**Component required:**

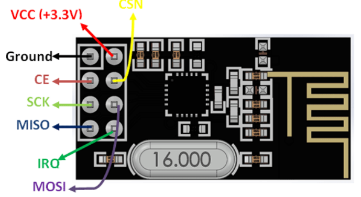
**Component cost**

Arduino uno 300 to 400

NRF+pa+lna 200 to 300

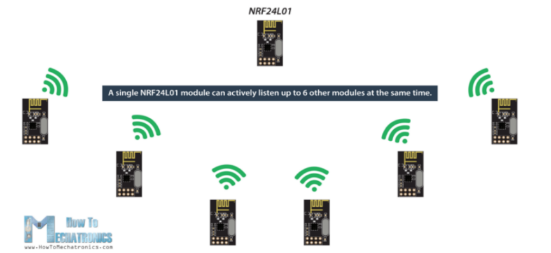
Node mcu 200 to 300

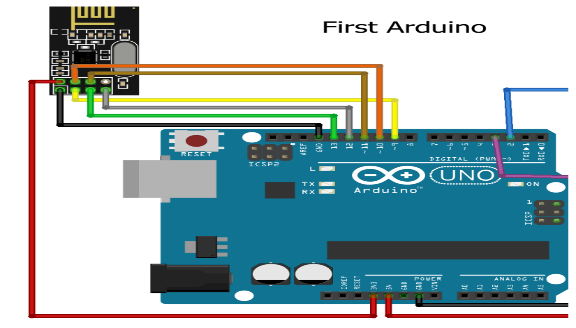
**PIN DIAGRAM OF NRF24L01**



* A single NRF24L01 module can actively listen up to 6 other modules at the same time. By using this feature of its it can connect it with total of 3125 nodes using tree topology

**Network design using nrf module:**





**Pin connection:**

nRF24l01 Arduino uno

VCC 3.3v

GND GND

MOSI 11

MISO 12

SCK 13

CE 9

CSN 10

**PIN DESCRIPTION OF NRF:**

* GND is the Ground Pin. It is usually marked by encasing the pin in a square so it can be used as a reference for identifying the other pins.
* VCC supplies power for the module. This can be anywhere from 1.9 to 3.9 volts. You can connect it to 3.3V output from your Arduino. Remember connecting it to 5V pin will likely destroy your nRF24L01+ module!
* CE (Chip Enable) is an active-HIGH pin. When selected the nRF24L01 will either transmit or receive, depending upon which mode it is currently in.
* CSN (Chip Select Not) is an active-LOW pin and is normally kept HIGH. When this pin goes low, the nRF24L01 begins listening on its SPI port for data and processes it accordingly.
* SCK (Serial Clock) accepts clock pulses provided by the SPI bus Master.
* MOSI (Master Out Slave In) is SPI input to the nRF24L01.
* MISO (Master In Slave Out) is SPI output from the nRF24L01.
* IRQ is an interrupt pin that can alert the master when new data is available to process.

**Aurdino nrf communication:**

**Transmitter Code**

1. /\*
2. \* Arduino Wireless Communication Tutorial
3. \* Example 2 - Transmitter Code
4. \*
5. \* by Dejan Nedelkovski, www.HowToMechatronics.com
6. \*
7. \* Library: TMRh20/RF24, https://github.com/tmrh20/RF24/
8. \*/
9. #include <SPI.h>
10. #include <nRF24L01.h>
11. #include <RF24.h>
12. #define led 12
13. RF24 radio(7, 8); // CE, CSN
14. **const** byte addresses[][6] = {"00001", "00002"};
15. boolean buttonState = 0;
16. **void** setup() {
17. pinMode(12, OUTPUT);
18. radio.begin();
19. radio.openWritingPipe(addresses[1]); // 00002
20. radio.openReadingPipe(1, addresses[0]); // 00001
21. radio.setPALevel(RF24\_PA\_MIN);
22. }
23. **void** loop() {
24. delay(5);
25. radio.stopListening();
26. **int** potValue = analogRead(A0);
27. **int** angleValue = map(potValue, 0, 1023, 0, 180);
28. radio.write(&angleValue, **sizeof**(angleValue));
29. delay(5);
30. radio.startListening();
31. **while** (!radio.available());
32. radio.read(&buttonState, **sizeof**(buttonState));
33. **if** (buttonState == HIGH) {
34. digitalWrite(led, HIGH);
35. }
36. **else** {
37. digitalWrite(led, LOW);
38. }
39. }

**Receiver Code**

1. /\*
2. \* Arduino Wireless Communication Tutorial
3. \* Example 2 - Receiver Code
4. \*
5. \* by Dejan Nedelkovski, www.HowToMechatronics.com
6. \*
7. \* Library: TMRh20/RF24, https://github.com/tmrh20/RF24/
8. \*/
9. #include <SPI.h>
10. #include <nRF24L01.h>
11. #include <RF24.h>
12. #include <Servo.h>
13. #define button 4
14. RF24 radio(7, 8); // CE, CSN
15. **const** byte addresses[][6] = {"00001", "00002"};
16. Servo myServo;
17. boolean buttonState = 0;
18. **void** setup() {
19. pinMode(button, INPUT);
20. myServo.attach(5);
21. radio.begin();
22. radio.openWritingPipe(addresses[0]); // 00001
23. radio.openReadingPipe(1, addresses[1]); // 00002
24. radio.setPALevel(RF24\_PA\_MIN);
25. }
26. **void** loop() {
27. delay(5);
28. radio.startListening();
29. **if** ( radio.available()) {
30. **while** (radio.available()) {
31. **int** angleV = 0;
32. radio.read(&angleV, **sizeof**(angleV));
33. myServo.write(angleV);
34. }
35. delay(5);
36. radio.stopListening();
37. buttonState = digitalRead(button);
38. radio.write(&buttonState, **sizeof**(buttonState));
39. }
40. }

**working of dynamic source routing**:

1. **Route Discovery:**

Let a source S wants to send data to Destination D. Source will broadcast a RREQ packet which will contain a unique RREQ packet id, destination address, and its own address. When a node receives packet it will first compare the packet destination address to its own address and

1. If node is not destination node

• It will check whether it has seen the packet already if it has it will discard that packet. This will be done by comparing the incoming packets unique RREQ packet id. Otherwise

• It will append its address in the packet and broadcast it again.

2. If node is destination node

• Node will make a new RREP packet

• It will copy the route record and RREQ ID from the RREQ into RREP packet and also it will append its address in the path field and then node will unicast the packet along the path found.

• Upon receiving the RREP packet the intermediate node in the path will check the RREP source route and will unicast the packet to the next node in the source route. Also the node will append its own address in the RREP path field.

• When the source receives the RREP packet it will copy the addresses in the path field to its address cache and will send the data in the path found.

1. **Route Caching**

In route discovery when source receives the RREP from the destination it saves the path from the destination to itself in its route cache which it can use immediately and also in future to send data to node which is the path between source and the destination. Also when each node sends the RREQ packet it also finds the path from the source to itself and it stores the path in its cache which it can use when the node needs the same path to send data in future. The same thing can also be done in RREP packet sending. In case when a node receives the multiple packet for same destination from two different source it can save both of them in its cache. This information about route can be used when one of the path gets broken or damaged. In addition to this route caching can also optimize the route discovery process consider for a scenario where a RREQ request for some destination is received by some intermediate node and that node has the route from itself to the destination instead of sending the RREQ packet by broadcasting the node can append the address from its cache to the RREQ packet and send the RREP packet to the destination.

1. **Route Maintenance**

After finding the route source S will send the data to D along the path it has known by route discovery i.e. Source Route. Let Source S have to send the data for Destination D through A, B, C intermediate nodes i.e. source route is S-A-B-C-D. S will send the data to A and will wait for the acknowledgement from A for the successful delivery of the data. similarly A will send data to B and will receive acknowledgement. Now suppose if B send data to C and no acknowledgement is received from C, B will wait for some predefined amount of time for the acknowledgement to come and if B does not receive the acknowledgement it will send the RERR packet to all the nodes in the path from which it received packet. All the nodes which will receive the packet will update their corresponding route cache for that path and remove the path C-D. If any of the node receiving RERR has an alternate path in its route cache it will try to send the data again in that path if it succeeds the source will update the new path in its cache. Otherwise Source S has to again initiate route discovery.

The route discovery and route maintenance steps involve three types of messages

* Route Request (RREQ) Whenever a source node wants to discover route to a destination, it will broadcast RREQ message. This message is then broadcasted by the subsequent nodes until the destination receives this RREQ packet.
* Route Reply (RREP) As soon as the destination receives a route request to itself, it originates a route reply (RREP) message and forwards it to the source through the path found in the RREQ packet.
* Route Error (RERR) During the packet delivery, if the original path has changed, then the node, that is not able to send the packet, will send a Route Error (RERR) packet to the source (origin) of the packet.

**Code of dynamic source routing:**

void forwardRouteReply(string source\_ip, string destination\_ip,

unsigned long long int packet\_id, string route\_list)

{

vector<string> ip\_list = StringOperations::split(route\_list, ","

);

int self\_pos = -1;

int count = ip\_list.size();

int i;

for(i = 0; i < ip\_list.size() - 1; i++)

{

if(ip\_list[i] == ip\_list[i + 1])

{

ip\_list.erase(ip\_list.begin() + i + 1);

}

}

for(i = 0; i < count; i++)

{

if(ip\_list[i] == ip)

{

self\_pos = i;

break;

}

}

for(i = 0; i < self\_pos; i++)

{

if(ip\_list[i] != ip && ip\_list[self\_pos - 1] != ip)

{

cache.updateRoute(ip\_list[i], ip\_list[self\_pos - 1]);

}

}

for(i = self\_pos + 1; i < count; i++)

{

if(ip\_list[i] != ip && ip\_list[self\_pos + 1] != ip)

{

cache.updateRoute(ip\_list[i], ip\_list[self\_pos + 1]);

}

}

if(destination\_ip != ip) // Or self\_pos != 0

{

string arr[] =

{

string(ROUTE\_REPLY), source\_ip, destination\_ip,

StringOperations::to\_string(packet\_id), route\_list

};

string message = StringOperations::join(vector<string>(arr, arr

+5), DELIMITER);

sendDataImmediate(ip\_list[self\_pos - 1], message);

return;

}

else

{

if(packet\_queues.find(source\_ip) != packet\_queues.end())

{

for(int i = 0; i < packet\_queues[source\_ip].size(); i++)

{

pthread\_t th;

pthread\_create(&th, NULL, sendDataPacket, packet\_queues[

source\_ip][i]);

}

}

}

}

void forwardRouteRequest(string source\_ip, string destination\_ip,

unsigned long long int packet\_id, string route\_list)

{

pair<string, unsigned long long int> pkt\_info = make\_pair(

source\_ip, packet\_id);

if(processed\_packets.find(pkt\_info) != processed\_packets.end())

// Already processed

{

return;

}

processed\_packets.insert(pkt\_info);

if(ip == destination\_ip)

{

sendRouteReply((route\_list == "")? ip: ((route\_list.find(ip) ==

string::npos)? route\_list + "," + ip: route\_list));

return;

}

if(cache.isRouteCached(destination\_ip))

{

if(route\_list == "")

{

route\_list = ip;

}

else if(route\_list.find(ip) == string::npos)

{

route\_list += "," + ip;

}

if(route\_list.find(cache.fetchRoute(destination\_ip)) == string

::npos)

{

route\_list += "," + cache.fetchRoute(destination\_ip);

}

#ifdef DEBUG

cerr << "Path is taken from cache: " << route\_list << endl;

#endif

sendRouteReply(route\_list);

return;

}

if(route\_list == "")

{

route\_list = ip;

}

else if(route\_list.find(ip) == string::npos)

{

route\_list = route\_list + "," + ip;

}

broadcastRouteRequest(ROUTE\_REQUEST, source\_ip, destination\_ip,

packet\_id, route\_list);

}

void forwardData(Data \*data, string sender\_ip)

{

pair<string, unsigned long long int> pkt\_info = make\_pair(data->

getPacket()->getSource(), data->getPacket()->getPacketID());

sendDataImmediate(sender\_ip, string(MAC\_ACK) + DELIMITER +

pkt\_info.first + DELIMITER + data->getPacket()->

getPacketIDString() + DELIMITER + data->getPacket()->

getDestination());

if(processed\_packets.find(pkt\_info) == processed\_packets.end())

{

processed\_packets.insert(pkt\_info);

if(data->getPacket()->getDestination() == ip)

{

cout << "Message from " << data->getPacket()->getSource() <<

": ’" << data->getPacket()->getContent() << "’" << endl

<< PROMPT\_STRING;

fflush(stdout);

if(cache.isRouteCached(pkt\_info.first))

{

sendDataImmediate(cache.fetchRoute(pkt\_info.first), string(

UDP\_ACK) + DELIMITER + pkt\_info.first + DELIMITER +

data->getPacket()->getPacketIDString() + DELIMITER + ip

);

}

else

{

sendRouteRequest(pkt\_info.first);

}

return;

}

else

{

addToQueue(data);

if(cache.isRouteCached(data->getPacket()->getDestination()))

{

pthread\_t th;

pthread\_create(&th, NULL, sendDataPacket, data);

}

else

{

char id[25];

sprintf(id, "%llu", next\_packet\_id++);

sendDataImmediate(cache.fetchRoute(data->getPacket()->

getSource()), string(ROUTE\_ERROR) + DELIMITER + ip +

DELIMITER + string(id) + DELIMITER + data->getPacket()

->getSource() + DELIMITER + ip);

}

}

}

else

{

return;

}

}

void forwardUACK(string orig\_src, string orig\_id, string orig\_dest

)

{

if(orig\_src == ip)

{

unsigned long long int id = atoll(orig\_id.c\_str());

cout << "Message with id " << orig\_id << " delivered to " <<

orig\_dest << endl << PROMPT\_STRING;

fflush(stdout);

for(int i = 0; i < packet\_queues[orig\_dest].size(); i++)

{

if(packet\_queues[orig\_dest][i]->getPacket()->getPacketID() ==

id)

{

packet\_queues[orig\_dest][i]->acknowledge();

packet\_queues[orig\_dest][i]->in\_queue = false;

packet\_queues[orig\_dest].erase(packet\_queues[orig\_dest].

begin() + i);

break;

}

}

}

else

{

if(cache.isRouteCached(orig\_src)) // To avoid pointer error.

{

sendDataImmediate(cache.fetchRoute(orig\_src), string(UDP\_ACK)

+ DELIMITER + orig\_src + DELIMITER + orig\_id + DELIMITER

+ orig\_dest);

}

}

}

void forwardRouteError(string rerr\_src, unsigned long long int

rerr\_pkt\_id, string data\_src, string path)

{

pair<string, unsigned long long int> pkt\_info = make\_pair(

rerr\_src, rerr\_pkt\_id);

if(processed\_packets.find(pkt\_info) != processed\_packets.end())

// Already processed

{

return;

}

processed\_packets.insert(pkt\_info);

vector<string> ip\_list = StringOperations::split(path, ",");

cache.removePath(ip\_list[0]);

for(int i = 0; i < ip\_list.size(); i++)

{

cache.updateRoute(ip\_list[i], ip\_list[0]);

}

if(ip != data\_src)

{

char id[25];

sprintf(id, "%llu", rerr\_pkt\_id);

sendDataImmediate(cache.fetchRoute(data\_src), string(

ROUTE\_ERROR) + DELIMITER + rerr\_src + DELIMITER + string(id

) + DELIMITER + data\_src + DELIMITER + ip + "," + path);

}

}

**Link for this code :**

<http://www.cse.iitd.ac.in/~mcs142144/documents/DSR_thesis.pdf>

explanation is also provided in it.