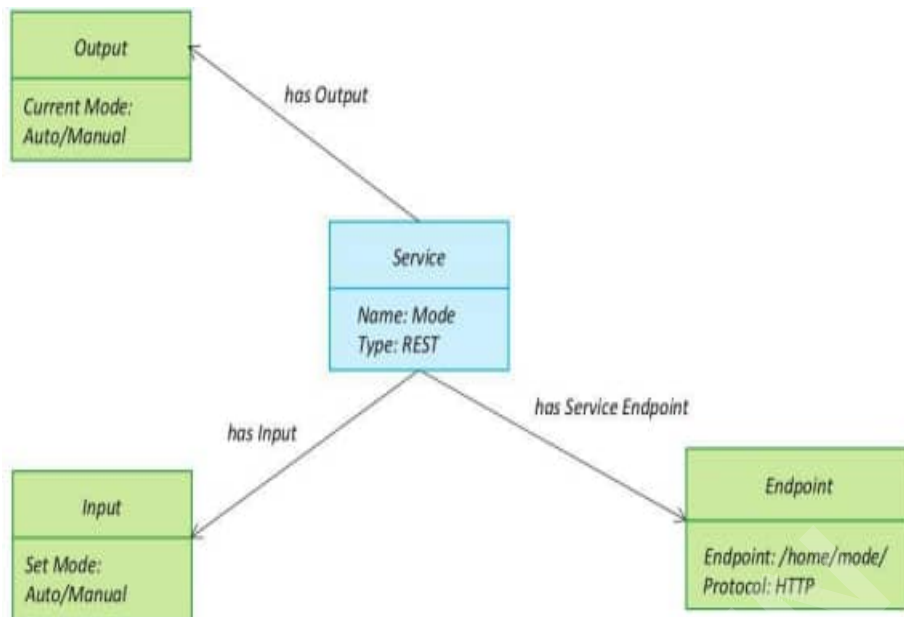


Figure 4.13: Service specification for home automation IoT system - mode service

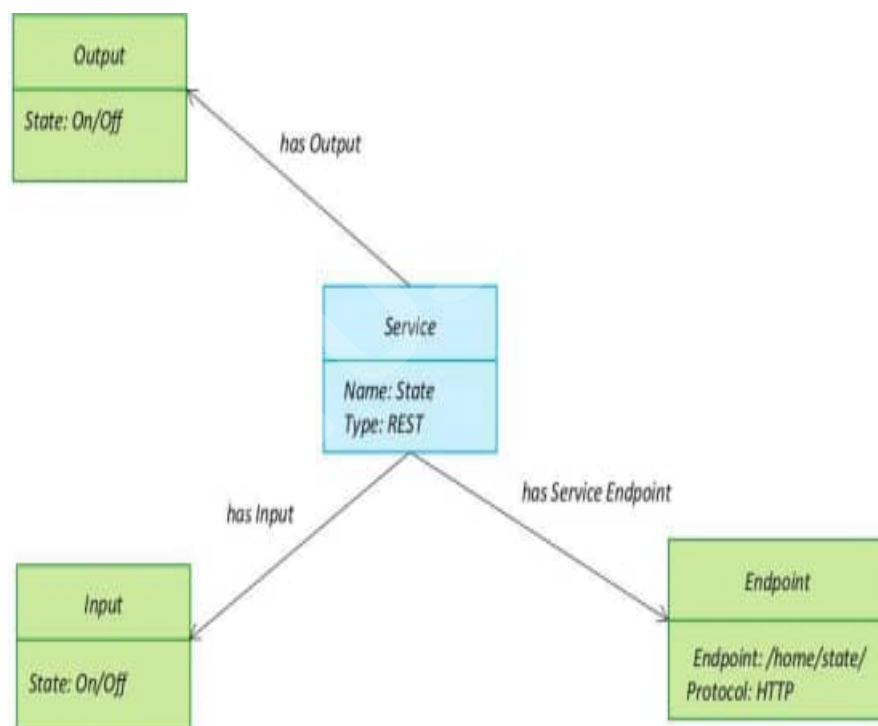


Figure 4.14: Service specification for home automation IoT system - state service

figures 4.12 and 4.13 show the **service specifications** for the Mode and State REST services. In Figure 4.12, the Mode service takes an **input** that sets the mode (Auto or Manual) and produces an **output** indicating the current operating mode. The service has an HTTP endpoint /home/mode/ through which communication

occurs. Similarly, in Figure 4.13, the State service accepts an **input** specifying the desired light state (On or Off) and generates an **output** reporting the present light state. It is exposed through the HTTP endpoint `/home/state/`. Both services are implemented following REST principles, making them lightweight and easy to integrate with other applications or future cloud components.

Implementation step for smart

Step1: Map services to Django models.

Step2: Implement the model serializers. Serializers allow complex data (such as model instances) to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types.

Step3: Write ViewSets for the Django models. ViewSets combine the logic for a set of related views in a single class. The ViewSets for the models (ModeViewSet and StateViewSet) are included in the views file.

Step4: Registering the viewsets with a router class. Routers automatically determine how the URLs for an application should be mapped to the logic that deals with handling incoming requests.

Figure 4.15 illustrates the schematic diagram of the home automation IoT system, which includes devices and components such as a Raspberry Pi mini computer, an LDR sensor, and a relay switch actuator. Figure 4.16 presents the specifications of the controller's native service running on the Raspberry Pi. In auto mode, the controller service monitors the light level, switches the light on or off accordingly, and updates the status in the database. In manual mode, it retrieves the current state from the database and controls the light based on that information. A Python implementation of the controller service is provided in Box 4.4.

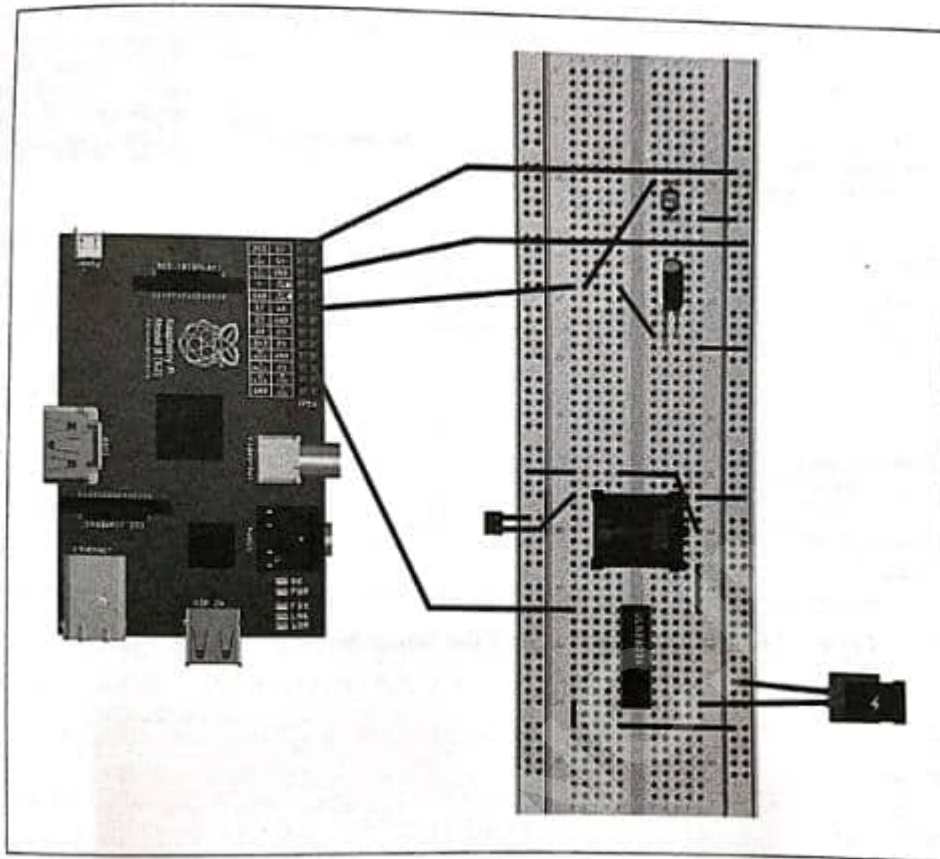


Figure 4.15 shows the schematic diagram of the home automation IoT system, illustrating the integration of the device, sensor, and actuator components.

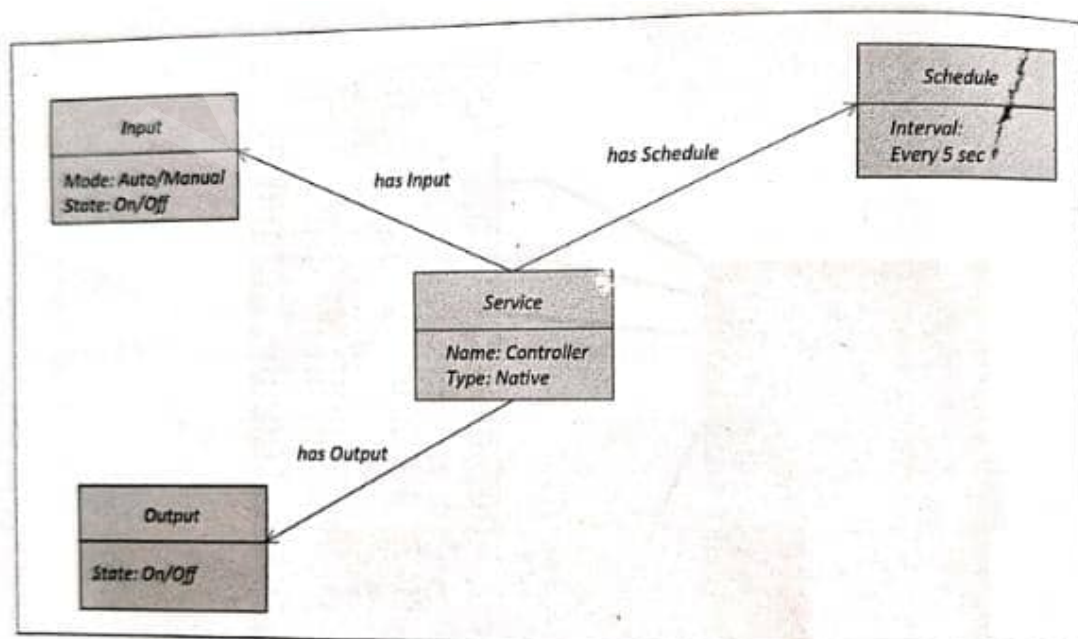


Figure 4.16: Controller service of the home automation system

Box 4.4: Python code for controller native service - controller.py

```
import time

import datetime

import sqlite3

import spidev

import RPi.GPIO as GPIO

#Initialize SQLite

con = sqlite3.connect ('database.sqlite' )

cur = con.cursor ()

#LDR channel on MCP3008

LIGHT_CHANNEL = 0

#GPIO Setup

GPIO.setmode (GPIO.BCM)

LIGHT_PIN = 25

# Open SPI bus

spi = spidev.SpiDev ()

spi.open (0,0)

#Light Level Threshold

threshold=200

#Function to read LDR connected to MCP3008

def readLDR():

    light_level = ReadChannel (LIGHT_CHANNEL)

    lux = ConvertLux(light_level, 2)
```

```

        return lux

#Function to convert LDR reading to Lux

def ConvertLux(data,places) :

    R=10 #10k-ohm resistor connected to LDR

    volts = (data *3'.3) / 1023

    volts = round(volts,places)

    lux=500* (3.3-volts) / (R*volts)

    return lux

# Function to read SPI data from MCP3008 chip
def ReadChannel (channel):

    adc = spi.xfer2([1, (8t+channel) <<4,0])

    data = ((adc [1] &3) << 8) + adc[2]

    return data

#Get current state from DB

def getCurrentMode():

    cur.execute('SELECT * FROM myapp_mode' )

    data = cur.fetchone() #(1, u'auto')

    return data[1]

#Get current state from DB

def getCurrentState():

    cur.execute('SELECT * FROM myapp_state' )

    data = cur.fetchone () #{(1, ul'on')}

    return data[1]

```

```
#Store current state in DB

def setCurrentState(val):

    query=' UPDATE myapp_state set name="'+val+'"'

    cur.execute (query)

def switchOnLight (PIN) :

    GPIO.setup (PIN, GPIO.OUT)

    GPIO.output (PIN, True)

def switchOffLight (PIN):

    GPIO.setup (PIN, GPIO.OUT)

    GPIO.output (PIN, False)

def runManualMode():

    #Get current state from DB

    currentState=getCurrentState ()

    if currentState=='on':

        switchOnLight (LIGHT_PIN)

    elif currentState==' off':

        switchOffLight (LIGHT_PIN)

def runAutoMode ():

    #Read LDR

    light level=readLDR ()

    if lightlevel < ldr_threshold:

        switchOnLight (LIGHT_PIN)

    else:
```

```
        switchOffLight (LIGHT_PIN)

    print 'Manual' + ' — '+getCurrentState()

#Controller main function
def runController():

    currentMode=get CurrentMode ()

    if currentMode==' auto':

        runAutoMode ()

    elif currentMode=='manual':

        runManualMode ()

    return true

while True:

    runController ()

    time.sleep (5)
```

4.7.2 Home Intrusion Detection

The home intrusion detection system is designed to identify unauthorized entries or movements inside a house using various sensors. It uses PIR (Passive Infrared) sensors to detect motion in rooms and door sensors to detect when doors are opened. When an intrusion or motion is detected, the system records the event and sends alerts if needed.

Figure 4.17 shows the process diagram of the system. Each room is equipped with a PIR sensor, and each door has a door sensor. These sensors are regularly monitored, and any detected activity is stored and used to generate alerts.

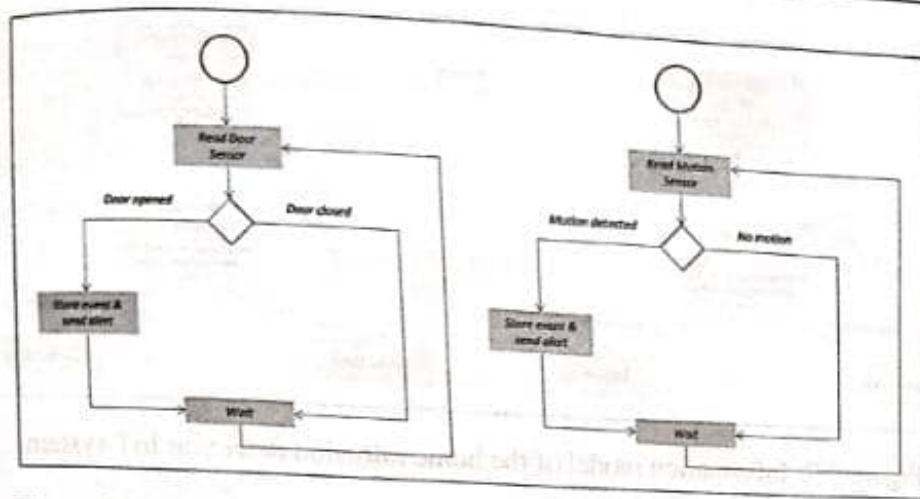


Figure 4.17: Process specification of the home intrusion detection IoT system

Figure 4.18 represents the domain model of the intrusion detection system. It includes both physical entities (like rooms and doors) and their virtual entities in the system. The device used is a single-board mini computer connected to the PIR and door sensors. The domain model also shows the services that work together to make the system function.

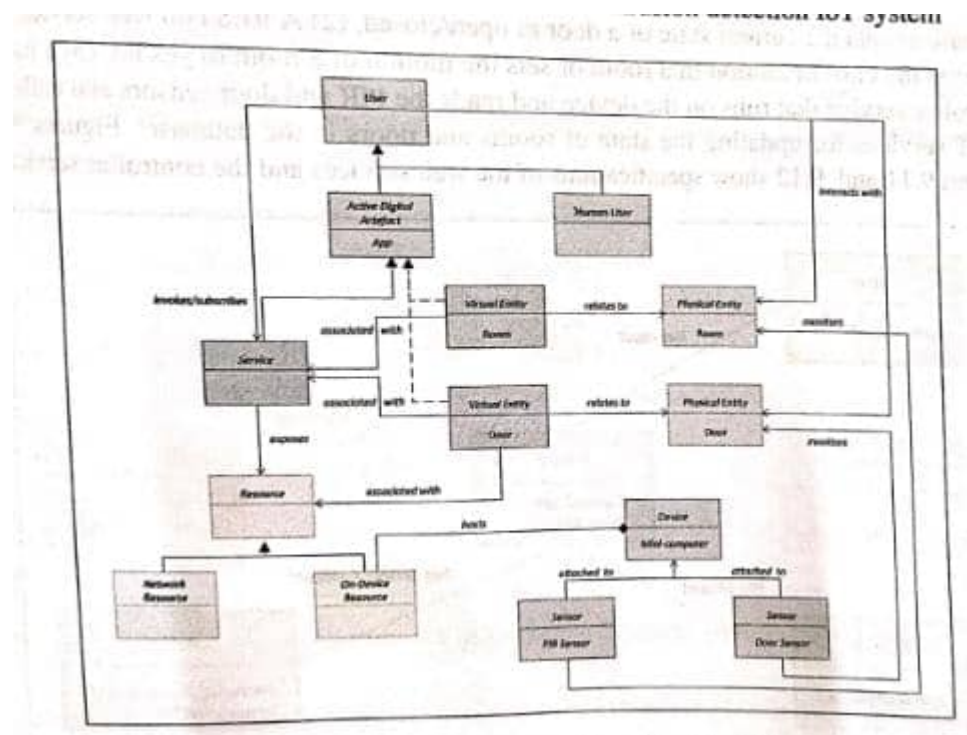


Figure 4.18: Domain model of the home intrusion detection IoT system

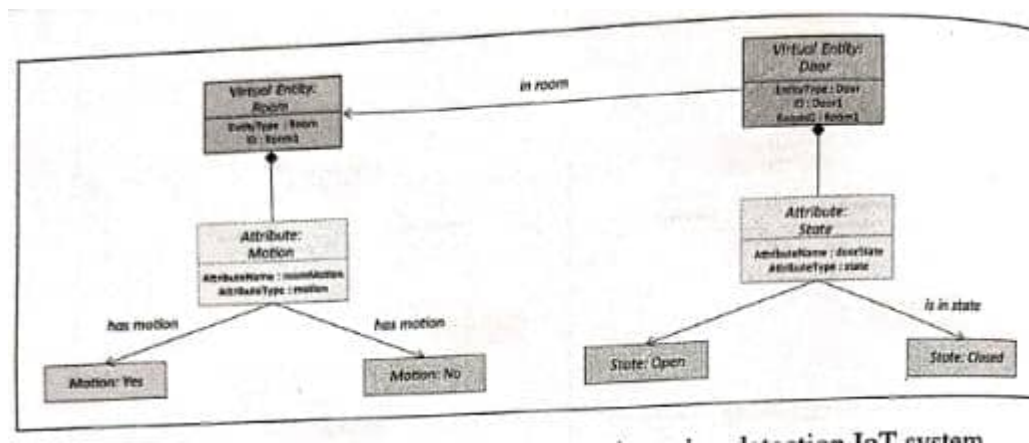


Figure 4.19: Information model of the home intrusion detection IoT system

Figure 4.19 shows the information model, which defines the attributes of the virtual entities. The room virtual entity has an attribute called motion, and the door virtual entity has an attribute called state. These attributes indicate whether there is motion in a room or whether a door is open or closed.

The system provides three main services:

- A RESTful web service to get or update the door's state (open/closed).
- A RESTful web service to get or update the motion status in a room (yes/no).
- A native controller service that runs on the device, reads the PIR and door sensors, and updates the database by calling the REST services.

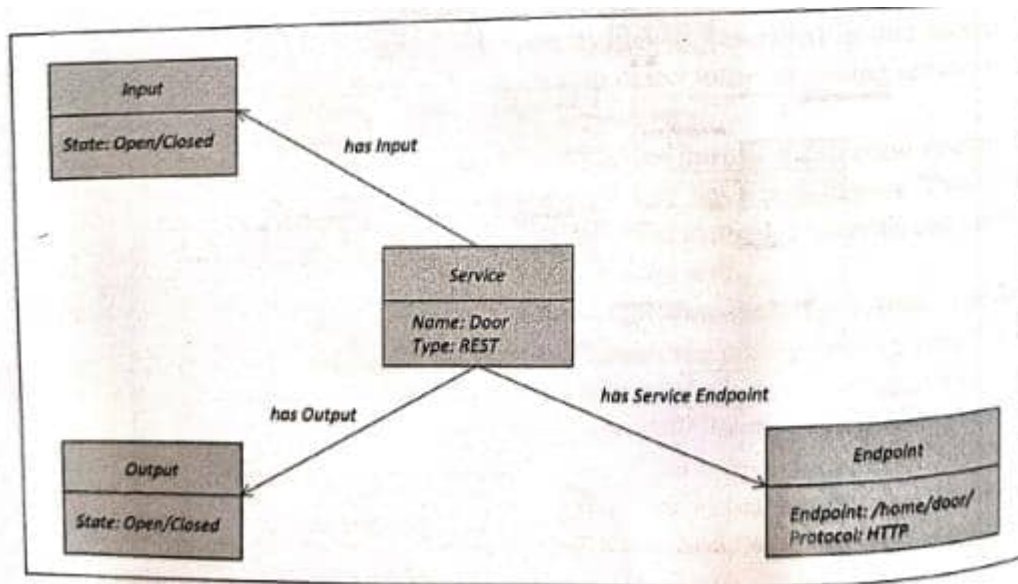


Figure 4.20:

Service specification for the home intrusion detection IoT system - door service

Figures 4.20, 4.21, and 4.22 show the specifications of these web services and the controller service.

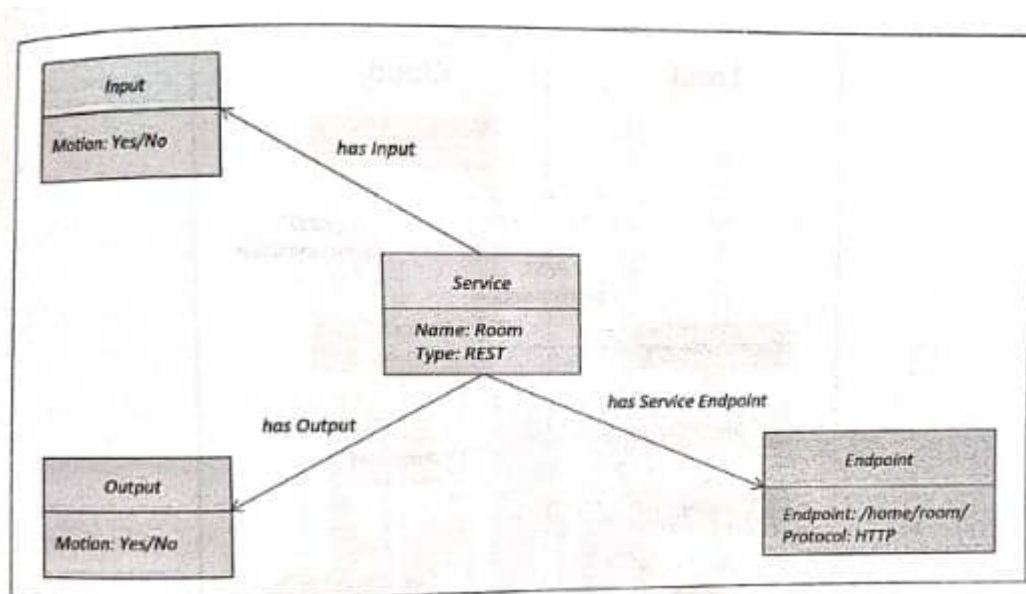


Figure 4.21: Service specification for the home intrusion detection IoT system - door service.

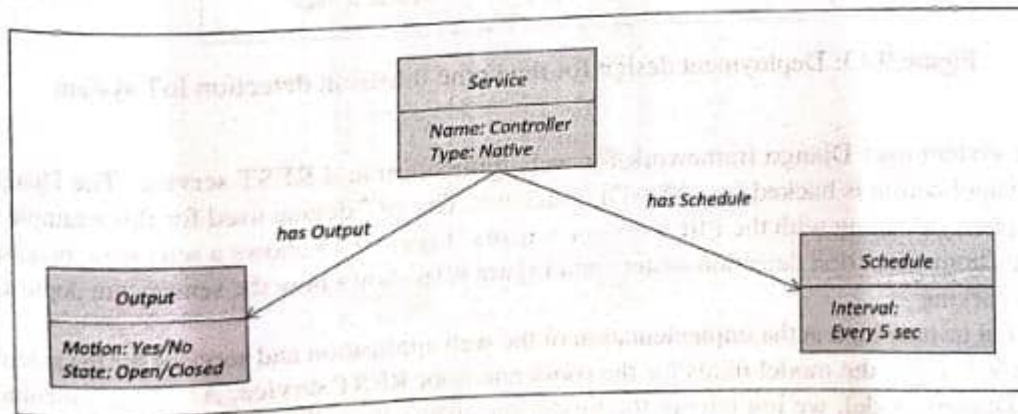


Figure 4.22: Controller service of the home intrusion detection IoT system

Figure 4.23 illustrates the deployment design of the home intrusion detection system, which is categorized as a Level-2 IoT system. The functional and operational views of the system are presented in Figure 4.24, showing the different deployment and operation options and how they are mapped to their respective functional groups. The system is implemented using the Django framework for both the web application and REST services, with a MySQL database serving as the backend. The IoT setup includes a Raspberry Pi connected to PIR and door sensors. Figure 4.25 provides a schematic diagram of the system, while Figure 9.16 shows the sensor deployment in a parking area.

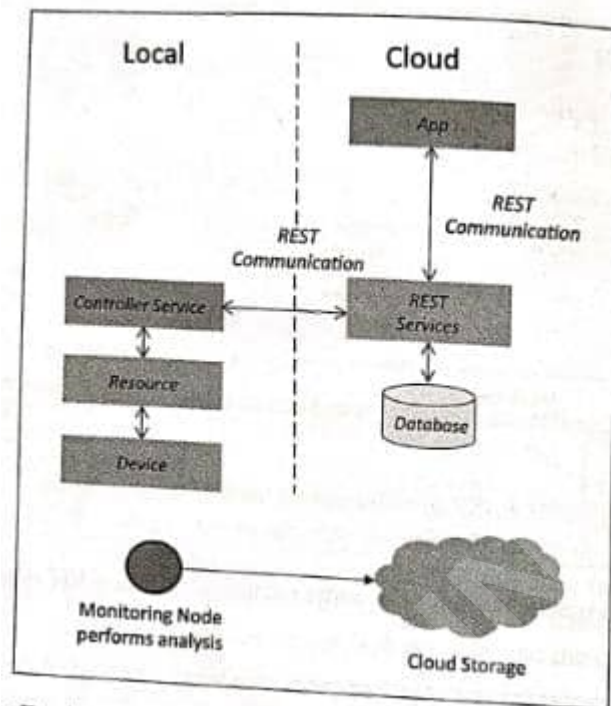


Figure 4.23: Deployment design for the home intrusion detection IoT system

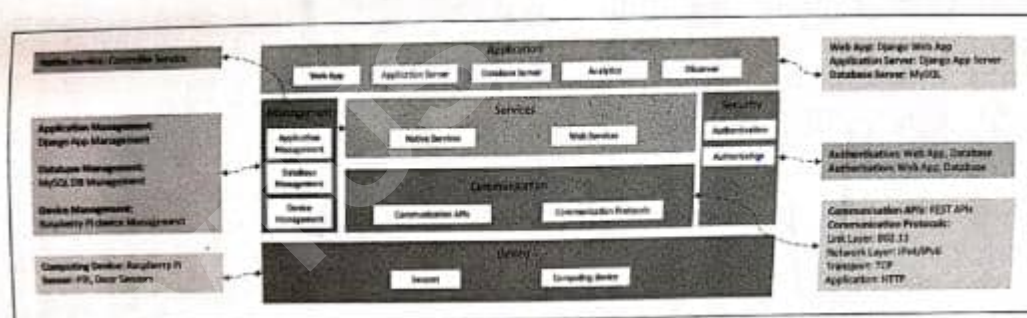


Figure 4.24: Functional & operational view of Specifications for home instruction detection System

For implementation of the web application and services for the system,

Step1 presents the model fields for the room and door REST service.

Step2 presents After implementing the Django model, we implement the model serializers that allows model instances to be converted to native Python datatypes. Serializers for room and door REST services

Step 3 presents Django views for REST services and home intrusion detection application. The home view is responsible for rendering the home page of the intrusion detection application, which displays the current status of each room. A request is sent to the state REST service to retrieve the latest state information for each room.

Step 4 presents Django URL patterns for REST services and home intrusion. Because ViewSets are used instead of regular views, the URL configuration is generated automatically by registering the viewsets with a router.

Step 5 presents Django template for home intrusion detection, which is rendered through the home view

Step 6 presents the Python implementation of the native controller service running on the Raspberry Pi. The runController function executes every second, reads data from the PIR and door sensors, and updates the room and door states by sending a PUT request to the respective REST services.

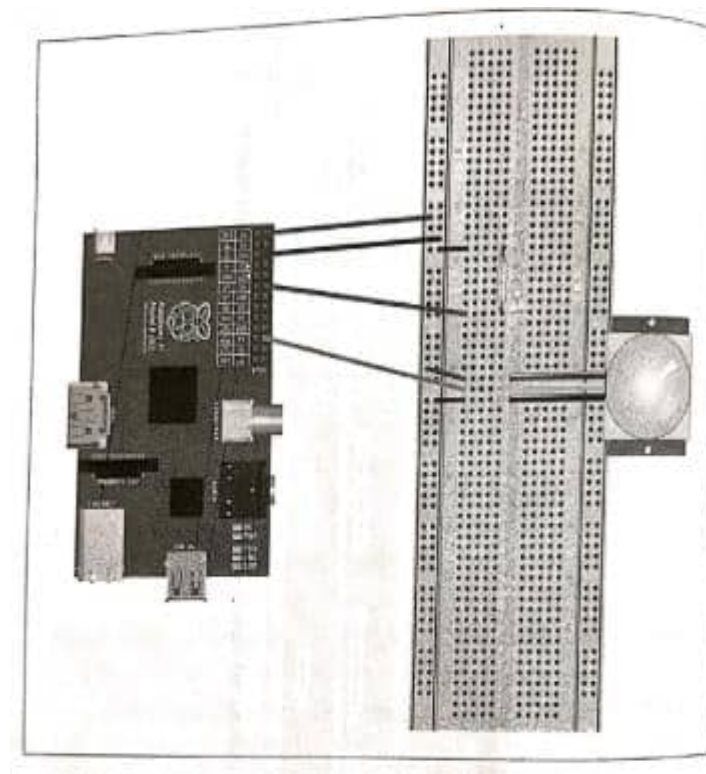


Figure 4.25: Schematic diagram of the home intrusion detection IoT system prototype, showing the device and ultrasonic sensor.

4.8 Cities

4.8.1 Smart Parking

Smart parking is an IoT-based system designed to help drivers easily find empty parking slots in a city. Each parking slot is fitted with a sensor that can detect whether the slot is empty or occupied. These sensors continuously collect data and send it to a local controller.

The local controller gathers information from all the sensors and forwards it through the Internet to a central server. The server processes this data and updates the smart parking application. Drivers can use this application on their smartphones, tablets, or in-car navigation systems to view real-time availability of parking spaces.

This system helps reduce traffic congestion, saves time for drivers, and makes city transportation more efficient.

Figure 4.26. illustrates how the smart parking system works, where each parking slot has an ultrasonic sensor placed above it to detect whether a vehicle is present. These sensors are checked regularly, and the system updates the database with the current status of each slot as either empty or occupied.

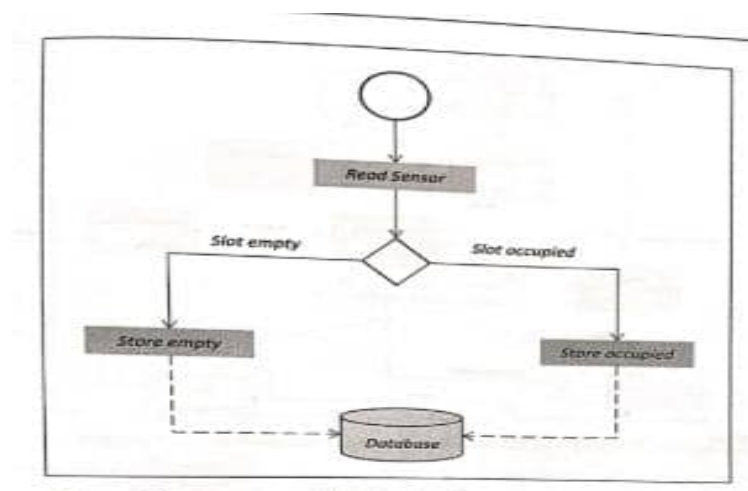


Figure 4.26: Process specification for the smart parking IoT system

Figure 4.27 presents the domain model, which includes the physical parking slot, its virtual representation, and a single-board mini computer connected to the ultrasonic sensor that functions as the device. It also shows the services that support the system's operation.

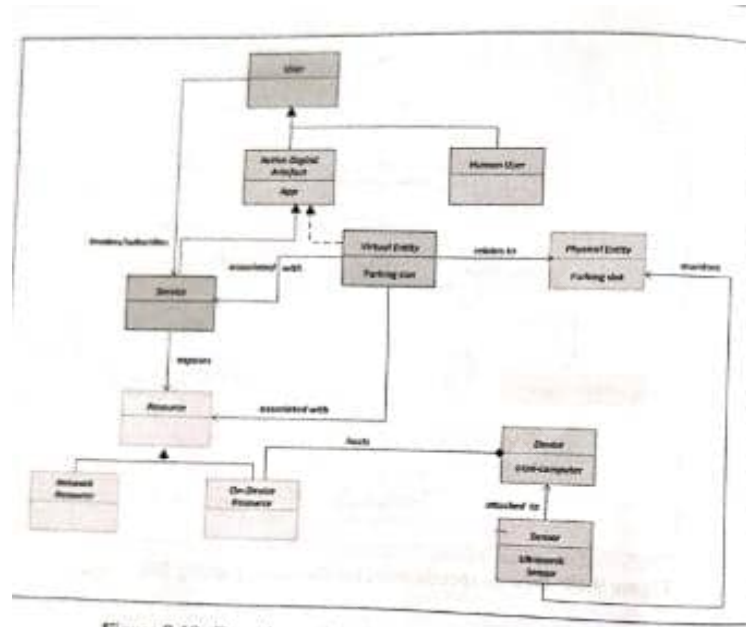


Figure 4.27: Domain model of the smart parking IoT system

Figure 4.28 displays the information model, which specifies that the virtual parking slot has one key attribute its state with only two possible values: empty or occupied.

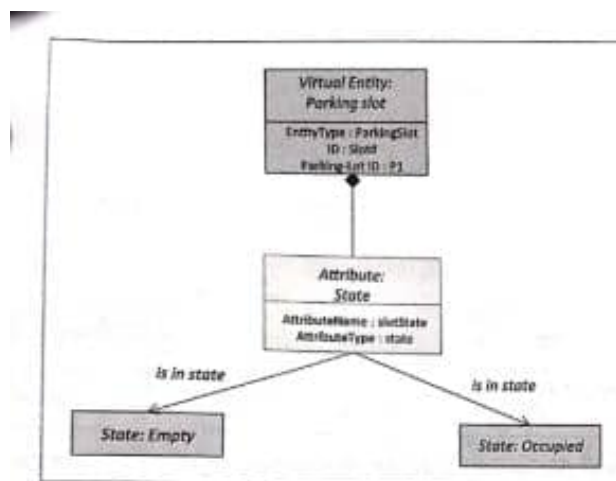


Figure 4.28: Information model of the smart parking IoT system

The next step in building the smart parking system is to define the service specifications, which are created from the process specification and the information model. The system provides two main services: one service monitors

each parking slot using ultrasonic sensors and updates the status to a cloud database through a REST web service, and the second service retrieves the current slot status through a controller native service. Figures 4.29 and 4.30 describe these controller and state services. The functional and operational views of this system follow the same pattern as the views used in the home intrusion detection system shown in Figure 4.24. The smart parking web application and REST services are developed using the Django framework, supported by a MySQL database. The IoT device used is a Raspberry Pi connected to ultrasonic sensors. Figure 4.31 shows how sensors are placed in the parking area, and Figure 4.32 provides a schematic diagram of the entire system. The implementation begins with creating the web application and services. Box 9.13 describes the model fields for the state REST service, and after defining the Django model, a serializer is implemented to convert model data into Python native types. Box 9.14 shows the serializer used for the state REST service.

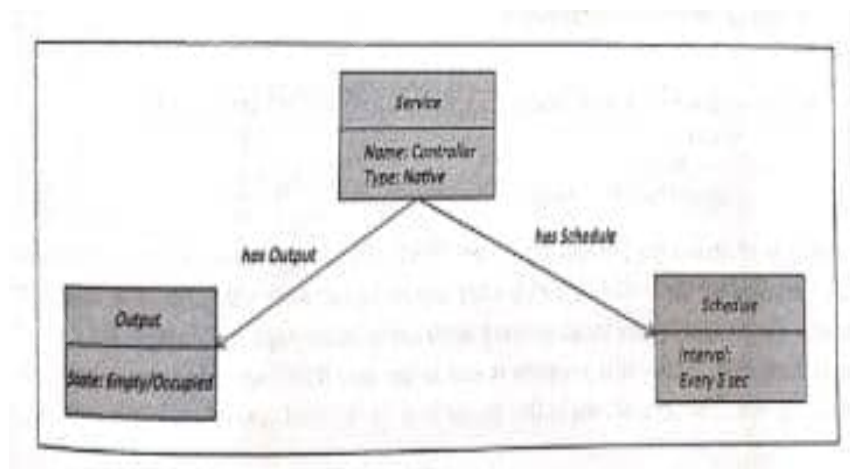


Figure 4.29: Controller service of the smart parking IoT system

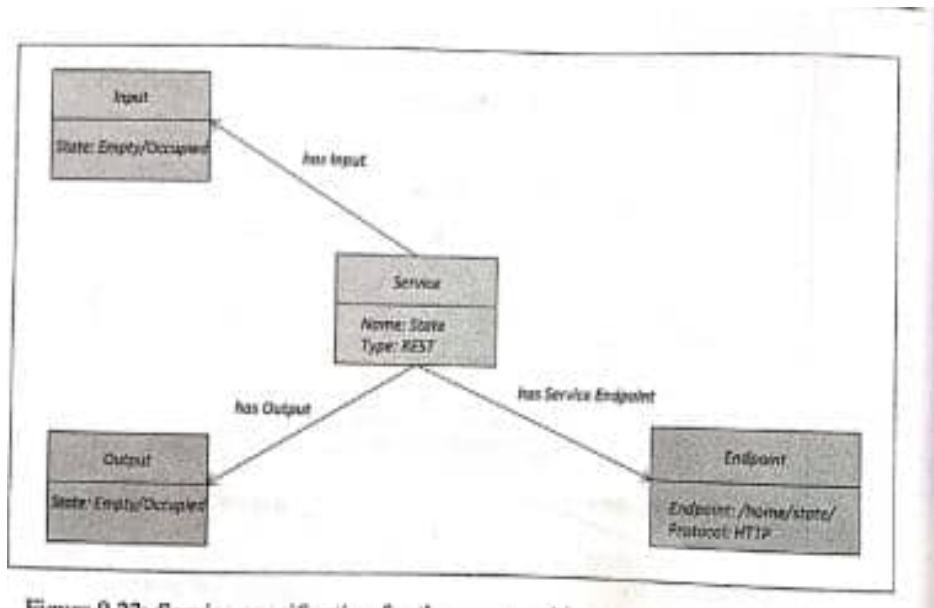


Figure 4.30: Service specification for the smart parking IoT system - state service

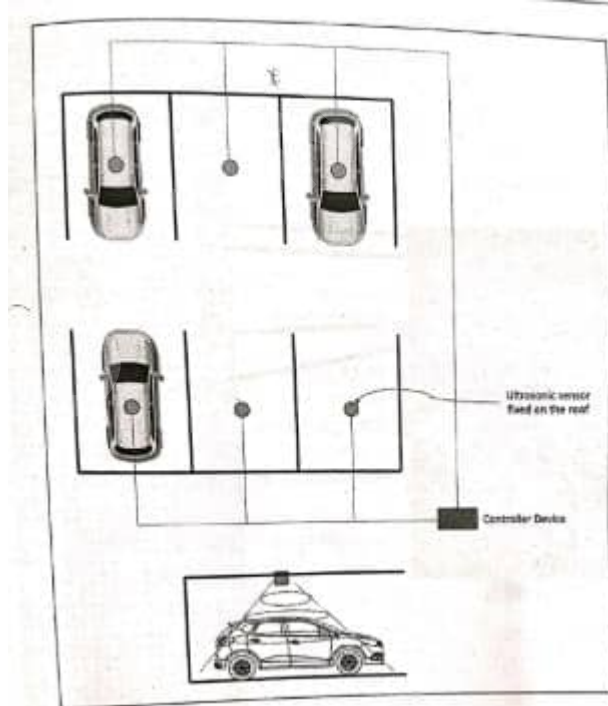


Figure 4.31: Deployment of sensors for smart parking system

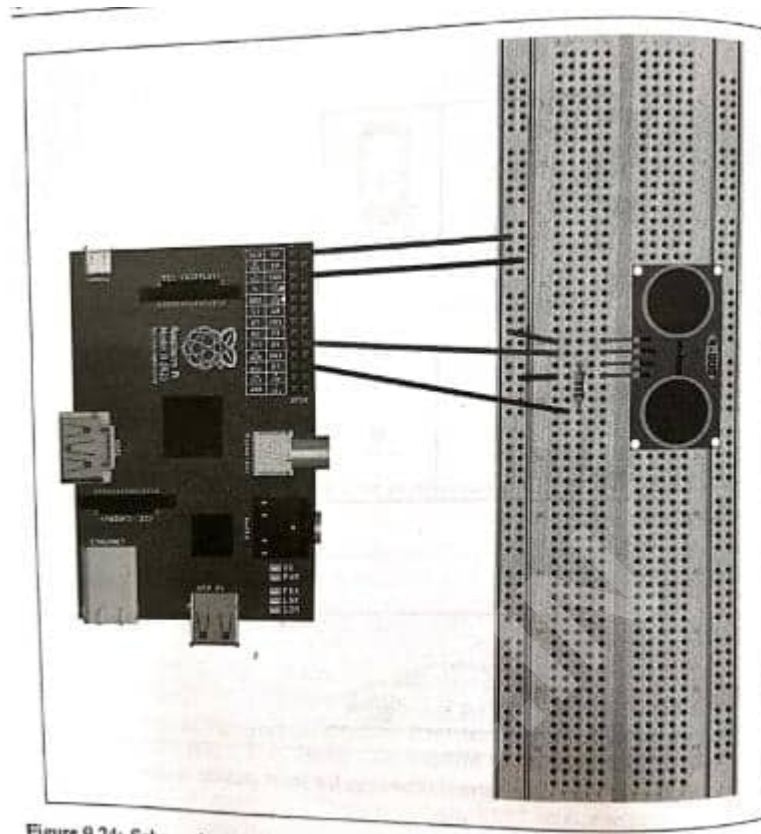


Figure 4.32: Schematic diagram of the smart parking IoT system prototype, showing the device and ultrasonic sensor

9.5 Agriculture

9.5.1 Smart Irrigation

Smart irrigation is an IoT-based system that uses soil moisture sensors to measure the moisture level in the soil and supply water only when the moisture drops below a set limit. The sensor data is also stored in the cloud, where it can be analyzed to plan proper watering schedules. In this section, an example implementation of a smart irrigation system is explained. Its deployment design is similar to the one shown in Figure 9.26, with several nodes placed at different points in the field to monitor soil moisture. These end nodes send the collected data to the cloud, where it is stored in a database and displayed through a cloud-based application. Figure 4.33 shows the structure of a typical end node, which consists of a Raspberry Pi mini-computer and a soil moisture sensor. A solenoid valve is used to manage the flow of water through the irrigation pipe, and it opens automatically when the soil

moisture level falls below the threshold. Box 4.5 includes the Python code used for the controller native service of the smart irrigation system.

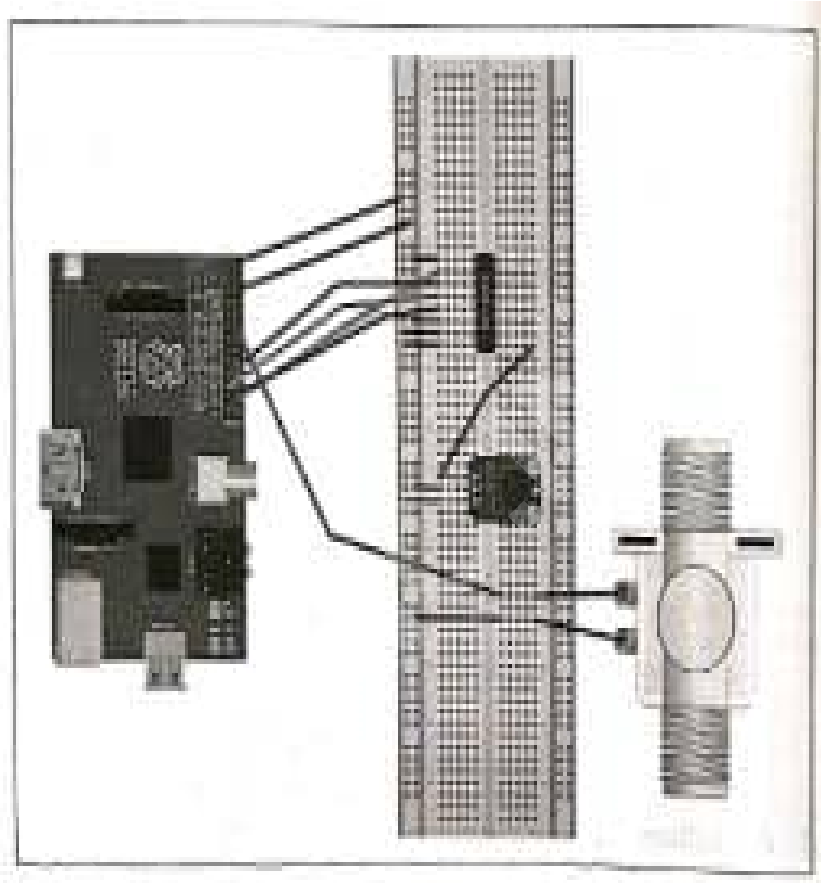


Figure 4.33: Schematic diagram of a smart irrigation system end-node showing the device and sensor

Box 4.5: Python code for controller native service - smart irrigation system

```
import time
import datetime
import sys
import json
import requests
import xively
import subprocess
import spidev
```

```

global mositure_datastream
#LDR channel on MCP3008
mositure_channel = 0
GPIO.setmode (GPIO.BCM)
TRIGGER_PIN=18
threshold = 10
# Open SPI bus
spi = spidev.SpiDev()
spi.open (0,0)
#Initialize Xively Feed
FEED_ID ="enter feed id"
API_KEY ="enter key"
api = xively.XivelyAPIClient (API KEY)
# Function to read SPI data from MCP3008 chip
def ReadChannel (channel):
    adc = spi.xfer2([1, (8+channel) «4, 0])
    data = ((adc[1]&3).« 8) + adc[2]
    return data
#Function to read sensor connected to MCP3008
def readMositure():
    level = ReadChannel (mositure_ channel)
    return level
#Controller main function
def runController():
    global mositure_datastream
    level=readMositure ()
    #Check moisture level
    if (level<threshold) :

```

```

        GPIO.output (TRIGGER_PIN, True)
    else:
        GPIO.output (TRIGGER_PIN, False)
    mositure_datastream.current_value = level
    mositure_datastream.at = datetime.datetime.utcnow ()
    print "Updating Xively feed with mositure: %s" % mositure
    try:
        mositure_datastream.update ()
    except requests.HTTPError as e:
        print "HTTPError( {0}): {1}".format (e.errno, e.strerror)
#Function to get existing or create new Xively data stream
def get_mosituredatastream (feed) :
    try:
        datastream = feed.datastreams.get ("mositure")
        return datastream
    except:
        datastream = feed.datastreams.create("mositure", tags="mositure")
        return datastream
#Controller setup function
def setupController():
    global mositure_datastream
    feed = api.feeds.get (FEED_ID)
    feed.location.lat="30.733315"
    feed.location.lon="76.779418"
    feed.tags="Soil Moisture"
    feed.update ()
    mositure_datastream = get_mosituredatastream (feed)
    mositure_datastream.max_value = None

```

```
    moisture_datastream.min_value = None
setupController ()
while True:
    runController()
    time.sleep (10)
```