Unit 4

1. Building blocks of IoT devices.

IoT Device A "Thing" in Internet of Things (IoT) can be any object that has a unique identifier and which can send/receive data (including user data) over a network (e.g., smart phone, smart TV, computer, refrigerator, car, etc.).

IoT devices are connected to the Internet and send information about themselves or about their surroundings (e.g., information sensed by the connected sensors) over a network (to other devices or servers/storage) or allow actuation upon the physical entities/environment around them remotely.

Example: A home automation device that allows remotely monitoring the status of appliances and controlling the appliances.

An IoT device can consist of a number of modules based on functional attributes, such as:

- Sensing: Sensors can be either on-board the IoT device or attached to the device. IoT device can collect various types of information from the on-board or attached sensors such as temperature, humidity, light intensity, etc. The sensed information can be communicated either to other devices or cloud-based servers/storage.
- Actuation: IoT devices can have various types of actuators attached that allow taking actions upon the physical entities in the vicinity of the device. For example, a relay switch connected to an IoT device can turn an appliance on/off based on the commands sent to the device.
- Communication: Communication modules are responsible for sending collected data to other devices or cloud-based servers/storage and receiving data from other devices and commands from remote applications.
- Analysis & Processing: Analysis and processing modules are responsible for making sense of the collected data.

Figure 7.1 shows a generic block diagram of a single-board computer (SBC) based IoT device that includes CPU, GPU, RAM, storage and various types of interfaces and peripherals.

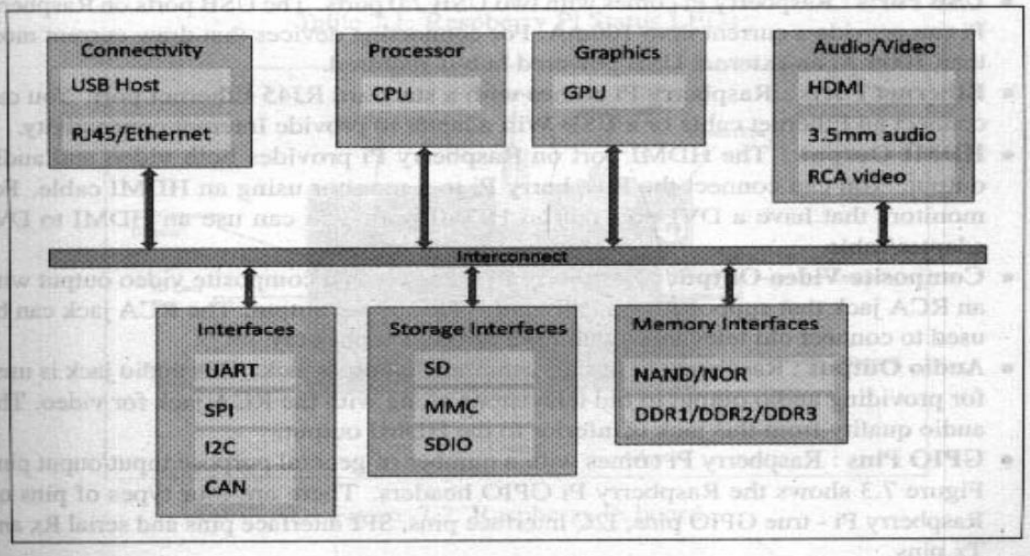


Figure 7.1: Block diagram of an IoT Device

2. Raspberry Pi Interface

7.5 Raspberry Pi Interfaces

Raspberry Pi has serial, SPI and I2C interfaces for data transfer as shown in Figure 7.3.

7.5.1 Serial

The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.

7.5.2 SPI

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices. In an SPI connection, there is one master device and one or more peripheral devices. There are five pins on Raspberry Pi for SPI interface:

- **MISO (Master In Slave Out)** : Master line for sending data to the peripherals.
- **MOSI (Master Out Slave In)** : Slave line for sending data to the master.
- **SCK (Serial Clock)** : Clock generated by master to synchronize data transmission
- **CE0 (Chip Enable 0)** : To enable or disable devices.
- **CE0 (Chip Enable 1)** : To enable or disable devices.

7.5.3 I2C

The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clock line).

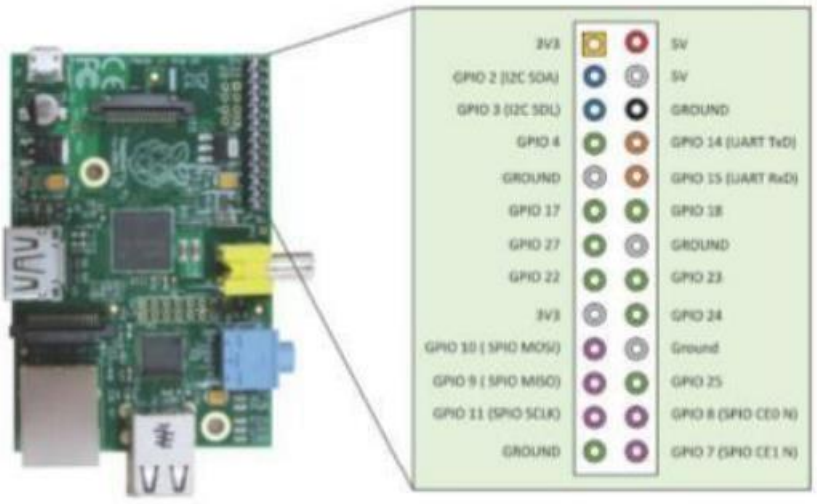
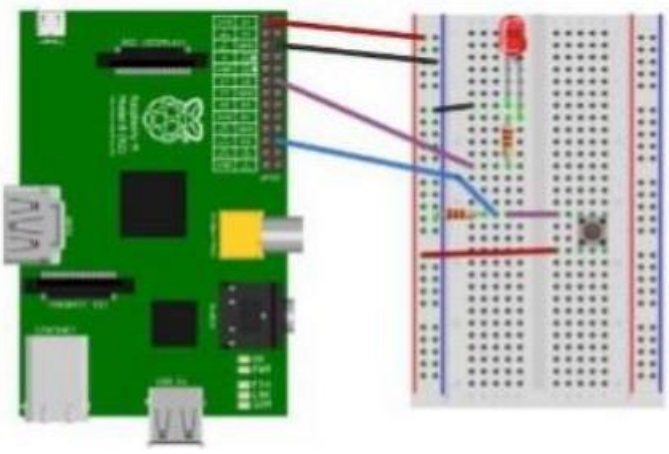


Figure 7.3: Raspberry Pi GPIO headers.

3. Interfacing Led and switch program with raspberry pi

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

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

#LEDPin
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 ()
```



Unit 5

1. Django Architecture and Python web Application frame work

Django is an open-source web application framework for developing web applications in Python. A web application framework in general is a collection of solutions, packages and best practices that allows development of web applications and dynamic websites. Django is based on the Model-Template-View architecture and provides a separation of the data model from the business rules and the user interface. Django provides a unified API to a database backend. Thus, web applications built with Django can work with different databases without requiring any code changes. With this flexibility in web application design combined with the powerful capabilities of the Python language and the Python ecosystem,
Django is best suited for cloud applications. Django consists of an object-relational mapper, a web templating system and a regular-expression based URL dispatcher.

Django Architecture

Django is Model-Template-View (MTV) framework.

1. **Model**: The model acts as a definition of some stored data and handles the interactions with the database. In a web application, the data can be stored in a relational database, non-relational database, an XML file, etc. A Django model is a Python class that outlines the variables and methods for a particular type of data.

2. **Template**: In a typical Django web application, the template is simply an HTML page with a few extra placeholders. Django‘s template language can be used to create various forms of text files (XML, email, CSS, JavaScript, CSV,etc.)

3. **View**: The view ties the model to the template. The view is where you write the code that actually generates the web pages. View determines what data is to be displayed, retrieves the data from the database and passes the data to the template.

2. Amazon Web services for IOT (or) Amazon EC2, Autoscaling, S3, RDS

Amazon EC2

Python Example Boto is a Python package that provides interfaces to Amazon Web Services (AWS). In this example, a connection to EC2 service is first established by calling boto.ec2.connect_to_region. The EC2 region, AWS access key and AWS secret key are passed to this function. After connecting to EC2, a new instance is launched using the conn.run_instances function. The AMI-ID, instance type, EC2 key handle and security group are passed to this function**.**

```
#Python program for launching an EC2 instance
import boto.ec2
from time import sleep
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

conn = boto.ec2.connect_to_region(REGION, aws_access_key_id=ACCESS_KEY,
                                  aws_secret_access_key=SECRET_KEY)

reservation = conn.run_instances(image_id=AMI_ID, key_name=EC2_KEY_HANDLE,
                                 instance_type=INSTANCE_TYPE,
                                 security_groups = [ SECGROUP_HANDLE, ] )
```

Amazon autoscaling:

8.6.2 Amazon AutoScaling

Amazon AutoScaling allows automatically scaling Amazon EC2 capacity up or down according to user defined conditions. Therefore, with AutoScaling users can increase the number of EC2 instances running their applications seamlessly during spikes in the application workloads to meet the application performance requirements and scale down capacity when the workload is low to save costs. AutoScaling can be used for auto scaling IoT applications and IoT platforms deployed on Amazon EC2.

Steps to creating an Autoscaling group:

1. **AutoScaling Service**: A connection to AutoScaling service is first established by calling boto.ec2.autoscale.connect_to_regionfunction.
2. **Launch Configuration**: After connecting to AutoScaling service, a new launch configuration is created by calling conn.create_launch_con f iguration. Launch configuration contains instructions on how to launch new instances including the AMIID, instance type, security groups,etc.
3. **AutoScaling Group**: After creating a launch configuration, it is then associated with a new AutoScaling group. AutoScaling group is created by calling conn.create_auto_scaling_group. The settings for AutoScaling group such as the maximum and minimum number of instances in the group, the launch configuration, availability zones, optional load balancer to use with the group,etc.

Amazon AutoScaling – Python Example

```
#Creating auto-scaling policies
scale_up_policy = ScalingPolicy(name='scale_up',
        adjustment_type='ChangeInCapacity',
        as_name='My-Group',
        scaling_adjustment=1,
        cooldown=180)
scale_down_policy =ScalingPolicy(name='scale_down',
        adjustment_type='ChangeInCapacity',
        as_name='My-Group', scaling_adjustment=-1,
        cooldown=180)
conn.create_scaling_policy(scale_up_policy)
conn.create_scaling_policy(scale_down_policy)
```

Amazon S3

Amazon S3 is an online cloud-based data storage infrastructure for storing and retrieving a very large amount of data. S3 provides highly reliable, scalable, fast, fully redundant and affordable storage infrastructure. S3 can serve as a raw datastore (or "Thing Tank") for IoT systems for storing raw data, such as sensor data, log data, image, audio and video data.

Let us look at some examples of using S3. Box 8.26 shows the Python code for uploading a file to Amazon S3 cloud storage. In this example, a connection to S3 service is first established by calling *boto.connect_s3* function. The AWS access key and AWS secret key are passed to this function. This example defines two functions *upload_to_s3_bucket_path* and *upload_to_s3_bucket_root*. The *upload_to_s3_bucket_path* function uploads the file to the S3 bucket specified at the specified path. The *upload_to_s3_bucket_root* function uploads the file to the S3 bucket root.

Box 8.26: Python program for uploading a file to an S3 bucket

```
import boto.s3

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

conn = boto.connect_s3(aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

def percent_cb(complete, total):
   print ('.')

def upload_to_s3_bucket_path(bucketname, path, filename):
   mybucket = conn.get_bucket(bucketname)
   fullkeyname=os.path.join(path,filename)
   key = mybucket.new_key(fullkeyname)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

def upload_to_s3_bucket_root(bucketname, filename):
   mybucket = conn.get_bucket(bucketname)
   key = mybucket.new_key(filename)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

upload_to_s3_bucket_path('mybucket2013', 'data', 'file.txt')
```

Amazon RDS:

Amazon RDS is a web service that allows you to create instances of MySQL, Oracle or Microsoft SQL Server in the cloud. With RDS, developers can easily set up, operate, and scale a relational database in the cloud.

RDS can serve as a scalable datastore for IoT systems. With RDS, IoT system developers can store any amount of data in scalable relational databases. Let us look at some examples of using RDS. Box 8.27 shows the Python code for launching an Amazon RDS instance. In this example, a connection to RDS service is first established by calling *boto.rds.connect_to_region* function. The RDS region, AWS access key and AWS secret key are passed to this function. After connecting to RDS service, the *conn.create_dbinstance* function is called to launch a new RDS instance. The input parameters to this function include the instance ID, database size, instance type, database username, database password, database port, database engine (e.g. MySQL5.1), database name, security groups, etc. The program shown in Box 8.27 waits till the status of the RDS instance becomes *available* and then prints the instance details such as instance ID, create time, or instance end point.

■ Box 8.27: Python program for launching an RDS instance

```
import boto.rds
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
INSTANCE_TYPE="db.t1.micro"
ID = "MySQL-db-instance"
USERNAME = 'root'
PASSWORD = 'password'
DB_PORT = 3306
DB_SIZE = 5
DB_ENGINE = 'MySQL5.1'
DB_NAME = 'mytestdb'
SECGROUP_HANDLE="default"

print "Connecting to RDS"

conn = boto.rds.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Creating an RDS instance"

db = conn.create_dbinstance(ID, DB_SIZE, INSTANCE_TYPE,
USERNAME, PASSWORD, port=DB_PORT, engine=DB_ENGINE,
db_name=DB_NAME, security_groups = [
SECGROUP_HANDLE, ] )
print db

print "Waiting for instance to be up and running"

status = db.status
while not status == 'available':
   sleep(10)
   status = db.status

if status == 'available':
   print " \n RDS Instance is now running.  Instance details are:"
   print "Intance ID: " + str(db.id)
   print "Intance State:  " + str(db.status)
   print "Intance Create Time:  " + str(db.create_time)
   print "Engine:  " + str(db.engine)
   print "Allocated Storage:  " + str(db.allocated_storage)
   print "Endpoint:  " + str(db.endpoint)
```