**INTERNET CONTROLLED RANGELESS REAL-TIME SURVEILLANCE ROBOT WITH ANDROID APPLICATION**

****

**INSTITUTE OF ENGINEERING AND MANAGEMENT**

**ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT**

**presented by**

**AVISHEK BISWAS**

**(ROLL NO: 10400312085, REGN. NO: 121040110279)**

**GUIDED BY:**

**PROFESSOR RATNA CHAKRABARTY**

**Assistant Professor, Electronics and Communication Engineering,**

**Institute of Engineering and Management**

CERTIFICATION

This is to certify that the project report entitled “INTERNET CONTROLLED RANGELESS REAL TIME SURVEILLANCE ROBOT WITH ANDROID APPLICATION” submitted by Avishek Biswas (University Roll: 10400312085) student of Institute of Engineering and Management(IEM), in fulfilment of degree of Bachelor of Technology in Electronics and Communication Engineering(ECE) is a bonafide work carried out by them under the supervision and guidance of Assistant Professor Ratna Chakrabarty, department of Electronics and Communication Engineering. The content of this report has not been submitted to any other university or institute for the award of any other degree.

I am glad to inform that the work is entirely original and it’s performance found to be quite satisfactory.

**Ratna Chakrabarty Amlan Kusum Nayak**

Assistant Professor and Assistant Head of the Department, The Principal,

Electronics and Communication Engineering Department, Institute of Engineering and Management

Institute of Engineering and Management.

**Dr Malay Gangopadhyay**

Professor and Head of the Department,

Electronics and Communication Engineering Department,

Institute of Engineering and Management.

DECLARATION

The entire document of this project is complete on its own and authorized by Institute of Engineering and Management, Kolkata. In many cases we took help of various application notes circulated on the internet by the manufacturer.

I confirm that no data or figure presented in this document is privileged under the intellectual property or copyright act. The author stands in strong support for the originality of this project.

ABSTRACT

The goal of this project is to make a range-less robotic car with a surveillance camera. This means, that it can control the movement of this robot from anywhere in the world at anytime through internet. The robot will have an Android phone used as a camera source attached to it. We will develop an application in this phone such that it streams real time footage to our media server. We will develop another client-based Android Application to retrieve the live footage from our Media Server and also send commands to the robot via MQTT server which will make it move accordingly. The distance between the controller app and the robot is not a factor as long as both devices have internet access.

ACKNOWLEDGEMENT

I would like to thank my parents, Dr Amitava Biswas and Mrs Tapashi Biswas for their undying support and well wishes. I would be nowhere without them. Being part of such a project was a great learning curve for me, and I am really excited about the future application of this effort. Without the tutelage and guidance of our mentor, Professor Ratna Chakraborty, I could not have made it this far in this project. Lastly, I would like to thank my friends Abhishek Dey and Sudip Chakrabarty for continuously inspiring me and motivating me to try my best at everything I attempt.

-- Avishek Biswas

CONTENTS

1. **INTRODUCTION** 1.1-1.4  
     
   THEORY
2. **MQTT** 2.1-2.17
3. **NODE MCU** 3.1-3.18
4. **TRANSPORT PROTOCOL AND RTSP** 4.1-4.7  
     
     
   WORKING
5. **SETTING UP CLOUD BASED MQTT SERVER** 5.1
6. **SETTING UP WOWZA SERVER** 6.1-6.3
7. **SETTING UP ROBOT CHASSIS AND MOTOR DRIVER** 7.1-7.3
8. **SETTING UP THE NODE MCU** 8.1-8.11
9. **ANDROID APPLICATION FOR LIVE STREAMING** 9.1-9.6
10. **ANDROID APPLICATION FOR MQTT PUBLISHING AND VIEWING LIVE STREAM** 10.1-10.14  
      
      
    RESULTS AND DISCUSSION
11. **BRINGING EVERYTHING TOGETHER** 11.1-11.4
12. **IN CONCLUSION** 12.1
13. **DISCUSSION AND FUTURE WORK** 13.1
14. **REFERENCE** 14.1
15. **APPENDIX** App.1-App.12

LIST OF FIGURES

1. 1.1: Basic block diagram of project
2. 2.1: MQTT co-located server
3. 2.2: Independently located nodes MQTT server
4. 2.3:Message format for control packet
5. 2.4:Header structure
6. 2.5: Flags for publish packet
7. 2.6:Mqtt binding
8. 2.7:Request/response over MQTT
9. 3.1:NodeMCU
10. 3.2:Pin Configuration of NodeMCU
11. 3.3:Pin Layout
12. 3.4:Functional Description of NodeMCU
13. 3.5:Power Management
14. 3.6:USB to Serial Chip
15. 3.7: Package IC Configuration
16. 3.8: Internal Architecture of Node MCU
17. 4.1:UDP
18. 4.2:TCP Overhead Details
19. 4.3:RTSP Handshake
20. 6.1:Wowza Streaming Engine
21. 6.2: Wowza Cloud
22. 7.1: Robot Chassis
23. 7.2:Wheel
24. 7.3:Motor
25. 7.4: L293D pin diagram
26. 8.1: Flashing the NodeMCU
27. 8.2: Motor Driver Module
28. 8.3:Battery Holder
29. 8.4: NodeMCU Power Supply
30. 8.5: Connecting NodeMCU and Motor Driver
31. 8.6: Circuit Diagram
32. 10.1: Android APP UI
33. 11.1: Android APP MQTT CONTROL
34. 11.2: Android APP live streaming
35. 11.3: Android App with Live streaming and Media Controls
36. 11.4: Detailed block diagram.

LIST OF TABLES

1. 2.1: Protocol segments for co-located node
2. 2.2: protocol segment for independently located nodes
3. 3.1:Technical specifications of NodeMCU
4. 3.2:Minimum Flash Memory
5. 3.3: Frequency Channel
6. 3.4: UART Pin Definition
7. 3.5:Pin Definition of PWM
8. 3.6:Pin Definition of IR Control
9. 8.2:NodeMCU Motor Driver Connections
10. 8.3: How Robot Moves?

# INTERNET CONTROLLED RANGELESS SURVEILLANCE ROBOT WITH ANDROID APPLICATION

The goal of our project is to make a range-less robotic car with a spy-cam. This means, that we can control the movement of this robot from anywhere in the world at anytime through internet. The robot will have a camera source attached at it's front, which will enable us to view the live footage of what the robot is "seeing". We will also create an android application that will help us control the movements of the robot and view the live streaming in real time.

Now, what do we mean by range-less? Technically it should mean that this robot can be controlled from anywhere at any time. Sadly, this is only partially true. As mentioned in the title, our robot only works in presence of internet. The motherboard of our robot should connect to the WiFi of the location to function properly. Furthermore, the client who sends the command to the server that facilitates movement of the robot also requires internet connection. If both of our client (the robot and the android user) has connection to the internet, our robot is perfectly range-less. One can be separated across continents, and the robot will still work.

In this introductory essay, we will just outline the different concepts, hardware and software we need to get our job done.

**Concepts discussed:**

* **MQTT (Message Queue Telemetry Transport):** This is a subscribe/publish based messaging protocol that we used to interact with the robot via internet.
* **Transport Protocols and RTSP (Real Time Streaming Protocol):** We use RTSP to stream live footage from android camera to our media server. We later retrieve this footage to our client app using RTMP(Real Time Messaging Protocol).

**Hardware requirements:**

* **NODE MCU(ESP8266):** This is the brain of all our operations. A very powerful chip, this acts as a WiFi module that can connect to the WiFi of the location. Also, this module acts as a motherboard to our system by reading input data from our MQTT server and processing it accordingly.
* **Motor Driver(L293D):** This is a very powerful motor driver that is used to drive DC motors.
* **DC Motors:** Since we are making a movable robot, we need motors to facilitate the rotation of the wheels. We are using two 12V 200rpm motors.
* **Robotic Car Chassis:** We are using a readymade robot car chassis, with a 19.5mX10.5m body, two wheels and an accessory front wheel.
* **Power Supply:** We need two power sources, one 12V source to run our Motor Driver and another 5V source to power our Node MCU.
* **Android Phone to act as IP Camera:** We are using an android phone as a source camera to stream live footage to our media server in real time. It is required that this phone supports Android 4.4 or higher. We will create an application that conducts the live streaming directly.
* **Android Phone to provide Input and view live stream:** This is our main client interface. We need an android phone to connect to our MQTT server and publish messages that the Node MCU can read to process the Motor Driver input. We will also view the live stream in this application. This phone therefore acts as both an input and output device.
* **Connecting wires**

**SOFTWARES AND OTHER NETWORK REQUIREMENTS:**

* **ESPlorer:** ESPlorer is an IDE (Integrated Development Environment) that we will need to code in the memory of our Node MCU chip. We discuss the .lua code in great details in the following chapters.
* **Cloud based MQTT server:** We will need this MQTT server to connect the client android app and the client robot.
* **Cloud based Media Server:** We will need a media server that will stream live videos across client-server and server-client ends. We have used Wowza Media Server to fulfill this requirement.
* **Eclipse with ADT (ANDROID DEVELOPER TOOLS):** To code our android application, we will need an IDE. Eclipse is our choice.
* **Some open source Android Libraries:** We need 3 open source libraries to simplify our project: libstreaming (we will use this to stream from Android to Server), vitamio(we will use open-source library to view live footage from our Media Server to a flash based player in the client app), mqtt-paho-client (an excellent open-source library to connect, subscribe and publish messages in our MQTT server).
* **WiFi:** Without WiFi, our Node MCU cannot access our MQTT server. In other words, nothing will work.

Having outlined all the units, we will proceed in the following modules to build a system out of these individual parts.

BASIC FLOWCHART

C:\Users\Avi\Downloads\Flowchar1.png

FIGURE 1.1: BASIC FLOWCHART

**THEORY**

**TS 118 110 V1.0.0 (2015-02)**

**CHAPTER 2**

**MQTT**

**Definitions and abbreviations**

Definitions

For the purposes of the present document, the following terms and definitions apply:

**originator** [**:**](#page6)actorthat initiates a Request

NOTE: An Originator can either be an Application or a CSE.

**receiver** [**:**](#page6)actorthat receives the Request

NOTE: A Receiver can be a CSE or an Application.

**resource** [**:**](#page6)uniquelyaddressable entity in oneM2M System such as by the use of a Universal Resource Identifier(URI)

NOTE: A resource can be accessed and manipulated by using the specified procedures.

**Abbreviations**

For the purposes of the present document, the abbreviations given in TS 118 101 and the following apply:

**ADN** Application Dedicated Nodebbgb

**ADN-AE** AE which resides in the Application Dedicated Node

**AE** Application Entity

**ASN** Application Service Node

**ASE-AE** A pplication Entity that is registered with the CSE in the Application Service Node

**ASN-CSE** CSE which resides in the Application Service Node

**CSE** Common Service Entity

**CSF** Common Service Function

**EF** Enabler Function

**IN** Infrastructure Node

**IN-AE** Application Entity that is registered with the CSE in the Infrastructure Node

**IN-CSE**  CSE which resides in the Infrastructure Node

**MN** Middle Node

**MN-CSE** CSE which resides in the Middle Node

**NSE** Network Service Entity

**TLS** Transport Level Security

MQTT is a light weight publish/subscribe messaging transport protocol, particularly well suited to event-oriented interactions. It was specifically designed for constrained environments such as those found in Machine to Machine (M2M) and Internet Of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium.

MQTT includes reliability features which allow recovery from loss of network connectivity without requiring explicit involvement of the applications that are using it, however it does require an underlying network protocol that provides ordered, lossless, bi-directional connections.

The features of MQTT include:

* The use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.
* Bidirectional communications. An entity can subscribe to receive messages without having a reliable IP address. This could be used to allow unsolicited requests to be sent to a Receiver, or an asynchronous response to be sent to an Originator, where the Originator or Receiver does not have an externally accessible IP address. It thus eliminates the need for long polling and can reduce the need for triggering.
* A messaging transport that is agnostic to the content of the payload. The message payload can be text or binary.
* A Session concept that can survive loss of network connectivity and can persist across multiple consecutive network connections. Messages can be stored and subsequently forwarded when connectivity is restored.
* Three levels of reliability (referred to as "qualities of service") for message delivery within a Session:
  + "At most once", where messages are delivered according to the best efforts of the operating environment. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
  + "At least once", where messages are assured to arrive but duplicates might occur. This is best suited to messages which have idempotent semantics.
  + "Exactly once", where message are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
* A small transport overhead and protocol exchanges designed to minimize network traffic, with consequent additional savings on battery power when compared to HTTP.
* A Retained Message option, allowing new subscribers to get the last message to have been published on a topic prior to their subscription.
* A mechanism to notify interested parties when an abnormal disconnection occurs.

Use of MQTT

This binding makes use of MQTT to provide reliable two-way communications between two parties (AEs and CSEs). It uses the following features of MQTT:

* Durable Sessions, providing Store and Forward in cases where network connectivity is not available.
* MQTT's "QoS 1" message reliability level. This provides reliability without incurring the overhead implied by QoS 2.
* NAT traversal (neither of the two parties is required to have prior knowledge of the other party's IP address).
* Dynamic topic creation and wild-carded subscription filters.

It does not use the following features:

* One-to-many publish/subscribe.
* Retained Messages.
* Will Messages.
* QoS 0 or QoS 2 message reliability levels.

**Binding overview**

The MQTT protocol binding specifies how the Mca or Mcc request and response messages are transported across the MQTT protocol. Both communicating parties (AEs and CSEs) typically make use of an MQTT client library, and the communications are mediated via the MQTT server. There is no need for the client libraries or the server to be provided by the same supplier, since the protocol they use to talk to each other is defined by the MQTT specification  [[1]](#page6).

Furthermore, the binding does not assume that the MQTT client libraries or server implementations are necessarily aware that they are being used to carry Mca, Mcc or any other oneM2M-defined primitives.

The binding is defined in terms of the MQTT protocol flows that take place between the client libraries and the MQTT server in order to effect the transport of an Mca or Mcc message.

There are two scenarios depending on the location of MQTT server: MQTT server co-located within a node, and MQTT server located independently from nodes.

1. **TS 118 110 V1.0.0 (2015-02)**

***Scenarios***

**MQTT server co-located within a node**

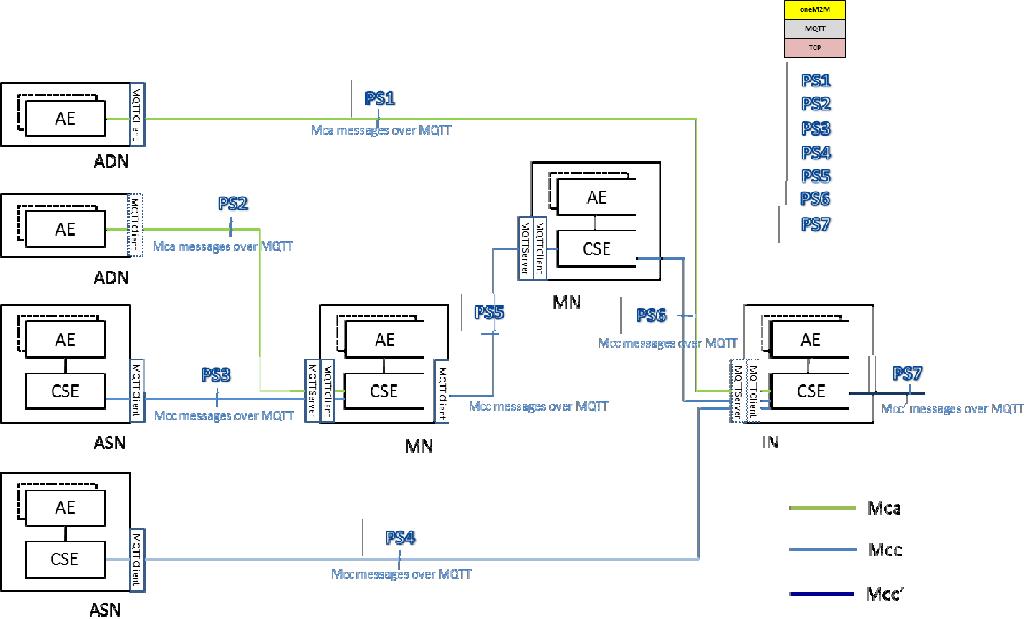


Figure 2.1: MQTT server co-located scenario

The above shows a protocol segment view of the MQTT server co-located scenario. In this scenario, all oneM2M nodes (ADN, ASN, MN, IN) include a MQTT client. MQTT servers are provided within MN and IN.

In this scenario, the protocol segments are illustrated as follows.

|  |  |  |
| --- | --- | --- |
| **Protocol Segment** | **oneM2M Message Transported** | **MQTT Interaction** |
| PS1 | Mca (AE of ADN to CSE of IN) | Client in ADN to Server in IN |
| PS2 | Mca (AE of ADN to CSE of MN) | Client in ADN to Server in MN |
| PS3 | Mcc (CSE of ASN to CSE of MN) | Client in ASN to Server in MN |
| PS4 | Mcc (CSE of ASN to CSE of IN) | Client in ASN to Server in IN |
| PS5 | Mcc (CSE of MN to CSE of MN) | Client in MN to Server in MN |
| PS6 | Mcc (CSE of MN to CSE of IN) | Client in MN to Server in IN |
| PS7 | Mcc' (CSE of IN to CSE of IN) | Client in IN to Server in IN |

Table 2.1: PROTOCOL SEGMENTS FOR MQTT SERVERS CO LOCATED IN A NODE

**MQTT server located independently from nodes**

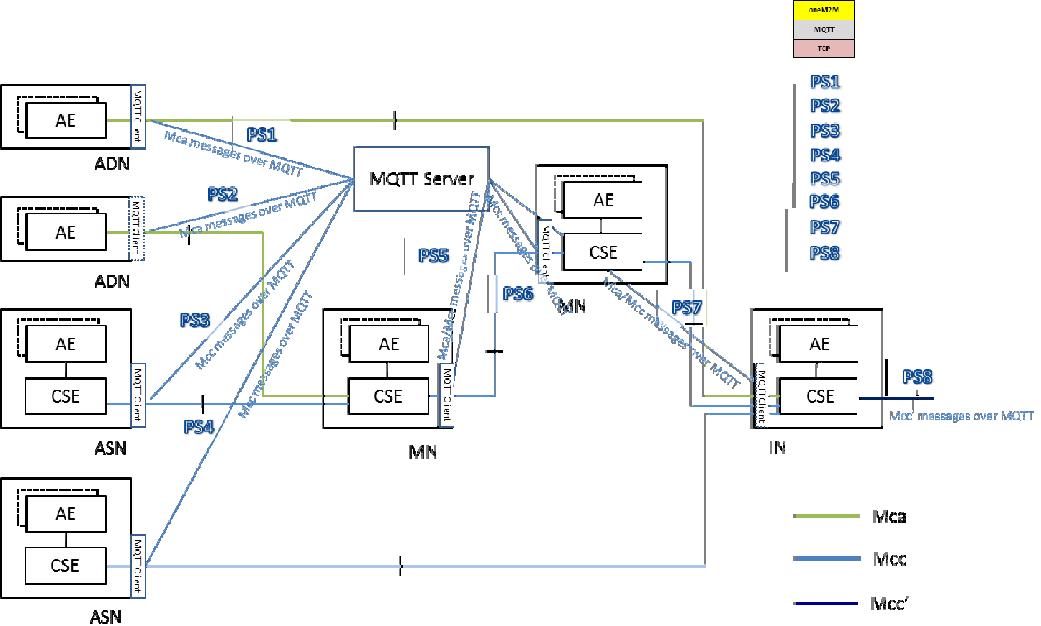


Figure 2.2: MQTT server independently located scenario

The above figure shows a protocol segment view in which the MQTT server is located independently from the oneM2M nodes. In this scenario, all oneM2M nodes (ADN, ASN, MN, IN) include a MQTT client. MQTT servers exists independently, which means the servers are located outside of the nodes.

In this scenario, the protocol segments are illustrated as follows.

|  |  |  |
| --- | --- | --- |
| **Protocol Segment** | **oneM2M Message Transported** | **MQTT Interaction** |
| PS1 | Mca (AE of ADN to CSE of IN) | Client in ADN to Server |
| PS2 | Mca (AE of ADN to CSE of MN) | Client in ADN to Server |
| PS3 | Mcc (CSE of ASN to CSE of MN) | Client in ASN to Server |
| PS4 | Mcc (CSE of ASN to CSE of IN) | Client in ASN to Server |
| PS5 | Mcc (CSE of MN to CSE of MN) | Client in MN to Server |
|  | Mcc (CSE of MN to CSE of ASN) |  |
|  | Mca (CSE of MN to AE of ADN) |  |
| PS6 | Mcc (CSE of MN to CSE of MN) | Client in MN to Server |
| PS7 | Mcc (CSE of IN to CSE of MN) | Client in IN to Server |
|  | Mcc (CSE of IN to CSE of ASN) |  |
|  | Mca (CSE of IN to AE of ADN) |  |
| PS8 | Mcc' (CSE of IN to CSE of IN) | Client in IN to Server |

TABLE 2.2: PROTOCOL SEGMENTS FOR SERVERS INDEPENDENTLY LOCATED

The next four sections show the four configurations in which the MQTT binding can be used in the co-located scenario, followed by similar configurations in the independently-located scenario.

MQTT includes reliability features which allow recovery from loss of network connectivity without requiring explicit involvement of the applications that are using it, however to do this it requires an underlying network protocol that provides ordered, lossless, bi-directional connections. The MQTT specification does not mandate a particular underlying protocol, so this binding specification restricts the choice of underlying protocol: it shall be one of the following:

* Raw TCP/IP .
* TCP/IP with Transport Level Security (TLS)
* WebSocket - either with or without the use of TLS.

Connecting to MQTT

In order to communicate, the two client parties (Application Entity or AE and Common Service Entry(CSE) or CSE and CSE) shall connect to a common MQTT server. The MQTT server shall be hosted in one of the two nodes or shall exist as an independent external entity.

Once each party has located the address of the MQTT server, it then connects to it using the standard MQTT CONNECT protocol packet. The following additional considerations apply:

* The CONNECT packet contains a Client ID. The Client Ids have to be unique at least among all clients that connect to a given MQTT server instance (this is a requirement imposed by the MQTT protocol). This condition will be satisfied if an AE uses its AE-ID and a CSE uses its CSE-ID

The AE-ID or CSE-ID may not be known during the initial registration process, in which case the client shall use some other appropriate unique ID.

* A client shall set the "Clean Session" flag in the CONNECT packet to false. This means that MQTT Session state related to that client will be retained by the MQTT Server in the event of a disconnection (deliberate or otherwise) of that client.
* A client shall not set the "Will Flag", so Will Messages are not enabled.
* A client may choose to provide a non-zero MQTT KeepAlive value or to provide a KeepAlive of 0 (this disables the MQTT KeepAlive).
* The MQTT server may require that a client provides a User Name and a password (or other credential).

A client might choose to keep the MQTT connection open permanently (restarting it as soon as possible after any unforeseen connection loss), it might choose to connect only when it wants to act as an Originator, or it might choose to connect based on the <schedule> associated with a relevant oneM2M resource.

Once a client has connected to the MQTT server it can then communicate (subject to authorization policies) with any other client connected to its server. There is no need for it to create another connection if it wants to communicate with a different counter-party.

When a client determines that it no longer wishes to participate in an MQTT Session with its MQTT Server it shall perform the following steps:

* Disconnect from that server, if it is currently connected.
* Reconnect with the cleanSession flag set to true.

**16**  **TS 118 110 V1.0.0 (2015**

* Disconnect again.

These steps delete any state that the MQTT server might be holding on behalf of the client.

**Sending and Receiving Messages**

**Request and Response Messages**

An MQTT Control Packet consists of up to three parts: a fixed header, a variable header, a payload. The control packet format is depicted below. MQTT does not have a data model to describe or constrain the content of its Application Message payloads (to that extent it is similar to a TCP socket). Mca and Mcc request messages shall be serialized into XML or JSON following the serialization.

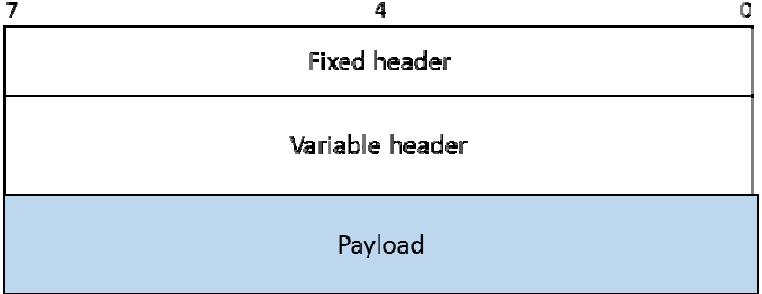


Figure 2. 3: Message format of MQTT Control Packet

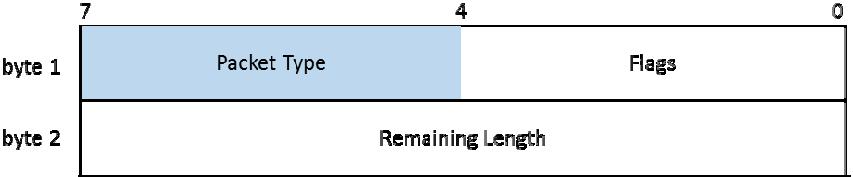


Figure 2.4: Fixed header structure

The packet type field in figure is used to define the MQTT Control packet type. It is a 4-bit field which possible values are listed in the table below. When a oneM2M Request/Response message is bound to MQTT, its packet type shall have value 3, i.e. the Request/Response message is delivered in an MQTT PUBLISH packet.

|  |  |  |
| --- | --- | --- |
| Reserved | 0 | Reserved for future use |
| CONNECT | 1 | Client request to connect to server |
| CONNACK | 2 | Connect acknowledgement |
| PUBLISH | 3 | Publish message |
| PUBACK | 4 | Publish message acknowledgement |
| PUBREC | 5 | Publish received (QoS=2) |
| PUBREL | 6 | Publish release (QoS=2) |
| PUBCOMP | 7 | Publish complete (QoS=2) |
| SUBSCRIBE | 8 | Client subscribe request |
| SUBACK | 9 | Subscribe acknowledgement |
| UNSUBSCRIBE | 10 | Client unsubscribe request |
| UNSUBACK | 11 | Unsubscribe acknowledgement |
| PINGREQ | 12 | Ping request |
| PINGRESP | 13 | Ping response |
| DISCONNECT | 14 | Client disconnection request |
| Reserved | 15 | Reserved for future use |

TABLE 2.3: MQTT Control Packet Types

**TS 118 110 V1.0.0 (2015-02)**

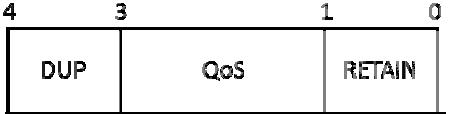


FIGURE 2.5:Flags for the PUBLISH packet

This figure shows the flags specific to MQTT PUBLISH packet. The QoS field represents the QoS level of the PUBLISH packet with possible values of 1, 2 or 3. However, since oneM2M messages are idempotent, they should be sent with MQTT QoS 1.

NOTE: MQTT packets are subjected to a theoretical maximum message size of 256MB, but it is good practice not to send packets that are bigger than a 100kB. If a larger amount of data needs to be sent, it should be segmented into multiple PUBLISH packets.

**Sending a Request**

A request is transmitted by sending it as an MQTT PUBLISH protocol packet to the MQTT Server. It uses a Topic that identifies both the Originator and the Receiver of the request as follows:

* /oneM2M/req/<from>/<to>
  + "oneM2M" is a literal string identifying the topic as being used by oneM2M.
  + <from> is the SP-relative-AE-ID or SP-relative-CSE-ID of the Originator, omitting any leading/.
  + <to> is the SP-relative-CSE-ID of the Receiver (Transit or Hosting CSE), omitting any leading/.
  + "req" is a literal string identifying this as a request.

The payload of the MQTT PUBLISH packet is used to carry the request primitive.

Listening for and responding to a Request

A Receiver listens for requests arriving via MQTT by subscribing to its Topics using a wildcarded topic filter of the following form:

* /oneM2M/req/+/<to>
  + "oneM2M" is a literal string identifying the topic as being used by oneM2M.
  + + is a wildcard which matches any Originator.
  + <to> is the SP-relative-CSE-ID of the Receiver (Transit or Hosting CSE), omitting any leading/.
  + "req" is a literal string identifying this as a request.

When it receives a request, the Receiver shall perform the Core Transport operations associated with the request, including any access control policy checks. In particular it shall check the request expiration timestamp (if any) contained in the request, since it is possible that that time might have passed while the message was being stored by MQTT.

It transmits a response by sending an MQTT PUBLISH packet to a response topic. This takes the form:

* /oneM2M/resp/<from>/<to>
  + "oneM2M" is a literal string identifying the topic as being used by oneM2M.
  + <to> is the SP-relative-CSE-ID of the Receiver (Transit or Hosting CSE), omitting any leading/.
  + <from> is the SP-relative-AE-ID or SP-relative-CSE-ID of the Originator, omitting any leading/.
  + "resp" is a literal string identifying this as a response.

The Originator shall subscribe to this Topic (either explicitly or using a wildcarded filter) in order to see the response. **TS 118 110 V1.0.0 (2015-02)**

The payload of the MQTT PUBLISH packet is used to carry the response primiive.

Initial Registration

In some security scenarios, an Originator might not initially know its AE-ID or CSE-ID. Initial registration exchanges can use the communication pattern, except that they use Topics containg a credential ID rather than an AE-ID or CSE-ID, as follows:

* /oneM2M/reg\_req/<from>/<to>; and
* /oneM2M/reg\_resp/<from>/<to>
  + "oneM2M" is a literal string identifying the topic as being used by oneM2M.
  + <from> is Credential-ID.
  + <to> is the SP-relative-CSE-ID of the Receiver (Transit or Hosting CSE), omitting any leading/.
  + "reg\_req" and "reg\_resp" are literal strings identifying them as registration requests and responses.

Request/Response Message Flow

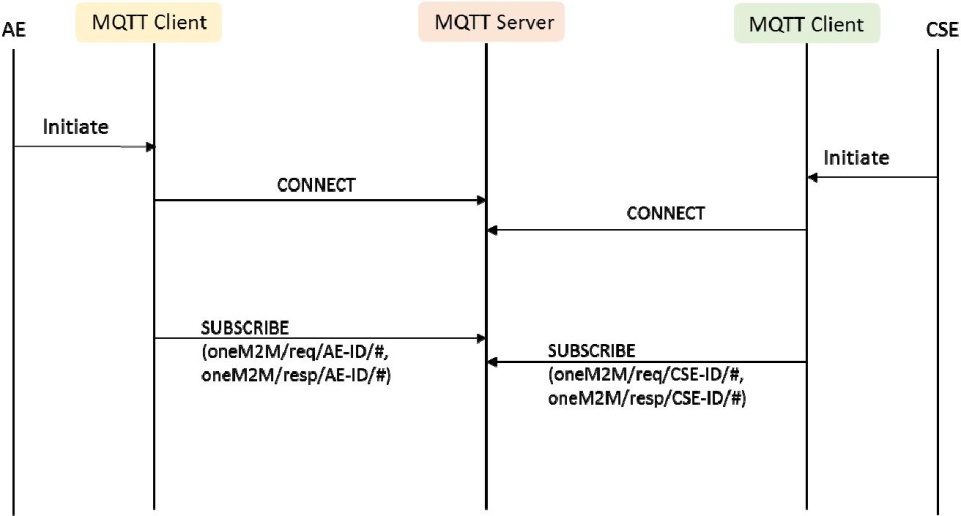


Figure 2.6: Initiating Process in MQTT binding

In the MQTT protocol, each client shall subscribe to the MQTT server to receive messages. As shown in

figure 6.4.5.1-1, the AE or CSE initiates the MQTT binding process by passing the AE-ID or CSE-ID respectively to the MQTT client library. After the MQTT client library receives this request, it tries to connect to the MQTT server. In this process, the AE-ID or CSE-ID is used as the client identifier within the MQTT CONNECT packet.

After the MQTT client successfully connects to the server, each MQTT client shall subscribe to the MQTT server. In this process, the AE-ID or CSE-ID with the number sign ('#' U+0023) wildcard is used as the topic name within the MQTT SUBSCRIBE packet. This enables subscription to topics with any resource URI under AE-ID or CSE-ID. The full topic names subscribed to are "oneM2M/req/ID/#" and "oneM2M/resp/ID" where ID is AE-ID or CSE-ID. Therefore, AE or CSE receives messages if the request/response messages are published with oneM2M/req/ID/resource and oneM2M/resp/ID/ where ID is AE-ID or CSE-ID and resource is the resource's URI.

**TS 118 110 V1.0.0 (2015-0**

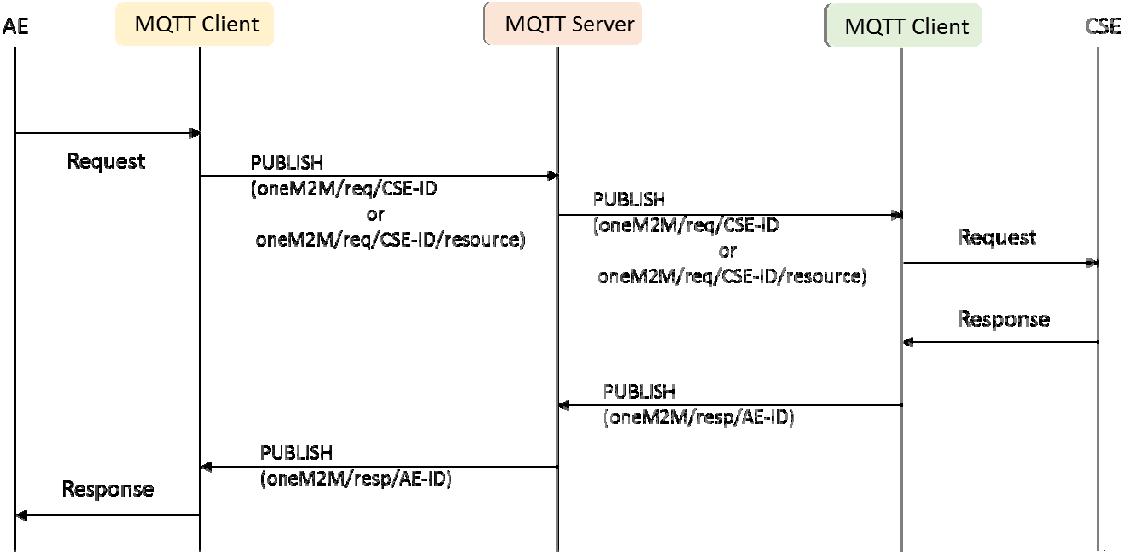


FIGURE 2.7 Request/Response message delivery over MQTT

As an example, above figure illustrates the request/response message delivery over MQTT protocol between AE and CSE via Mca reference point in oneM2M. The message flow is as follows.

In this scenario, AE wants to send a Request message to the registered CSE. After that, the AE'S MQTT client library sends an MQTT PUBLISH packet to the MQTT server with "oneM2M/req/CSE-ID" or "oneM2M/req/CSE-ID/resource" as the topic name. The MQTT PUBLISH packet includes {"op", "fr", "to", "ri", "cn", "ty"} and any other optional message parameters in its payload.

After the MQTT server receives the MQTT PUBLISH packet from the MQTT client, the server refers to the topic name and delivers the message to the intended MQTT client. Finally, the MQTT client library delivers the message to the CSE.

Regarding the Response message, the MQTT messages are sent using CSE-ID/AE-ID as topic. In order to send a Response message to the requestor, the CSE uses the "fr" information from the received request message and sends an MQTT PUBLISH packet to the MQTT server with "oneM2M/resp/AE-ID" as the topic name in this scenario. The MQTT PUBLISH packet includes {"rs", "ri", "cn"} and any other optional message parameters in its payload.

Similar to the request message delivery from AE to CSE, the MQTT server delivers the published packet to the intended MQTT client by referring to the topic name. Finally, AE receives the response message from the MQTT client.

Primitive Mapping

Request primitives

A oneM2M request primitive is made up of a number of control parameters and (optionally) a content part. All the parameters in these parts are serialized into the payload of an MQTT Publish Packet.

All the parameters that are present in the primitive shall be serialized, in particular the request shall contain the **Operation** parameter to indicate whether it is a Create, Read, Update or Delete.

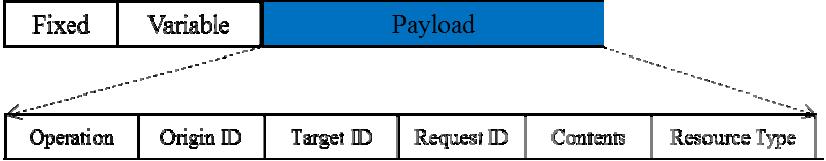


Figure 2.8:MQTT Request example

Content Format Negotiation

The request primitive can contain an additional parameter called xM2M-Accept which indicates the Content Formats that the Originator is prepared to accept in a Response.

If a Hosting CSE supports the Content-Format specified in Accept option of the request, the Hosting CSE shall respond with that Content-Format. If the Hosting CSE doesn't support the Content-Format specified in Accept option of the request, "Not Acceptable" shall be sent as the ResponseStatusCode unless another error code takes precedence for this response.

Possible values for the Accept options are listed below:

* application/xml
* application/json

**Security**

Introduction

The MQTT servers authenticate the clients (both CSEs and AEs) that connect to them and authorize access to Topics used for communicate. The clients do not authenticate each other, instead they use the MQTT server to do this.

**Background.** The MQTT binding makes use of one or more MQTT Servers to transport messages between AEs andCSEs (or between CSEs) as described in clause 5. The AE/CSEs both act as MQTT Clients of an MQTT Server that mediates delivery of messages between the two. As described in clause 6, the topic in the MQTT PUBLISH packet either indicates the Originator's identity and Receiver's identity, or identifies the Originator's Credential and the Receiver Identity (in the case that the Originator has not yet been assigned an identifier).

**Trust and the MQTT Server.** When the oneM2M binding to MQTT is used, some security functions are performed bythe MQTT Server, as described further in this clause. In particular:

1. The MQTT Server authenticates the AEs and CSEs as they connect as MQTT Clients. These MQTT Clients themselves never directly authenticate the CSE or AE that is using another MQTT Client - instead they trust the MQTT Server to authenticate the MQTT Clients, and trust the MQTT Server to route the messages between the Originator and Receiver indicated in the topic.
2. The MQTT Server enforces access control policies to ensure that unidentified or unauthorized clients are not able to publish messages to oneM2M topics or subscribe to receive messages from them.

**118**

Authorization

There are two levels of authorization in the oneM2M binding to MQTT.

* *The MQTT Server is responsible for verifying identifiers, for routing messages to the expected CSE or AE, and providing the correct Credential-ID during initial registration.*

The MQTT Server is responsible for verifying that the Client Identifier field in a MQTT CONNECT packet matches the expected AE-ID, CSE-ID or Credential-ID.

The MQTT Server is responsible for controlling those topics to which an MQTT may subscribe and receive published MQTT packets, and those topics to which an MQTT may publish MQTT packets.

Since the topic includes the Receiver's CSE-ID, the Originator can trust that the MQTT packets are routed to and from the expected Receiver. If the topic includes the Originator's CSE-ID or AE-ID, then the Receiver can trust that the MQTT packets are routed to and from the expected Originator.

If the topic includes the Originator's Credential-ID (which should only occur at initial registration), then the Receiver can use this Credential-ID to determine the CSE-ID or AE-ID or list of allowed AE-ID(s) which are to be used in assigning an CSE-ID or AE-ID to the Originator (as described in TS 118 103). This Credential-ID can be trusted to have been verified by the MQTT Server.

* *The Receiver is responsible for authorizing requests to specific resources, and assigning CSE-ID or AE-ID during initial registration.*

When the MQTT topic includes the Originator's CSE-ID or AE-ID, then the Receiver is responsible for making access control decisions on requests to perform operations on specific resources hosted on the Receiver. The access control decisions are dictated by the applicable accessControlPolicy resources and the Originator's CSE-ID or AE-ID (and other factors not relevant to the present discussion). This authorization process is as defined in the Architecture Specification [,](#page6) the Core Protocol Specification and Security Solutions specification.

When the MQTT topic includes the Originator's Credential-ID (which should only occur at initial registration), then the Receiver is responsible for assigning a CSE-ID or AE-ID to the Originator (which may be dependent on the Originator's Credential-ID).

Authentication

An MQTT Client and MQTT Server shall apply Transport Layer Security (TLS) using any of the Security Association Establishment Frameworks in TS 118 103.

The Security Association Establishment Frameworks provide mutual authentication of the MQTT Client and MQTT Server. The Security Association Establishment Frameworks are described using two main entities Entity A and Entity B: in the case of the oneM2M binding to MQTT, Entity A is a CSE or AE using an MQTT Client, and Entity B is an MQTT Server.

NOTE: In TS 118 103 Entity A is described as establishing the CSE-ID of Entity B as a result of Security Association Establishment. The application to MQTT differs because Entity A establishes the identity of the MQTT Server instead. However, the procedures are still applicable.

The Remote Security Provision Frameworks in TS 118 103  [[7]](#page6) may be applied to provision a symmetric key shared by a CSE/AE using an MQTT Client and an MQTT Server, with the MQTT Server assuming the role of the Enrolment Target.

**Identification of Originator and Receiver.**  TS 118 103  [[7]](#page6) describesa variety of approaches by whichsuccessful Security Association Establishment results in Entity B determining the CSE-ID or AE-ID or list of allowed AE-ID(s) for the CSE/AE using the Entity A. These approaches can also be used in the oneM2M binding to MQTT.

We assume that the MQTT Server is configured with the information necessary to determine the CSE-ID of the Receiver following successful Security Association Establishment with the Receiver's MQTT Client.

**TS 118 110 V1.0.0 (2015-02)**

In some scenarios, the MQTT Server can be configured with appropriate information to verify the CSE-ID or AE-ID of the Originator. However, in cases where the Originator has not yet been assigned its CSE-ID or AE-ID, and the MQTT Server has also not been provided with the CSE-ID or AE-ID of the Originator, then the Receiver is responsible for determining the applicable CSE-ID or AE-ID. In these cases, the MQTT Server forms a Credential-ID, identifying the Credential used to authenticate the Originator, and includes this in the topic when forwarding to the initial registration request to the Receiver's MQTT Client. The Receiver extracts the Credential-ID, and the procedures in

TS 118 101 and TS 118 103 determines the CSE-ID or AE-ID of the Originator.

Authorization by the MQTT Server

This procedure describes how an MQTT Server authorizes topics to which an MQTT Client may subscribe. The M2M Service Provider is responsible for configuring the MQTT Server with the relevant information used in these authorization decisions.

1. In the case that the MQTT Server determines that the MQTT Client represents a CSE, and if the CSE has been assigned a CSE's SP-Relative-CSE-ID (which we denote by <my-SP-Relative-CSE-ID>), then:
   1. the MQTT Server authorizes the MQTT Client to subscribe to the topic /oneM2M/req/+/<my-SP-Relative-CSE-ID>;
   2. for the set of <Registree ID Stem> values corresponding to SP-Relative CSE-ID or AE-ID Stem values of zero or more CSE(s) and/or AE(s) currently registered to this CSE (and known to the MQTT Server), the MQTT Server authorizes the MQTT Client to subscribe to the topics /oneM2M/resp/<Registree ID Stem>/<my-SP-Relative-CSE-ID>; and
   3. if this CSE is registered to a CSE, and the SP-Relative-CSE-ID of this CSE is <Registrar-SP-Relative-CSE-ID >, then the MQTT Server authorizes the MQTT Client to perform the following:

to subscribe to;

/oneM2M/resp/<my-SP-Relative-CSE-ID>/<Registrar-SP-Relative-CSE-ID>; to publish to;

/oneM2M/resp/<my-SP-Relative-CSE-ID>/<Registrar-SP-Relative-CSE-ID>.

1. In the case that the MQTT Server determines that the MQTT Client represents a AE which has been assigned an S-Type AE-ID Stem equal to < AE-ID-Stem>, then
   1. If the MQTT Server determines that the AE is currently registered, and the AE's Registrar CSE has SP-Relative-CSE-ID equal to <Registrar-SP-Relative-CSE-ID> then the MQTT Server authorizes the MQTT Client to perform the following:
   2. To subscribe to /oneM2M/resp/<AE-ID-Stem>/<Registrar-SP-Relative-CSE-ID>; and
   3. To publish to /oneM2M/req/<AE-ID-Stem>/<Registrar-SP-Relative-CSE-ID>.
2. Otherwise, the MQTT Server authorizes the MQTT Client to perform the following:
   1. To subscribe to /oneM2M/reg\_resp/<AE-ID-Stem-Credential-ID>/+; and
   2. To publish to /oneM2M/req\_req/<AE-ID-Stem-Credential-ID>/+.

where <AE-ID-Stem-Credential-ID> is generated from <AE-ID-Stem> as per TS 118 103

**3**  **TS 118 110 V1.0.0 (2015-02)**

1. In the case that the MQTT Server determines that the MQTT Client represents a AE with C-Type AE-ID-Stem equal to <AE-ID-Stem> (which implies that the AE is registered), and the AE's Registrar CSE has SP-Relative-CSE-ID equal to <Registrar-SP-Relative-CSE-ID> then the MQTT Server authorizes the MQTT Client to perform the following:
   1. To subscribe to /oneM2M/resp/<AE-ID-Stem>/<Registrar-SP-Relative-CSE-ID>; and
   2. To publish to /oneM2M/req/<AE-ID-Stem>/<Registrar-SP-Relative-CSE-ID>.
2. In all other cases, the MQTT Server authorizes the MQTT Client to perform the following:
   1. To subscribe to /oneM2M/reg\_resp/<Credential-ID>/+; and
   2. To publish to /oneM2M/req\_req/<Credential-ID>/+.

CHAPTER 3

NODE MCU

Introduction toNodeMCU Development kit

NodeMCU is a firmware that runs on ESP8266 as NodeJS server and lets you write your code in LUA Script language. NodeMCU DevKit is basically an “upgrade” for ESP8266 module with extended number of pins and a USB-To-Serial interface. It can be powered by a micro usb cable or by its pins. It uses the exact model of ESP8266-12 as core. Basically it’s a pocket NodeJS server. The price range is about 6-14 USD. Also it doesn’t come with the NodeMCU firmware (ironic). We need to flash and write the firmware into it.

Pros:

* Small
* Very cheap
* Has USB-To-Serial interface
* Open source

Cons:

* No official documentation
* Mixed information floats around
* The official forum mostly consists of Chinese speaking people
* Firmware still has many bugs

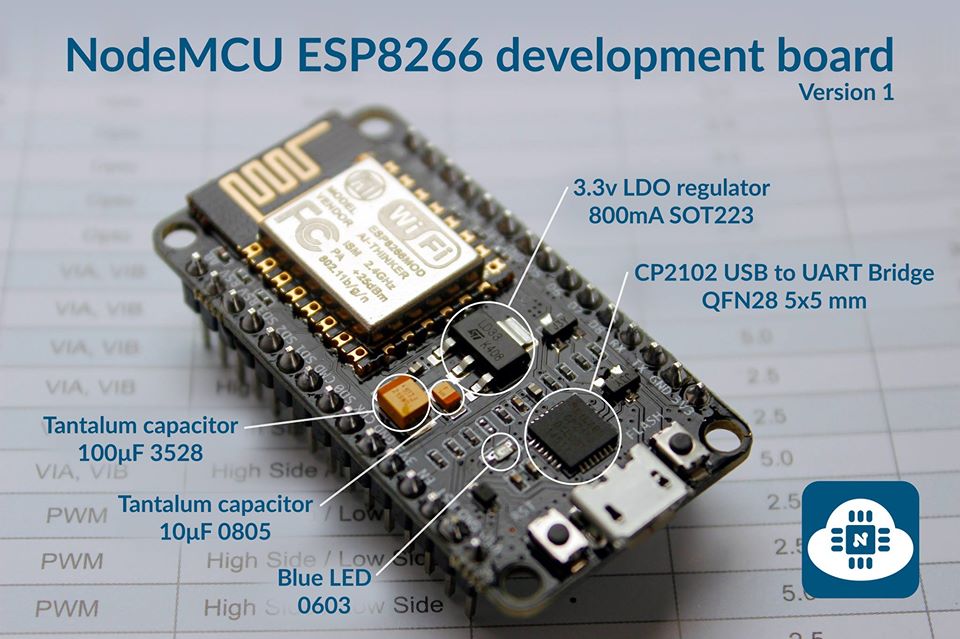


Figure 3.1 NODE MCU

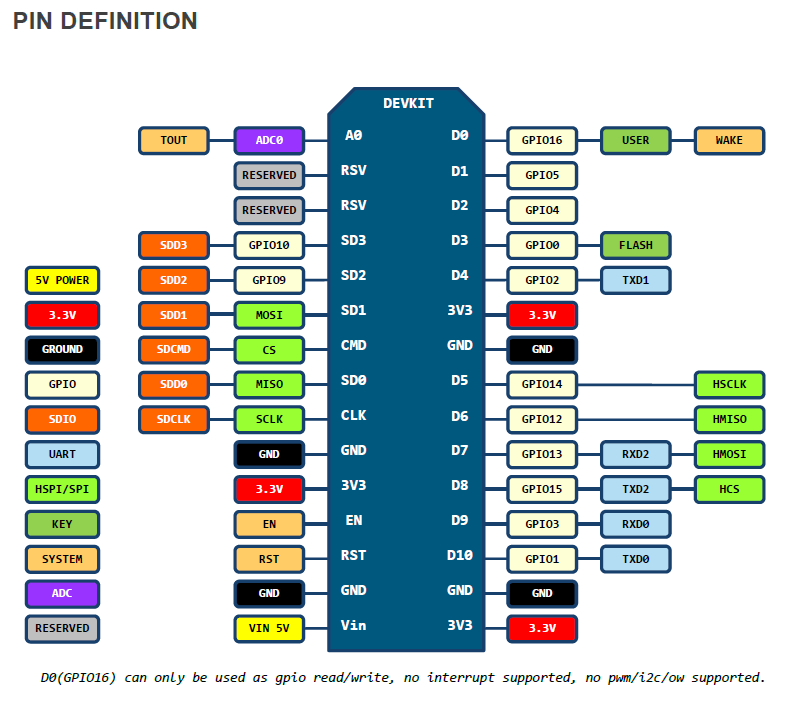


Figure 3.2: PIN CONFIGURATION

Key Components of the NodeMCU devkit

The NodeMCU development kit is a readymade development board based on the ESP8266 wifi chip which consists of the following key components that are inbuilt within the board so that we do not need to connect them separately with the ESP8266. These key components are as follows :

* The ESP8266 wifi chip core
* CH340 USB to serial chip
* SPX3819 Low voltage LDO voltage regulator

**ESP8266 wifi chip core**



Espressif’s ESP8266 delivers highly integrated Wi-Fi SoC solution to meet users’ continuous demands for efficient power usage, compact design and reliable performance in the Internet of Things industry.

With the complete and self-contained Wi-Fi networking capabilities, ESP8266 can perform either as a standalone application or as the slave to a host MCU. When ESP8266 hosts the application, it promptly boots up from the flash. The integrated high-speed cache helps to increase the system performance and optimize the system memory. Also, ESP8266 can be applied to any micro-controller design as a Wi-Fi adaptor through SPI / SDIO or I2C / UART interfaces.

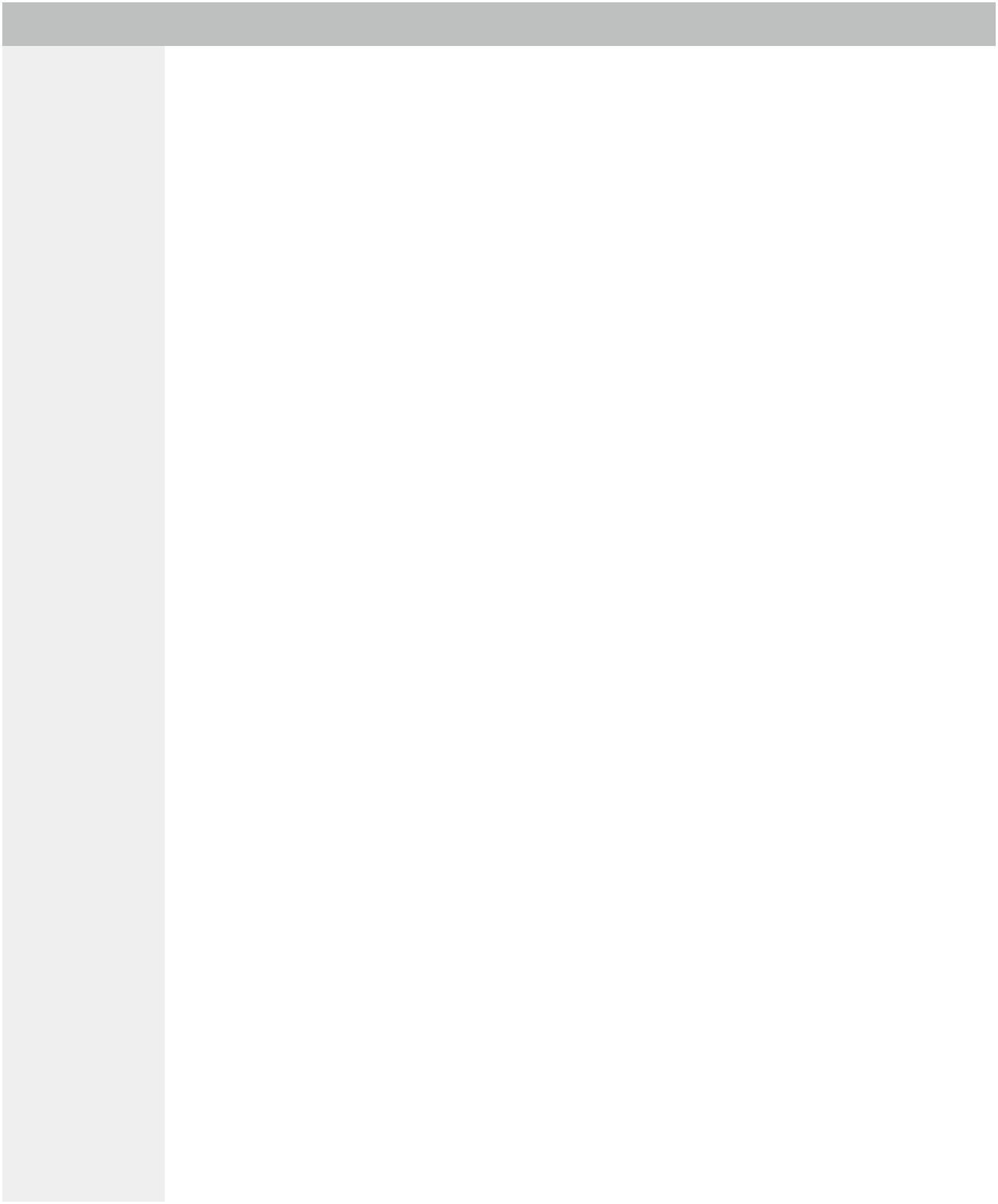
ESP8266 integrates antenna switches, RF balun, power amplifier, low noise receive amplifier, filters and power management modules. The compact design minimizes the PCB size and requires minimal external circuitries.

Besides the Wi-Fi functionalities, ESP8266 also integrates an enhanced version of Tensilica’s L106 Diamond series 32-bit processor and on-chip SRAM. It can be interfaced with external sensors and other devices through the GPIOs. Software Development Kit (SDK) provides sample codes for various applications.

Espressif Systems’ Smart Connectivity Platform (ESCP) enables sophisticated features including fast switch between sleep and wake-up mode for energy-efficient purpose, adaptive radio biasing for low-power operation, advance signal processing, spur cancellation and radio co-existence mechanisms for common cellular, Bluetooth, DDR, LVDS, LCD interference mitigation.

**Main Technical Specifications**

Table 3.1: Main Technical Specifications

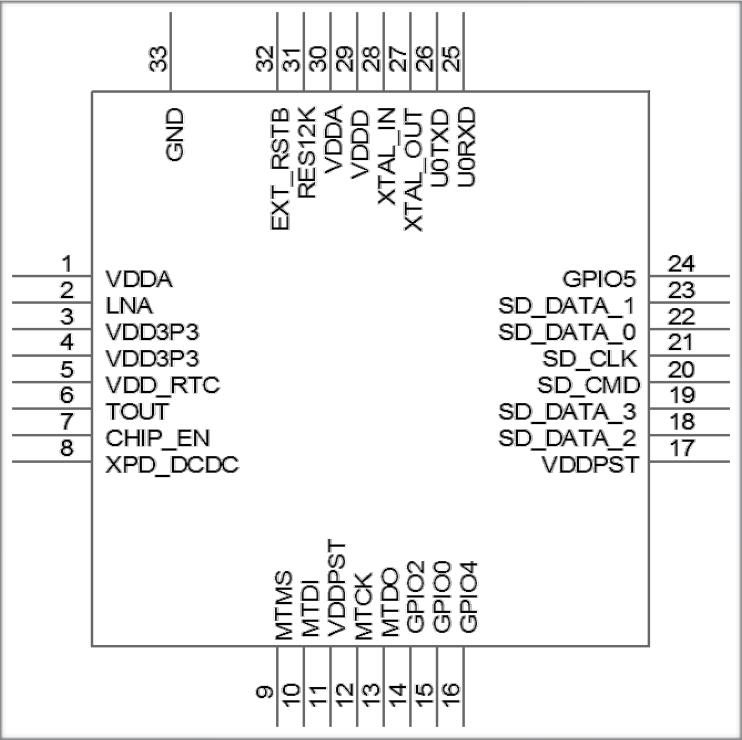


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Categories** | | **Items** | **Parameters** |  |
|  |  |  |  |  |
|  |  | Standards | FCC/CE/TELEC/SRRC |  |
|  |  |  |  |  |
|  |  | Protocols | 802.11 b/g/n/e/i |  |
|  |  |  |  |  |
|  |  | Frequency Range | 2.4 G2.5 G (2400M2483.5M) |  |
|  |  |  |  |  |
|  |  |  | 802.11 b: +20 dBm |  |
|  |  | Tx Power |  |  |
| Wi-Fi | | 802.11 g: +17 dBm |  |
|  |  |  |
|  | 802.11 n: +14 dBm |  |
|  |  |  |  |
|  |  |  |  |  |
|  |  |  | 802.11 b: -91 dbm (11 Mbps) |  |
|  |  |  |  |  |
|  |  | Rx Sensitivity | 802.11 g: -75 dbm (54 Mbps) |  |
|  |  |  |  |  |
|  |  |  | 802.11 n: -72 dbm (MCS7) |  |
|  |  |  |  |  |
|  |  | Antenna | PCB Trace, External, IPEX Connector, Ceramic Chip |  |
|  |  |  |  |  |
|  |  | CPU | Tensilica L106 32-bit micro controller |  |
|  |  |  |  |  |
|  |  | Peripheral Interface | UART/SDIO/SPI/I2C/I2S/IR Remote Control |  |
|  |  |  |  |
|  |  | GPIO/ADC/PWM |  |
|  |  |  |  |
|  |  |  |  |  |
|  |  | Operating Voltage | 3.0 V ~ 3.6 V |  |
|  |  |  |  |  |
| Hardware | | Operating Current | Average value: 80 mA |  |
|  |  |  |  |  |
|  |  | Operating Temperature Range | -40°C ~ 125°C |  |
|  |  |  |  |  |
|  |  | Storage Temperature Range | -40°C ~ 125°C |  |
|  |  |  |  |  |
|  |  | Package Size | QFN32-pin (5 mm x 5 mm) |  |
|  |  |  |  |  |
|  |  | External Interface | - |  |
|  |  |  |  |  |
|  |  | Wi-Fi Mode | station/softAP/SoftAP+station |  |
|  |  |  |  |  |
|  |  | Security | WPA/WPA2 |  |
|  |  |  |  |  |
|  |  | Encryption | WEP/TKIP/AES |  |
|  |  |  |  |  |
| Software | | Firmware Upgrade | UART Download / OTA (via network) |  |
|  |  |  |
| Software Development | Supports Cloud Server Development / Firmware and |  |
|  |  |  |
|  |  | SDK for fast on-chip programming |  |
|  |  |  |  |
|  |  |  |  |  |
|  |  | Network Protocols | IPv4, TCP/UDP/HTTP/FTP |  |
|  |  |  |  |  |
|  |  | User Configuration | AT Instruction Set, Cloud Server, Android/iOS App |  |

**Pin Definitions**



Figure 2-1 shows the pin layout for 32-pin QFN package.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | | Figure 3.3.: Pin Layout |  |
|  | | | | |  |
|  |  | |  | |  |
| **SL** | **Name** | | **Function** | **Features** |  |
|  |  | |  |  |  |
| 1 | VDDA | | P | Analog Power 3.0 V ~ 3.6 V |  |
|  |  | |  |  |  |
|  |  | |  | RF Antenna Interface. |  |
| 2 | LNA | | I/O | Chip Output Impedance=50 Ω |  |
| No matching required. It is suggested to retain the π-type |  |
|  |  | |  | matching network to match the antenna. |  |
|  |  | |  |  |  |
| 3 | VDD3P3 | | P | Amplifier Power 3.0 V ~ 3.6 V |  |
|  |  | |  |  |  |
| 4 | VDD3P3 | | P | Amplifier Power 3.0 V ~ 3.6 V |  |
| 5 | VDD\_RTC | | P | NC (1.1 V) |  |
|  |  | |  | ADC pin. It can be used to test the power-supply voltage of |  |
| 6 | TOUT | | I | VDD3P3 (Pin3 and Pin4) and the input power voltage of TOUT |  |
| (Pin 6). However, these two functions cannot be used |  |
|  |  | |  | simultaneously. |  |
|  |  | |  |  |  |
| 7 |  | |  | Chip Enable |  |
| CHIP\_PU | | I | High: On, chip works properly |  |
|  |  | |  | Low: Off, small current consumed |  |
|  |  | |  |  |  |
| 8 | XPD\_DCDC | | I/O | Deep-sleep wakeup (need to be connected to EXT\_RSTB); |  |
| GPIO16 |  |
|  |  | |  |  |
|  |  | |  |  |  |
|  |

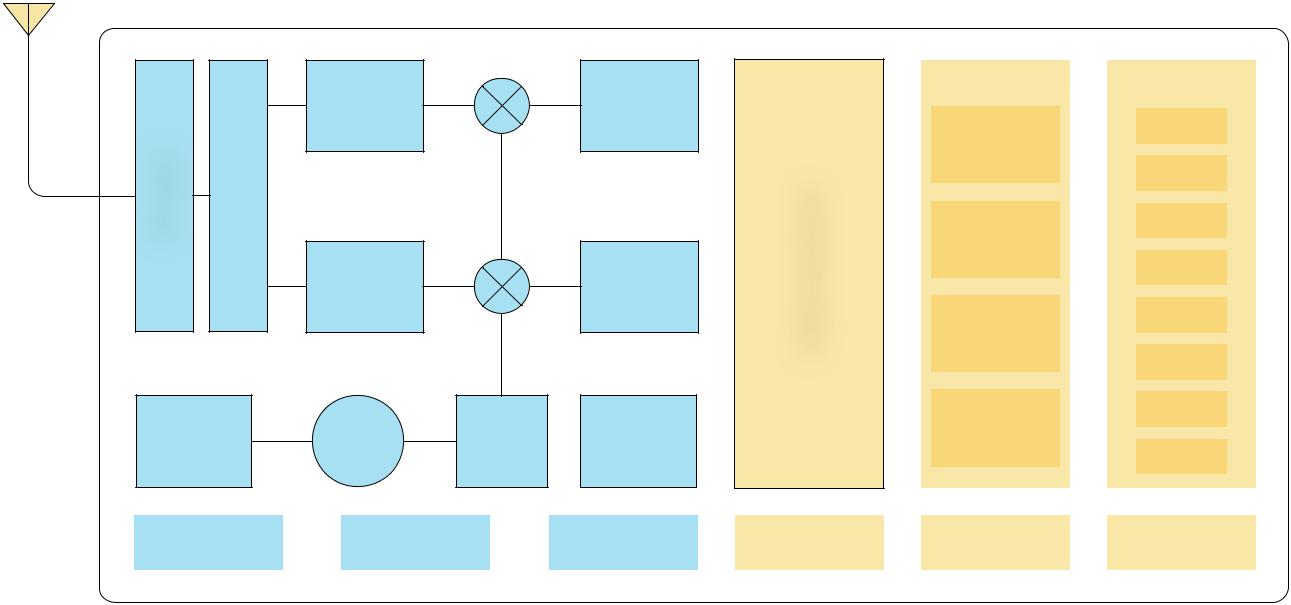
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SL | NAME | Function | Features |  |
| 9 | MTMS | I/O | GPIO14; HSPI\_CLK |  |
| 10 | MTDI | I/O | GPIO12; HSPI\_MISO |  |
| 11 | VDDPST | P | Digital/IO Power Supply (1.8 V ~ 3.3 V) |  |
|  |  |  |  |  |
| 12 | MTCK | I/O | GPIO13; HSPI\_MOSI; UART0\_CTS |  |
| 13 | MTDO | I/O | GPIO15; HSPI\_CS; UART0\_RTS |  |
| 14 | GPIO2 | I/O | UART Tx during flash programming; GPIO2 |  |
| 15 | GPIO0 | I/O | GPIO0; SPI\_CS2 |  |
|  |  |  |  |  |
| 16 | GPIO4 | I/O | GPIO4 |  |
| 17 | VDDPST | P | Digital/IO Power Supply (1.8 V ~ 3.3 V) |  |
| 18 | SDIO\_DATA\_2 | I/O | Connect to SD\_D2 (Series R: 200Ω); SPIHD; HSPIHD; GPIO9 |  |
| 19 | SDIO\_DATA\_3 | I/O | Connect to SD\_D3 (Series R: 200Ω); SPIWP; HSPIWP; GPIO10 |  |
|  |  |  |  |  |
| 20 | SDIO\_CMD | I/O | Connect to SD\_CMD (Series R: 200Ω); SPI\_CS0; GPIO11 |  |
| 21 | SDIO\_CLK | I/O | Connect to SD\_CLK (Series R: 200Ω); SPI\_CLK; GPIO6 |  |
| 22 | SDIO\_DATA\_0 | I/O | Connect to SD\_D0 (Series R: 200Ω); SPI\_MSIO; GPIO7 |  |
| 23 | SDIO\_DATA\_1 | I/O | Connect to SD\_D1 (Series R: 200Ω); SPI\_MOSI; GPIO8 |  |
|  |  |  |  |  |
| 24 | GPIO5 | I/O | GPIO5 |  |
| 25 | U0RXD | I/O | UART Rx during flash programming; GPIO3 |  |
| 26 | U0TXD | I/O | UART Tx during flash progamming; GPIO1; SPI\_CS1 |  |
| 27 | XTAL\_OUT | I/O | Connect to crystal oscillator output, can be used to provide BT |  |
| clock input |  |
|  |  |  |  |
|  |  |  |  |  |
| 28 | XTAL\_IN | I/O | Connect to crystal oscillator input |  |
|  |  |  |  |  |
| 29 | VDDD | P | Analog Power 3.0 V ~ 3.6 V |  |
| 30 | VDDA | P | Analog Power 3.0 V ~ 3.6 V |  |
| 31 | RES12K | I | Serial connection with a 12 kΩ resistor and connect to the |  |
| Ground |  |
|  |  |  |  |
|  |  |  |  |  |
| 32 | EXT\_RSTB | I | External reset signal (Low voltage level: Active) |  |



**Functional Description**



The functional diagram of ESP8266 is shown as in Figure 3-4.



|  |  |  |  |
| --- | --- | --- | --- |
|  | RF |  | Analog |
|  | Receive |  | receive |
| RF | Switch |  |  |
|  | RF |  | Analog |
|  | Transmit |  | Transmit |
| PLL | VCO | 1/2 | PLL |

|  |
| --- |
| Digital baseband |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | MAC | |  |  | Interface | |  |
|  |  |  |  |  |  | |  |
|  | Registers |  |  |  | UART |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | GPIO |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  | CPU |  |  |  | I2C |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | I2S |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  | Sequencers |  |  |  | SDIO |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | PWM |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | |  |
|  | Accelerator |  |  |  | ADC |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | SPI |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
| PMU |  | Crystal |  | Bias circuits |  | SRAM |  | PMU |  | Flash | |
|  |  |  |  |  |  |  |  |  |  |  |  |

**CPU, Memory, and Flash**

**CPU**

ESP8266 integrates Tensilica L106 32-bit micro controller (MCU) and ultra-low-power 16-bit RSIC. The CPU clock speed is 80 MHz. It can also reach a maximum value of 160 MHz. Real Time Operation System (RTOS) is enabled. Currently, only 20% of MIPS has been occupied by the Wi-Fi stack, the rest can all be used for user application programming and development. The CPU includes the interfaces as below.

* + Programmable RAM/ROM interfaces (iBus), which can be connected with memory controller, and can also be used to visit flash.
  + Data RAM interface (dBus), which can connected with memory controller.
  + AHB interface which can be used to visit the register.

**Memory**

ESP8266 Wi-Fi SoC integrates memory controller and memory units including SRAM and ROM. MCU can access the memory units through iBus, dBus, and AHB interfaces. All memory units can be accessed upon request, while a memory arbiter will decide the running sequence according to the time when these requests are received by the processor.

According to our current version of SDK, SRAM space available to users is assigned as below.

* + RAM size < 50 kB, that is, when ESP8266 is working under the station mode and connects to the router, programmable space accessible in heap + data section is around 50kB.
  + There is no programmable ROM in the SoC, therefore, user program must be stored in an external SPI flash.

**External Flash**

ESP8266 uses external SPI flash to store user programs, and supports up to 16 Mbytes memory capacity theoretically.

The minimum flash memory of ESP8266 is shown in Table 3-1.

|  |  |
| --- | --- |
|  | Table 3-2. Minimum Flash Memory |
| OTA | Minimum Flash Memory |
|  |  |
| disabled | 512 kB |
| enabled | 1 MB |



**Clock**

**High Frequency Clock**

The high frequency clock on ESP8266 is used to drive both transmit and receive mixers. This clock is generated from internal crystal oscillator and external crystal. The crystal frequency ranges from 26 MHz to 52 MHz.

The internal calibration inside the crystal oscillator ensures that a wide range of crystals can be used, nevertheless the quality of the crystal is still a factor to consider to have reasonable phase noise and good Wi-Fi sensitivity. Refer to Table 3-2 to measure the frequency offset.

**Radio**

ESP8266 radio consists of the following blocks.

* + 2.4 GHz receiver
  + 2.4 GHz transmitter
  + High speed clock generators and crystal oscillator
  + Real time clock
  + Bias and regulators
  + Power management

**Channel Frequencies**

The RF transceiver supports the following channels according to IEEE802.11b/g/n standards.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Table 3-4. Frequency Channel | |  |
| Channel No. |  | Frequency (MHz) | Channel No. | Frequency (MHz) |
|  |  |  |  |  |
| 1 | 2412 |  | 8 | 2447 |
| 2 | 2417 |  | 9 | 2452 |
| 3 | 2422 |  | 10 | 2457 |
| 4 | 2427 |  | 11 | 2462 |
| 5 | 2432 |  | 12 | 2467 |
| 6 | 2437 |  | 13 | 2472 |
| 7 | 2442 |  | 14 | 2484 |



**2.4 GHz Receiver**

The 2.4 GHz receiver down-converts the RF signals to quadrature baseband signals and converts them to the digital domain with 2 high resolution high speed ADCs. To adapt to varying signal channel conditions, RF filters, automatic gain control (AGC), DC offset cancelation circuits and baseband filters are integrated within ESP8266.

**2.4 GHz Transmitter**

The 2.4 GHz transmitter up-converts the quadrature baseband signals to 2.4 GHz, and drives the antenna with a high-power CMOS power amplifier. The function of digital calibration further improves the linearity of the power amplifier, enabling a state of art performance of delivering +19.5 dBm average power for 802.11b transmission and +16 dBm for 802.11n transmission.

Additional calibrations are integrated to offset any imperfections of the radio, such as:

* Carrier leakage
* I/Q phase matching
* Baseband nonlinearities

These built-in calibration functions reduce the product test time and make the test equipment unnecessary.

**Wi-Fi**

ESP8266 implements TCP/IP, the full 802.11 b/g/n/e/i WLAN MAC protocol and Wi-Fi Direct specification. It supports not only basic service set (BSS) operations under the distributed control function (DCF) but also P2P group operation compliant with the latest Wi-Fi P2P protocol. Low level protocol functions are handled automatically by ESP8266.

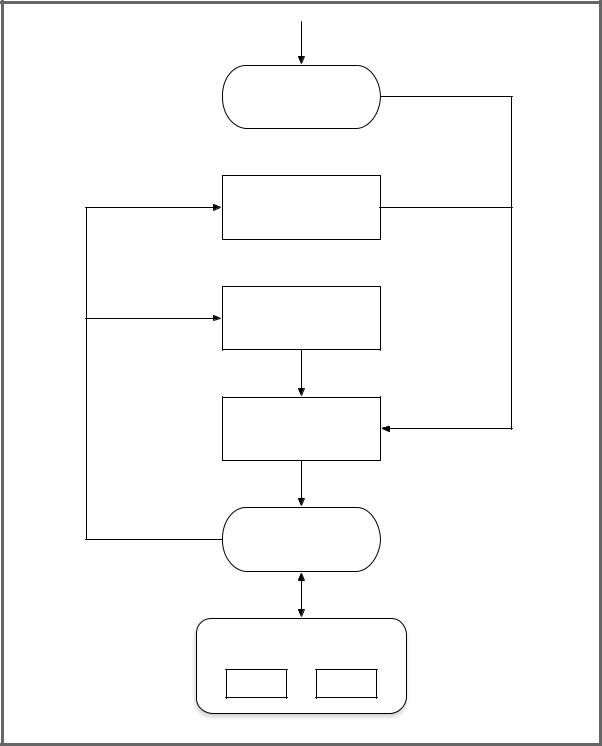
* RTS/CTS
* acknowledgement
* fragmentation and defragmentation
* aggregation
* frame encapsulation (802.11h/RFC 1042)
* automatic beacon monitoring / scanning, and
* P2P Wi-Fi direct

Passive or active scanning, as well as P2P discovery procedure is performed autonomously once initiated by the appropriate command. Power management is handled with minimum interaction with host to minimize active duty period.

**Power Management**

ESP8266 is dedicated designed for mobile devices, wearable electronics and the Internet of Things applications with advanced power management technologies.

The low-power architecture operates in 3 modes: active mode, sleep mode and deep sleep mode. ESP8266 consumes about than 20μA in deep sleep mode (with RTC clock still running) and less than 1.0mA (DTIM=3) or less than 0.6mA (DTIM=10) to stay connected to the access point.



Sleep Criteria

CHIP\_PU

CHIP\_PU

Off

Deep Sleep

Sleep Criteria

Sleep XTAL Off

WAKEUP Events

Wakeup

XTAL\_SETTLE

CPU On

Work

Tx  Rx

Figure 3.5:Power Management

**Peripheral Interface**



**General Purpose Input/Output Interface (GPIO)**

ESP8266 has 17 GPIO pins which can be assigned to various functions by programming the registers.

Each GPIO can be configured with internal pull-up or pull-down, or set to high impedance, and when configured as an input, the data are stored in software registers; the input can also be set to edge-trigger or level trigger CPU interrupts. In short, the IO pads are bi-directional, non-inverting and tristate, which includes input and output buffer with tristate control inputs.

These pins can be multiplexed with other functions such as I2C, I2S, UART, PWM, IR Remote Control, etc.

For low power operations, the GPIOs can also be set to hold their state. For instance, when the chip is powered down, all output enable signals can be set to hold low.

Optional hold functionality can be built into the IO if requested. When the IO is not driven by the internal or external circuitry, the hold functionality can be used to hold the state to the last used state. The hold functionality introduces some positive feedback into the pad. Hence, the external driver that drives the pad must be stronger than the positive feedback. The required drive strength is small — in the range of 5μA to pull apart the latch.

**Universal Asynchronous Receiver Transmitter (UART)**

ESP8266 has two UART interfaces UART0 and UART, the definitions are as below.

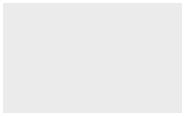
Table 3.4:Pin Definitions of UART



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pin Type | Pin Name | Pin Num | IO | Function Name |  |
|  |  |  |  |  |  |
|  | U0RXD | 25 | IO3 | U0RXD |  |
|  |  |  |  |  |  |
| UART0 | U0TXD | 26 | IO1 | U0TXD |  |
|  |  |  |  |  |
| MTDO | 13 | IO15 | U0RTS |  |
|  |  |
|  |  |  |  |  |  |
|  | MTCK | 12 | IO13 | U0CTS |  |
|  |  |  |  |  |  |



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pin Type | Pin Name | Pin Num | IO | Function Name |  |
|  |  |  |  |  |  |
| UART1 | GPIO2 | 14 | IO2 | U1TXD |  |
|  |  |  |  |  |
| SD\_D1 | 23 | IO8 | U1RXD |  |
|  |  |
|  |  |  |  |  |  |



Data transfers to/from UART interfaces can be implemented via hardware. The data transmission speed via UART interfaces reaches 115200 x 40 (4.5 Mbps).

UART0 can be used for communication. It supports fluid control. Since UART1 features only data transmit signal (Tx), it is usually used for printing log.

*Note*

*By default, UART0 outputs some printed information when the device is powered on and booting up. The baud rate of the printed information is relevant to the frequency of the external crystal oscillator. If the frequency of the crystal oscillator is 40 MHz, then the baud rate for printing is 115200; if the frequency of the crystal oscillator is 26 MHz, then the baud rate for printing is 74880. If the printed information exerts any influence on the functionality of the device, it is suggested to block the printing during the power-on period by changing (U0TXD,U0RXD) to (MTDO,MTCK).*

**Pulse-Width Modulation (PWM)**

ESP8266 has four PWM output interfaces. They can be extended by users themselves. The pin definitions of the PWM interfaces are defined as below.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Table 3-5. Pin Definitions of PWM | | |
| Pin Name | Pin Num | IO | Function Name |
|  |  |  |  |
| MTDI | 10 | IO12 | PWM0 |
|  |  |  |  |
| MTDO | 13 | IO15 | PWM1 |
|  |  |  |  |
| MTMS | 9 | IO14 | PWM2 |
|  |  |  |  |
| GPIO4 | 16 | IO4 | PWM3 |
|  |  |  |  |

The functionality of PWM interfaces can be implemented via software programming. For example, in the LED smart light demo, the function of PWM is realized by interruption of the timer, the minimum resolution reaches as much as 44 ns. PWM frequency range is adjustable from 1000 μs to 10000 μs, i.e., between 100Hz and 1 kHz. When the PWM frequency is 1 kHz, the duty ratio will be 1/22727, and over 14 bit resolution will be achieved at 1 kHz refresh rate.

**IR Remote Control**

One Infrared remote control interface is defined as below.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Table 3.6: Pin Definitions of IR Remote Control | | |
| Pin Name | Pin Num | IO | Function Name |
|  |  |  |  |
| MTMS | 9 | IO14 | IR Tx |
|  |  |  |  |





|  |  |  |  |
| --- | --- | --- | --- |
| Pin Name | Pin Num | IO | Function Name |
|  |  |  |  |
| GPIO5 | 24 | IO5 | IR Rx |
|  |  |  |  |

The functionality of Infrared remote control interface can be implemented via software programming. NEC coding, modulation, and demodulation are used by this interface. The frequency of modulated carrier signal is 38 kHz, while the duty ratio of the square wave is 1/3. The transmission range is around 1m which is determined by two factors: one is the maximum value of rated current, the other is internal current-limiting resistance value in the infrared receiver. The larger the resistance value, the lower the current, so is the power, and vice versa. The transmission angle is between 15° and 30° which is determined by the radiation direction of the infrared receiver.

**ADC (Analog-to-Digital Converter)**

ESP8266 is embedded with a 10-bit precision SARADC. TOUT (Pin6) is defined as below.

|  |  |  |
| --- | --- | --- |
|  | Pin Definition of ADC | |
| Pin Name | Pin Num | Function Name |
|  |  |  |
| TOUT | 6 | ADC Interface |
|  |  |  |



**LED Light and Button**

ESP8266 features 17 GPIOs, all of which can be assigned to support various functions of LED lights and buttons. Definitions of some GPIOs that are assigned with certain functions in demo application design are shown as below.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Table 3.6: Pin Definition of LED and Button | | |
| Pin Name | Pin Num | IO | Function Name |
|  |  |  |  |
| MTCK | 12 | IO13 | Button (Reset) |
|  |  |  |  |
| GPIO0 | 15 | IO0 | Wi-Fi Light |
|  |  |  |  |
| MTDI | 10 | IO12 | Link Light |
|  |  |  |  |



Altogether three interfaces have been defined, one is for the button, while the other two are for LED light. Generally, MTCK is used to control the reset button, GPIO0 is used as an signal to indicate the Wi-Fi working state, MTDI is used as a signal light to indicate communication status between the device and the server.

**CH340 USB to serial chip**

CH340 is a USB bus convert chip and it can realize USB convert to serial interface, USB convert to IrDA infrared or USB convert to printer interface.

In serial interface mode, CH340 supplies common MODEM liaison signal, used to enlarge asynchronous serial interface of computer or upgrade the common serial device to USB bus directly. More detail about USB convert to printer interface please referring to the second manual CH340DS2.

In infrared mode, add infrared transceiver to CH340 can compose USB infrared adapter, realize SIR infrared communication.

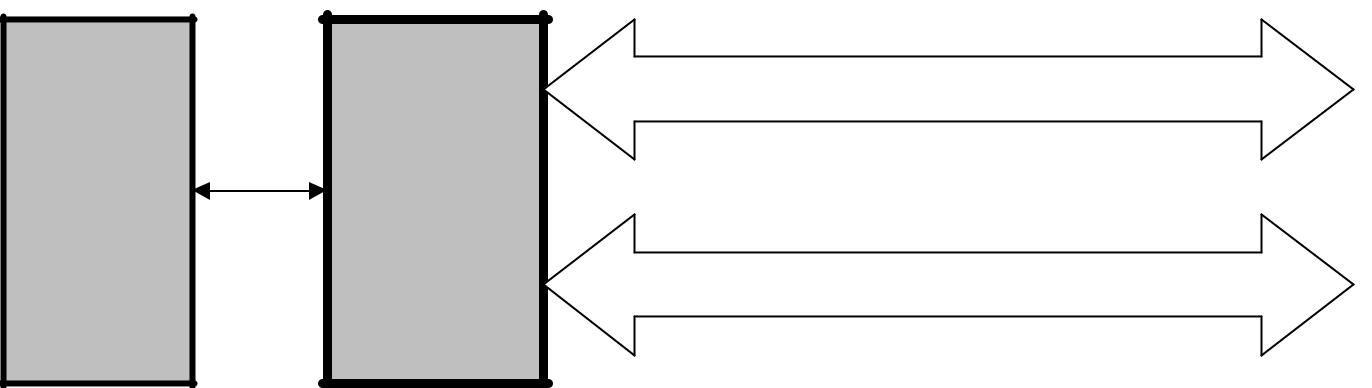


Figure 3.6: USB TO SERIAL CHIP

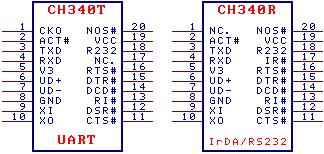
UART/RS232/IrDA infrared SIR

|  |  |  |  |
| --- | --- | --- | --- |
| Computer | USB | CH340 |  |
| or other | Convert |  |
|  |  |
| USB host |  | Chip | Parallel printer to usb printer |

**Features**

* + Full speed USB device interface, conforms to USB Specification Version 2.0, only needs crystal and capacitance external.
  + Emulate standard serial interface, used to upgrade the former peripheral device, or add excess serial interface through USB.
  + Totally compatible with serial application program in computer endpoint Windows operation system.
  + Hardware full duplex serial interface, set transceiver buffer, supports communication baud rate varies from 50bps to 2Mbps.
  + Supports common MODEM liaison signal RTS, DTR, DCD, RI, DSR and CTS.
  + Through adding level converter equipment to supply RS232, RS485, RS422 and other interface.
  + Supports IrDA criterion SIR infrared communication, supports baud rate varies from 2400bps to 115200bps.
  + For it is through USB converts to serial interface, only compatible with application layer not totally.
  + Software compatible with CH341, using drive of CH341 directly.
  + Support 5V and 3.3V source voltage.
  + Supply SSOP-20 package without lead, compatible with RoHS.

**3. Package**



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Package shape | Width of plastic | | Pitch of Pin | | Instruction of package | Ordering type |
|  |  |  |  |  |  |  |
| SSOP-20 | 5.30mm | 209mil | 0.65mm | 25mil | Shrink small outline package of 20-pin | CH340T |
| SSOP-20 | 5.30mm | 209mil | 0.65mm | 25mil | Shrink small outline package of 20-pin | CH340R |
|  |  |  |  |  |  |  |

**4. Pins( Table 3.7)**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Pin No. | |  | Pin Name | |  | Pin Type | | Pin Description(description in bracket is only about |  |
|  |  |  | CH340R) |  |
|  |  |  |  |  |  |  |  |  |  |
| 19 | |  |  | VCC | |  | POWER | | Positive power input port, requires an external 0.1uF |  |
|  |  |  | power decoupling capacitance |  |
|  |  |  |  |  |  |  |  |  |  |
| 8 | |  |  | GND | |  | POWER | | Public ground, ground connection for USB |  |
|  |  |  |  |  |  |  |  |  | connects of VCC to input outside power while 3.3V, |  |
| 5 | |  |  | V3 | |  | POWER | | connects of 0.01uF decoupling capacitance outside |  |
|  |  |  |  |  |  |  |  |  | while 5V |  |
| 9 | |  |  | XI | |  | IN | | Input of crystal oscillator, attachment of crystal and |  |
|  |  |  | crystal oscillator capacitance outside |  |
|  |  |  |  |  |  |  |  |  |  |
| 10 | |  |  | XO | |  | OUT | | Opposite output of crystal oscillator, attachment |  |
|  |  |  | crystal and crystal oscillator capacitance outside |  |
|  |  |  |  |  |  |  |  |  |  |
|  | 6 |  |  | UD+ |  |  | USB signal |  | Directly connect to D+ data wire of USB bus, set up |  |
|  |  |  |  |  |  | pull-up resistor internal |  |
|  |  |  |  |  |  |  |  |  |  |
| 7 | |  |  | UD- | |  | USB signal | | Directly connect to D- data wire of USB bus |  |
| 20 | |  |  | NOS# | |  | IN | | Forbid USB device suspending, active with low, set |  |
|  |  |  | pull-up resistor internal |  |
|  |  |  |  |  |  |  |  |  |  |
| 3 | |  |  | TXD | |  | OUT | | Serial data output(opposite phasic output of CH340R) |  |
| 4 | |  |  | RXD | |  | IN | | Serial data input, set up controlled pull-up and |  |
|  |  |  | pull-down resistor |  |
|  |  |  |  |  |  |  |  |  |  |
| 11 | |  |  | CTS# | |  | IN | | MODEM liaison input signal, clear sending, active |  |
|  |  |  | low(high) |  |
|  |  |  |  |  |  |  |  |  |  |
| 12 | |  |  | DSR# | |  | IN | | MODEM liaison input signal, data equipment ready |  |
|  |  |  | active with low(high) |  |
|  |  |  |  |  |  |  |  |  |  |
| 13 | |  |  | RI# | |  | IN | | MODEM liaison input signal, oscillate ring to prompt, |  |
|  |  |  | active with low(high) |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 14 | DCD# | OUT | MODEM liaison input signal, carrier wave detection, |  |  |
|  | active with low(high) |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | 15 | DTR# | OUT | MODEM liaison output signal, data endpoint is ready, |  |  |
|  | active with low(high) |  |  |
|  |  |  |  |  |  |
|  | 16 | RTS# | OUT | MODEM liaison output signal, request to send, active |  |  |
|  | with low(high) |  |  |
|  |  |  |  |  |  |
|  |  |  |  | CH340T:negative phasic clock output |  |  |
|  | 2 | ACT# | OUT | (CH340R:USB configuration is finished state output, |  |  |
|  |  |  |  | active with low) |  |  |
|  | 18 | R232 | IN | Assistant RS232 enable, active with high, set up |  |  |
|  | pull-down resistor internal |  |  |
|  |  |  |  |  |  |
|  |  | NC. | NC. | CH340T: unconnected, must be suspended |  |  |
|  |  |  |  |  |  |  |
|  | 17 | IR# | IN | CH340R:Serial interface mode set input, set up pull-up |  |  |
|  |  | resistor internal, low level is SIR infrared serial |  |  |
|  |  |  |  | interface, high level is common serial interface |  |  |
|  | 1 | CKO. | OUT(NC.) | CH340T: clock output |  |  |
|  |  |  |  |  |  |
|  | NC |  | CH340R:unconnected, must be suspend |  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |  |

**Function description**

CH340 chip set up USB pull-up resistor internal, UD+ and UD- pins must be connected to USB bus directly.

CH340 chip set up power up reset circuit internal.

When CH340 chip is working normally, the outside must supply 12MHz clock signal to XI pin. In generally, clock signal is generated by inverter in CH340 through oscillating of crystal keeping frequency. A crystal of 12MHz between XI and XO, XI and XO connect a high frequency oscillator capacitance to ground respectively can compose the peripheral circuit.

CH340 chip supports 5V and 3.3V power voltage. When using 5V source power, the VCC input 5V power and the pin of V3 must connect with 4700pF or 0.01uF decoupling capacitance. When using 3.3V power voltage, connects V3 with VCC, and input 3.3V power voltage. And the other circuit voltage which is connected with CH340 is no more than 3.3V.

CH340 automatically supports USB device suspending to save power consume. NOS# is low**–**level can forbid USB device suspending.

In asynchronous serial interface mode, CH340 chip contains these pins: data transfer pin, MODEM liaison signal pin and assistant pin.

Data transfer pin contains: TXD pin and RXD pin. When serial interface is idle, RXD must be high-level. If R232 is high-level, use assistant RS232 function, then RXD pin automatically inserts a inverter internal, and low-level is in default. When serial interface output is free, the TXD in CH340H and CH340T is high level, TXD in CH340R is low-level.

MODEM liaison signal pin contains: CTS#, DSR#, RI#, DCD# and RTS#. All these MODEM liaison signal are controlled by computer application program and application program defines function.

Assistant pin contains: IR#, R232, CKOH, CKOL and ACT#. When IR# is low-level, starts infrared serial interface mode. R232 is used to control assistant RS232 function. When R232 is high-level, RXD pin automatically insert a inverter internal, and output opposite phase clock from CKOH and CKOL. ACT# is USB device configuration finished state output (such as USB infrared adapter is ready) when R232 is low-level. IR# and R232 only be detected once a time after power reset.

CH340 set separate transceiver buffer internal and supports simplex, semiduplex and full duplex asynchronous serial communication. Serial data contains one low-level start bit , eight or nine data bit and one high-level stop bit. Supporting odd check/even check/flag check/blank check. CH340 supports common baud rate: 50,75,100,110,134.5,150,300,600,900,1200,1800,2400,3600,4800,9600,14400,19200,28800, 33600,38400, 56000,57600,76800,115200,128000,153600,230400,460800,921600,1500000,2000000 and so on. The baud rate error of serial transfer signal is less than 0.3%, and permission baud rate error of serial receive signal is not less than 0.2%.

In the WINDOWS operation system of computer endpoint, drive program of CH340 can communicate standard serial interface. So the mostly original serial interface application program is totally compatible, and without any modify.

CH340 can be used to upgrade the former serial interface peripheral equipment, or add extra serial interface for computer via USB bus. Supply RS232, RS485, RS422 and other interface via adding level change device.

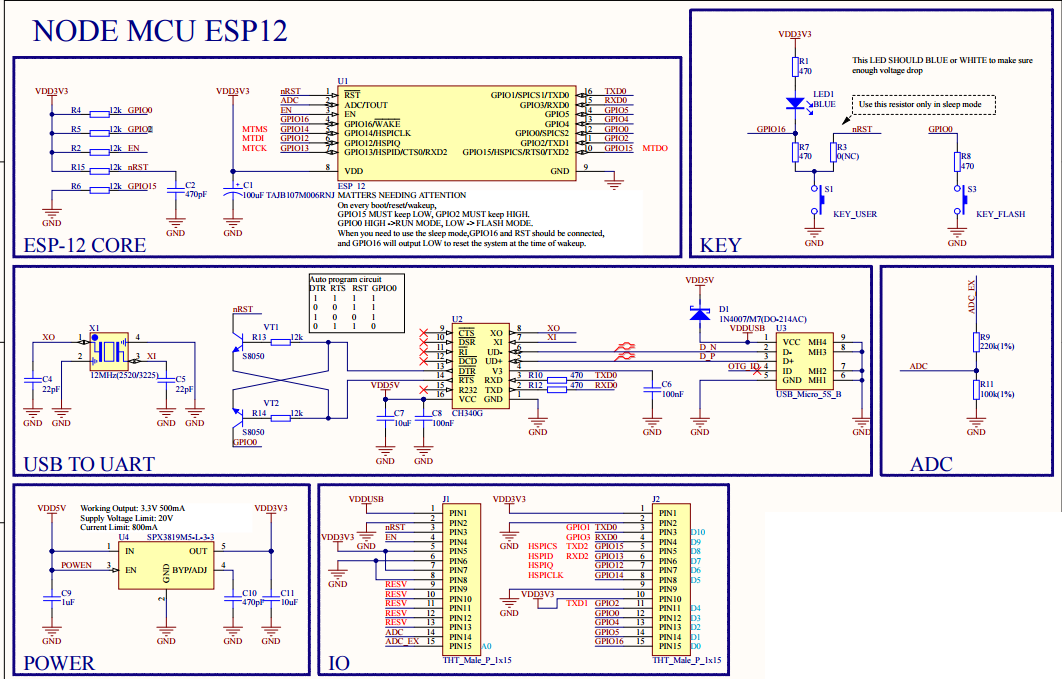
Only add infrared transceiver, CH340 can add SIR infrared adapter for computer via USB bus, realize infrared communication between computer and peripheral equipment which is according to IrDA criterion.

**SPX3819 Low voltage LDO voltage regulator**

The SPX3819 is a positive voltage regulator with a low dropout voltage and low noise output. In addition, this device offers a very low ground current of 800µA at 100mA output. The SPX3819 has an initial tolerance of less than 1% max and a logic compatible ON/OFF switched input. When disabled, power consumption drops to nearly zero. Other key features include reverse battery protection, current limit, and thermal shutdown. The SPX3819 includes a reference bypass pin for optimal low noise output performance. With its very low output temperature coefficient, this device also makes a superior low power voltage reference.

Internal architecture of the NodeMCU board along with its internal components

Figure 3.8: Internal Architecture of NodeMCU



# BASICS OF REAL TIME STREAMING PROTOCOLS

A camera source is required to stream live video footage from the robot to the user. The purpose of this chapter is to explain the process of setting up a Real Time Streaming service from an android camera fixed at the robot's chassis to a Cloud-based media server. The streaming from media server to the client's android phone is discussed in the next module.

# RTSP STREAMING: REAL TIME STREAMING PROTOCOL

RTSP or Real Time Streaming Protocol is the most fundamental network protocol required for live streaming applications. It finds a huge importance in entertainment and communication with media servers. RTSP makes it possible to deliver video and audio in different qualities in real time across a client-server-client interface. RTSP also allows basic media commands such as "play", "pause", "change quality", "set volume" and timeline switching like common media players.

Before we get into the crust of why RTSP is so widely popular, let us first examine two other network protocols: UDP (User Datagram Protocol) and TCP (Transmission Control Protocol).

**UDP:**

User Datagram Protocol is one of the core members of the Internet Protocol Suite working in the Transport Layer. UDP provides a light-weight, fast, best effort message transmission from one application to another through the network layer without creating any distinct application-to-application connection channel.

The UDP header consists of four fields each of 2 bytes in length, containing Source port, Destination Port, UDP length and checksum.

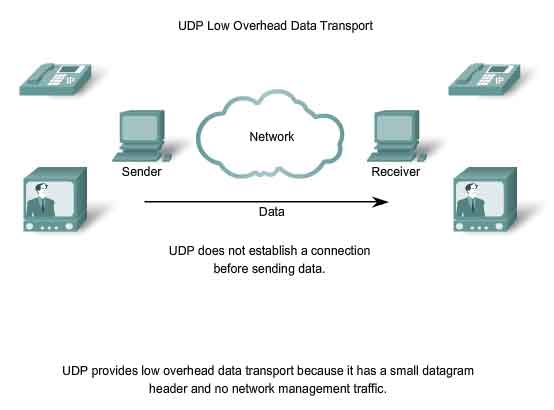


FIGURE 4.1:UDP

Because of its lightweight header and no terminal-to-terminal connection establishment, UDP can offer a very efficient communication transport to some applications, but has no inherent congestion control or reliability. This means that when the network is congested, many packets of data can be lost or ignored by the network. Also, there is no feedback system that allows the receiver port to acknowledge the received data, nor the sender port to confirm a successful transmission. Moreover, the order at which data was sent may get altered when receiving at the other terminal. Lastly, since there is no pre-defined connection between the sender and the receiver, UDP does not provide any communications security.

UDP is message oriented and is suitable for simple query-response protocols. UDP is stateless, that is it treats each request as an independent transaction and does not depend on past request or response. This allows UDP to allow multiple client user transactions suitable for streaming media applications.

**TCP:**

TCP (Transmission Control Protocol) commonly referred to as TCP/IP protocol is another transport layer protocol in the Internet Protocol (IP) suite. Like Internet Protocols, it defines how electronic devices (like computers) should be connected over the Internet, and how data should be transmitted between them

Unlike UDP, TCP is a connection based protocol. This means that a connection must be established between the source and destination ports before transmission of data is possible, thus providing a host-to-host connectivity in the Transport Layer of the Internet Model. The sender sends a request for connection to the receiver port. The receiver port accepts this request by responding back. The connection is then established with another feedback sent from the sender to the receiver. This process of confirming the transmission of data at the receiver end is called Positive Acknowledgement with Re-transmission, and it pretty much guarantees reliability of successful transmission.

TCP detects common network problems like message-duplication, loss of message, congested networks and other protocol problems. Then it requests retransmission of lost data, rearranges out-of-order data, and even helps minimize network congestion to reduce the occurrence of the other problems. The destination application never receives a latter message before receiving all the previous messages. Thus, TCP abstracts the application's communication from the underlying networking details.

Because of this features, TCP is a stateful protocol. A stateful protocol is a protocol in which part of the data that is exchanged between the client and the server systems includes state information. Both systems keep track of the state of the communication session.

TCP is great for a reliable, accurate delivery of data. The problems lie when some data gets lost or delayed due to network issues. Since the destination application never receives the data until all the predecessor packages have been received, it can cause delays. While TCP is optimized for accurate delivery, timing and speed is not it's strong suit.

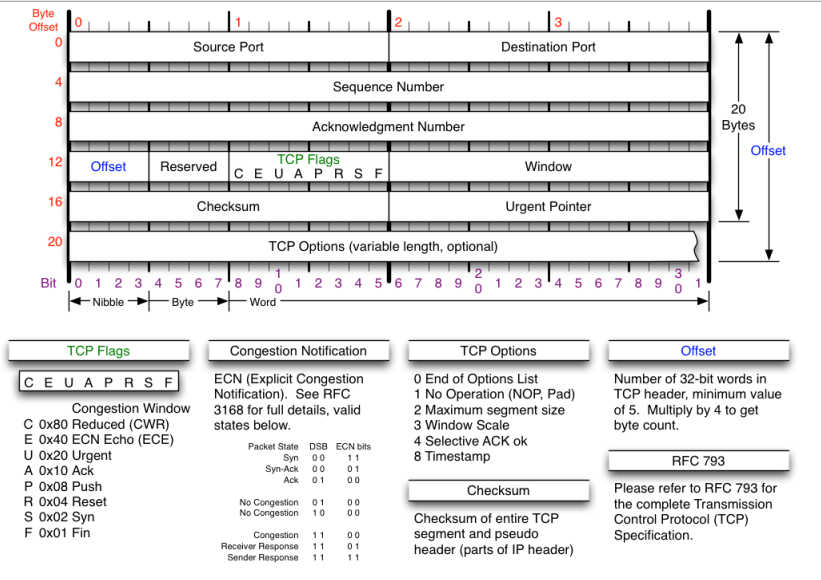


FIGURE 4.2:TCP OVERHEAD DETAILS

**GETTING BACK TO RTSP**

In most transport protocols, the client asks for some piece of content, the content is delivered using *TCP* or *UDP*, and then the client application can display the content to the user. While these mechanisms are suitable for a large number of applications in the Internet, there also exists a requirement to deliver content, be it images, audio, video, or a combination of all three, in real time.

Imagine if a user were to try to watch a full-screen video file of a one-hour movie using *HTTP* or *FTP* as the Application layer protocol. The movie file might be several hundred megabytes, if not several gigabytes, in size. Even with modern broadband services deliverable to the home, this type of large file size does not fit well in the "download then play" model.

*RTSP* uses a combination of reliable transmission over *TCP* (used for control) and best-efforts delivery over *UDP* (used for content) to stream content to users. By this, we mean that the file delivery can start and the client-side application can begin displaying the audio and video content before the complete file has arrived. In terms of our one-hour movie example, this means that the client can request a movie file and watch a "live" feed similar to how one would watch a TV. Along with this "on demand" type service, *RTSP* also enables the delivery of live broadcast content that would not be possible with traditional download and play type mechanisms.

The transmission of streaming data itself is not a task of RTSP. Most RTSP servers use the Real-time Transport Protocol (RTP) in conjunction with Real-time Control Protocol (RTCP) for media stream delivery.

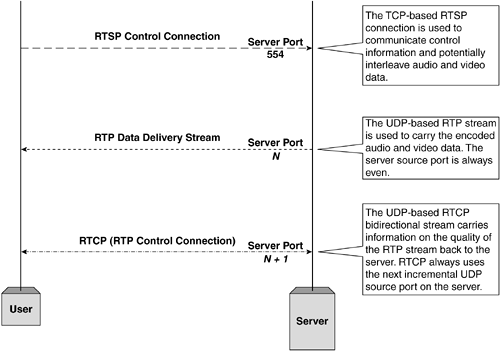


FIGURE 4.3: RTSP PROTOCOL: RELATION BETWEEN RTP AND RTCP

**RTP (REAL TIME TRANSPORT PROTOCOL):**  RTP is an UDP based transport protocol that provides end-to-end delivery services for data with real-time characteristics such as audio and video streaming. These services include payload type identification, sequence numbering, time-stamping and delivery monitoring. The *RTP* flow when delivering the content is unidirectional from the server to the client

#### Real Time Control Protocol (RTCP):

RTCP is a complimentary protocol to RTP and is a bidirectional UDP-based mechanism to allow the client to communicate stream-quality information back to the object server. While it does not transport any media data itself, the primary function of RTCP is to provide feedback on the Quality of Service in media distribution by periodically sending statistics information to participants in a streaming multimedia session.

RTSP defines control sequences useful in controlling multimedia playback. RTSP has state; an identifier is used when needed to track concurrent sessions. RTSP uses TCP to maintain an end-to-end connection and, while most RTSP control messages are sent by the client to the server, some commands travel from server to client as well. The *RTSP* protocol the rtsp:// scheme in it's URL structure. *RTSP* has a number of additional headers (such as DESCRIBE, SETUP, and PLAY) and also allows data transport out-of-band and over a different protocol, such as *RTP* described earlier.

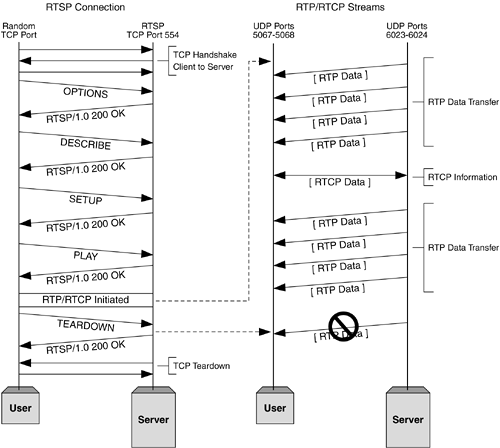


FIGURE 4.4: RTSP HEADERS AND CONNECTION

The basic steps involved in the process of streaming are as follows:

1. The client establishes a *TCP* connection to the servers for *RTSP*.
2. The client will then commence issuing a series of *RTSP* header commands that have a similar format to *HTTP*, each of which is acknowledged by the server. Within these *RTSP* commands (such as DESCRIBE, OPTIONS), the client will describe to the server details of the session requirements.
3. Once the negotiation of transport parameters has been completed, the client will issue a PLAY command to instruct the server to commence delivery of the *RTP* data stream.
4. Once the client decides to close the stream, a TEARDOWN command is issued along with the Session ID instructing the server to cease the *RTP* delivery associated with that ID

For our live streaming purposes we use RTSP. We set up a media server that accepts RTSP sources for live streaming. We will set up an android application that captures images through it's camera and streams it to our media server using RTSP protocol.

Before we move on to the actual coding aspects of our app, let us look into RTMP (Real Time Messaging Protocol). We will use RTMP to retrieve the live feed from the server to the client app that controls the robot's movement through MQTT.

Real Time Messaging Protocol (RTMP) is a proprietary communications protocol used primarily by Flash, but implemented by some other software as well. It's usually used over TCP, though this isn't a requirement. It operates in the application through session layers. Its importance is a direct result of the ubiquity of Flash, and it will decline as the use of Flash does. Android no longer supports RTMP or Flash directly. We have used third-party open-source libraries like libstreaming and vitamio to simulate RTMP streams for our app. After establishing a TCP connection, an RTMP connection is established first performing a handshake through the exchange of 3 packets from each side.

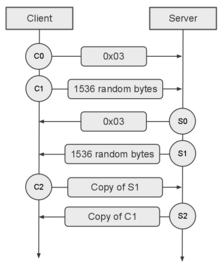


FIGURE 4.5: RTMP HANDSHAKE

After the "three-way" handshake is established with the server, a connection is formed between the server and the client and the video message can be streamed using the  "createStream" invocation followed by a ping message, followed by a "play" invocation.

The takeaway from this chapter is that we need an rtsp:// ping to stream live footage from our android phone to the server, and an rtmp:// ping with a Flash based player to view the live footage from the server in our android phones.

We discuss setting up a Media Server in chapter 6. The applications to stream from client to server and back to client will be discussed in chapter 7 and 8 respectively.

**WORKING**

# SETTING UP MQTT SERVER

For the purpouse of our project, we needed a simplistic and lightweight server which could support a type of connection where we could send a message over the public internet to it via a client which could be easily read by our robot in real time. The MQTT protocol fit this requirement perfectly as it is not only lightweight but also supports the PUBLISH and SUBSCRIBE methodology where the controller client would ‘publish’ its signals to a specific TOPIC and the robot can access those signals by simply subscribing to the topic.

But since the signals would be sent over the public internet, it would be more preferable to use a service for which we did not have to run a separate computer for the server. Hence a CLOUD BASED MQTT server was the best choice and we availed the services of cloudmqtt.com which is an MQTT service that is hosted in the form of a cloud server.

In the account we created in cloudmqtt.com, the server details are:

SERVER ADDRESS: m12.cloudmqtt.com

SERVER PORT: 12879

USERNAME: pphdjxet

PASSWORD: \_ur1NqE1w5gq

We created a topic called “test” in our account. This is the topic where we intend to publish and read MQTT messages from.

We will later require all the information mentioned above to connect to our cloud MQTT server and publish the relevant messages. The Node MCU will subscribe to the “test” topic and process the messages.

# SETTING UP A CLOUD BASED MEDIA SERVER FOR VIDEO STREAMING

Before we get into the actual streaming process, we need to set up a cloud-based media server to facilitate our Client-Server interactions. We have chosen WOWZA Media Server to fulfill the purposes for our live streaming.

Wowza media server is one of the most popular and critically acclaimed media server choice in the world right now, having won the Streaming Media Reader's Choice Award 2015 and the Cloud-based Streaming Server Award in the same year. Wowza Streaming Engine is a robust, customizable media server software that powers reliable high-quality video and audio streaming.  It takes in any format, transcodes it once, and reliably delivers it in multiple formats to any device, anywhere, anytime. Wowza software fits with nearly every streaming architecture, and provides a flexible pricing model and deployment options allows users to scale easily as needs change. Wowza supports RTSP live source streaming and RTMP streaming to flash players as well.

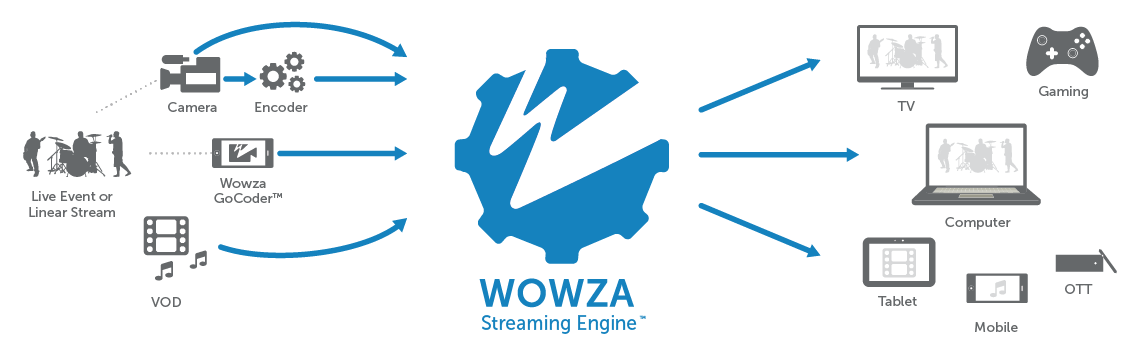


FIGURE 6.1: WOWZA STREAMING ENGINE

**PROCESS OF SETTING UP A WOWZA CLOUD BASED SERVER ACCOUNT**

In this section we will explain how to set up a Wowza cloud based account. Since we have used the Wowza System for educational purposes and we do not actively endorse or promote the Wowza Organization, we will keep the discussion as brief as possible. Extensive tutorials are present in Wowza's official website.

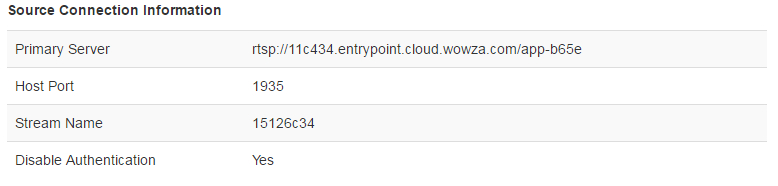
Wowza provides a free trial version of Cloud server available to use for a month and 5 transcoding hours. After setting up an account, the user is shown the media control panel. Create a new live stream and give it a name(we named it "myStream"). We set the server broadcast location to Asia/Japan which was the closest spot to our location. Setting the broadcast location closer will make the streaming faster. Set the source of the live stream as "Other RTSP source" or "GoCoder". GoCoder is a ready-made Wowza Android/iOS application used for live streaming. We will however make our own live streaming application instead of using GoCoder. The coding will be discussed in the next module.   
  
We set the source aspect ratio to 4:3 and resolution at 320X240 (240p). The video is deliberately taken to be of low resolution since we do not want unnecessary hindrance in our live streaming. We can check the disable authentication field to Yes or No. If one selects NO, one has to create an source account with a user name and password. This username and password will be later authenticated by the server before allowing a source feed. We have avoided the authentication process for simplicity. Lastly, set playback destination size and host page title. Confirm the changes and one is ready to go.

TABLE 6.1:WOWZA SERVER DETAILS

A

After setting up the stream, the following information about source connection is visible in the myStream page. As we clearly see from the information, our server can be accessed by rtsp://11c434.entrypoint.cloud.wowza.com. The app-b65e is actually a channel name for our workspace. Wowza allows users to create multiple workspaces. Our stream name is encoded as 15126c34. It can however also be accessed as "myStream", like we originally set it up to be. The Host Port is set at 1935, which is the default Wowza port used for all their applications. Since we have disabled authentication, we would not need to provide external source username and password to access streaming rights. We can get away with disabling authentication here since our project is experimental, educational and small-scale.

We click on the "Start Stream" button to make the streaming server go live.

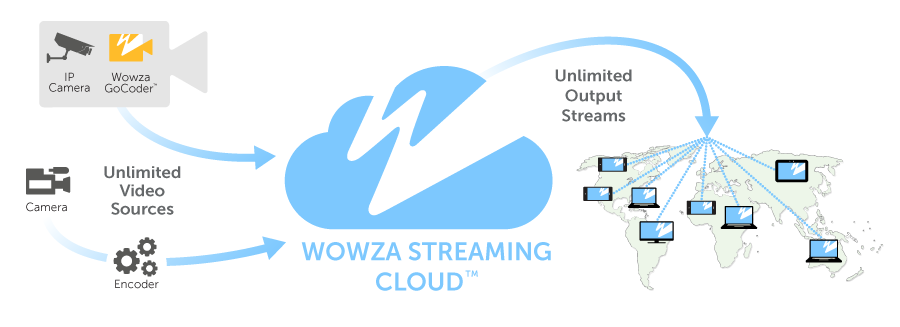


FIGURE 6.2: WOWZA STREAMING CLOUD

We are all set to now develop our live streaming app. We have the required knowledge of RTSP and RTMP protocols. We have set up our media server that will facilitate live streaming from android phone to the server. We will use Real Time Streaming Protocol (RTSP) to connect from android camera fixed on our robot chassis to stream live footage to server; and Real Time Messaging Protocol (RTMP) to retrieve the live feed to our client app in real time. This client app will also contain features to manipulate the movements of our robot using MQTT servers. In the next module, we explain the first app required to transmit live feed from phone to server.

CHAPTER 7

SETTING UP THE ROBOT CHASSIS AND THE MOTOR DRIVER

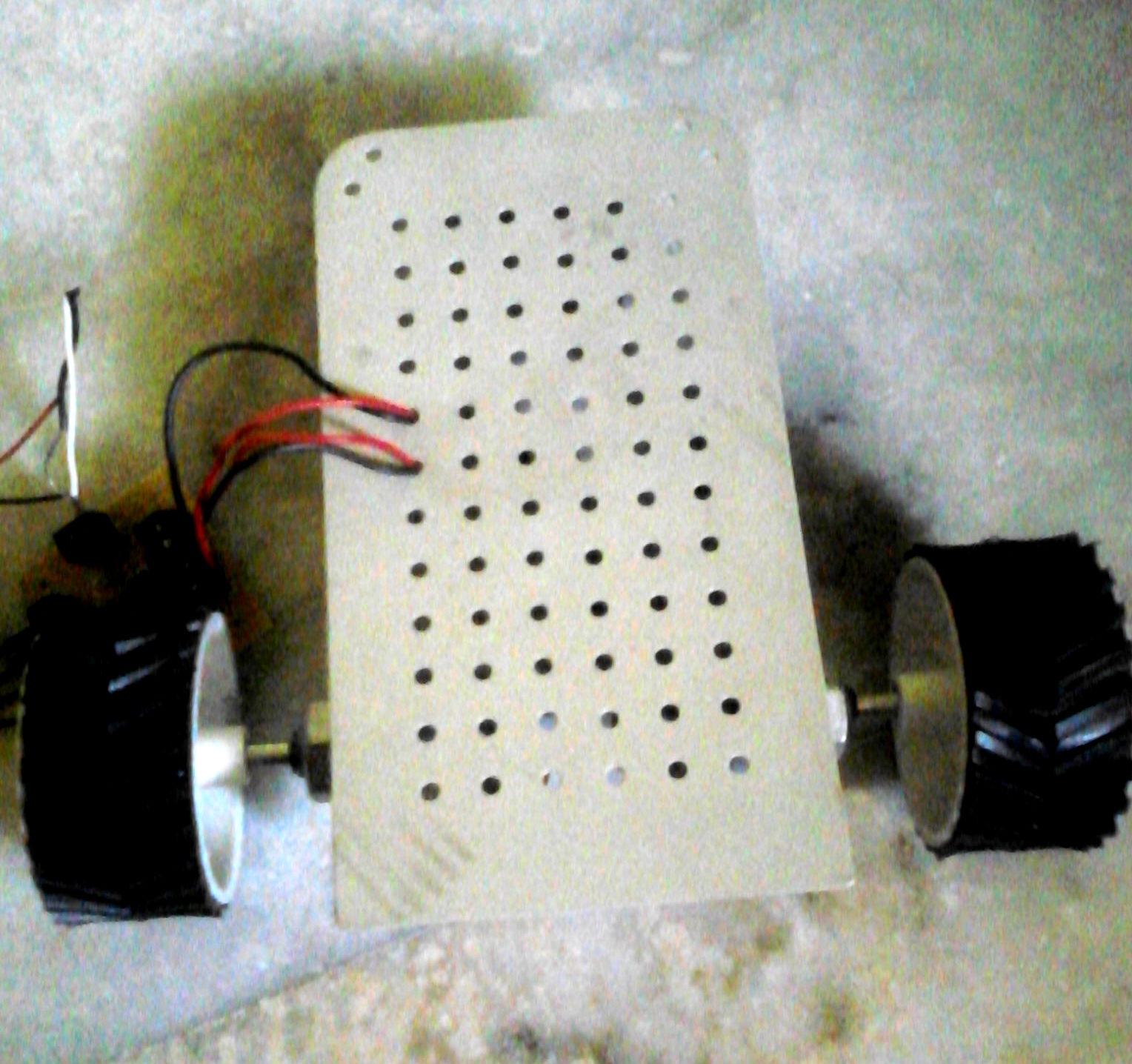


FIGURE 7.1: ROBOT CHASSIS

In this part we will discuss how the main body or the chassis of our robot was connected with the other components, namely the NodeMCU, power supply, the camera cellphone and the motor driver. We used a readymade chassis with wheels and motors attached. Therefore, installation of the motor is beyond the scope of this project. The chassis was drilled with holes to help attach the motor driver. The dimension of the chassis was measured at 19.5cmX10.5cm. This is ample space for accommodating all of our components properly.



FIGURE 7.2: The Wheel

For the robot chassis, we have used a readymade robot chassis. We used two rear wheels (with motors) and one accessory front wheel. The wheels have zigzagged patterns that smoothens traversal.



FIGURE 7.3: THE MOTOR

The motors used are rated at 12V-200 rpm. Each motor has two terminal wires coming out of it and requires a 12V voltage difference across it's terminal to run at the rated rpm. We now need a motor driver to moderate the voltage difference across the two terminals and conduct the revolutions of the motor.

**MOTOR DRIVER (L293D) MODULE**

L293D is a typical Motor driver or Motor Driver IC which allows DC motor to drive on either direction. L293D is a 16-pin IC which can control a set of two DC motors simultaneously in any direction. It means that you can control two DC motor with a single L293D IC.

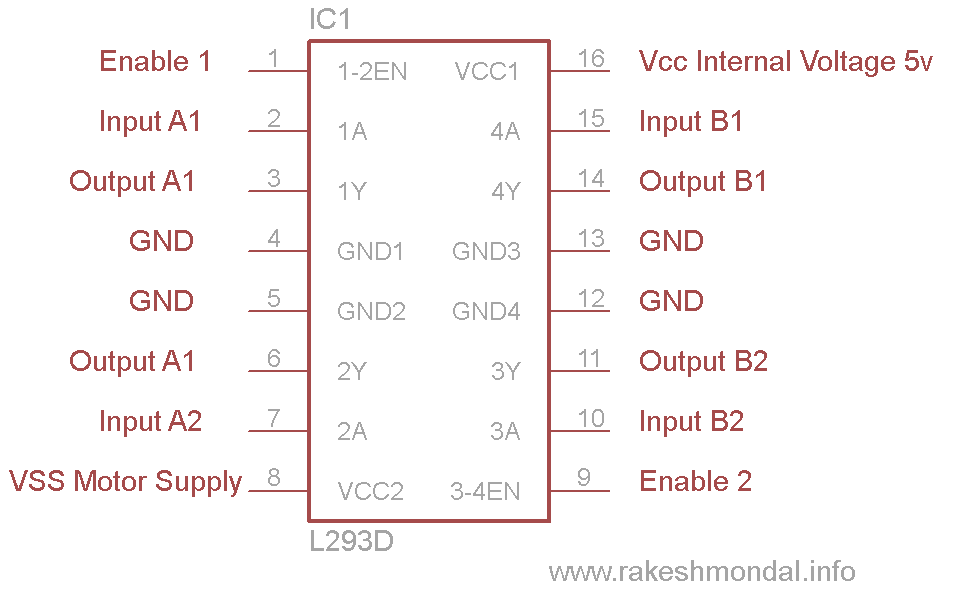


FIGURE 7.4: L293D MOTOR DRIVER IC PIN DIAGRAM

There are 4 input pins for l293d, pin 2,7 on the left and pin 15 ,10 on the right as shown on the pin diagram. Left input pins will regulate the rotation of motor connected across left side and right input for motor on the right hand side. The motors are rotated on the basis of the inputs provided across the input pins as LOGIC 0 or LOGIC 1.

Let's consider a Motor connected on left side output pins (pin 3,6). For rotating the motor in clockwise direction the input pins has to be provided with Logic 1 and Logic 0.

• **Pin 2** = **Logic 1** and **Pin 7**= **Logic 0** | Clockwise Direction  
• **Pin 2** = **Logic 0**and **Pin 7**= **Logic 1** | Anticlockwise Direction  
•**Pin 2**= **Logic 0** and **Pin 7** = **Logic 0** | Idle [No rotation] [Hi-Impedance state]  
• **Pin 2**= **Logic 1** and **Pin 7** = **Logic 1** | Idle [No rotation]

The Motor Driver is discussed at more detail in Chapter 9.

CHAPTER 8

SETTING UP THE NODE MCU

**Selection of the microcontroller**

For the purpose of our project, we needed a microcontroller (with an USB to serial chip) and a Wi-Fi module along with the microcontroller to connect to the internet with. The ESP8266 chip was the best choice for the Wi-Fi module. However, the ESP8266 also has its own processing abilities which were enough for the purpose of our project. The Node MCU development board based on the ESP8266 was a board with inbuilt USB to serial chip and voltage regulator with which we could utilize the processing power of the ESP8266 itself, thus eliminating the need for a separate microcontroller and a separate Wi-Fi module by combing the two. Thus, the NodeMCU development board fulfills all the requirements for our project and hence it was the perfect choice.

**Downloading the necessary drivers for the NodeMCU development board**

Before we could begin to work with the NodeMCU board we had to download the necessary drivers required by the USB to serial converter chip of the board to work with Windows. The name of the driver was CH340G drivers and it was available in the github repository for NodeMCU. The link is mentioned in the reference page.

**Flashing the NodeMCU development board**

Flashing the NodeMCU board means burning into the the NodeMCU flash memory firmware that allows you to program the ESP8266 modules with LUA script. For this the following steps were taken:

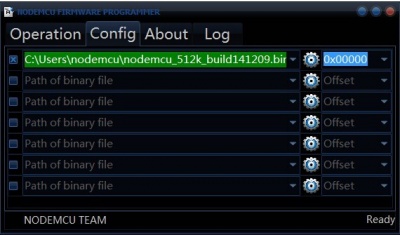
* The flasher tool called NodeMCU flasher was downloaded
* The latest NodeMCU firmware .bin file available was downloaded
* The NodeMCU was connected to the computer using a USB cable.
* The correct port number for the NodeMCU board was specified in the flasher.
* In the Advanced tab the Baud rate was selected as 115200, the flash size was selected as 32M, The SPI speed was specified as 40MHz and the SPI mode was specified as DIO
* In the Config tab the memory location was specified as 0x0000 and the downloaded firmware was selected in the left hand side as shown in figure 8.1 from the location in the computer where it was downloaded .  
    
  

FIGURE 8.1

* The board was put into flash mode by clicking in the Flash button in the Operation tab and we waited until the flashing completed when the flashing completion bar was full and the flashing was confirmed to have been completed. (as shown in following picture)

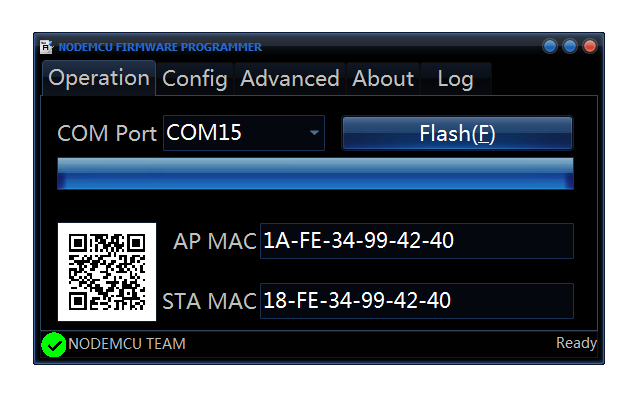


Figure 8.2

**How does the robot actually move?**

The basic flow diagram for the robot’s control is:

User publishes in MQTT Server 🡪 Node MCU subscribes to the topic and reads the value and determines which direction the user wants to move 🡪 Node MCU changes the state of GPIO 2,3,4,5 accordingly (explained later) to accommodate the required translation🡪The output of pin 2,3,4,5 is fed to the motor driver input🡪the motor driver sets the rotation of the left and right wheels in clockwise or counter-clockwise direction to facilitate movement.

We have already discussed in the previous chapter how the motor driver L293D chip reacts to different states.

Motor Driver Outputs A1 and A2 control the fate of left wheel and output B1 and B2 control the fate of right wheel.

**TABLE 8.1 : MOTOR DRIVER PIN I/O**

**Pin 2** = **Logic 1** and **Pin 7**= **Logic 0** | Clockwise Direction  
• **Pin 2** = **Logic 0**and **Pin 7**= **Logic 1** | Anticlockwise Direction  
•**Pin 2**= **Logic 0** and **Pin 7** = **Logic 0** | Idle [No rotation] [Hi-Impedance state]  
• **Pin 2**= **Logic 1** and **Pin 7** = **Logic 1** | Idle [No rotation]

We want our robot to move forward when the message is “w”, backwards when the message is “s”, right when the message is “d” and left when the message is “a”.

**TABLE 8.2: NODE MCU - MOTOR DRIVER CONNECTIONS**

|  |  |
| --- | --- |
| **GPIO PINS** | **MOTOR DRIVER INPUTS** |
| 2 | A2 (pin 2) |
| 3 | A1 (pin 7) |
| 4 | B1 (pin 15) |
| 5 | B2 (pin 10) |

This is the full overview of the user input and corresponding pin state is discussed.

**TABLE 8.3: FULL OVERVIEW OF RELATIONSHIP BETWEEN USER INPUT AND ROBOT MOVEMENT**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| User Input | NodeMCU  GPIO2/Motor  Driver Pin 3 | NodeMCU  GPIO2/ Motor  Driver Pin 3 | NodeMCU  GPIO2/Motor  Driver Pin 3 | NodeMCU  GPIO2/Motor  Driver Pin 3 | Left Wheel | Right  Wheel | Robot Movement |
| w | H | L | H | L | C | C | Forward |
| s | L | H | L | H | CC | CC | Back |
| a | L | H | H | L | CC | C | Left |
| d | H | L | L | H | C | CC | Right |
| h | L | L | L | L | Idle | Idle | Halt |

LEGEND: H:High, L:Low, C:Clockwise, CC:CounterClockwise

**Programming the NodeMCU development board**

For programming the NodeMCU development board, ESPlorer IDE was used that is ideal for programming the ESP8266 or ESP8266 based boards. In this software, first the serial port was opened at the baud rate of 9600. After the detection message was displayed in the terminal, the program was written in new LUA scipt(s)

The init.lua script is the script that always runs first when NodeMCU is powered up. We did not write our main code in the init.lua file since it is generally regarded as a bad practice. Instead we coded a separate file “script.lua” and simply redirected our init.lua file to our script.lua file.

This can be done by simply writing the following function in init.lua:

dofile(“script.lua”)

**SCRIPT.LUA**

The main objective of this code is to

a)connect to the WiFi

b) subscribe to the “test” topic

c) read the most recently published message

d)make the right GPIO pin high according to the message.

The full code can be found in the appendix. The significant portions are included here.

We connect to the Local Wifi using the code:

wifi.setmode(wifi.STATION)

wifi.sta.config("WIFINAME","WIFIPASSWORD")

We connect to our MQTT server using the following lines:

m:connect("m12.cloudmqtt.com",12879, 0, function(conn)

print("connected")

m:subscribe("test", 0, function(conn)

print("subscribed")

end)

end)

We must declare the pins that we will use as output modes. In our case that is GPIO 2,3,4 and 5.

gpio.mode (2, gpio.OUTPUT) -- ia1 - GPIO4

gpio.mode (3, gpio.OUTPUT) -- ia2 - GPIO0

gpio.mode (4, gpio.OUTPUT) -- ib2 - GPIO2

gpio.mode (5, gpio.OUTPUT) -- ib1 - GPIO5

A pin is set high or low by the command: gpio.write(PIN NUMBER, STATE)

In the following snippet, we read the message present in the topic “test” and change the ouput of pins 2,3,4 and 5 accordingly if the message (stored in variable data) is “w”.

m:on("message", function(conn, topic, data)

print(topic .. ":" )

if data == "w" then

gpio.write(2, gpio.HIGH)

gpio.write(3, gpio.LOW)

gpio.write(4, gpio.HIGH)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

end

For halting, we must change the output of all the pins to low.

if data == "h" then

gpio.write(2, gpio.LOW)

gpio.write(3, gpio.LOW)

gpio.write(4, gpio.LOW)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

end

We continue the process for moving left by setting pin 2,3,4,5 to Low,High and High,Low respectively.

if data == "a" then

gpio.write(2, gpio.LOW)

gpio.write(3, gpio.HIGH)

gpio.write(4, gpio.HIGH)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

end

The process is repeated for backwards and right traversal as well. This concludes our NodeMCU set up. Now we must connect the Node MCU to the Motor Driver.

**Connecting and wiring the components together**

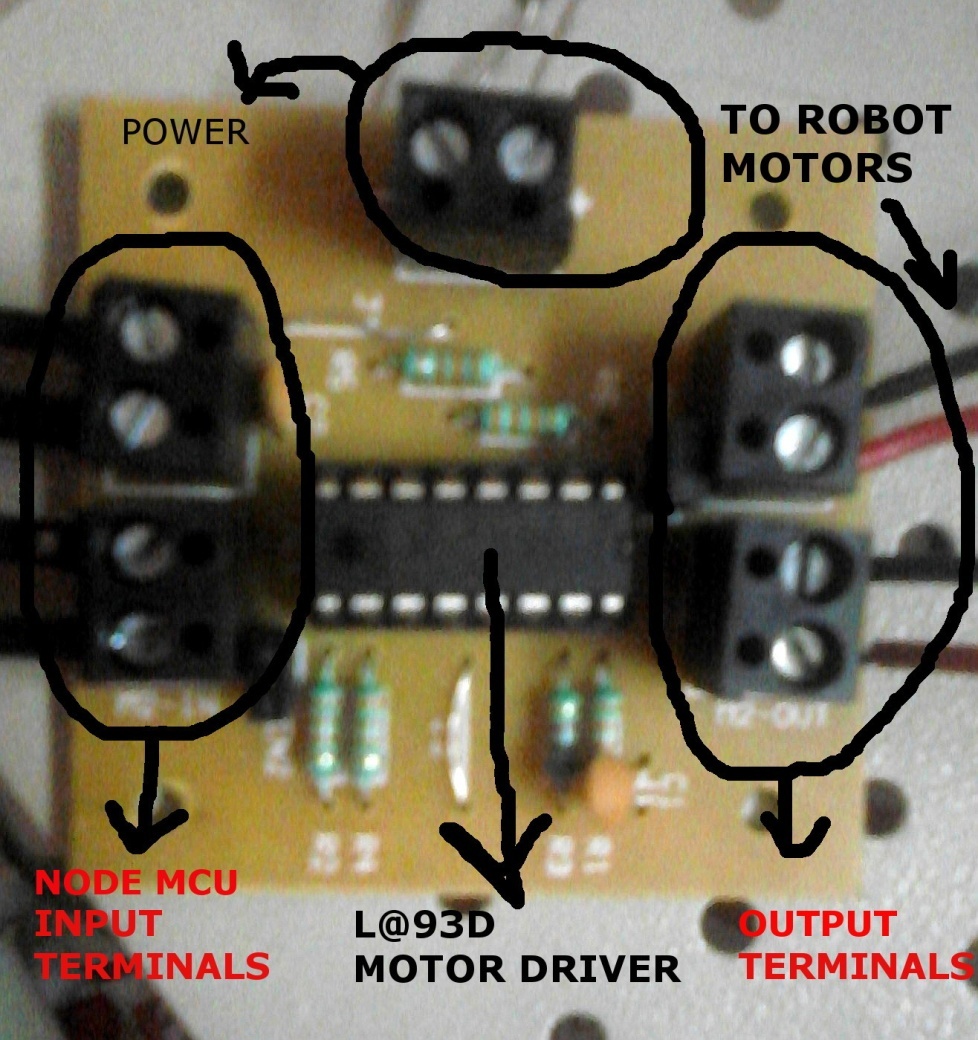


FIGURE 8.2: MOTOR DRIVER L293D MODULE

The L293D Motor Driver IC that we discussed in the previous chapter also comes in module form. There are two input nodes (four terminals) and two output nodes (four terminals). There is a power terminal that needs a 12V power supply to run. The two input terminals of a single node were connected with Pin 2(Input A1) and Pin 7(Input A2) to facilitate a single motor. The other input terminals were connected to pins 10 (Input B1) and 15 (Input B2). The inputs of these terminals were derived from our ESP8266 NodeMCU module.

The two output nodes of the Module were connected with the respective output pins of the motor driver IC. Output terminals 1 and 2 were connected with pin 3 (Output A1) and pin 6 (Output A2), and output terminals 3 and 4 were connected with pin 11 (Output B1) and pin 14 (Output B2). This output terminals were connected with the motors of the robot.

The power node was connected with a 12V power supply. The supply voltage was achieved by connecting eight 1.5V cells in series using two four-battery holders.



FIGURE 8.3: BATTERY HOLDER WITH 12V SUPPLY

The L293D module was screwed in with the robot chassis to keep it from falling off during movement.

As stated in previous modules, the NodeMCU development board is the brain and the motherboard of our system. It requires a 5V power input to function. We have used a 5V mobile charger power bank to supply the NodeMCU board power. This is a simple USB connection as shown in the figure 8.4.



FIGURE 8.4: NODE MCU POWER SUPPLY

The NodeMCU board will connect to our MQTT server after getting power supply. We are required to also connect the controlling pins with our motor driver input. As stated previously, our Lua Code is designed such that GPIO pins 2 and 3 controls motor one, and pins 4 and 5 control motor two. To match the high-low patterns of our code, we should connect GPIO pin 3, 2, 4, 5 to the motor driver (L293D Module) input pins 1, 2, 3 and 4 respectively. The aftermath of this connection is shown in Figure 8.5.

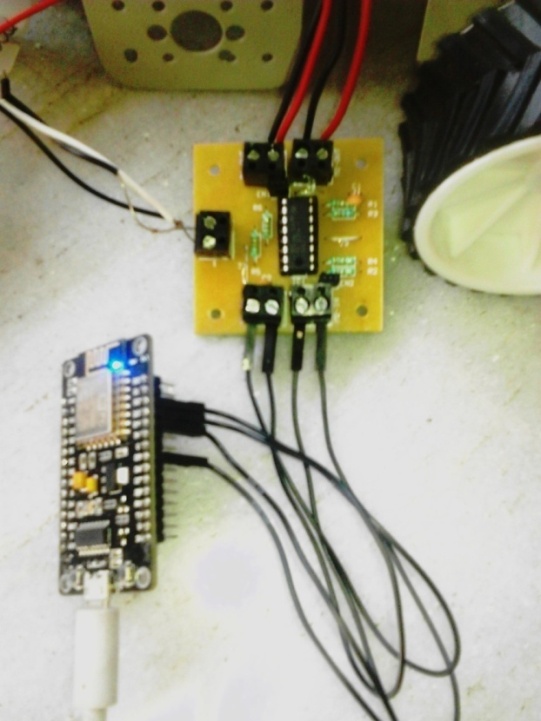


FIGURE 8.5: CONNECTING NODE MCU AND MOTOR DRIVER

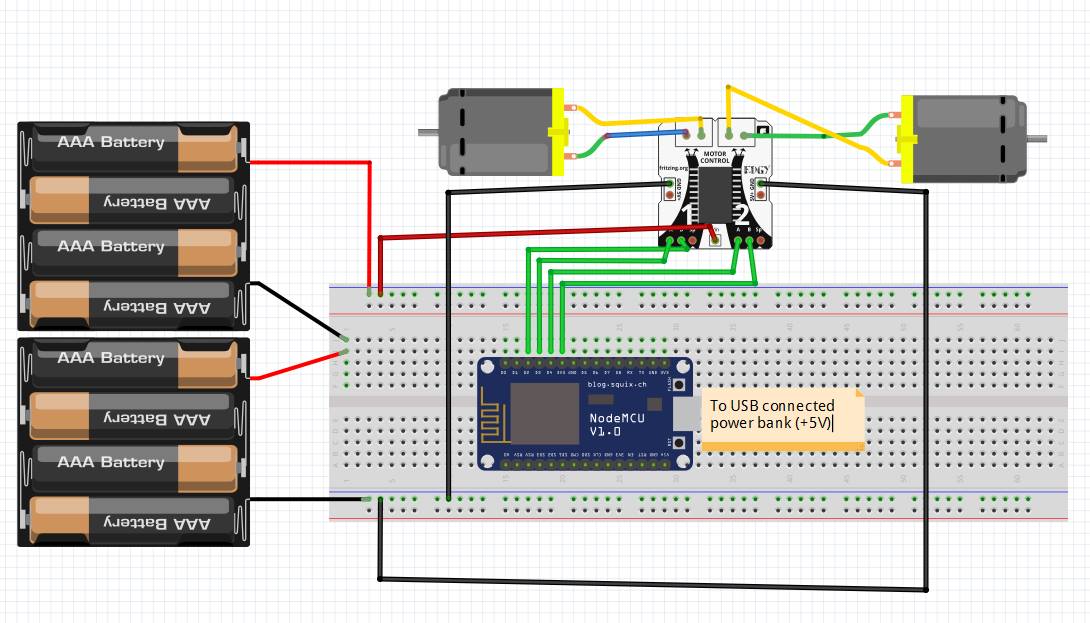
The circuit diagram of the above connection is as shown below.

FIGURE 8.6: CIRCUIT DIAGRAM

CHAPTER 9

ANDROID LIVE STREAMING APP FROM CLIENT TO SERVER

In this module we discuss how we made the client-to-server streaming app. The UI of this app is kept minimalistic, it just opens a camera view to record footage. The recorded footage is encoded and sent to our Wowza Cloud Server. We have used an open-source live-streaming library called "libstreaming" to complete our goals. The link for the Github project for Libstreaming is included in the reference page.

**BREIF INTRODUCTION TO LIBSTREAMING LIBRARY**

It is beyond the scope of this project to actively discuss the libstreaming library. But a brief introduction is in order to facilitate latter explanations.

**libstreaming** is an API that allows developers, with only a few lines of code, to stream the camera and/or microphone of an android powered device using RTP over UDP. We use the MediaCoder API to get the stream data from the peripherals. For this action Android 4.3 or above is required. Supported encoders include H.264, H.263, AAC and AMR.

The first step to start a streaming session to some peer is called 'signaling'. During this step we contact the receiver and send a description of the incoming streams. The most important reason to use libstreaming in this project was the fact that it has readymade classes to help connect to Wowza Server.

Also libstreaming supports RTSP sessions, which is the backbone of our streaming intentions.

We open a new project in Eclipse (our choice for Android IDE). We give it a suitable name ("Live Streaming App"). Having downloaded the libstreaming library from Github and having imported the same into our workspace, we right click on the newly created project -> Properties -> Android -> Add Libraries -> libstreaming. Confirm.

Let's start with the manifest file for our application first.

Since we will be using camera for our application, we must declare that in the manifest. Also we need to secure the permission to use internet in the app, required for streaming. Write External Storage permission is required to store video information from the peripherals temporarily on the phone's memory. Permission to use RECORD\_AUDIO ensures that sound will be recorded while video capturing. Lastly the android.permission.CAMERA is required to ensure usage of camera in the app.

<uses-feature

android:name="android.hardware.camera"

android:required="true" />

<uses-feature

android:name="android.hardware.camera.autofocus"

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

<uses-permission android:name="android.permission.RECORD\_AUDIO" />

<uses-permission android:name="android.permission.CAMERA" />

Next we define the UI of the app. The layout of the app is coded in the activity\_main.xml file present in res-> layout folder. This will be a simplistic UI, with just a surface view that will host our camera.

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/surface\_layout"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:layout\_alignParentLeft="true"

android:layout\_alignParentTop="true"

android:background="@android:color/black" >

<net.majorkernelpanic.streaming.gl.SurfaceView

android:id="@+id/surface"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center" />

</FrameLayout>

Since the phone itself will always remain on the robot, we don't need a complicated interactive interface here. This code is therefore short and simple. The UI will just contain a SurfaceView on where the camera will be hosted.

Note that the surface view is not a regular android surface view from the graphics library. This surface view is taken from the libstreaming project. The id chosen for our Surface View is called "surface", which will be used in our Activity code to link to this View.

Now that we have got the basics right, we open our MainActivity.java file and complete our code.

We should declare a global String variable containing the address of our server.The server address is stored in the format:

rtsp://Wowza-server-address:PORT\_Number/WORKSPACE/STREAM\_NAME

public static final String STREAM\_URL = "rtsp://11c434.entrypoint.cloud.wowza.com:1935/app-b65e/15126c34";

NOW we shall define our onCreate() method. This method is called whenever the user opens the application from a closed state.

setContentView(R.layout.activity\_main);**//*LINKS THE VIEW OF OUR APP TO OUR activity\_main.xml FILE***

mSurfaceView = (SurfaceView) findViewById(R.id.surface); **//*LINKS TO OUR SURFACE VIEW***

mSurfaceView.getHolder().addCallback(this);

initRtspClient(); ***// Initialize RTSP client***

The initRtspClient() is where all the code for RTSP session is present. Firstly, we need to create a RtspSession variable and set all it's parameters. These parameters include the type of AudioEncoder we want to use, the quality of audio, the size of the video, the quality of the video and of course an instance to the surface view where the video will be projected. In the end, we add a callback to this RtspSession instance.

private void initRtspClient() {

mSession = SessionBuilder.getInstance() **//THIS LINE CREATES THE RTSP SESSION**

.setContext(getApplicationContext())

.setAudioEncoder(SessionBuilder.AUDIO\_AAC)

.setAudioQuality(new AudioQuality(8000, 16000)) **//*SETS AUDIO QUALITY***

.setVideoEncoder(SessionBuilder.VIDEO\_H264) **//*SETS VIDEO TYPE***

.setSurfaceView(mSurfaceView)

.setPreviewOrientation(0)

.setCallback(this).build();

In order to participate in this session, Android needs a Client.

mClient = new RtspClient();

mClient.setSession(mSession);

mClient.setCallback(this);

Now we need to extract the IP, port and path name from our URL and set Client-Server details.

String ip, port, path; **// THE FOLLOWING CODE SNIPPETS DERIVES THE IP, PORT AND PATH FROM OUR STREAM\_URL**

Pattern uri = Pattern.compile("rtsp://(.+):(\\d\*)/(.+)");

Matcher m = uri.matcher(STREAM\_URL);

m.find();

ip = m.group(1); **// IP =** 11c434.entrypoint.cloud.wowza.com

port = m.group(2); **// PORT =** 1935

path = m.group(3); **//path =** app-b65e/15126c34

**SINCE WE HAVE DISABLED OUR AUTHENTICATION FROM THE SERVER WE DO NOT NEED TO PROVIDE USERNAME OR PASSWORD. HAD WE KEPT SOURCE AUTHENTICATION ENABLED, WE WOULD HAVE TO ADD THIS LINE IN OUR CODE:**

mClient.setCredentials(WowzaParams.PUBLISHER\_USERNAME,WowzaParams.PUBLISHER\_PASSWORD);

Now we need to set the stream path and begin streaming.

mClient.setServerAddress(ip, Integer.parseInt(port));

mClient.setStreamPath("/" + path);

mSession.startPreview(); **// Starts showing user the camera footage**

mClient.startStream(); **// Starts Streaming**

We have added other background components such as checking for errors in connection, and handling the different exceptions. The full code is present in the Appendix.

This concludes our Phone-to-Server Streaming Application. Now we need to create another app that can retrieve this live stream, and also control the movement of our robot through MQTT.

Meanwhile, we have just made a virtual IP Camera here. And we did not need to spend a coin.

CHAPTER 10

ANDROID APPLICATION TO CONTROL MOVEMENT OF THE ROBOT AND VIEW LIVE SPY-CAM FOOTAGE

The next step in our project is to create an Android App that can

* Publish MQTT messages to our topic so that it controls of the motion of the robot.
* Enable user to view live footage streamed from our media server
* Create an intuitive and simple User Interface to fulfill our goals.

**The MQTT problem**

In our ESP-8266 Node MCU module, we included the Lua Code that we had uploaded to the chip. The basic algorithm of that code was:

1. Connect to WiFi
2. Connect to MQTT server
3. Subscribe to topic "test"
4. Read the latest message published in "test"
   1. If the message is "w", make the robot move forward
   2. If the message is "s", make the robot move backward
   3. If the message is "a", make the robot rotate left
   4. If the message is "d", make the robot rotate right
   5. If the message is "h", make the robot stop.
5. Stop

Thus to tackle the MQTT problem, all we need to do is:

1. Create an User Interface that displays 4 Buttons- Up, Down, Left, Right.
2. When User holds down one of the keys (say Up Key), we publish a corresponding message ("w") to the topic "test" defined in our MQTT cloud server. (Similarly, we publish "s" for Down, "a" for Left and "d" for Right).
3. When User releases the button, we publish the message "h" to our MQTT server.
4. This approach ensures the robot to move whenever the user presses a particular button, and then halt when the button is released.

**MQTT PAHO OPEN SOURCE LIBRARY**

To help us tackle the MQTT problem, we have excellent Git open source project MQTT-PAHO. For our purposes, we will need two .jar files from that extensive library, namely the **org.eclipse.paho.client.mqttv3-1.0.2.jar** and the **org.eclipse.paho.android.service-1.0.2.jar.** These jar facilitates easy and simple connection with the MQTT server and equally effective message publishing.

The link for the Github project for paho-MQTT is mentioned in the Reference page.

**THE LIVE-STREAMING PROBLEM**

There are two pre-requisites to set-up before we implement the live streaming.

1. We make sure that our stream (myStream) is active on our Wowza Cloud Server Account. A stream is activated by pressing the "Start Stream" Button present in the Overview Tab of the stream page.
2. We make sure that the camera streaming application is switched on and the streaming has started healthily from that phone to our media server.

With those out of the way, we can now lay down the requisites of this aspect of the app.

1. Set up a Flash-based video view.
2. Access the Wowza Server with the RTMP protocol. (rtmp://11c434.entrypoint.cloud.wowza.com:1935/app-b65e/15126c34)
3. Display the video in real time.

**THE VITAMIO OPEN SOURCE LIBRARY**

Since Android does not support Flash anymore, we could not use any of Android's default API to display a RTMP stream. Instead we have to settle for an alternative: the excellent Vitamio library. Vitamio is an open multimedia framework for Android and iOS, with full and real hardware accelerated decoder and renderer. With its simple, clean and powerful API, we can stream from an RTMP server. Furthermore, the video-view of Vitamio Library provides basic media-player controls like play, pause etc.

The link for the Github project for Vitamio is mentioned in the Reference page.

**MAKING THE APP - MAKING EVERYTHING COME TOGETHER**

We create a new project in Eclipse (or any Android IDE) and we include the Vitamio Library and the two MQTT .jar files (**org.eclipse.paho.client.mqttv3-1.0.2.jar** and **org.eclipse.paho.android.service-1.0.2.jar)** to make our project have access to all the classes we require from those libraries.

First, we edit the Manifest file. We declare the relevant permissions that we require in the project. The most important of these are the android.permission.ACCESS\_NETWORK\_STATE to find out if the device is connected with the internet or not, and the android.permission.INTERNET which secures the permission of using the device's internet connection.

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE" />

Next*,*we must declare the Vitamio main activity (InitActivity) in our applications tab to allow our app access to the Vitamio classes.

<application>

...

...

<activity

android:name="io.vov.vitamio.activity.InitActivity"

android:configChanges="orientation|screenSize|smallestScreenSize|keyboard|keyboardHidden|navigation"

android:launchMode="singleTop"

android:theme="@android:style/Theme.NoTitleBar"

android:windowSoftInputMode="stateAlwaysHidden" />

<activity

....

....

Since MqttService falls under a android service category, it must be specified in the manifest under the applications tab.

<service android:name="org.eclipse.paho.android.service.MqttService" >

</service>

**LAYOUT AND USER INTERFACE**

Next, we set the layout parameters and user interface of our app in the activity\_main.xml file in res->layout folder. We want a video view on top, that will display the live streaming footage from our Wowza Server. We need four buttons to simulate up, down, left and right movement and align it accordingly. To create a more dynamic look, we will use different backgrounds for each button to visually intuit the user of it's function. Figure 10.1 shows how our app will look after we are done coding it.

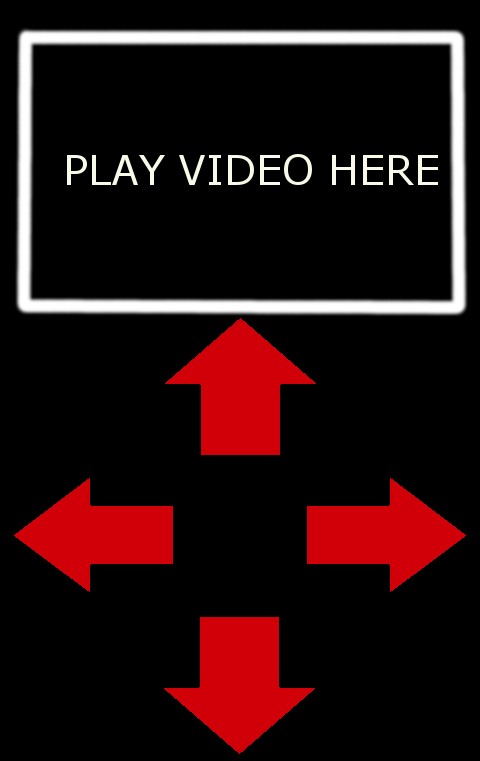


Figure 10.1

First, we create a linear layout to manage all our UI components. We add a Vitamio VideoPlayer as our first component.

<io.vov.vitamio.widget.VideoView

android:id="@+id/vitamio\_videoView"

android:layout\_width="match\_parent"

android:layout\_height="0dp"

android:layout\_weight="0.7"

/>

WE DECLARE THE FOUR BUTTONS : FORWARD, BACKWARD, LEFT TURN, RIGHT TURN. WE ALSO CHANGE THE BACKGROUND OF THESE BUTTONS WITH IMAGES.

<Button

android:id="@+id/buttonForward"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:background="@drawable/up" />

With the above code snippet, we create the forward button. It is present on the next level of our root LinearLayout.

Similarly, we add the down, left, right buttons:

<RelativeLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="horizontal"

>

<Button

android:id="@+id/buttonRight"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentRight="true"

android:layout\_alignParentTop="true"

android:background="@drawable/right"

/>

*<!..****RIGHT BUTTON CREATED****-->*

<Button

android:id="@+id/buttonLeft"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentLeft="true"

android:layout\_alignParentTop="true"

android:background="@drawable/left"

/>

*<!..****LEFT BUTTON CREATED****-->*

</RelativeLayout>

<Button

android:id="@+id/buttonBack"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:background="@drawable/down" />

*<!..****DOWN BUTTON CREATED****-->*

</LinearLayout>

</LinearLayout>

**MAIN ACTIVITY CODE**

Since we are using touch controls for Buttons, we need to implement the OnTouchListener interface in our Main Activity. We declare our MQTT server IP (along with PORT), our MQTT username, password which will all be required to connect to our MQTT cloud server and publish messages. We also need the address of our WOWZA MEDIA SERVER to stream live footage into our videoview.

public class MainActivity extends Activity implements OnTouchListener {

private String mqttServerAddress="tcp://m12.cloudmqtt.com:12879"; **//*MQTT SERVER IP***

private String mqttUserName="pphdjxet"; **//*MQTT USERNAME***

private String mqttPassword="\_ur1NqE1w5gq"; **//*MQTT PASSWORD***

private String path = "rtmp://11c434.entrypoint.cloud.wowza.com:1935/app-b65e/15126c34"; **// *MEDIASERVER***

In our onCreate() method, which is called when the app is opened, we need to link our video view to an instance, and set it's path. We also enable media controls like play, pause, stop and timeline controls by setting a media controller instance to our videoview.

setContentView(R.layout.activity\_main); **//*WE LINK THE VIEW OF OUR ACTIVITY TO THE XML FILE WE HAVE CREATED***

mVideoView = (VideoView) findViewById(R.id.vitamio\_videoView);

***// FIRST WE WORK OUT THE LIVE STREAMING PART***

mVideoView.setVideoPath(path); **// *WE HAVE SET THE VIDEO PATH TO OUR WOWZA SERVER***

mVideoView.setMediaController(new MediaController(this)); **//*ENABLES MEDIA CONTROLS***

mVideoView.requestFocus();

Now we begin to code for the MQTT part. First, we need to generate a client ID string that will be required during the MQTT session transaction. We assign our clientID and MQTT Server Address to the MQTT Client object.

String clientId = MqttClient.generateClientId();

client =new MqttAndroidClient(this.getApplicationContext(), mqttServerAddress, clientId);

We need an MQTTConnectOptions instance to specify the nature of our connection. In this instance, we store metadata such as the MQTT Version we want to employ(MqttV3), our username and our password.

options = new MqttConnectOptions();

options.setMqttVersion(MqttConnectOptions.MQTT\_VERSION\_3\_1);

options.setUserName(mqttUserName)

options.setPassword(.toCharArray());

We should now link our buttons with their respective UI components. Following that we should set a TouchListener to them so that we can handle button inputs like pressing and releasing.

fwd=(Button)findViewById(R.id.buttonForward);

fwd.setOnTouchListener(this);

We repeat the process for the bck,ryt and lft button instances. Next, we override the onTouch callback function. This function is called whenever the user presses or releases a button. The pressed object contains the button object that was touched, and the event object contains the nature of interaction the user had with the button. For pressing down the value of the action of the event is MotionEvent.ACTION\_DOWN, and for releasing it is MotionEvent.ACTION\_UP. As planned, we publish "w","s","a","d" when user presses and holds the respective button (up, down, left, right respectively) and when user releases said button, we publish "h" which results in halting of the robot in the back-end.

@Override

public boolean onTouch(View pressed, MotionEvent event) { **// CALLED WHEN A BUTTON IS TOUCHED**

if (event.getAction() == MotionEvent.ACTION\_DOWN) {

if (pressed==fwd){

sendMessage("w"); **// MQTT COMMAND TO GO FORWARD**

}

else if(pressed==lft){

sendMessage("a"); **// MQTT COMMAND TO GO LEFT**

}

else if(pressed==bck){

sendMessage("s"); **// MQTT COMMAND TO GO BACK**

}

else if(pressed==ryt){

sendMessage("d"); **// MQTT COMMAND TO GO RIGHT**

}

} else if (event.getAction() == MotionEvent.ACTION\_UP) {

**// ACTION\_UP occurs when user releases a button**

sendMessage("h"); **// MQTT COMMAND TO STOP**

}

return false;

}

In this function, we simply pick up the user's input and channel the appropriate response to the sendMessage() function.

The sendMessage() accepts a string input as it's parameter, connects to the MQTT Server with the MqttClient Object we had created, and publishes the recieved message to the "test" topic we defined in our MQTT server.

public void sendMessage(final String message){ **//This function will publish the String message to the topic "test"**

IMqttToken token = null;

try {

token = client.connect(options); **// First, we connect to the MQTT Server**

} catch (MqttException e1) {

e1.printStackTrace();

}

token.setActionCallback(new IMqttActionListener() {

@Override

public void onSuccess(IMqttToken asyncActionToken) {

String topic = "test"; **//TOPIC NAME**

**// TO PUBLISH THE PAYLOAD OR MESSAGE, WE NEED TO CONVERT IT INTO A BYTE ARRAY FIRST**

byte[] encodedPayload = new byte[0];

try {

encodedPayload = message.getBytes("UTF-8");

MqttMessage message = new MqttMessage(encodedPayload); **// CONVERTING TO MQTT Message Instance**

client.publish(topic, message); **// PUBLISHING MESSAGE TO TOPIC "TEST"**

} catch (UnsupportedEncodingException | MqttException e) {

e.printStackTrace();

}

}

@Override

public void onFailure(IMqttToken asyncActionToken, Throwable exception) {

**// Something went wrong e.g. connection timeout or firewall problems**

Toast.makeText(getApplicationContext(), "A Problem Occured", Toast.LENGTH\_SHORT).show();

}

});

}

The Android phone application for controlling the robot and viewing the live stream. In the next module we discuss setting up the chassis of the robot, attaching the wheels, and installing the motor driver.

**RESULTS**

**AND**

**DISCUSSIONS**

CHAPTER 11

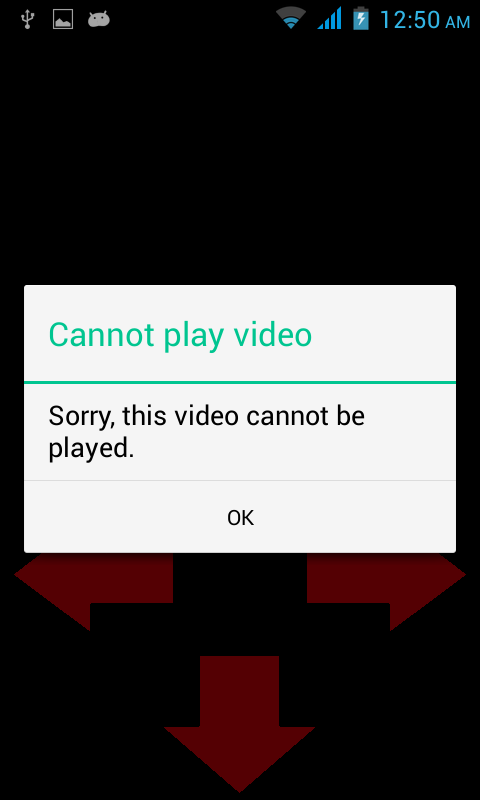
CONNECTING EVERYTHING AND MAKING IT WORK

Finally, when all the connections stated in chapter 8 are made (the NodeMCU to the MotorDriver and the power supply), it is checked if our Node MCU is connected properly. The blue LED glows when the Node MCU is ON. We can now use our MQTT project app to control the movement of our robot. As stated before, pressing the respective arrow keys will publish the appropriate message to the MQTT server. The Node MCU will subscribe to this topic and process the message by changing the state of it's pins. This will in turn change the voltage difference across the motor driver inputs for both wheels. And the chassis will move in the specified direction.

The process was explained at greater detail in chapters 7 and 8.

Although we have tackled the robot's movement and MQTT parts, the live streaming will not occur just yet. In fact if we open the app without starting our media server, we cannot see any video in our video view.

This state is shown in the figure.

  
 Figure 11.1

However, we can still publish MQTT signals by pressing the relevant arrow keys.

To watch the stream, we need to go to our Wowza Cloud Server, (as we had previously set up), and go to our workspace. We need to go to our streaming homepage and click on the "Start Stream" button to open our stream (myStream). Now, our Cloud Media Server is ready to detect RTSP video source.

After starting our Media Server, we need to attach the camera streaming android phone to our chassis. After attaching the phone, we should open our LiveStreaming Application (the one we had created before). Our live streaming is now complete. We may check our server to confirm if our server has detected our RTSP source.

Depending on the server traffic, there will be varying degrees of lag in streaming.

If we restart the Project App now, we will see the phone pick up our server stream properly, as shown in Figure 11.3.

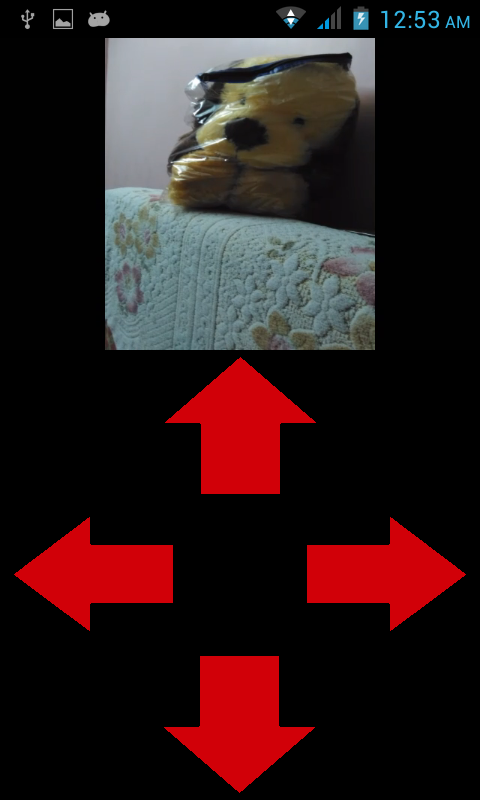


Figure 11.2

We can control media playback by simply touching the bottom of the screen, and the playback options will show up. We can control the movement of the robot by pressing the arrow keys, as before.

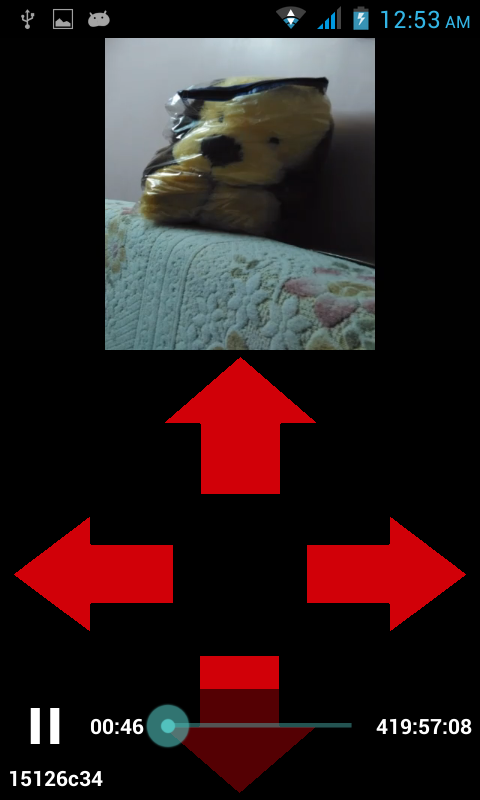


Figure 11.3

All the individual parts of our system are now up and running. In the next page, we have given a detailed flowchart of how every cog in the wheel works, and how they are interlinked.

BLOCK DIAGRAM

C:\Users\Avi\Downloads\Flowchar1 (1).png

Figure 11.4

CHAPTER 12

DISCUSSION AND FUTURE WORK

Surveillance bots have a huge market in modern technology. The concept of keeping an eye on something when you are not physically present in the scene is empowering and without a doubt a significant part of the future. The streaming delay due to client-server-client traffic is at an average about 10 seconds for normal internet speeds. This is a major area of improvement. With our Wowza Media Cloud Server is free, it is also very slow and invites a lot of traffic. A more expensive server plan will allow for not only better streaming speeds but also better streaming quality. The better the server and internet connection is, better is our surveillance functionality.

Our robot requires WIFI access of the location it is in to work properly. This means that while one cannot really use this project for military purposes, one can use it for basic home surveillance. The WIFI module Node MCU has opened many a door in automation and control circuits. Imagine the scenario of controlling an entire room with your android app over the internet. The so-called "smart-home" theory at a smaller scale is very much possible with Node MCU.

Our project is a small stepping off point more than a definitive end to the world's journey towards automation. The fact that one can control electronic circuits from a continent away opens up infinite doors of creativity and imagination. There are infinite ways to improve this project. One can add limbs to the robot to make it used for cleaning and fetching purposes. One can make a robotic battle multiplayer games or racing games while controlling the cars from a different part of the world. One can install a speaker to the system and transmit voice. There is a plethora of options from this base project.

IN CONCLUSION

This brings our project to it’s conclusion. We have made an “Internet-Controlled Range-Less Real Time Surveillance Robot with Android Application”. Our robot is controlled by the android application that we had developed via internet. We used the MQTT server for communicating across the blocks of our project. The NodeMCU chip forms the main brain of our project. It is a WIFI module that connects with remote WIFI signal and processes MQTT messages from server. This message is processed such that the robot moves in the direction that the user had previously asked for. The surveillance part of our project definition is covered by another Android application that we developed. This application is installed on the phone fixed at the chassis of the robot body. It streams real time footage to our Media server. The streamed footage is then retrieved by our client app and one can see the footage.

REFERENCE

www.wikipedia.org

esp8266.ru/esplorer/

github.com/4refr0nt/ESPlorer

github.com/yixia/VitamioBundle

github.com/fyhertz/libstreaming

www.14core.com

www.cloudmqtt.com/

www.eclipse.org/paho/

allaboutcircuits

letsmakerobots

hivemq.com

www.instructables.com

www.chrislarsen.me

www.foobarflies.io

www.adafruit.com

www.seeedstudio.com

www.espressif.com

doch.oasis-open.org

www.etsi.com

home-assistant.io

www.wowza.com

www.stackoverflow.com

www.csee.umbc.edu/

www.androidhive.info/

android-libs.com/lib/libstreaming

www.truiton.com/2015/03/stream-rtmp-live-android/

www.vitamio.org/en/

www.infoq.com/articles/practical-mqtt-with-paho

mqtt.org/tag/paho

ebay.in

www.robomart.com/robot-chassis

APPENDIX

THE CODES FOR ALL THE APPLICATIONS USED IN THIS PROJECT ARE GIVEN BELOW

# ANDROID LIVE STREAMING APP CODE

Manifest File

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.project.livestreamapp"

android:versionCode="1"

android:versionName="1.0" >

<uses-sdk

android:minSdkVersion="8"

android:targetSdkVersion="19" />

<uses-feature

android:name="android.hardware.camera"

android:required="true" />

*<!..*

*This is done to alert the user that the app will use the camera.*

*-->*

<uses-feature

android:name="android.hardware.camera.autofocus"

android:required="false" />

<supports-screens

android:largeScreens="true"

android:normalScreens="true"

android:smallScreens="true"

android:xlargeScreens="true" />

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

<uses-permission android:name="android.permission.RECORD\_AUDIO" />

<uses-permission android:name="android.permission.CAMERA" />

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name=".MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

</application>

</manifest>

**LAYOUT: activity\_main.xml**

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/surface\_layout"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:layout\_alignParentLeft="true"

android:layout\_alignParentTop="true"

android:background="@android:color/black" >

<net.majorkernelpanic.streaming.gl.SurfaceView

android:id="@+id/surface"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center" />

</FrameLayout>

**MAIN ACTIVITY CODE**

package com.project.livestreamapp;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import android.app.Activity;

import android.app.AlertDialog;

import android.content.DialogInterface;

import android.os.Bundle;

import android.view.Menu;

import android.view.MenuItem;

import android.view.SurfaceHolder;

import android.view.Window;

import android.view.WindowManager;

import net.majorkernelpanic.streaming.Session;

import net.majorkernelpanic.streaming.SessionBuilder;

import net.majorkernelpanic.streaming.audio.AudioQuality;

import net.majorkernelpanic.streaming.gl.SurfaceView;

import net.majorkernelpanic.streaming.rtsp.RtspClient;

public class MainActivity extends Activity implements RtspClient.Callback, Session.Callback, SurfaceHolder.Callback {

public final static String TAG = MainActivity.class.getSimpleName();

public static final String STREAM\_URL = "rtsp://11c434.entrypoint.cloud.wowza.com:1935/app-b65e/15126c34"; //***THIS IS THE SERVER //ADDRESS FOR OUR STREAM. THE FORMAT IS*** *rtsp://WOWZASERVERADDRESS:PORT/WORKSPACE/STREAM\_NAME*

private static SurfaceView mSurfaceView;

private Session mSession;

private static RtspClient mClient;

@Override

protected void onCreate(Bundle savedInstanceState) { **// *THIS IS THE FIRST CALLBACK THAT OCCURS WHEN USER OPENS OUR APP***

super.onCreate(savedInstanceState);

getWindow().addFlags(WindowManager.LayoutParams.FLAG\_KEEP\_SCREEN\_ON); **//*TO ENSURE THAT SCREEN REMAINS ON***

getWindow().addFlags(WindowManager.LayoutParams.FLAG\_FULLSCREEN);

requestWindowFeature(Window.FEATURE\_NO\_TITLE);

setContentView(R.layout.activity\_main);**//*LINKS THE VIEW OF OUR APP TO OUR activity\_main.xml FILE WE HAVE ALREADY CODED***

mSurfaceView = (SurfaceView) findViewById(R.id.surface); **//*LINKS TO OUR SURFACE VIEW***

mSurfaceView.getHolder().addCallback(this);

initRtspClient(); ***// Initialize RTSP client***

}

@Override

protected void onResume() {

super.onResume();

}

@Override

public void onDestroy(){ **//*CALLED WHEN USER QUITS APPLICATION***

super.onDestroy();

mClient.release();

mSession.release();

mSurfaceView.getHolder().removeCallback(this);

}

@Override

public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3) {

}

@Override

public void surfaceCreated(SurfaceHolder arg0) {

}

@Override

public void surfaceDestroyed(SurfaceHolder arg0) {

}

public void onBitrareUpdate(long bitrate) {

}

@Override

public void onSessionError(int reason, int streamType, Exception e) { **// *CALLED WHEN THERE IS AN ERROR IN THE APPLICATION***

switch (reason) {

case Session.ERROR\_CAMERA\_ALREADY\_IN\_USE:

break;

case Session.ERROR\_CAMERA\_HAS\_NO\_FLASH:

break;

case Session.ERROR\_INVALID\_SURFACE:

break;

case Session.ERROR\_STORAGE\_NOT\_READY:

break;

case Session.ERROR\_CONFIGURATION\_NOT\_SUPPORTED:

break;

case Session.ERROR\_OTHER:

break;

}

if (e != null) {

alertError(e.getMessage());

e.printStackTrace(); **//*THIS WAS USED FOR DEBUGGING THE APPLICATION. THIS IS OPTIONAL.***

}

}

@Override

public void onPreviewStarted() { }

@Override

public void onSessionConfigured() { }

@Override

public void onSessionStarted() { }

@Override

public void onSessionStopped() { }

@Override

public void onRtspUpdate(int message, Exception exception) {

switch (message) {

case RtspClient.ERROR\_CONNECTION\_FAILED:

case RtspClient.ERROR\_WRONG\_CREDENTIALS:

alertError(exception.getMessage()); ***//TO PROVIDE NECESSARY ERROR MESSAGES TO USER***

exception.printStackTrace();

break;

}

}

private void alertError(final String msg) { ***//THIS FUNCTION DISPLAYS ERROR MESSAGE msg TO USER***

final String error = (msg == null) ? "Unknown error: " : msg;

AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);

builder.setMessage(error).setPositiveButton("Ok",

new DialogInterface.OnClickListener() {

public void onClick(DialogInterface dialog, int id) {

}

});

AlertDialog dialog = builder.create();

dialog.show();

}

private void initRtspClient() {

mSession = SessionBuilder.getInstance() **//THIS LINE CREATES THE RTSP SESSION**

.setContext(getApplicationContext())

.setAudioEncoder(SessionBuilder.AUDIO\_AAC)

.setAudioQuality(new AudioQuality(8000, 16000)) **//*SETS AUDIO TYPE AND QUALITY***

.setVideoEncoder(SessionBuilder.VIDEO\_H264) **//*SETS VIDEO TYPE AND QUALITY***

.setSurfaceView(mSurfaceView)

.setPreviewOrientation(0)

.setCallback(this).build();

mClient = new RtspClient();

mClient.setSession(mSession);

mClient.setCallback(this);

mSurfaceView.setAspectRatioMode(SurfaceView.ASPECT\_RATIO\_PREVIEW);

String ip, port, path; **// THE FOLLOWING CODE SNIPPETS DERIVES THE IP, PORT AND PATH FROM OUR STREAM\_URL**

Pattern uri = Pattern.compile("rtsp://(.+):(\\d\*)/(.+)");

Matcher m = uri.matcher(STREAM\_URL);

m.find();

ip = m.group(1); **// IP =** 11c434.entrypoint.cloud.wowza.com

port = m.group(2); **// PORT =** 1935

path = m.group(3); **//path =** app-b65e/15126c34

// **SINCE WE HAVE DISABLED OUR AUTHENTICATION FROM THE SERVER WE DO NOT NEED TO PROVIDE USERNAME OR PASSWORD. HAD //WE NOT DONE THAT, WE WOULD HAVE TO ADD THIS LINE IN OUR CODE:**

**//**mClient.setCredentials(WowzaParams.PUBLISHER\_USERNAME,WowzaParams.PUBLISHER\_PASSWORD);

mClient.setServerAddress(ip, Integer.parseInt(port));

mClient.setStreamPath("/" + path);

mSession.startPreview(); **// Starts showing user the camera footage**

mClient.startStream(); **// Starts Streaming**

}

}

## PROJECT APP FOR CONTROLLING ROBOT THROUGH MQTT PUBLISHING AND VIEWING LIVE STREAM FROM SERVER

**MANIFEST FILE**

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.project.mqttexamples"

android:versionCode="1"

android:versionName="1.0" >

<uses-sdk

android:minSdkVersion="8"

android:targetSdkVersion="21" />

<uses-permission android:name="android.permission.WAKE\_LOCK" />

<uses-permission android:name="android.permission.INTERNET" />

<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE" />

<uses-permission android:name="android.permission.READ\_PHONE\_STATE" />

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

*<!--*

*We must declare the Vitamio main activity (InitActivity) in our applications tab to allow our app access to the Vitamio classes*

*-->*

<activity

android:name="io.vov.vitamio.activity.InitActivity"

android:configChanges="orientation|screenSize|smallestScreenSize|keyboard|keyboardHidden|navigation"

android:launchMode="singleTop"

android:theme="@android:style/Theme.NoTitleBar"

android:windowSoftInputMode="stateAlwaysHidden" />

<activity

android:name=".MainActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

*<!..*

*Since MqttService falls under a android service category, it must be specified in the manifest*

*-->*

<service android:name="org.eclipse.paho.android.service.MqttService" >

</service>

</application>

</manifest>

**LAYOUT FILE: activity\_main.xml**

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

android:background="#000"

>

<io.vov.vitamio.widget.VideoView

android:id="@+id/vitamio\_videoView"

android:layout\_width="match\_parent"

android:layout\_height="0dp"

android:layout\_weight="0.7"

/>

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="vertical"

android:gravity="center"

>"

<Button

android:id="@+id/buttonForward"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:background="@drawable/up" />

<RelativeLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="horizontal"

>

<Button

android:id="@+id/buttonRight"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentRight="true"

android:layout\_alignParentTop="true"

android:background="@drawable/right"

/>

<Button

android:id="@+id/buttonLeft"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentLeft="true"

android:layout\_alignParentTop="true"

android:background="@drawable/left"

/>

</RelativeLayout>

<Button

android:id="@+id/buttonBack"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:background="@drawable/down" />

</LinearLayout>

</LinearLayout>

**MAIN ACTIVITY CODE**

package com.project.mqttexamples;

import io.vov.vitamio.LibsChecker; import io.vov.vitamio.MediaPlayer; import io.vov.vitamio.widget.MediaController;

import io.vov.vitamio.widget.VideoView; import java.io.UnsupportedEncodingException;

import org.eclipse.paho.android.service.MqttAndroidClient; import org.eclipse.paho.client.mqttv3.IMqttActionListener;

import org.eclipse.paho.client.mqttv3.IMqttToken; import org.eclipse.paho.client.mqttv3.MqttClient;

import org.eclipse.paho.client.mqttv3.MqttConnectOptions; import org.eclipse.paho.client.mqttv3.MqttException;

import org.eclipse.paho.client.mqttv3.MqttMessage;

import android.app.Activity; import android.os.Bundle; import android.util.Log;

import android.view.MotionEvent;import android.view.View;

import android.view.View.OnClickListener; import android.view.View.OnTouchListener;

import android.widget.Button; import android.widget.Toast;

public class MainActivity extends Activity implements OnTouchListener {

private String mqttServerAddress="tcp://m12.cloudmqtt.com:12879"; **//*MQTT SERVER IP***

private String mqttUserName="pphdjxet"; **//*MQTT USERNAME***

private String mqttPassword="\_ur1NqE1w5gq"; **//*MQTT PASSWORD***

private String path = "rtmp://11c434.entrypoint.cloud.wowza.com:1935/app-b65e/15126c34"; **// *ADDRESS OF OUR SERVER***

VideoView mVideoView; **//*THIS WILL CONTAIN THE INSTANCE OF OUR VIDEO PLAYER***

Button fwd,lft,ryt,bck; **//*THESE WILL KEEP THE INSTANCES OF THE FOUR BUTTONS***

MqttAndroidClient client;

MqttConnectOptions options;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

if (!LibsChecker.checkVitamioLibs(this))

return;

setContentView(R.layout.activity\_main); **// *LINK THE VIEW OF THIS ACTIVITY TO THE XML FILE WE HAVE CREATED***

mVideoView = (VideoView) findViewById(R.id.vitamio\_videoView);

***// FIRST WE WORK OUT THE LIVE STREAMING PART***

mVideoView.setVideoPath(path); **// *WE HAVE SET THE VIDEO PATH TO OUR WOWZA SERVER***

mVideoView.setMediaController(new MediaController(this)); **//*ENABLES MEDIA CONTROLS LIKE PLAY PAUSE AND TIMELINE***

mVideoView.requestFocus();

mVideoView.setOnPreparedListener(new MediaPlayer.OnPreparedListener() {

@Override

public void onPrepared(MediaPlayer mediaPlayer) {

mediaPlayer.setPlaybackSpeed(1.0f);

}

});

**// *THE MQTT PART***

String clientId = MqttClient.generateClientId();

client =new MqttAndroidClient(this.getApplicationContext(), mqttServerAddress, clientId);

**//*Here we create a clientID and initialize it with the parameters of our server address***

options = new MqttConnectOptions();

options.setMqttVersion(MqttConnectOptions.MQTT\_VERSION\_3\_1); **//Since we are using mqttv3**

options.setUserName(mqttUserName); **//Specifying username and password**

options.setPassword(.toCharArray());

**//Linking the Button objects to the respective Buttons and make them responsive to touch**

fwd=(Button)findViewById(R.id.buttonForward);

bck=(Button)findViewById(R.id.buttonBack);

lft=(Button)findViewById(R.id.buttonLeft);

ryt=(Button)findViewById(R.id.buttonRight);

fwd.setOnTouchListener(this);

bck.setOnTouchListener(this);

lft.setOnTouchListener(this);

ryt.setOnTouchListener(this);

}

public void sendMessage(final String message){ **//This function will publish the String message to the topic "test"**

IMqttToken token = null;

try {

token = client.connect(options); **// First, we connect to the MQTT Server**

} catch (MqttException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

token.setActionCallback(new IMqttActionListener() {

@Override

public void onSuccess(IMqttToken asyncActionToken) {

String topic = "test"; **//TOPIC NAME**

**// TO PUBLISH THE PAYLOAD OR MESSAGE, WE NEED TO CONVERT IT INTO A BYTE ARRAY FIRST**

byte[] encodedPayload = new byte[0];

try {

encodedPayload = message.getBytes("UTF-8");

MqttMessage message = new MqttMessage(encodedPayload); **// CONVERTING TO MQTT Message Instance**

client.publish(topic, message); **// PUBLISHING MESSAGE TO TOPIC "TEST"**

} catch (UnsupportedEncodingException | MqttException e) {

e.printStackTrace();

}

}

@Override

public void onFailure(IMqttToken asyncActionToken, Throwable exception) {

**// Something went wrong e.g. connection timeout or firewall problems**

Log.d(TAG, "onFailure");

Toast.makeText(getApplicationContext(), "A Problem Occured", Toast.LENGTH\_SHORT).show();

}

});

}

@Override

public boolean onTouch(View pressed, MotionEvent event) { **// CALLED WHEN A BUTTON IS TOUCHED**

if (event.getAction() == MotionEvent.ACTION\_DOWN) {

if (pressed==fwd){

sendMessage("w"); **// MQTT COMMAND TO GO FORWARD**

}

else if(pressed==lft){

sendMessage("a"); **// MQTT COMMAND TO GO LEFT**

}

else if(pressed==bck){

sendMessage("s"); **// MQTT COMMAND TO GO BACK**

}

else if(pressed==ryt){

sendMessage("d"); **// MQTT COMMAND TO GO RIGHT**

}

} else if (event.getAction() == MotionEvent.ACTION\_UP) {

**// ACTION\_UP occurs when user releases a button**

sendMessage("h"); **// MQTT COMMAND TO STOP**

}

return false;

}

}

**NODE MCU SCRIPT.LUA**

clien1="clitttt"

wifi.setmode(wifi.STATION)

wifi.sta.config("WIFI NAME","WIFI PASSWORD")

gpio.mode (2, gpio.OUTPUT) -- ia1 - GPIO4

gpio.mode (3, gpio.OUTPUT) -- ia2 - GPIO0

gpio.mode (4, gpio.OUTPUT) -- ib2 - GPIO2

gpio.mode (5, gpio.OUTPUT) -- ib1 - GPIO5

gpio.write(2, gpio.LOW)

gpio.write(3, gpio.LOW)

gpio.write(4, gpio.LOW)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

m = mqtt.Client(clien1, 120, "pphdjxet","\_ur1NqE1w5gq")

m:on("connect", function(con) print ("connected") end)

m:on("offline", function(con) print ("offline") end)

m:connect("m12.cloudmqtt.com",12879, 0, function(conn)

print("connected")

m:subscribe("test", 0, function(conn)

print("subscribed")

end)

end)

m:on("message", function(conn, topic, data)

print(topic .. ":" )

if data == "w" then

gpio.write(2, gpio.HIGH)

gpio.write(3, gpio.LOW)

gpio.write(4, gpio.HIGH)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

end)

end

if data == "h" then

gpio.write(2, gpio.LOW)

gpio.write(3, gpio.LOW)

gpio.write(4, gpio.LOW)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

m:publish("topic","h",0,0,function(conn)

print("sent")

end)

end

if data == "s" then

gpio.write(2, gpio.LOW)

gpio.write(3, gpio.HIGH)

gpio.write(4, gpio.LOW)

gpio.write(5, gpio.HIGH)

tmr.delay(1000)

m:publish("topic","s",0,0,function(conn)

print("sent")

end)

end

if data == "a" then

gpio.write(2, gpio.LOW)

gpio.write(3, gpio.HIGH)

gpio.write(4, gpio.HIGH)

gpio.write(5, gpio.LOW)

tmr.delay(1000)

end

if data == "d" then

gpio.write(2, gpio.HIGH)

gpio.write(3, gpio.LOW)

gpio.write(4, gpio.LOW)

gpio.write(5, gpio.HIGH)

tmr.delay(1000)

end)

end

end)