JADAVPUR UNIVERSITY
Kolkata – 700 032

DEPARTMENT OF
ELECTRONICS AND TELE-COMMUNICATION ENGINEERING

COMMUNICATION NETWORK LABORATORY

Name of the Student:_____

Roll Number: _____

Year: _____ Session: _____ Semester: _____

Teacher in-Charge: _____

Co-Workers:

Name	Roll Number

LIST OF EXPERIMENTS

EXPT. No.	DATE	NAME OF THE EXPERIMENT	TEACHERS' INITIAL
1.A		**1. INTRODUCTION TO NS2** Node and Link Creation using NS2	
1.B		**2. INTRODUCTION TO DATA LINK LAYER** To Study the Transmission of Packets over Ethernet LAN and to verify CSMA/CD protocol using NS2	
1.C		**3. INTRODUCTION TO NETWORK LAYER** Verification of Distance Vector Routing Protocol using NS2	
1.D		**4. INTRODUCTION TO TRANSPORT LAYER** • To Study the performance of UDP using NS2 • To Study the performance of TCP using NS2 • To Study the the performance of UDP and TCP together using NS2	
1.E		**ASSIGNMENT/ EXERCISE ON NS2**	
2.		Implementation of PC to PC Serial Communication using RS-232C Serial Port	
3.A		**CRYPTOGRAPHY** Implementation of **RSA** Algorithm (Wireless Mode)	
3.B		**CRYPTOGRAPHY** Implementation of **RC4** Algorithm (Wireless Mode)	
4		To transmit and receive strings of Data / Messages through MQTT Protocol using HiveMQ and MQTTBox	
5		Sending SMS using GSM module and Arduino Uno Board	

1. INTRODUCTION TO NS2

Node and Link Creation using NS2

1.0 **Introduction**: In this section, you are going to develop a TCL script for ns which simulates a simple topology. You are going to learn how to set up nodes and links, how to send data from one node to another, how to monitor a queue and how to start NAM file and TRACE file from your simulation script to visualize your simulation.

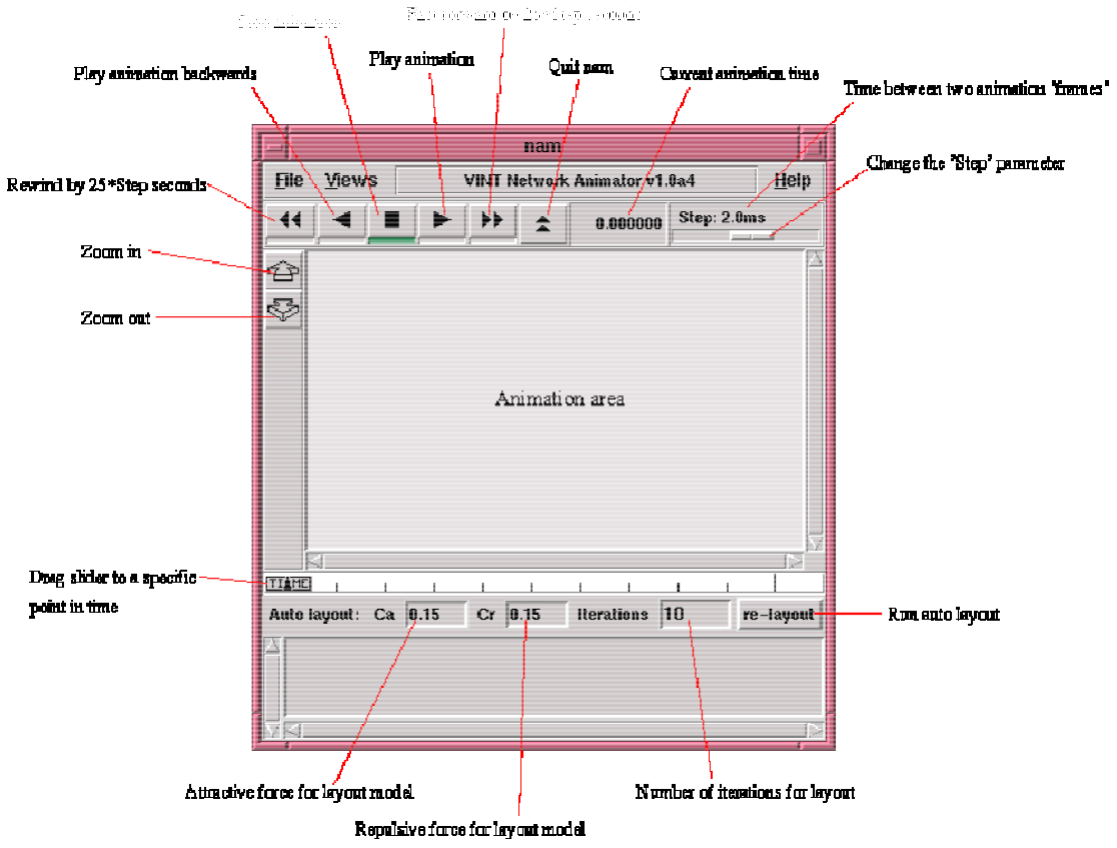
1.1 Introduction to nam and trace file

First of all, you need to create a simulator object. This is done with the command

```
set ns [new Simulator]
```

Now we open a file for writing that is going to be used for the nam trace data.

```
set nf [open out.nam w]
$ns namtrace-all $nf
```



The first line opens the file 'out.nam' for writing and gives it the file handle 'nf'. In the second line we tell the simulator object that we created above to write all simulation data that is going to be relevant for nam into this file.

The next step is to add a 'finish' procedure that closes the trace file and starts nam.

```
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exec namout.nam&
        exit 0
}
```

The next line tells the simulator object to execute the 'finish' procedure after 5.0 seconds of simulation time.

```
$ns at 5.0 "finish"
```

The last line finally starts the simulation.

```
$ns run
```

1.2 How to run/execute the code

You can actually save the file now and try to run it with 'ns example1.tcl'. You are going to get an error message like 'nam: empty trace file out.nam' though, because until now we haven't defined any objects (nodes, links, etc.) or events.

Experiment 1A : Node and Link creation using NS2

In this section we are going to define a very simple topology with two nodes that are connected by a link. The following two lines define the two nodes.

```
set n0 [$ns node]
set n1 [$ns node]
```

A new node object is created with the command '$ns node'. The above code creates two nodes and assigns them to the handles 'n0' and 'n1'.

The next line connects the two nodes.

```
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
```

This line tells the simulator object to connect the nodes n0 and n1 with a duplex link with the bandwidth 1Megabit, a delay of 10ms and a DropTail queue.

Now you can save your file and start the script with 'ns example1.tcl'. nam will be started automatically and you should see an output that resembles the picture below.

Sending data
In ns, data is always being sent from one 'agent' to another. So the next step is to create an agent object that sends data from node n0, and another agent object that receives the data on node n1.

```
#Create a UDP agent and attach it to node
n0 set udp0 [new Agent/UDP] $ns attach-
agent $n0 $udp0
# Create a CBR traffic source and attach it to udp0
```

```
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

These lines create a UDP agent and attach it to the node n0, then attach a CBR traffic generator to the UDP agent. CBR stands for 'constant bit rate'. Line 7 and 8 should be self-explaining. The packetSize is being set to 500 bytes and a packet will be sent every 0.005 seconds (i.e. 200 packets per second).

The next lines create a Null agent which acts as traffic sink and attach it to node n1.

```
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
```

Now the two agents have to be connected with each other.

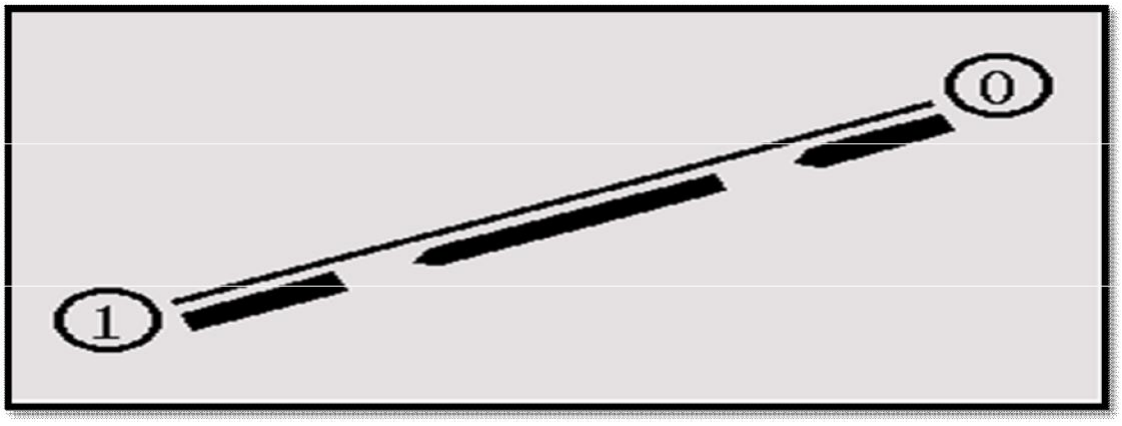
```
$ns connect $udp0 $null0
```

And now we have to tell the CBR agent when to send data and when to stop sending. Note: It's probably best to put the following lines just before the line '$ns at 5.0 "finish"'.

```
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

This code should be self-explaining again.

Now you can save the file and start the simulation again. When you click on the 'play' button in the nam window, you will see that after 0.5 simulation seconds, node 0 starts sending data packets to node 1. You might want to slow nam down then with the 'Step' slider.



Problem Statement

Create two Nodes with duplex link,naming as Node 0 and Node 1, with Bandwidth 1 Megabit, a delay of 10ms and a Drop Tail queue. Create UDP agents and attach it to Node 0 and Node 1. Attach CBR traffic to UDP agent. Set the packet size to 500 bytes and packet interval at0.005 s. In the time interval, t = 0.05s to t= 2.0s Node 0 sends packets to Node 1 and similarly from time interval t= 2.5 s to 5.0 s Node 1 sends packets to Node 0(keeping all parameters same). Simulation stops at t = 5.5 s.

Solution to Problem

```
set ns [new Simulator]

setntrace [open prog.tr w]
$ns trace-all $ntrace

setnamfile [open prog.nam w]
$ns namtrace-all $namfile

proc finish {} {
global ns ntracenamfile

$ns flush-trace
close $ntrace
close $namfile

execnamprog.nam&
exit 0
}
set n0 [$ns node]
set n1 [$ns node]
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

set null0 [new Agent/UDP]
$ns attach-agent $n1 $null0

set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500 $cbr0 set
interval_ 0.005
$cbr0 attach-agent $udp0

$ns connect $udp0 $null0

set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1

set null1 [new Agent/UDP]
$ns attach-agent $n0 $null1
```

```
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500 $cbr1 set
interval_ 0.005
$cbr1 attach-agent $udp1

$ns connect $null1 $udp1

$ns at 0.5 "$cbr0 start"

$ns at 2.0 "$cbr1 start"
$ns at 2.5 "$cbr0 stop"
$ns at 5.0 "$cbr1 stop"

$ns at 5.5 "finish"

$ns run
```

Exercise:

ODD groups

- **Set packet size = 1000**
- **Interval = .006**
- **Node 0 to Node 1 time interval = 0.3 s to 3 s**
- **Node 1 to Node 0 time interval = 3.5 s to 5s**
- **Stop the simulation at**

5s. EVEN groups

- **Set packet size = 800**
- **Interval = .008**
- **Node 0 to Node 1 time interval = 0.6 s to 2s**
- **Node 1 to Node 0 time interval = 2.5 s to 5s**
- **Stop the simulation at 5.5 s.**

1.1 Conclusion

You have learned to create Nodes, links, define link type, setting time intervals as per the requirement, starting and stopping of the simulation.

1.2 Reference

[a] TEL-WIMAX-NS2 project documentation [www.TEL.unl.edu]
[b] http://nsnam.isi.edu/nsnam/index.php/NS-2_Trace_Formats
[c] Tutorial for the Network Simulator “ns” by Marc Greis

Introduction to Data Link layer

To Study the Transmission of Packets over Ethernet LAN and to verify the CSMA/CD Protocol using NS2

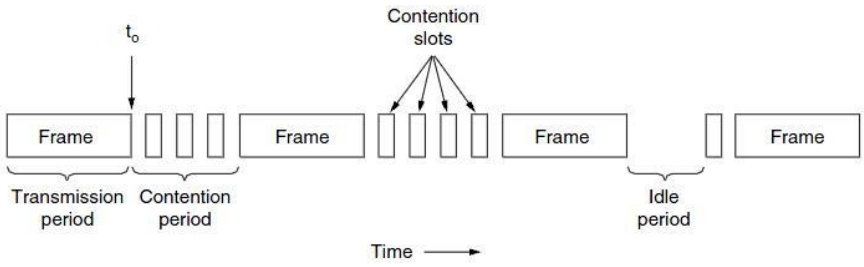
2.1 CSMA/CD

This protocol, known as **CSMA/CD** (**Carrier Sense Multiple Access with Collision Detection**), is the basis of the classic Ethernet LAN. It is important to realize that collision detection is an analog process.

The station's hardware must listen to the channel while it is transmitting. If the signal it reads back is different from the signal it is putting out, it knows that a collision is occurring. The implications are that a received signal must not be tiny compared to the transmitted signal (which is difficult for wireless, as received signals may be 1,000,000 times weaker than transmitted signals) and that the modulation must be chosen to allow collisions to be detected (e.g., a collision of two 0- volt signals may well be impossible to detect).

CSMA/CD, as well as many other LAN protocols, uses the conceptual model of Fig. At the point marked t_0, a station has finished transmitting its frame.

Any other station having a frame to send may now attempt to do so. If two or more stations decide to transmit simultaneously, there will be a collision. If a station detects a collision, it aborts its transmission, waits a random period of time, and then tries again (assuming that no other station has started transmitting in the meantime). Therefore, our model for CSMA/CD will consist of alternating contention and transmission periods, with idle periods occurring when all stations are quiet (e.g., for lack of work).



CSMA/CD can be contention, transmission or idle state

Experiment 1B :

To Study the Transmission of Packets over Ethernet LAN and to verify CSMA/CD Protocol using NS2

```
########################## TCL CODE ############################

#Create Simulator
set ns [new Simulator]


#Use colors to differentiate the traffic
$ns color 1 Blue
$ns color 2 Red


#Open trace and NAM trace file
Set ntrace [open prog5.tr w]
$ns trace-all $ntrace
Set namfile [open prog5.nam w]
$ns namtrace-all $namfile

#Finish Procedure
proc Finish {} {
global ns ntracenamfile

#Dump all trace data and close the files
$ns flush-trace
close $ntrace
close $namfile

#Execute the nam animation file
execnam prog5.nam &

exit 0

}


#Create 6 nodes
for {set i 0} {$i< 6} {incr i} {
set n($i) [$ns node]
}


#Create duplex links between the nodes
$ns duplex-link $n(0) $n(2) 2Mb 10ms DropTail
$ns duplex-link $n(1) $n(2) 2Mb 10ms DropTail
$ns duplex-link $n(2) $n(3) 4Mb 100ms DropTail

Set lan [$ns newLan "$n(3) $n(4) $n(5)" 1Mb 40ms LL

Queue/DropTail MAC/csma/cd Channel]

$ns duplex-link-op $n(0) $n(2) orient right-down

$ns duplex-link-op $n(1) $n(2) orient right-up
```

```
$ns duplex-link-op $n(2) $n(3) orient right

$ns queue-limit $n(2) $n(3) 20
$ns duplex-link-op $n(2) $n(3) queuePos 0.5

set tcp0 [new Agent/TCP]
$tcp0 set fid_ 1
#$tcp0 set window_ 8000
$tcp0 set packetSize_ 552
$ns attach-agent $n(0) $tcp0

set sink0 [new Agent/TCPSink]
$ns attach-agent $n(4) $sink0
$ns connect $tcp0 $sink0

#Apply FTP Application over TCP
set ftp0 [new Application/FTP]
$ftp0 set type_ FTP
$ftp0 attach-agent $tcp0
#Setup UDP Connection between n(1) and n(5)

set udp0 [new Agent/UDP]
$udp0 set fid_ 2
$ns attach-agent $n(1) $udp0

set null0 [new Agent/Null]
$ns attach-agent $n(5) $null0
$ns connect $udp0 $null0

#Apply CBR Traffic over UDP
set cbr0 [new Application/Traffic/CBR]
$cbr0 set type_ CBR
$cbr0 set packetSize_ 1000
#$cbr0 set rate_ 0.1Mb
$cbr0 set random_ false
$cbr0 attach-agent $udp0

#Schedule events
$ns at 0.1 "$cbr0 start"
$ns at 1.0 "$ftp0 start"
$ns at 20.0 "$ftp0 stop"
$ns at 24.0 "$cbr0 stop"
$ns at 25.0 "Finish"
#Run Simulation
$ns run
```

```
########################## AWK CODE ##############################

BEGIN {
      recvdSize = 0;
      startTime = 400;
      stopTime = 0;
}

{
      event = $1;
      time = $2;
      node_id = $3;
      pkt_size = $8;
      level = $4;

      #Store start time
      if(level ==”AGT” && event ==”s” &&pkt_size>= 512)
            { if (time <startTime) {
                  startTime = time;
            }
      }

      #Update total received packet’s size ans store
packet’s arrival time
      if(level ==”AGT” && event ==”r” &&pkt_size>= 512)
            { if (time >stopTime) {
                  stopTime = time;
            }

      #Rip off the header
            hdr_size = pkt_size %512;
            pkt_size -= hdr_size;

      #store received packet’s size
            recvdSize += pkt_size;
      }
}

END {
      printf("Average Throughput[kbps] = %0.2f \t\t StartTime = %0.2f
StopTime = %0.2f \n”, recvdSize/(stopTime- startTime)) *(8/1000),
startTime, stopTime);
}
```

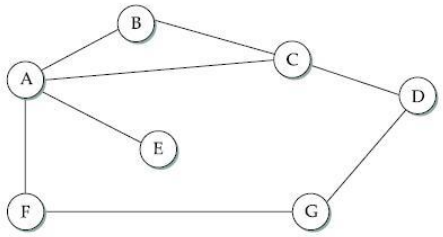
Introduction to Network Layer

Distance Vector Routing using NS2

3.1 Distance Vector Routing

In distance vector routing, each router maintains a routing table indexed by, and containing one entry for each router in the network. This entry has two parts: the preferred outgoing line to use for that destination and an estimate of the distance to that destination. The distance might be measured as the number of hops or using another metric, as we discussed for computing shortest paths. The router is assumed to know the ‘‘distance’’ to each of its neighbors. If the metric is hops, the distance is just one hop. If the metric is propagation delay, the router can measure it directly with special ECHO packets that the receiver just timestamps and sends back as fast as it can.

As an example, assume that delay is used as a metric and that the router knows the delay to each of its neighbors. Once every T msec, each router sends to each neighbor a list of its estimated delays to each destination. It also receives a similar list from each neighbor. Imagine that one of these tables has just come in from neighbor X, with Xi being X’s estimate of how long it takes to get to router i.
If the router knows that the delay to X is m msec, it also knows that it can reach router i via X in Xi + m msec. By performing this calculation for each neighbor, a router can find out which estimate seems the best and use that estimate and the corresponding link in its new routing table.



Distance-vector routing: an example network.

Information Stored at Node	Distance to Reach Node						
	A	B	C	D	E	F	G
A	0	1	1	∞	1	1	∞
B	1	0	1	∞	∞	∞	∞
C	1	1	0	1	∞	∞	∞
D	∞	∞	1	0	∞	∞	1
E	1	∞	∞	∞	0	∞	∞
F	1	∞	∞	∞	∞	0	1
G	∞	∞	∞	1	∞	1	0

Table1 : Initial distance stored at each node(Global view)

Destination	Cost	Next Hop
B	1	B
C	1	C
D	∞	—
E	1	E
F	1	F
G	∞	—

Table 2: Initial routing table of node A

Destination	Cost	Next Hop
B	1	B
C	1	C
D	2	C
E	1	E
F	1	F
G	2	F

Table 3: Final routing table of node A

Experiment 1C :

Verification of Distance Vector Routing using NS2

```
############################ TCL CODE ##############################


set ns [new Simulator]

                set nr [open thro.tr w]
                $ns trace-all $nr
                Set nf [open thro.nam w]
                $ns namtrace-all $nf


                proc finish { } {
                global ns nr nf
                  $ns flush-trace
                  close $nf
                close $nr
                execnamthro.nam&
                exit 0
                }


                for { set i 0 } { $i < 12} { incri 1 } {
                set n($i) [$ns node]
                }


                for {set i 0} {$i < 8} {incr i} {
                $ns duplex-link $n($i) $n([expr $i+1]) 1Mb 10ms DropTail
                }
                $ns duplex-link $n(0) $n(8) 1Mb 10ms DropTail
                $ns duplex-link $n(1) $n(10) 1Mb 10ms DropTail
                $ns duplex-link $n(0) $n(9) 1Mb 10ms DropTail
                $ns duplex-link $n(9) $n(11) 1Mb 10ms DropTail
                $ns duplex-link $n(10) $n(11) 1Mb 10ms DropTail
                $ns duplex-link $n(11) $n(5) 1Mb 10ms DropTail


                set udp0 [new Agent/UDP]
                $ns attach-agent $n(0) $udp0


                set cbr0 [new Application/Traffic/CBR]
                $cbr0 set packetSize_ 500
                $cbr0 set  interval_ 0.005
                $cbr0 attach-agent $udp0
                set null0 [new Agent/Null]
                ns attach-agent $n(5) $null0
                $ns connect $udp0 $null0
```

```
set udp1 [new Agent/UDP]
$ns attach-agent $n(1) $udp1
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500
$cbr1 set  interval_ 0.005
$cbr1 attach-agent $udp1

set null0 [new Agent/Null]
$ns attach-agent $n(5) $null0
$ns connect $udp1 $null0
$ns rtproto DV
$ns rtmodel-at 10.0 down $n(11) $n(5)
$ns rtmodel-at 15.0 down $n(7) $n(6)
$ns rtmodel-at 30.0 up $n(11) $n(5)
$ns rtmodel-at 20.0 up $n(7) $n(6)

$udp0 set fid_ 1
$udp1 set fid_ 2
$ns color 1 Red
$ns color 2 Green
$ns at 1.0 "$cbr0 start"
$ns at 2.0 "$cbr1 start"
$ns at 45 "finish"
$ns run
```

==========

Introduction to Transport layer

4.1 UDP

The User Datagram Protocol (UDP) provides a connectionless service for application-level procedures.

Thus, UDP is basically an unreliable service; delivery and duplicate protection are not guaranteed. However, this does reduce the overhead of the protocol and may be adequate in many cases. An example of the use of UDP is in the context of network management.
The strengths of the connection-oriented approach are clear. It allows connection- related features such as flow control, error control, and sequenced delivery.
Connectionless service, however, is more appropriate in some contexts. At lowerlayers (internet, network), a connectionless service is more. In addition, it represents a "least common denominator" of service to be expected at higher layers. Further, even at transport and above there is justification for a connectionless service.There are instances in which the overhead of connection establishment and termination is unjustified or even counterproductive.

Examples include the following:

• **Request-response:** Applications in which a transaction service is provided by a common server to a number of distributed TS users, and for which a single request-response sequence is typical. Use of the service is regulated at the application level, and lower-level connections are often unnecessary and cumbersome.
• **Real-time applications:** Such as voice and telemetry, involving a degree of redundancy and/or a real-time transmission requirement. These must not have connection-oriented functions such as retransmission.
Thus, there is a place at the transport level for both a connection-oriented and a connectionless type of service.

4.2 TCP

TCP is designed to provide reliable communication between pairs of processes (TCP users) across a variety of reliable and unreliable networks and internets. TCP provides two useful facilities forlabeling data: push and urgent.

TCP Mechanisms
We can group TCP mechanisms into the categories of connection establishment, data transfer, and connection termination.

Experiment 1D :

(i) To Study the Performance of UDP using NS2

```
################################TCL CODE#################################
#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows (for NAM)

$ns color 1 Blue
$ns color 2 Red

#Open the NAM trace file

setnf [open ju2.nam w]
$ns namtrace-all $nf


#Open the traffic trace file to record all events

setnd [open ju2.tr w]

$ns trace-all $nd

 #Define a 'finish' procedure


proc finish {} {

global ns nfnd
 $ns flush-trace

 #Close the NAM trace file

close $nf

close $nd


 #Execute NAM on the trace file


execnam ju2.nam &

exit 0

 }



#Create six nodes

set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
```

```
$ns color 1 Blue
$ns color 2 Red



$ns at 0.0 "$n0 label Sender"
$ns at 0.0 "$n1 label Sender"
$ns at 0.0 "$n2 label Sender"
$ns at 0.0 "$n3 label Sender"
$ns at 0.0 "$n5 label Receiver"

#Create links between the nodes

$ns duplex-link $n0 $n4 3Mb 10ms DropTail
$ns duplex-link $n1 $n4 3Mb 10ms DropTail
$ns duplex-link $n2 $n4 3Mb 10ms DropTail
$ns duplex-link $n3 $n4 3Mb 10ms DropTail
$ns duplex-link $n4 $n5 12Mb 20ms RED


#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n5 $n4 8

#Give node position (for NAM)
$ns duplex-link-op $n0 $n4 orient right-down
$ns duplex-link-op $n1 $n4 orient right-down
$ns duplex-link-op $n2 $n4 orient right-up
$ns duplex-link-op $n3 $n4 orient right-up


#Monitor the queue for link (n2-n3). (for NAM)

$ns duplex-link-op $n4 $n5 queuePos 0.5


#$ns duplex-link-op $n0 $n2 queuePos 0.5

#$ns duplex-link-op $n1 $n2 queuePos 0.5



#Create a UDP agent and attach it to node
n0 set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0


# Create a CBR traffic source and attach it to udp0
```

```
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500 $cbr0 set
interval_ 0.005
$cbr0 attach-agent $udp0


#Create a UDP agent and attach it to node
n1 set
udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1

# Create a CBR traffic source and attach it to udp1

set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500 $cbr1 set
interval_ 0.005
$cbr1 attach-agent $udp1



#$ns attach-agent
 $n1 $null0

#$udp0 set fid_ 1

#$udp1 set fid_ 2

#$udp0 set packetSize_ 500

#$udp1 set packetSize_ 500

#$udp0 set rate_ 1.0Mb

#$udp1 set rate_ 1.0Mb

#set sink0 [new Agent/UDPSink]
#set sink1 [new Agent/UDPSink]

set null0 [new Agent/Null]

$ns attach-agent $n3 $null0


set null1 [new Agent/Null]

$ns attach-agent
$n3 $null1

$ns connect $udp0 $null0
```

```
$ns connect $udp1 $null1

$ns at 0.0 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
$ns at 0.0 "$cbr1 start"
$ns at 4.5 "$cbr1 stop"
#Call the finish procedure after 5 seconds of simulation time


$ns at 5.0 "finish"

#Run the simulation

$ns run
```

(ii) To Study the Performance of TCP using NS2

```
################################TCL CODE#################################

#Create a simulator object

set ns [new Simulator]


#Define different colors for data flows (for NAM)

$ns color 1 Blue

$ns color 2 Red


#Open the NAM trace file

setnf [open ju1.nam w]

$ns namtrace-all $nf


#Open the traffic trace file to record all events

setnd [open ju1.tr w]

$ns trace-all $nd


#Define a 'finish' procedure
```

```
proc finish {} {
global ns nfnd
        $ns flush-trace
        #Close the NAM trace file
close $nf
close $nd
        #Execute NAM on the trace file
execnam ju1.nam &
exit 0
}
#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
$ns color 1 Blue
$ns color 2 Red

$ns at 0.0 "$n0 label Sender"
$ns at 0.0 "$n1 label Sender"
$ns at 0.0 "$n3 label Receiver"
#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns duplex-link $n2 $n3 2Mb 20ms RED

#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n3 $n2 4
```

```
#Give node position (for NAM)

$ns duplex-link-op $n0 $n2 orient right-down

$ns duplex-link-op $n1 $n2 orient right-up

$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for link (n2-n3). (for NAM)

$ns duplex-link-op $n2 $n3 queuePos 0.5
#$ns duplex-link-op $n0 $n2 queuePos 0.5
#$ns duplex-link-op $n1 $n2 queuePos 0.5

#Setup a TCP connection
set tcp0 [new Agent/TCP]
set tcp1 [new Agent/TCP]
$tcp0 set fid_ 1
$tcp1 set fid_ 2
$tcp0 set packetSize_ 1000
$tcp1 set packetSize_ 1000
$tcp0 set rate_ 1.0Mb
$tcp1 set rate_ 1.0Mb

$ns attach-agent $n0 $tcp0
$ns attach-agent $n1 $tcp1

set sink0 [new Agent/TCPSink]
set sink1 [new Agent/TCPSink]

$ns attach-agent $n3 $sink0
$ns attach-agent $n3 $sink1

$ns connect $tcp0 $sink0

$ns connect $tcp1 $sink1

#Setup a FTP over TCP connection
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0

set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1


$ns at 0.5 "$ftp0 start"

$ns at 4.5 "$ftp0 stop"

$ns at 1.0 "$ftp1 start"

$ns at 4.0 "$ftp1 stop"
```

```
#Call the finish procedure after 5 seconds of simulation time

$ns at 5.0 "finish"

#Run the simulation

$ns run
```

(iii) To Study the Performance of UDP and TCP together using NS2

```
################################TCL CODE#################################


#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red

#Open the NAM trace file
setnf [open out.nam w]
$ns namtrace-all $nf
setnd [open ju5.tr w]
$ns trace-all $nd

#Define a 'finish' procedure
proc finish {} {
global ns nfnd
        $ns flush-trace
close $nd
        #Close the NAM trace file
close $nf
        #Execute NAM on the trace file
execnamout.nam&
exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

$ns color 1 Blue

$ns color 2 Red


$ns at 0.0 "$n0 label Sender"
```

```
$ns at 0.0 "$n1 label Sender"

$ns at 0.0 "$n3 label Receiver"
#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns duplex-link $n2 $n3 4Mb 20ms DropTail

#Set Queue Size of link (n2-n3) to 10

$ns queue-limit $n2 $n3 4

#Give node position (for NAM)

$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for link (n2-n3). (for NAM)

$ns duplex-link-op $n2 $n3 queuePos 0.5


#Setup a TCP connection

settcp [new Agent/TCP]
$tcp set class_ 2
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
$ns connect $tcp $sink
$tcp set fid_ 1

#Setup a FTP over TCP connection

set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP


#Setup a UDP connection

setudp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null
$udp set fid_ 2

#Setup a CBR over UDP connection

setcbr [new Application/Traffic/CBR]
$cbr attach-agent $udp

$cbr set packetSize_ 1000


#Schedule events for the CBR and FTP
agents $ns at 0.0 " $cbr start" $ns at
10.0 "$ftp start"
$ns at 80.0 "$ftp stop"
```

```
$ns at 90.0 "$cbr stop"

#Call the finish procedure after 5 seconds of simulation
time $ns at 100.0 "finish"


#Run the simulation
$ns run
```

==========

1. E. Assignment / Exercise

1. *Repeat the experiment of DSR with 10 nodes, with defining shortest path from Node0 to Node4. After half of the simulation time given to you, down the link between Node2 and Node3. Again make the link up after one fourth simulation time. Explain the meaning of each code used in your program in detail.*

2. *Taking the number of nodes as 6 for ODD groups and 8 for EVEN groups , packet size 500 for ODD groups and 800 for EVEN groups ,simulate all three programs of experiment 1D with full explanation of each code used.*

3. *Explain the relevance of TRACE file with explaining all its parameters.*

4. *Explain the effect of Changing the packet size, bit rate etc.*

Ex. No.: 2

Date:_____

IMPLEMENTATION OF PC TO PC SERIAL COMMUNICATION USING RS232C SERIAL PORT

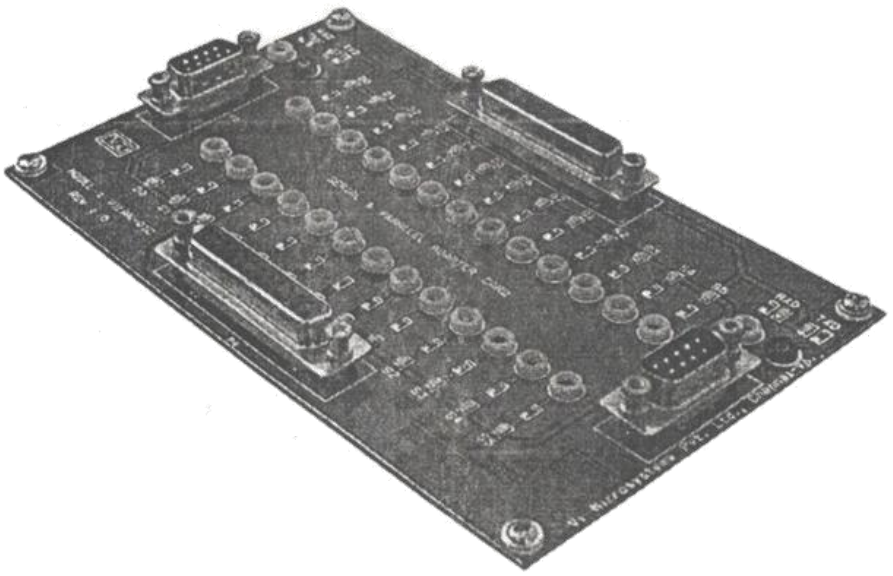
OBJECTIVE:

To implement and study the PC to PC Serial Communication using RS232C Serial Port.

THEORY:

SERIAL INTERFACE CARD:

This Interface card has been designed to teach the basics of Serial Communication between two PCs.



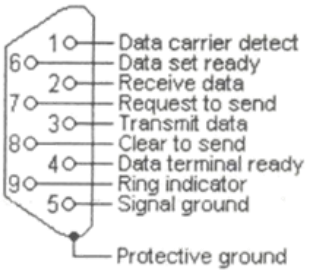
RS232 SERIAL CONNECTOR PIN ASSIGNMENT :

The RS232 connector was originally developed to use 25 pins. In this DB25 connector pin out, provisions were made for a secondary serial RS232 communication channel. In practice, only one serial communication channel with accompanying handshaking is present. Only very few computers have been manufactured, where both serial RS232 channels are implemented. Examples of this are the Sun SparcStation 10 and 20 models and the Dec Alpha Multia. Also on a number of Telebit modem models the secondary channel is present. It can be used to query the modem status while the modem is on-line

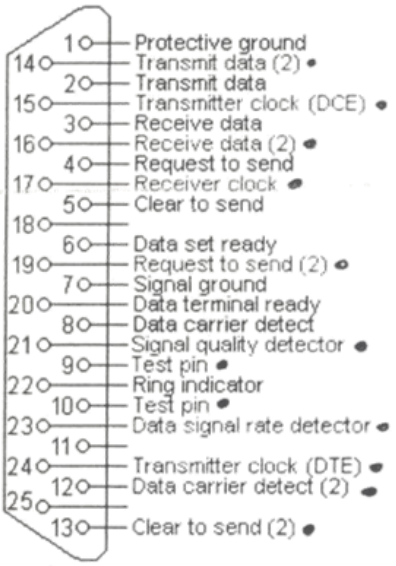
and busy communicating. On personal computers, the smaller DB9 version is more commonly used today. The diagrams show the signals common to both connector types in black. The defined pins only present on the larger connector are shown in red. Note, that the protective ground is assigned to a pin at the large connector where the connector outside is used for that purpose with the DB9 connector version.

The pinout is also shown for the DEC modified modular jack. This type of connector has been used on systems built by Digital Equipment Corporation; in the early days one of the leaders in the mainframe world. Although this serial interface is differential (the receive and transmit have their own floating ground level which is not the case with regular RS232) it is possible to connect RS232 compatible devices with this interface because the voltage levels of the bit streams are in the same range. Where the definition of RS232 focussed on the connection of DTE, data terminal equipment (computers, printers, etc.) with DCE, data communication equipment (modems), MMJ was primarily defined for the connection of two DTE's directly.

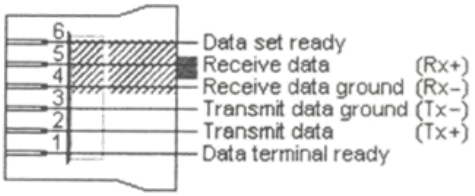
RS232 DB9 pinout



RS232 DB25 pinout



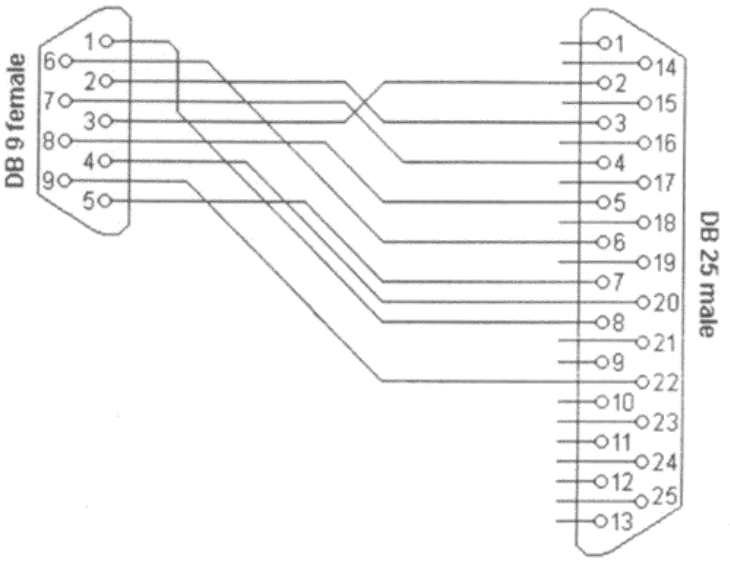
DEC MMJ pinout



RS232 DB25 TO DB9 CONVERTER

The original pinout for RS232 was developed for a 25 pins sub D connector. Since the introduction of the smaller serial port on the IBM-AT, 9 pins RS232 connectors are commonly used. In mixed applications, a 9 to 25 pins converter can be used to connect connectors of different sizes. As most of the computers are equipped with the DB9 serial port version, all wiring examples on this website will use that connector as a default. If you want to use the example with a DB25, simply replace the pin numbers of the connector according to the conversion table below.

RS232 DB9 to DB25 converter

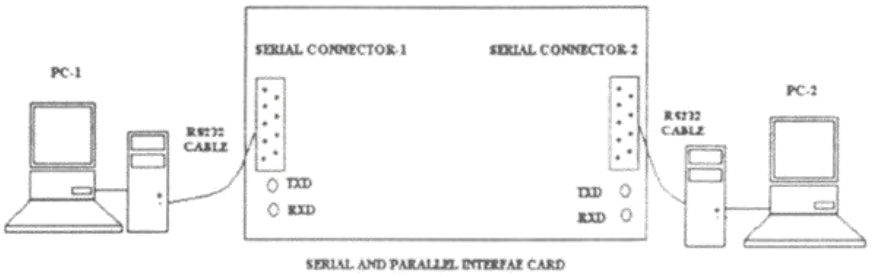


DB9 - DB25 conversion

DB9	DB25	Function
1	8	Data carrier detect
2	3	Receive data
3	2	Transmit data
4	20	Data terminal ready
5	7	Signal ground
6	6	Data set ready
7	4	Request to send
8	5	Clear to send
9	22	Ring indicator

Serial Communication Between Two PCs

This Card Contains Two 9 pin D-Type connectors for serial communication between Two PCs. TXD and RXD lines are terminated at PS/2 connector. **CTS and RTS** lines are internally connected.

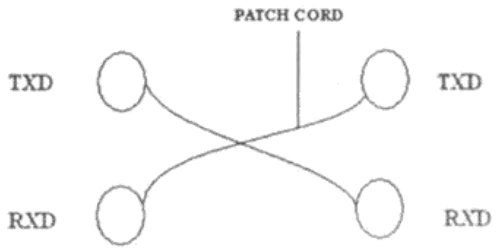


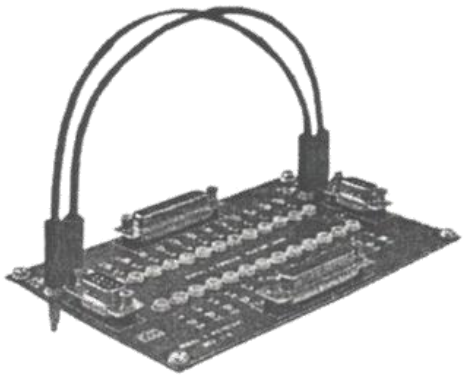
Requirements:

1. Two RS232 cable.
2. Two PCs with RS232 connector
3. Two Patch Cords.

Procedure:

1. One End Rs232 Cable is connected to PC-1 and other end is connected to serial connector1.
2. Like wise another one cable is connected on serial connector2.
3. Connect the patch cord as follows





PROGRAM FOR SERIAL COMMUNICATION USING RS232

Send and receive the Data in Both Directions

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#define PORT1 0x3F8
/* Defines Serial Ports Base Address */
/* COM1 0x3F8 */
/* COM2 0x2F8 */
/* COM3 0x3E8 */
/* COM4 0x2E8 */
void main(void)
{
int c;
int ch;
outportb(PORT1 + 1 , 0); /* Turn off interrupts - Port1 */
/* PORT 1 - Communication Settings */
outportb(PORT1 + 3 , 0x80); /* SET DLAB ON */
outportb(PORT1 + 0 , 0x03); /* Set Baud rate - Divisor Latch Low Byte */
/* Default 0x03 = 38,400 BPS */
/* 0x01 = 115,200 BPS */
/* 0x02 = 57,600 BPS */
/* 0x06 = 19,200 BPS */
/* 0x0C = 9,600 BPS */
/* 0x18 = 4,800 BPS */
/* 0x30 = 2,400 BPS */
outportb(PORT1 + 1 , 0x00); /* Set Baud rate - Divisor Latch High Byte */
outportb(PORT1 + 3 , 0x03); /* 8 Bits, No Parity, 1 Stop Bit */
outportb(PORT1 + 2 , 0xC7); /* FIFO Control Register */
outportb(PORT1 + 4 , 0x0B); /* Turn on DTR, RTS, and OUT2 */
printf("\nSample Comm's Program. Press ESC to quit \n");
do { c = inportb(PORT1 + 5); /* Check to see if char has been */
/* received. */
if (c & 1) {ch = inportb(PORT1); /* If so, then get Char */
printf("%c",ch);} /* Print Char to Screen */
if (kbhit()){ch = getch(); /* If key pressed, get Char */
outportb(PORT1, ch);} /* Send Char to Serial Port */
} while (ch !=27); /* Quit when ESC (ASC 27) is pressed */
}
```

5.4. PROGRAM-1

1. SERIAL COMMUNICATION USING RS232

a. To write the program for Send the Data in Serially

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#define PORT1 0x3F8

/* Defines Serial Ports Base Address */
/* COM1 0x3F8                        */
/* COM2 0x2F8                        */
/* COM3 0x3E8                        */
/* COM4 0x2E8                        */

void main(void)
{

int c;
int ch;
clrscr();
printf("\n\r-------------------------------------------------");
printf("\n\r                                                 ");
printf("\n\r          SERIAL COMMUNICATION                   ");
printf("\n\r            ( SERIAL OUT)                        ");
printf("\n\r-------------------------------------------------");

outportb(PORT1 + 1 , 0);  /* Turn off interrupts - Port1 */

/*       PORT 1 - Communication Settings       */
outportb(PORT1 + 3 , 0x80); /* SET DLAB ON */
outportb(PORT1 + 0 , 0x03); /* Set Baud rate - Divisor Latch Low Byte */
                /* Default 0x03 =  38,400 BPS */
                /*       0x01 = 115,200 BPS */
                /*       0x02 =  57,600 BPS */
                /*       0x06 =  19,200 BPS */
                /*       0x0C =   9,600 BPS */
                /*       0x18 =   4,800 BPS */
                /*       0x30 =   2,400 BPS */
outportb(PORT1 + 1 , 0x00); /* Set Baud rate - Divisor Latch High Byte */
outportb(PORT1 + 3 , 0x03); /* 8 Bits, No Parity, 1 Stop Bit */
outportb(PORT1 + 2 , 0xC7); /* FIFO Control Register */
outportb(PORT1 + 4 , 0x0B); /* Turn on DTR, RTS, and OUT2 */
```

```
printf("\nPress ESC to quit \n");
printf("\n\r Enter the Data \n\r");
do {
    if (kbhit())
    {
    ch = getch();       /* If key pressed, get Char */
    printf("%c",ch);
    outportb(PORT1, ch);
    }           /* Send Char to Serial Port */

  } while (ch !=27); /* Quit when ESC (ASC 27) is pressed */
}
```

b. Read the Data From Serial Port

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8

 /* Defines Serial Ports Base Address       */
 /* COM1 0x3F8                          */
 /* COM2 0x2F8              */
 /* COM3 0x3E8              */
 /* COM4 0x2E8              */

void main(void)
{
 int c;
 int ch;
 clrscr();
 printf("\n\r-------------------------------------------------------");
 printf("\n\r                                                       ");
 printf("\n\r          Serial Communication                  ");
 printf("\n\r             (Serial in)                              ");
 printf("\n\r-------------------------------------------------------");

 outportb(PORT1 + 1 , 0);   /* Turn off interrupts - Port1 */

 /*       PORT 1 - Communication Settings       */

 outportb(PORT1 + 3 , 0x80);  /* SET DLAB ON */
 outportb(PORT1 + 0 , 0x03);  /* Set Baud rate - Divisor Latch Low Byte */
            /* Default 0x03 =  38,400 BPS */
```

```
                /*      0x01 = 115,200 BPS */
                /*      0x02 =  57,600 BPS */
                /*      0x06 =  19,200 BPS */
                /*      0x0C =   9,600 BPS */
                /*      0x18 =   4,800 BPS */
                /*      0x30 =   2,400 BPS */
outportb(PORT1 + 1 , 0x00); /* Set Baud rate - Divisor Latch High Byte */
outportb(PORT1 + 3 , 0x03); /* 8 Bits, No Parity, 1 Stop Bit */
outportb(PORT1 + 2 , 0xC7); /* FIFO Control Register */
outportb(PORT1 + 4 , 0x0B); /* Turn on DTR, RTS, and OUT2 */

printf("\n\r  Press ESC to quit \n");
printf("\n\r");
printf("\n\r Received Data :.....\n\r");
do {
    c = inportb(PORT1 + 5);         /* Check to see if char has been */
                        /* received.                    */
    if (c & 1) {ch = inportb(PORT1); /* If so, then get Char         */
          printf("%c",ch);}    /* Print Char to Screen         */

    if (kbhit())ch = getch();

  } while (ch !=27); /* Quit when ESC (ASC 27) is pressed */
}
```

c. Send and receive the Data in Both Direction

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>

#define PORT1 0x3F8

 /* Defines Serial Ports Base Address */
 /* COM1 0x3F8                    */
 /* COM2 0x2F8                 */
 /* COM3 0x3E8                 */
 /* COM4 0x2E8                 */

void main(void)
{
 int c;
 int ch;
 outportb(PORT1 + 1 , 0);   /* Turn off interrupts - Port1 */
```

OBSERVATION:

1. Demonstrate the Transmission and Reception of Characters on PC1 and PC2 in both directions simultaneously (Full Duplex mode).

2. Verify & Demonstrate the Transmission and Reception of Characters on PC1 and PC2 in Half Duplex mode by writing the appropriate C Program.

CONCLUSION:

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Examples of parallel transfers are printers and hard disks; each uses cables with many wire strips. Although in such cases a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used. In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. Serial communication of the 8051 is the topic of this chapter. The 8051 has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

In this chapter we first discuss the basics of serial communication. In Section 10.2, 8051 interfacing to RS232 connectors via MAX232 line drivers is discussed. Serial communication programming of the 8051 is discussed in Section 10.3

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

When a microprocessor communicates with the outside world, it provides the data in byte-sized chunks. In some cases, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the printer. This can work only if the cable is not too long, since long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart. Figure 10-1 diagrams serial versus parallel data transfers.

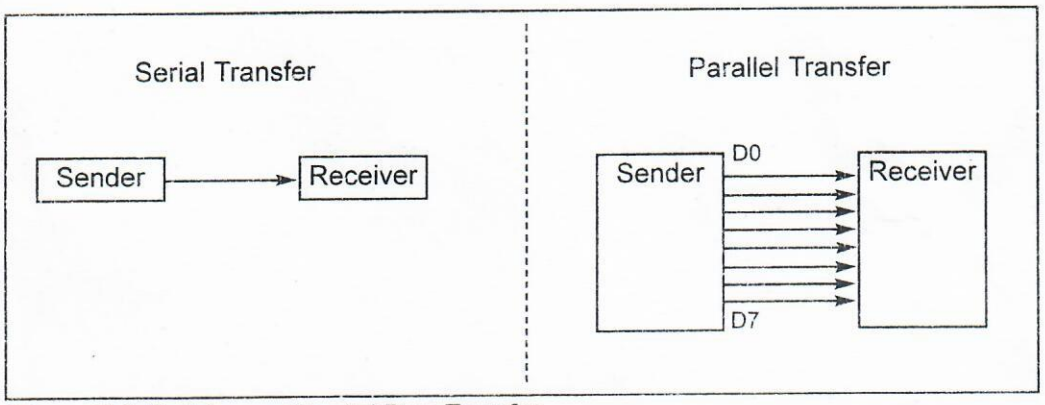


Figure 10-1. Serial versus Parallel Data Transfer

The fact that in serial communication a single data line is used instead of the 8-bit data line of parallel communication makes it not only much cheaper but also makes it possible for two computers located in two different cities to communicate over the telephone.

For serial data communication to work, the byte of data must be converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted

over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data and pack them into a byte. Of course, if data is to be transferred on the telephone line, it must be converted from 0s and 1s to audio tones, which are sinusoidal-shaped signals. This conversion is performed by a peripheral device called a *modem*, which stands for "modulator/demodulator."

When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation. This is how IBM PC keyboards transfer data to the motherboard. However, for long-distance data transfers using communication lines such as a telephone, serial data communication requires a modem to *modulate* (convert from 0s and 1s to audio tones) and *demodulate* (converting from audio tones to 0s and 1s).

Serial data communication uses two methods, asynchronous and synchronous. The *synchronous* method transfers a block of data (characters) at a time while the *asynchronous* transfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, there are special IC chips made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The 8051 chip has a built-in UART, which is discussed in detail in Section 10.3.

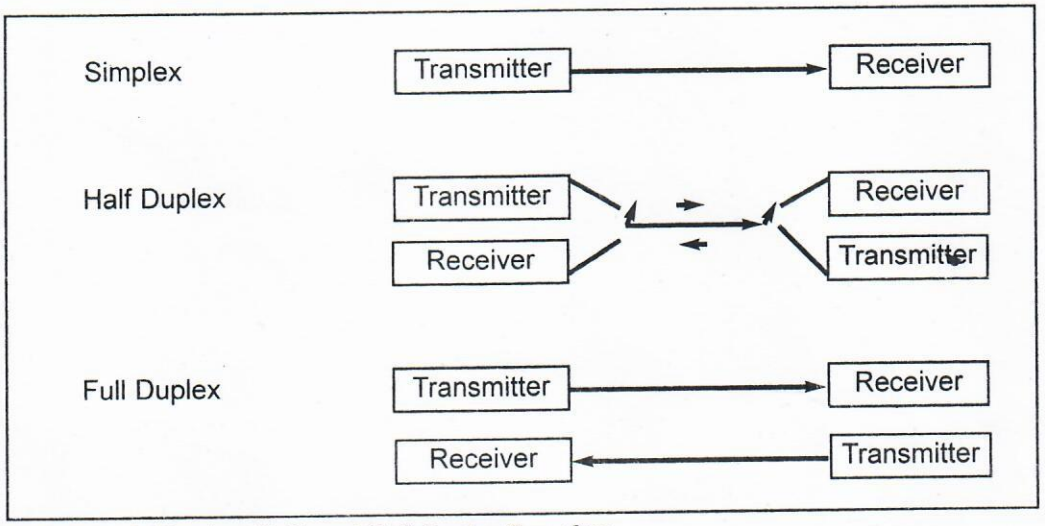


Figure 10-2. Simplex, Half-, and Full-Duplex Transfers

Half- and full-duplex transmission

In data transmission if the data can be transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex*. If the data

can go both ways at the same time, it is *full duplex*. Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously. See Figure 10-2.

Asynchronous serial communication and data framing

The data coming in at the receiving end of the data line in a serial data transfer is all 0s and 1s; it is difficult to make sense of the data unless the sender and receiver agree on a set of rules, a *protocol*, on how the data is packed, how many bits constitute a character, and when the data begins and ends.

Start and stop bits

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed in between start and stop bits. This is called *framing*. In data framing for asynchronous communications, the data, such as ASCII characters, are packed in between a start bit and a stop bit. The start bit is always one bit but the stop bit can be one or two bits. The start bit is always a 0 (low) and the stop bit(s) is 1 (high). For example, look at Figure 10-3 in which the ASCII character "A" (8-bit binary 0100 0001) is framed in between the start bit and a single stop bit. Notice that the LSB is sent out first.

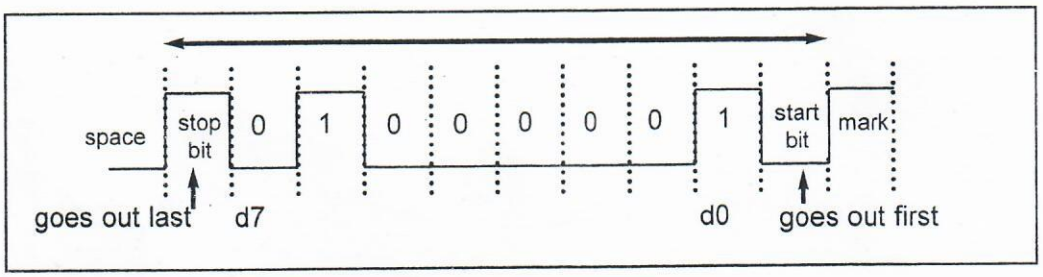


Figure 10-3. Framing ASCII "A" (41H)

Notice in Figure 10-3 that when there is no transfer, the signal is 1 (high), which is referred to as *mark*. The 0 (low) is referred to as *space*. Notice that the transmission begins with a start bit followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years due to the extended ASCII characters, 8-bit data has become common. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. However, in modern PCs the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, for each 8-bit character there are an extra 2 bits, which gives 25% overhead.

In some systems in order to maintain data integrity, the parity bit of the character byte is included in the data frame. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd-parity bit the number of data bits, including the parity bit, has an odd number of 1s. Similarly, in an even-parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even-parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not necessarily equal. This is due to the fact that baud rate is the modem terminology and is defined as the number of signal changes per second. In modems, there are occasions when a single change of signal transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. However in recent years, Pentium-based PCs transfer data at rates as high as 56K bps. It must be noted that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

RS232 standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969, respectively. In this book we refer to it simply as RS232. Today, RS232 is the most widely used serial I/O interfacing standard. This standard is used in PCs and numerous types of equipment. However, since the standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by −3 to −25 V, while a 0 bit is +3 to +25 V, making −3 to +3 undefined. For this reason, to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage level, and vice versa. MAX232 IC chips are commonly referred to as line drivers. RS232 connection to MAX232 is discussed in Section 10.2.

RS232 pins

Table 10-1 provides the pins and their labels for the RS232 cable, commonly referred to as the DB-25 connector. In labeling, DB-25P refers to the plug connector (male) and DB-25S is for the socket connector (female). See Figure 10-4.

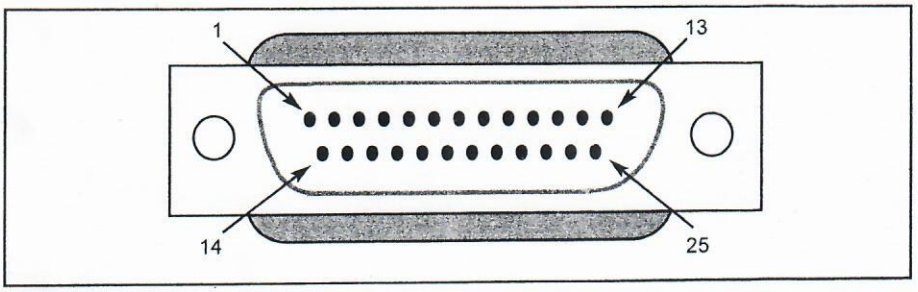


Figure 10-4. RS232 Connector DB-25

Since not all the pins are used in PC cables, IBM introduced the DB-9 version of the serial I/O standard, which uses 9 pins only, as shown in Table 10-2. The DB-9 pins are shown in Figure 10-5.

Data communication classification

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receive data, while DCE refers to communication equipment, such as modems, that are responsible for transferring the data. Notice that all the RS232 pin function definitions of Tables 10-1 and 10-2 are from the DTE point of view.

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground, as shown in Figure 10-6. Notice in that figure that the RxD and TxD pins are interchanged.

Table 10-1: RS232 Pins (DB-25)

Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send ($\overline{\text{RTS}}$)
5	Clear to send ($\overline{\text{CTS}}$)
6	Data set ready ($\overline{\text{DSR}}$)
7	Signal ground (GND)
8	Data carrier detect ($\overline{\text{DCD}}$)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready ($\overline{\text{DTR}}$)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

Examining RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Just as in the case of the printer, due to the fact that in serial data communication the receiving device may have no room for the data, there must be a way to inform the sender to stop sending data. Many of the pins of the RS-232 connector are used for handshaking signals. Their description is provided below only as a reference and they can be by-passed since they are not supported by the 8051 UART chip.

1. DTR (data terminal ready). When the terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-low signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.
2. DSR (data set ready). When DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Thus, it is an output from the modem (DCE) and input to the PC (DTE). This is an active-low signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.
3. RTS (request to send). When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-low output from the DTE and an input to the modem.
4. CTS (clear to send). In response to RTS, when the modem has room for storing the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.
5. DCD (carrier detect, or DCD, data carrier detect). The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modem and an input to the PC (DTE).
6. RI (ring indicator). An output from the modem (DCE) and an input to a PC (DTE) indicates that the telephone is ringing. It goes on and off in synchronization with the ringing sound. Of the 6 handshake signals, this is the least often used, due to the fact that modems take care of answering the phone. However, if in a given system the PC is in charge of answering the phone, this signal can be used.

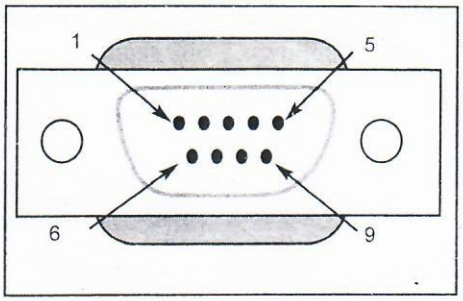


Figure 10-5. DB-9 9-Pin Connector

Table 10-2: IBM PC DB-9 Signals

Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)

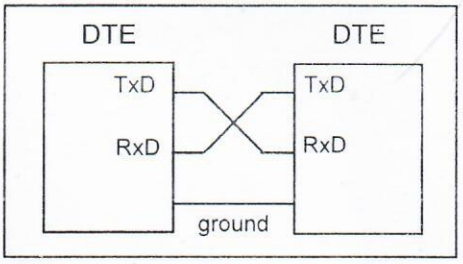


Figure 10-6. Null Modem Connection

From the above description, PC and modem communication can be summarized as follows: While signals DTR and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, if the modem is ready (has room) to accept the data, it sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as hardware control flow signals.

This concludes the description of the 9 most important pins of the RS232 handshake signals plus TxD, RxD, and ground. Ground is also referred to as SG (signal ground).

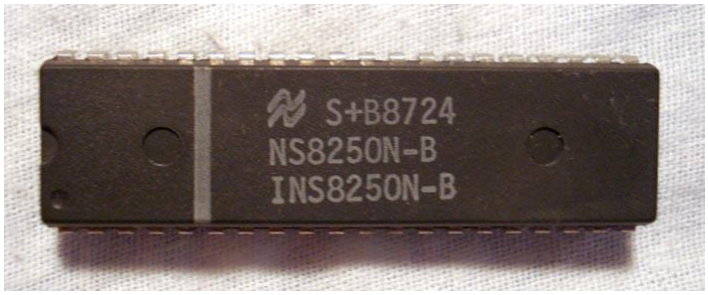
IBM PC/compatible COM ports

IBM PC/compatible computers based on x86 (8086, 286, 386, 486, and Pentium) microprocessors normally have two COM ports. Both COM ports have RS232-type connectors. Many PCs use one each of the DB-25 and DB-9 RS232 connectors. The COM ports are designated as COM 1 and COM 2. In recent years, COM 1 is used for the mouse and COM 2 is available for devices such as a modem. We can connect the 8051 serial port to the COM 2 port of a PC for serial communication experiments.

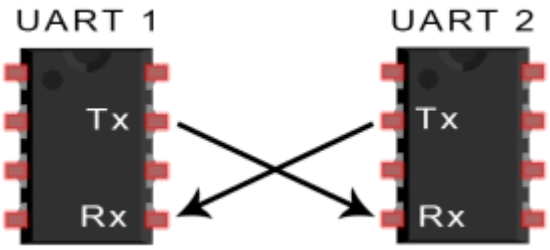
With this background in serial communication, we are ready to look at the 8051. In the next section we discuss the physical connection of the 8051 and RS232 connector, and in Section 10.3 we see how to program the 8051 serial communication port.

Review Questions

1. The transfer of data using parallel lines is _____ (faster, slower) but _____ (more expensive, less expensive).
2. True or false. Sending data to a printer is duplex.
3. True or false. In full duplex we must have two data lines, one for transfer and one for receive.
4. The start and stop bits are used in the _____ (synch, asynch) method.
5. Assuming that we are transmitting the ASCII letter "E" (0100 0101 in binary) with no parity bit and one stop bit, show the sequence of bits transferred serially.
6. In Question 5, find the overhead due to framing.
7. Calculate the time it takes to transfer 10,000 characters as in Question 5 if we use 9600 bps. What percentage of time is wasted due to overhead?
8. True or false. RS232 is not TTL-compatible.
9. What voltage levels are used for binary 0 in RS232?
10. True or false. The 8051 has a built-in UART.
11. On the back of x86 PCs, we normally have ____ COM port connectors.
12. The PC COM ports are designated by DOS and Windows as _____ and _____.



National Semiconductor's 8250B UART



The UART chip has a total of 12 different registers that are mapped into 8 different Port I/O locations.

RBR, receiver buffer register , THR, transmitter holding register ,

IER, interrupt enable register

IIR, interrupt identification register FCR, FIFO control register , LCR, line control register

MCR, modem control register , LSR, line status register , MSR, modem status register

SCR, scratch register , DLL, divisor latch LSB , DLM, divisor latch MSB

For a "typical" PC system, the following are the Port I/O addresses and IRQs for each serial COM port:

Common UART IRQ and I/O Port Addresses		
COM Port	**IRQ**	**Base Port I/O address**
COM1	IRQ4	$3F8
COM2	IRQ3	$2F8
COM3	IRQ4	$3E8
COM4	IRQ3	$2E8

UART Registers				
Base Address	**DLAB**	**I/O Access**	**Abbrv.**	**Register Name**
+0	0	Write	THR	Transmitter Holding Buffer
+0	0	Read	RBR	Receiver Buffer
+0	1	Read/Write	DLL	Divisor Latch Low Byte
+1	0	Read/Write	IER	Interrupt Enable Register
+1	1	Read/Write	DLH	Divisor Latch High Byte
+2	x	Read	IIR	Interrupt Identification Register
+2	x	Write	FCR	FIFO Control Register
+3	x	Read/Write	LCR	Line Control Register
+4	x	Read/Write	MCR	Modem Control Register
+5	x	Read	LSR	Line Status Register
+6	x	Read	MSR	Modem Status Register
+7	x	Read/Write	SR	Scratch Register

Divisor Latch Byte Values (common baud rates)			
Baud Rate	**Divisor (in decimal)**	**Divisor Latch High Byte**	**Divisor Latch Low Byte**
50	2304	$09	$00
110	1047	$04	$17
220	524	$02	$0C
300	384	$01	$80
600	192	$00	$C0
1200	96	$00	$60
2400	48	$00	$30
4800	24	$00	$18
9600	12	$00	$0C
19200	6	$00	$06
38400	3	$00	$03
57600	2	$00	$02
115200	1	$00	$01

FIFO Control Register (FCR)

Bit	Notes			
7 & 6	**Bit 7**	**Bit 6**	**Interrupt Trigger Level (16 byte)**	**Trigger Level (64 byte)**
	0	0	1 Byte	1 Byte
	0	1	4 Bytes	16 Bytes
	1	0	8 Bytes	32 Bytes
	1	1	14 Bytes	56 Bytes
5	Enable 64 Byte FIFO (16750)			
4	Reserved			
3	DMA Mode Select			
2	Clear Transmit FIFO			
1	Clear Receive FIFO			
0	Enable FIFOs			

Line Control Register (LCR)

Bit	Notes			
7	Divisor Latch Access Bit			
6	Set Break Enable			
3, 4 & 5	**Bit 5**	**Bit 4**	**Bit 3**	**Parity Select**
	0	0	0	No Parity
	0	0	1	Odd Parity
	0	1	1	Even Parity
	1	0	1	Mark
	1	1	1	Space
2	0	One Stop Bit		
	1	1.5 Stop Bits or 2 Stop Bits		
0 & 1	**Bit 1**	**Bit 0**	**Word Length**	
	0	0	5 Bits	
	0	1	6 Bits	
	1	0	7 Bits	
	1	1	8 Bits	

Modem Control Register (MCR)

Bit	Notes
7	Reserved
6	Reserved
5	Autoflow Control Enabled (16750)
4	Loopback Mode
3	Auxiliary Output 2
2	Auxiliary Output 1
1	Request To Send
0	Data Terminal Ready

Ex. No. : 3

Date:_____

IMPLEMENTATION OF RSA ALGORITHM (WIRELESS MODE)

OBJECTIVE:

To implement RSA Algorithm and study its performance.

THEORY:

RSA algorithm consists of three main functional schemes:

1. Key Generation Algorithm.
2. Encryption process.
3. Decryption Process.

1. Key Generation Algorithm:

Key Generation algorithm is the most important part of RSA algorithm. In this stage, the public key as well as the private key are generated. Different steps of key generation algorithm are:

1. Two large random prime numbers P & Q are generated.
2. N = P*Q and $\phi(N)$ = (P-1)*(Q-1) are computed.
3. An integer 'E' $(1 < E < \phi(N))$ is chosen such that gcd $(E, \phi(N)) = 1$.
4. The secret exponent 'D' $(1<D<\phi(N))$ is computed such that $d \equiv e^{-1} \pmod{\phi(N)}$
5. The public key is (N, E) and the private key is (N, D),
   where, N is known as modulus,
   $\phi(N)$ is known as Euler's totient function
   E is known as public exponent or encryption exponent.
   D is known as secret exponent or decryption exponent.

2. Encryption:

Different steps of encryption process are:

1. Public key (N, E) of recipient is obtained.
2. Plain text message 'T' is represented by its corresponding ASCII value (M).
3. Cipher Data 'x' [M pow E mod N] is obtained.
4. Cipher Data 'x' is sent to recipient.

3. Decryption:

Different steps of decryption process are:

1. The decrypt 'M' [x pow D mod N] is computed using the private key (N, D).
2. The plain text 'T' is extracted from the decrypt 'M'.

PROCEDURE:

For Transmitter section:

1. Set the remote IP.
2. Connect the remote PC having the receiver in wireless mode.
3. Compute the public key and private key using the given prime numbers.
4. Encrypt the plain text to obtain the ASCII value and the Cipher Data.
5. Send the Cipher Data.

For Receiver section:

1. Receiver receives the Cipher Data.
2. Decrypt the Cipher Data using the private key.
3. Recover the plain text message.

OBSERVATION:

For Transmitter section:

Remote IP	:
Prime No. (P)	:
Prime No. (Q)	:
N (P*Q)	:
PHIE [(P-1)*(Q-1)]	:
Public exponent 'E'	:
Secret Exponent 'D'	:
Public Key (N, E)	:
Private Key (N, D)	:
Plain Text	:
ASCII value	:
Cipher Data	:

For Receiver Section:

Local IP	:
Private Key (N, D)	:
Cipher Data	:
Decrypt	:
Plain Text	:

CONCLUSION:

Ex. No. :

Date:_____

IMPLEMENTATION OF RC 4 ALGORITHM (WIRELESS MODE)

OBJECTIVE:

To implement RC 4 Algorithm and study its performance.

THEORY:

The main advantages of RC 4 algorithm in prcatical applications are considered to be in its impressive speed and simplicity.

RC4 generates a pseudorandom stream of bits (a keystream) which, for encryption, is combined with the plaintext using XOR; decryption is performed in the same way.

To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

1. A permutation of all 256 possible bytes (denoted "S" below).
2. Two 8-bit index-pointers (denoted "i" and "j").

The permutation is initialized with a variable length key, typically between 40 and 256 bits, using the *key-scheduling* algorithm (KSA). Once this has been completed, the stream of bits is generated using the *pseudo-random generation algorithm* (PRGA).

The Key-Scheduling Algorithm (KSA):

The key-scheduling algorithm is used to initialize the permutation in the array "S". "keylength" is defined as the number of bytes in the key and can be in the range $1 \leq$ keylength ≤ 256, typically between 5 and 16, corresponding to a key length of 40 – 128 bits. First, the array "S" is initialized to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA algorithm, but also mixes in bytes of the key at the same time.

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap(S[i],S[j])
endfor
```

The Pseudo-Random Generation Algorithm (PRGA):

The lookup stage of RC4. The output byte is selected by looking up the values of S(i) and S(j), adding them together modulo 256, and then looking up the sum in S; S(S(i) + S(j)) is used as a byte of the key stream, K.

For as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA increments *i*, adds the value of S pointed to by *i* to *j*, exchanges the values of S[*i*] and S[*j*], and then outputs the value of S at the location S[i] + S[j] (modulo 256). Each value of S is swapped at least once every 256 iterations.

```
i := 0
j := 0
while (True)
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap(S[i],S[j])
    output S[(S[i] + S[j]) mod 256]
endwhile
```

Implementation:

Many stream ciphers are based on linear feedback shift registers (LFSRs), which while efficient in hardware are less so in software. The design of RC4 avoids the use of LFSRs, and is ideal for software implementation, as it requires only byte manipulations. It uses 256 bytes of memory for the state array, S[0] through S[255], k bytes of memory for the key, key[0] through key[k-1], and integer variables, i, j, and k. Performing a *modulus 256* can be done with a bitwise AND with 255 (or on most platforms, simple addition of bytes ignoring overflow). All information transmission is protected by a password.

PROCEDURE:

For Transmitter section:

1. Set the remote IP.
2. Connect the remote PC having the receiver in wireless mode.
3. Set the Password.
4. Encrypt the plain text to obtain the ASCII value and the Cipher Data.
5. Send the Cipher Data.

For Receiver section:

1. Receiver receives the Cipher Data.
2. Decrypt the Cipher Data using the password.
3. Recover the plain text message.

OBSERVATION:

For Transmitter Section:

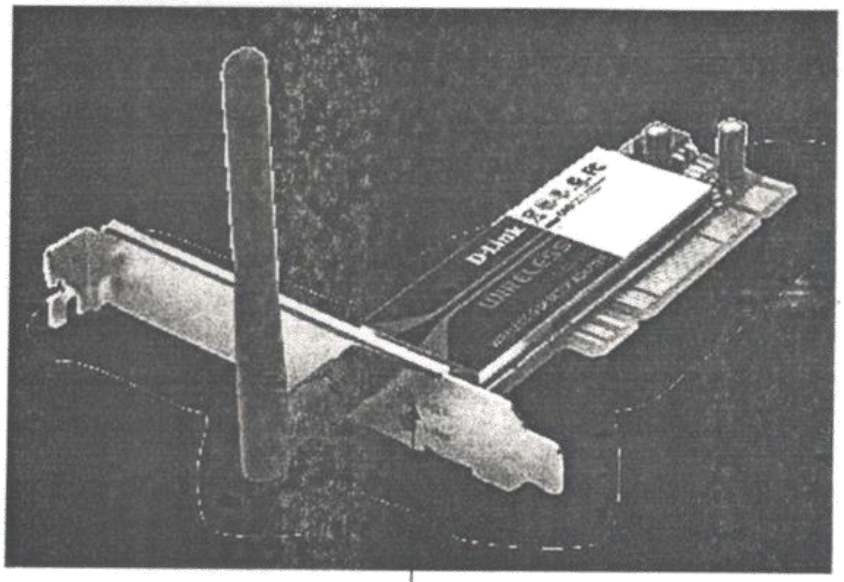
Remote IP	:
Plain Text	:
Password	:
Plain Text byte(ASC)	
XOR key byte (M)	:
Cipher Text (Chr M)	:

For Receiver Section:

Local IP	:
Password	:
Cipher Text	:
Cipher Text byte (ASC) XOR key byte (N)	:
Clear Text (Chr N)	:

CONCLUSION:

D-Link DWA-510 Wireless G Desktop Adapter



Hardware Overview:

LEDs:

1. **ACT:** A blinking light indicates transmission of data.
2. **Link:** A steady light indicates a connection to a wireless network.

System Requirements:

1. A desktop computer with an available PCI slot
2. Windows® 2000, XP, 98SE, Me or Vista
3. 300MHz processor and at least 64MB of RAM
4. An 802.11g or 802.11b access point (*for Infrastructure mode*), or another 802.11g or 802.11b wireless adapter

Introduction:

The D-Link Wireless G DWA-510 Desktop adapter features the very latest in advanced wireless silicon chip technology to deliver a maximum wireless signal rate of up to 54Mbps* in the 2.4GHz frequency. The DWA-510 also works with 802.11b standard wireless devices and when used with other D-Link Wireless G products delivers throughput speeds capable of handling heavy data payloads.

The DWA-510 also includes a configuration utility to discover available wireless networks and create and save detailed connectivity profiles for those networks most often accessed.

The DWA-510 is a powerful 32-bit Desktop adapter that installs quickly and easily into desktop PCs and when used with other D-Link Wireless G products automatically connect to the network.
Like all D-Link wireless adapters, the DWA-510 can be used in Ad-Hoc mode to connect directly with other 2.4GHz wireless computers for peer-to-peer file sharing or in Infrastructure mode to connect with a wireless access point or wireless router for access to the Internet in your office or home network. The DWA-510 is an ideal solution enabling wireless networking capabilities on desktops PCs for the home or office.

Features:

1. Faster Wireless Networking - Faster data transfers mean increased productivity. With the DWA-510 in your desktop PC, you will have the flexibility of wireless networking speeds that save you time and money.
2. Compatible with 802.11b and 802.11g Devices - Fully compatible with the IEEE 802.11b and 802.11g standards, the DWA-510 can connect with existing 802.11b or 802.11g compliant routers, access points and cards. That means you can still communicate with colleagues and friends while you have the ability to link to even more wireless networks.
3. 32-bit PCI Performance/Plug & Play Connectivity - The DWA-510 is a powerful 32-bit PCI adapter that installs quickly and easily into desktop PCs, and when used with other D-Link Wireless G products will automatically connect to the network out of the box.
4. User-friendly configuration and diagnostic utilities.

Ex. No.4 To Transmit and Receive strings of Data / Messages through MQTT Protocol using HiveMQ and MQTTBox

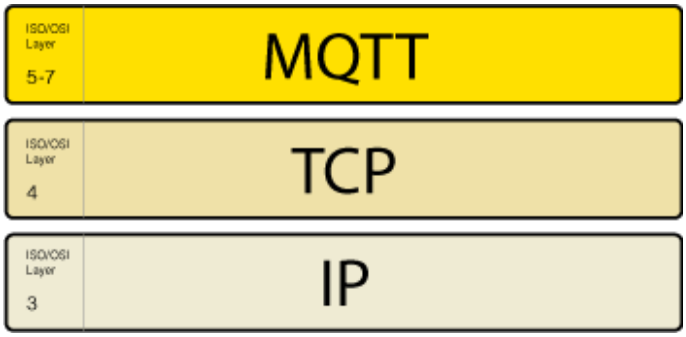
Introduction

MQTT stands for Message Queuing Telemetry Transport.MQTT was invented by Dr Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom (now Eurotech), in 1999. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

Theory

The protocol uses a publish/subscribe architecture in contrast to HTTP with its request/response paradigm. Publish/Subscribe is event-driven and enables messages to be pushed to clients. The central communication point is the MQTT broker, it is in charge of dispatching all messages between the senders and the rightful receivers. Each client that publishes a message to the broker, includes a topic into the message. The topic is the routing information for the broker. Each client that wants to receive messages subscribes to a certain topic and the broker delivers all messages with the matching topic to the client. Therefore, the clients don't have to know each other, they only communicate over the topic. This architecture enables highly scalable solutions without dependencies between the data producers and the data consumers.

The difference to HTTP is that a client doesn't have to pull the information it needs, but the broker pushes the information to the client, in the case there is something new. Therefore, each MQTT client has a permanently open TCP connection to the broker. If this connection is interrupted by any circumstances, the MQTT broker can buffer all messages and send them to the client when it is back online. As mentioned before the central concept in MQTT to dispatch messages are topics.



Because MQTT decouples the publisher from the subscriber, client connections are always handled by a broker.

Client

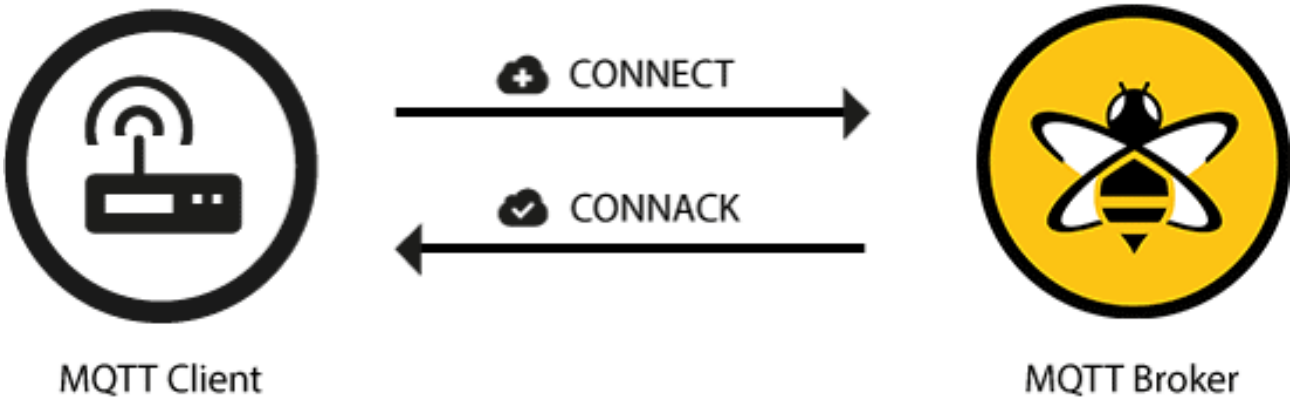
MQTT client: Both publishers and subscribers are MQTT clients. The publisher and subscriber labels refer to whether the client is currently publishing messages or subscribing to messages (publish and subscribe functionality can also be implemented in the same MQTT client). An MQTT client is any device (from a micro controller up to a full-fledged server) that runs an MQTT library and connects to an MQTT broker over a network. For example, the MQTT client can be a very small, resource-constrained device that connects over a wireless network and has a bare-minimum library. The MQTT client can also

be a typical computer running a graphical MQTT client for testing purposes. Basically, any device that speaks MQTT over a TCP/IP stack can be called an MQTT client. The client implementation of the MQTT protocol is very straight forward and streamlined. The ease of implementation is one of the reasons why MQTT is ideally suited for small devices. MQTT client libraries are available for a huge variety of programming languages. For example, Android, Arduino, C, C++, C#, Go, iOS, Java, JavaScript, and .NET.

Broker

The counterpart of the MQTT client is the MQTT broker. The broker is at the heart of any publish/subscribe protocol. Depending on the implementation, a broker can handle up to thousands of concurrently connected MQTT clients. The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients. The broker also holds the sessions of all persisted clients, including subscriptions and missed messages (more details). Another responsibility of the broker is the authentication and authorization of clients. Usually, the broker is extensible, which facilitates custom authentication, authorization, and integration into backend systems. Integration is particularly important because the broker is frequently the component that is directly exposed on the internet, handles a lot of clients, and needs to pass messages to downstream analyzing and processing systems. In brief, the broker is the central hub through which every message must pass.

Here we use HiveMQ as a broker and MQTTBox to set-up and manage clients.



MQTT Client Settings:

When creating new MQTT client from MQTTBox app, there are wide range of connection settings one can specify. Most of the settings are set by default to most used values, however they are customisable and the terms are explained below:

1. MQTT Client Name:

Name to identify MQTT client and display on dashboard. It can be any string value.

e.g.: client_test_1

2. Client Id:

The client identifier is an identifier of each MQTT client connecting to a MQTT broker. It should be unique per client for given broker. The broker uses it for identifying the client and the current state of the client. It is auto generated by default. Trying to connect two MQTT clients with same client identifier, connection will be rejected by broker. If 2 instances of MQTTBox app are open, it must be made sure they have unique client id's otherwise the clients will be rejected by broker and may show offline.

e.g.: client_id_1

3. Append timestamp to MQTT client id?

If checked, timestamp will be appended to client Id. This is enabled by default. Many a times, users open multiple instances of MQTTBox app on same or different machines with same MQTT client settings including same client id unknowingly. This causes both client connections to be rejected by broker because of same client id. If this option is enabled, MQTTBox will append timestamp to every client id making it almost unique which helps save time debugging unnecessary issues.

4. Broker is MQTT v3.1.1 compliant?

If you are connecting to a MQTT broker that only supports older versions of MQTT protocol 3.1 or less (v3.1.1 is latest and current standard), you should un-check this option. By default, it is checked and assumes MQTT broker is compliant with current MQTT standard v3.1.1 (or higher).

5. Protocol:

Network protocol used by MQTT client to connect with MQTT broker.

MQTTBox supports TCP, SSL/TLS, MQTT, MQTTS, WebSockets (WS) and Secure WebSockets (WSS). Depending on the platform, all Protocol may not be supported because of platform limitations. (For this experiment, TCP/IP will be used.)

6. Host:

MQTT host to connect. One needs to specify right host and port number depending on MQTT connection protocol selected. MQTT client may not get connected if wrong port number is mentioned or port numbers are interchanged.

e.g.: test.mosquitto.org:8080

7. Clean Session?

The clean session flag indicates the broker whether the client wants to establish a persistent session or not. If you need a persistent session, meaning, that the broker will store all subscriptions for the client and also all missed messages, when subscribing with Quality of Service (QoS) 1 or 2, un-check/No this option. By default, this is checked or Yes, which means broker will start new session, won't store anything for the client and will also purge all information from a previous persistent session.

8. Auto connect on app launch?

If you check Yes, this option, client will try to connect to broker automatically when MQTTBox app is launched and can be in "Connected", or "Connection Error" state depending on if client was connected to broker or not. If un-checked/No this option, client will be in "Not connected" state and you need to manually connect client to broker.

9. Username:

Username required by broker, if any. MQTT allows to send username for authenticating and authorization of client.

10. Password:

Password required by broker, if any. MQTT allows to send password for authenticating and authorization of client. Please note, all passwords are saved in plain text. Make sure you never save production passwords inside MQTTBox app. In fact, all fields are saved as plain text, make sure you never save any sensitive information/settings inside MQTTBox app.

11. Reschedule Pings?

If checked/yes, reschedules ping messages after sending packets.

12. Queue outgoing QoS zero messages:

If checked yes, if connection is broken between client and broker, client queue's outgoing QoS zero messages. All these messages will be published when connection is established.

13. Reconnect Period (in milliseconds):

Interval between two reconnections

14. Connect Timeout (in milliseconds):

Time to wait before a CONNACK is received

15. KeepAlive (in seconds):

The keep alive is a time interval, the clients commits to by sending regular PING Request messages to the broker. The broker response with PING Response and this mechanism will allow both sides to determine if the other one is still alive and reachable. By default, it is set to 10 seconds. Set to 0 to disable.

16. Will Settings:

The will message is part of the last will of MQTT Client. It allows to notify other clients, when a client disconnects ungracefully. A connecting client will provide his will in form of an MQTT message and topic in the CONNECT message. If the client gets disconnected ungracefully, the broker sends this message on behalf of the client automatically.

16.1 Will Settings - Topic: The topic to publish will payload. A simple string, which can have hierarchically structured with forward slashes as delimiters.

e.g.: topic_test_1 or home/kitchen/humidity

16.2 Will Settings - QoS: Publish payload with QoS set. By default, it is 0.

16.3 Will Settings - Retain: Retain flag for will payload.

16.4 Will Settings - Payload: The message to publish when the client disconnects badly.

Publisher Settings:

1. Topic to publish: is the topic to publish to. A simple string, which can have hierarchically structured with forward slashes as delimiters.

e.g.: topic_test_1 or home/kitchen/humidity

2. QoS: A Quality of Service Level (QoS) for this message. The level (0,1 or 2) determines the guarantee of a message delivered.

3. Retain: This flag determines if the message will be saved by the broker for the specified topic as last known good value. New clients that subscribe to that topic will receive the last retained message on that topic instantly after subscribing.

To delete retain message of topic, send a retained message with a zero byte payload on that topic.

4. Payload: This is the actual content of the message to publish to topic.

Subscriber settings:

Topic: is a String topic to subscribe to. You can specify Single Level(+) and Multilevel(#) subscription to topics.

e.g.: topic_test_1 or home/+/humidity or home/#

QoS: A Quality of Service Level (QoS) for this message. The level (0,1 or 2) determines the guarantee of a message delivered.

For further reading:

https://www.hivemq.com/docs/4.1/hivemq/introduction.html

Software Requirements

Although the hardware requirements for this does not extend beyond a simple PC running Windows or Linux and an internet connection, there are a few software requirements to get started.

1. Environment: Java - OpenJDK JRE/JDK 11 or newer is required.

2. MQTT Broker: HiveMQ is being used in this case. (Some alternatives are Mosquitto, Mosca, emqttd, Python Test Broker, VerneMQ)

3. MQTTBox: This helps create MQTT clients to publish or subscript topics, create MQTT virtual device networks, load test MQTT devices or brokers and offers some additional features.

Setting up and testing:

1. ***Java Runtime Environment:*** Download and install a compatible version of JRE
2. ***HiveMQ broker:***
   2.1. Download the HiveMQ zip file from *https://www.hivemq.com/downloads/download-hivemq/*
   2.2. Unzip the file
   2.3. Open the "*config.xml*" file in the directory ~*hivemq-4.1.0\hivemq-4.1.0\conf* (using any standard text editors)
   2.4. Change the IP under "bind-address" from <0.0.0.0> to the IP of your computer
   2.5. Go to ~*hivemq-4.1.0\hivemq-4.1.0\bin* and run the "*run.bat*" as administrator

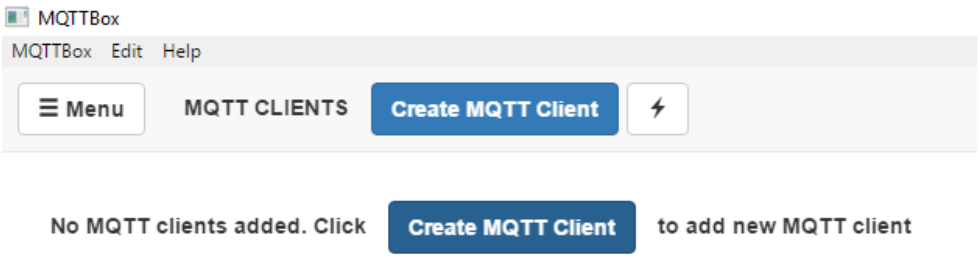
```
<?xml version="1.0"?>
<hivemq>

    <listeners>
        <tcp-listener>
            <port>1883</port>
            <bind-address>0.0.0.0</bind-address>
        </tcp-listener>
    </listeners>

    <anonymous-usage-statistics>
        <enabled>true</enabled>
    </anonymous-usage-statistics>

</hivemq>
```

3. ***MQTTBox app:***
   3.1. It can be installed from the Windows Store itself. Alternatively one can download the installer file from their website *http://workswithweb.com/html/mqttbox/downloads.html* which comes with instructions to install for non-Windows systems as well.
   3.2. Once installed, launch the app and click on "Create MQTT Client" inside the app and follow the steps given below for a basic setup with minimal customization:



3.2.1. Fill in a name under “MQTT Client Name”
3.2.2. Change the protocol to “mqtt/tcp” from the drop-down menu
3.2.3. Fill in a topic name under “Will-Topic” (optional)
3.2.4. Change the “Host” to the IP of your machine (or in the case of multiple machines, the machine acting as “publisher” will put their own IP and the machines acting as “subscribers” will put the IP of the “publisher”) and click on save

3.3. Once a MQTT client is created, two boxes will appear one for the publisher and one for the subscriber.

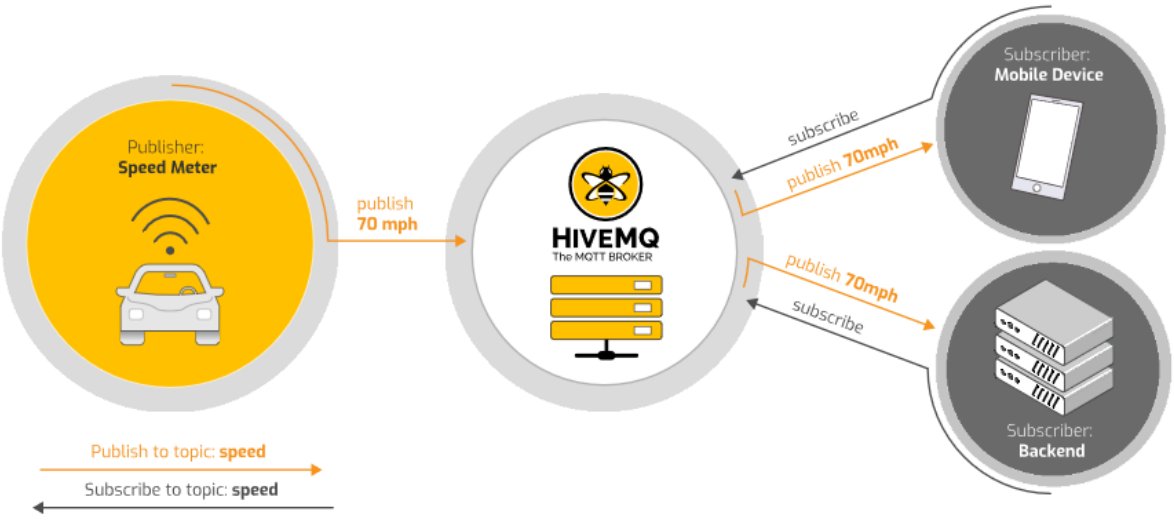
3.4. Publisher setup:

3.4.1. Give a topic name in the box designated to “topic to publish”
3.4.2. Check the “retain” box
3.4.3. Under “Payload” type the message you wish to send
3.4.4. Before hitting “publish” make sure at least one person is subscribed to the same topic as you are publishing in order to verify that it is working

3.5. Subscriber setup:

3.5.1. Under “topic” fill in the topic of interest and set the corresponding QoS accordingly and click on “Subscribe”.
3.5.2. You may type a message from the publisher window and check if it is working now.

An example flow diagram:



Observations:

Conclusions:

Assignments:

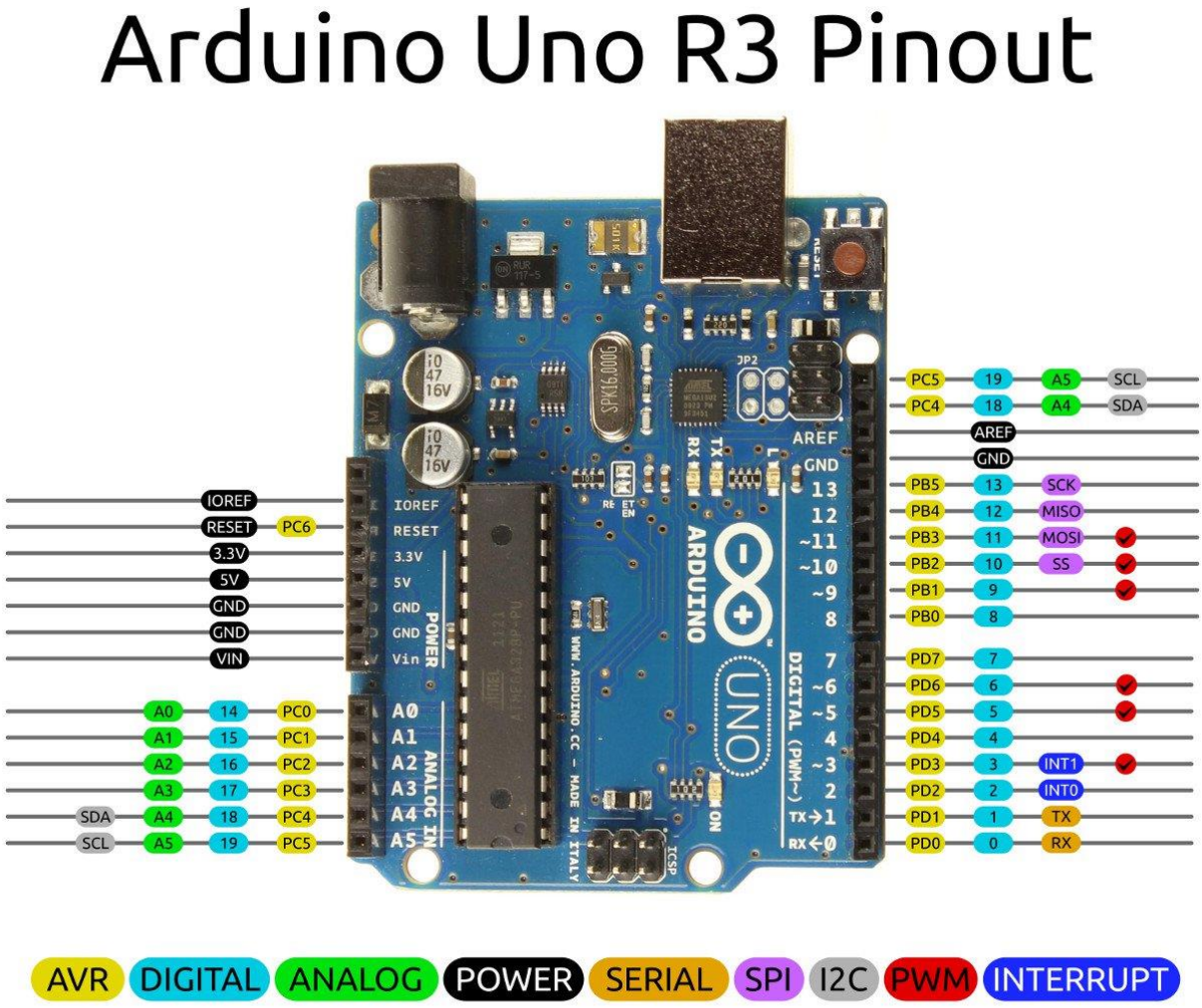
1. **Publish DHT sensor data using Arduino Uno through MQTT.**
2. **Publish Ultrasonic sensor data using Node MCU through MQTT.**

Ex. No. 5 Sending SMS using GSM module and Arduino Uno Board

Introduction

Arduino:

The Arduino UNO is an open-source microcontroller board based on the Microchip ATmega328P microcontroller and developed by Arduino.cc. The board is equipped with sets of digital and analog input/output (I/O) pins that may be interfaced to various expansion boards (shields) and other circuits. The board has 14 digital pins, 6 analog pins, and programmable with the Arduino IDE (Integrated Development Environment) via a type B USB cable. It can be powered by the USB cable or by an external 9-volt battery, though it accepts voltages between 7 and 20 volts.



2014 by Bouni
Photo by Arduino.cc

GSM module:

SIM900A is a miniature cellular module which allows for GPRS transmission, sending and receiving SMS and making and receiving voice calls. Low cost and small footprint and quad band frequency support make this module perfect solution for any project that require long range connectivity. After connecting power module boots up, searches for cellular network and login automatically. On board LED displays connection state (no network coverage - fast blinking, logged in - slow blinking).

NOTE: Power consumption peeks up to 2A. Maximum voltage on UART in this module is 2.8V. Higher voltage will kill the module. So, an external power supply is provided using an AC adapter.

Note: This may not be identical to the board in the laboratory. Only the TX, RX and Gnd need to be connected to the Arduino. Power is to be supplied externally.

Hardware requirements:

1. **Arduino Uno board**
2. **GSM Module**
3. **Jumper wires**
4. **SIM card (adapters too in case the SIM card is a micro or nano type)**
5. **Power adapter for the GSM module**

Software requirements:

1. **Arduino IDE**

Setup:

Hardware connections:

1. Connect the Arduino to the computer using the connector cable.
2. Without connecting the Arduino and GSM module, insert the SIM card in the GSM module.
3. Connect the GSM module to power source using the power adapter, and power up the device. (The connection can be checked via the led as well as calling the number whose SIM card is inserted in the module.)
4. Open Arduino IDE and write the required program, select the correct board (Arduino Genuino Uno in this case) and port (COM5/COM6/whichever is applicable) under "Tools" menu.
5. Upload the sketch (code) to the Arduino.
6. Now connect the TX pin of the GSM module to the D8 pin of the Arduino board and RX pin of the GSM module to the D7 pin of the Arduino using the jumper wires. Connect the Ground pins of the two boards as well.
7. Click on the serial monitor and select the Baud rate specified in the program (9600 in this case) from the bottom right corner to view the output. Verify with the recipient's phone.
   Note:
   a. Remember to change the recipient's phone number in the Program.
   b. Adjust the antenna on the GSM module for better cellular network receptivity.
   c. If you wish to send text messages in a loop, it can be done simply by copying/cutting the code in "void setup()" and inserting the code in "void loop()".

ARDUINO Code Required For The Implementation:

```
void setup()
  {
    Serial.begin(9600);     //Set baud rate to 9600
  }
  void loop()
  {
  //You can skip the first three lines if you are sure your sim900 gsm setup is
  //perfect and there is no error in circuit connection and power requirements.
  //delay(1200);
  //Serial.print("AT");
  //delay(3000);
  Serial.println();
  Serial.println("AT+CMGF=1");  // Sets the SMS mode to text
  delay(3000);
  Serial.println();
  Serial.print("AT+CMGS=");     // Send the SMS number. To whom the message to send (ie.
                                                                the Receiver mobile #.
  Serial.print("\"+91XXXXXXXXXX\"");
  Serial.println();
  delay(3000);
  Serial.print("Hello Sir"); // SMS-Message body
  delay(4000);
  Serial.println();
  Serial.write(26);              //CTRL+Z key combination to send message
  while(1);                    // For ever control
  }
```

Observations:

Conclusions:

Assignments:

1. **Send the Roll number of any of your group members, instead of test message in the example.**
2. **Write a Program for sending an SMS, if any movement is detected by a PIR sensor using Arduino Uno.**