

Matrix1Util_28 tutorial

Table of Contents

Is This Plug-in For Me?	2
Networking Concepts	3
IP Versions	3
Protocols	3
The TCP Protocol	3
The UDP Protocol	4
Other protocols	5
Blocking and non-blocking modes	5
Network Byte Order	6
Basic Tutorial	8
Our first socket programs - An Echo Client and Server	8
Tutorial 1 : Our first socket program	8
Tutorial 2 : The server to go with it	12
Tutorial 3 : Improving the server	18
Using UDP - Expanding the Echo Server	25
Tutorial 4: A UDP Client program	25
Tutorial 5 : Adding UDP to the Server	32
Tutorial 6 : Making the Client safer	38
Glossary	42

Matrix1Util_28 tutorial

Is This Plug-in For Me?

Let's face it, there are far simpler plug-ins out there than this one, so you'll need to make the decision on your own. You already have the choice of the following:

- Multiplayer commands (the commands provided by TGC)
- Multiplayer plus commands (extras provided by TGC for FPSC support)
- Multisync (By Benjamin – there are two versions)
- DarkTCP (by SirFire)
- EZ_Serv or DBP_COMMS_EXP (by CattleRustler)

(If I've missed anyone, I apologise – I'm just pointing out some alternatives to this plug-in, not providing a).

One thing these all have in common is that they are very high level. You connect to another DBPro program/server, you send some data, you receive some data, you disconnect and that's pretty much it. What they don't do is give you the ability to connect to anything else, and they don't give you any choices in how the network transfer is done. That's where this plug-in come in.

So if you need to provide software that needs to interact with non-DBPro stuff, if you need low-level control over what's being transmitted over the network and how, if you need to connect to cross-platform stuff, then this is the plug-in you'll need to use.

Beware though – if you use this plug-in, you'll need to learn how to write networking programs properly, as unlike the other plug-ins, this one won't 'just work' unless you put the work into it too.

Good luck.

Matrix1Util_28 tutorial

Networking Concepts

The goal of this page is to provide an overview of the TCP/IP protocols provided by this library, and to give you enough understanding of them to be able to start writing your own network code.

To those of you who are already familiar with standard socket programming, this will be familiar territory, so you can either jump straight to the tutorials chapters to see the library in action, or you can advance to the section on the UDP protocol.

IP Versions

There are two versions of IP addressing out there – IPV4 (the original) and IPV6. Windows XP & Vista support both, however the majority of network devices only support the first. This plug-in also currently only supports the first.

Protocols

The TCP Protocol

TCP, the most popular network protocol on the planet, has the following properties:

- It provides connections between clients and servers. A TCP client establishes a connection with a given server, exchanges data with that server across the connection, and then terminates the connection.
- It provides reliability. When TCP sends data to the other end of the connection, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmits the data, and waits a longer amount of time. After a number of retransmissions, TCP will give up and report an error (typically between 4 and 10 minutes total time).
- It provides a sequence to the data. Each time that a packet of data is transmitted on the network, it is given a sequence number. If the packets of data are received out of order the TCP system will reorder the two packets based on these sequence numbers before passing them on to the application. If duplicate sequence numbers are received, the duplicates are automatically discarded.
- It provides flow-control. Each end of the connection tells the other exactly how much space is available within their internal buffers, guaranteeing that the sender cannot overflow the receiver's buffer.
- Finally, TCP provided a full-duplex connection. This means that an application can both send and receive data in both directions on a

Matrix1Util_28 tutorial

connection at the same time.

What all this means in practice is that when you transmit data it is guaranteed to arrive in the order it was transmitted in, and without any missing pieces. This makes it relatively easy to use compared to UDP.

It is up to you to put meaning into the data that you transmit. If you transmit a floating point number from your client, and your server is expecting an integer, the server will not generate an error, but will simply read the data provided as if it was an integer.

To deal with this, you will need to design and implement a protocol for the data that you transmit. Although this may seem difficult, it is in fact relatively easy. An example is the HTTP protocol used by Web browsers and Web servers all over the world (for those of you who are experienced with this protocol, the following explanation has been deliberately simplified).

The first piece of data passed is the initial request line, that specifies what you want to do. The most common type of request is the GET request, and the data will look a little like this:

```
GET /path/to/file/index.html HTTP/1.0
```

The line will then be terminated with a CRLF sequence (ASCII code 13, followed by ASCII code 10). This will then be followed by zero or more largely optional header lines, again terminated with the CRLF sequence. Finally, an empty line of data completes the request. The response from the server also consists of an initial header line to report the status of the response (reporting that it did or did not locate the requested data), zero or more header lines, an empty line, and finally, the requested data if the request was successful. You can find a full breakdown of the HTTP protocol if you wish at the following site: <http://www.jmarshall.com/easy/http/>

In this example the CRLF characters are used to 'close' specific parts of the protocol. Other techniques include sending an integer for the length of the following data, or sending an integer signalling the type of the following data.

You can read the tutorials (1 to 3) to see a standard TCP echo server and client being put together with a full explanation of the code. Although these pieces of code are relatively short and simple, they demonstrate most of the things that you need to know to write a TCP application.

The UDP Protocol

The UDP protocol is far simpler than the TCP protocol, and this actually makes it harder to produce code for. UDP can be described using the following points:

- It is connectionless. It allows you to transmit data to a machine that you have not connected to previously.

Matrix1Util_28 tutorial

- It is datagram based. This means that anything that you transmit is essentially treated as a record.
- It is unreliable. Anything that you transmit to another machine is not guaranteed to be delivered ... or may be delivered several times. If the packet does arrive, there is no guarantee that it will arrive after the packet transmitted before it, or before the packet following.
 - Where can packet be lost? They can be dropped by your machine if it is too busy, network routers if they are busy or the packet is too large, the destination machine if the input buffer is full or by corruption due to transmission error.
 - Large batches of information can be broken into several packets that are re-assembled at the destination machine. If any part of the whole message is missing (one packet out of potentially many) then the whole message will be dropped.

Despite the unreliability of UDP, it does have its uses. Because it doesn't carry the overhead of TCP (with its acknowledgements, sequence numbers, retransmission, long timeouts) it is fast, and is the preferred option for games.

If you need your UDP transmissions to be reliable, then you will need to keep your messages small, and add this functionality. If you need the data to be received in sequence, then you will need to add this too. In practice however, it is possible to accept reliability, and ignore the re-ordering of data when using UDP for games if care is taken.

When reading UDP data, you should always attempt to read your largest datagram/record size to ensure that you get the complete record. This is done automatically for you when you attempt to read a string or a memblock from the connection.

You can read the tutorials (4, 5 and 6) to see a standard UDP echo server and client with a full breakdown and explanation of the code.

Other protocols

Other protocols are possible (there are well over 100), but this plug-in does not currently support them as they are not generally useful for a DBPro program.

Blocking and non-blocking modes

Connections are usually created in a mode called 'blocking' mode. This means that if there is not enough data to fill your request ready to read from the connection, the read command will block (ie it will wait until either there is enough data, or until the connection is remotely closed if it is using the TCP protocol).

To counteract this, the connection can be switched to a non-blocking mode that will read as much data from the connection as it can, and then return immediately.

Matrix1Util_28 tutorial

The disadvantage of blocking connections can be removed by using several functions that allow you to check to see if data is read to be read, check if the connection has been closed (TCP only), and that there is enough room inside the write buffer to transmit data (again, TCP only). **All** of the tutorials use blocking connections in this way rather than switch to non-blocking connections, as blocking connections are easier to work with. For example, a read of a blocking connection that tells you it received zero new characters has been closed by the remote machine. For a non-blocking connection it can mean either that the remote socket is closed, or that there is no data available to read.

Generally, there is no advantage in using a non-blocking connection over a blocking connection when using this plug-in, but the option is available for those who want to use it.

Note that if you are using a non-blocking socket and attempt to read more data than is required to fulfil your request, no data will be read (ie, you attempt to read a 4 byte integer and there are only 2 bytes available to read) – the plug-in has been engineered to give you correct results without loss of data.

Network Byte Order

The network byte order is a standard applied to 2 and 4 byte integer values (signed and unsigned) to ensure that the numbers transmitted to the remote host machine have the same value before and after transmission – this is important because you may be running a server process on a Linux machine with PowerPC or Sparc chips, which have a different byte order.

The way this is usually handled is to pass each value through a standard conversion function before transmission which will transform the value into network byte order, transmit the data, and then at the receiving end to pass the received value through another function to transform the value into the local byte ordering scheme.

One other thing to consider is that there is no globally accepted standard way to transmit floating point values, however there is a standard available called XDR (eXternal Data Representation - RFC1014) which can be used instead.

Functions have been provided in the plug-in to allow your data transmissions of integers, dwords, words and floats to conform to this standard.

As you can probably figure out for yourselves, there are two circumstances where all this extra work is not necessary:

- When transmitting data to a machine where the byte order of the local and remote hosts are identical.
- When transmitting data as text.

Note that this plug-in does absolutely none of this data conversion on your behalf, with 2 exceptions:

Matrix1Util_28 tutorial

- Transmitted memblocks are preceded with their length when using TCP – this is in network byte order
- Transmitted strings when set to sized transmission are preceded by their length when using TCP – this is in network byte order

Matrix1Util_28 tutorial

Basic Tutorial

Our first socket programs - An Echo Client and Server

An echo server is a simple concept – whatever data that the program receives will be returned, like an echo in a valley. This makes it ideal for teaching the basic concepts for networking.

Tutorial 1 : Our first socket program

The first program we will write will be the TCP client program. This program will simply connect to the server, send a string, read a string back and display it, and then close the connection.

```
EchoClient01.dba
1.  #constant ECHO_PORT      7
2.  #constant SERVER_NAME   "localhost"
3.
4.  IPAddress as dword
5.  Connection as dword
6.  ReturnStr as string
7.
8.  print "Press a key to start the client"
9.  wait key
10.
11.  IPAddress = HOSTNAME TO IP( SERVER_NAME )
12.  if IPAddress = 0 then Abort("Unable to determine the IP address")
13.
14.  Connection = NEW CONNECT SOCKET( IPAddress, ECHO_PORT)
15.  if Connection = 0
16.      Abort("Unable to get a connection to the server")
17.  endif
18.
19.  if SEND SOCKET STRING( Connection, "Hello World" ) <= 0
20.      Abort("Unable to transmit")
```


Matrix1Util_28 tutorial

```
21. endif
22.
23. SHUTDOWN SOCKET SEND Connection
24.
25. ReturnStr = RECV SOCKET STRING$( Connection )
26. if SOCKET ERROR() > 0
27.     Abort("Unable to receive")
28. endif
29.
30. print "I got this back : "; ReturnStr
31.
32. DELETE SOCKET Connection
33.
34. print "Press a key to exit"
35. wait key
36. end
37.
38. function Abort(Msg as string)
39.     print "Error : "; MSG
40.     print "Error : ("; SOCKET ERROR(); ") "; SOCKET ERROR$()
41.     wait key
42.     end
43. endfunction
```

Lines 1-2.

Define a few constants for use later in the program. We are defining the port number and the machine name or hostname that we will be communicating with. Here, we are using the localhost name, which is always available for you to use to communicate with processes on your own machine.

Lines 11-12.

Retrieve the IP Address for the hostname that we want to communicate with by calling the HOSTNAME TO IP() function. We check the results to ensure that we got a valid IP Address. The function Abort() appears later in the program.

Lines 14-16

Attempt to connect to the remote machine using the IP Address we have retrieved and the port number by using the NEW CONNECT SOCKET()

Matrix1Util_28 tutorial

function. This returns a socket handle that we will use to communicate with the remote machine. We also check to ensure that the socket has connected correctly, by checking the value of the handle. It will be zero if the connection has failed.

Lines 19-21

Now we transmit the string "Hello World" to the remote machine using the `SEND SOCKET STRING()` function. This function returns the number of characters that it managed to transmit. If the value is zero or lower then we definitely know an error occurred.

Line 23

Shutting down the send side of our socket using the `SHUTDOWN SOCKET SEND` command is the standard way to tell the server that we are done. This will cause the server to detect the equivalent of an end-of-file condition on the socket connection when it attempts to read data.

Lines 25-28

Read the returned string from the connection using the `RECV SOCKET STRING$()` function. We then use the `SOCKET ERROR()` function to check to see if an error has occurred. If it has, we terminate the program. Also, note that without special handling, the program will wait at this command until a string has been received, or an error has occurred on the socket.

Lines 30-36

Here we simply display the string that we've read, use the `DELETE SOCKET` command on the connection to close the socket and free its resources, and then finish the program.

Lines 38-43

This is the `Abort` function that we will have been calling if any problems had been detected. After displaying the user message, it then retrieves the last `SOCKET ERROR()` value, and displays it with the descriptive version of the error using the `SOCKET ERROR$()` function. Note that this function does not delete the socket as was done in the main part of the code. This is because any open socket is automatically closed cleanly when the program ends.

A little more explanation is needed for the reason behind using the `SHUTDOWN SOCKET SEND` command. We could have sent a string, received the reply and then deleted the connection, but this is a bad habit that we should avoid. When you carry out the `DELETE SOCKET`, both the send and receive sides of the socket are closed immediately. This means that any data that is currently in transit from the remote host to your program will simply be lost. You should try to close the send side of your socket first, then only delete the socket once you detect the end-of-file condition while reading the responses.

One other thing that you should have noticed is that the socket handle and IP addresses are held in dword variables defined as dwords. This is the 'safe' way to hold these values, and you should try to do this, however it should still work correctly with integers also.

Matrix1Util_28 tutorial

If you enter and run this program right now, you will get an error reported:

Error : Unable to get a connection to the server

Error : (10061) Connection refused

This means that although you were trying to talk, no-one was listening. We'll fix that in tutorial 2 by writing a server process.

Matrix1Util_28 tutorial

Tutorial 2 : The server to go with it

We have the client program, but it's pretty much useless without the server program to go with it. There are a few more commands to learn, but if you managed to follow and understand the previous piece of code you should have no problems here.

There is only one new command used here, so we'll leave the explanation of it until the code breakdown.

```
EchoServer01.dba
1.  #constant ECHO_PORT      7
2.  #constant SERVER_NAME   "localhost"
3.
4.  IPAddress as dword
5.  Connection as dword
6.  EchoStr as string
7.
8.  print "Press a key to start the server"
9.  wait key
10.
11.  IPAddress = HOSTNAME TO IP( SERVER_NAME )
12.  if IPAddress = 0 then Abort("Unable to determine the IP address")
13.
14.  Connection = NEW ACCEPT SOCKET( IPAddress, ECHO_PORT )
15.  if Connection = 0
16.      Abort("Unable to get a connection to a client")
17.  endif
18.
19.  repeat
20.      EchoStr = RECV SOCKET STRING$( Connection )
21.
22.      if SOCKET ERROR() = 0
23.
24.          print "Echoing back : "; EchoStr
25.
26.          if SEND SOCKET STRING( Connection, EchoStr ) < 0
27.              Abort("Unable to transmit")
```

Matrix1Util_28 tutorial

```
28.         endif
29.
30.     else
31.
32.         if SOCKET ERROR() > 1 then Abort("Unable to receive")
33.
34.     endif
35.
36. until SOCKET ERROR() = 1
37.
38. DELETE SOCKET Connection
39.
40. print "Press a key to exit"
41. wait key
42. end
43.
44. function Abort(Msg as string)
45.     print "Error : "; MSG
46.     print "Error : ("; SOCKET ERROR(); ") "; SOCKET ERROR$()
47.     wait key
48.     end
49. endfunction
```

Lines 1-12.

You've seen these lines before in the client program.

Lines 14-17.

This is where we use the NEW ACCEPT SOCKET() function to accept an incoming connection attempt. There are several forms of this function. The one that we are using specifies the only local IP address to which a connection will be accepted, but you can remove the IP address parameter to allow connections on all interfaces (internal like localhost, or external like your Ethernet or dial-up interfaces). To keep things easy for the moment, we restrict the connection to the internal interface only. There are some drawbacks to using this command, but we will go through those later.

Note that this is another command that will cause your program to wait, until a client program connects.

Line 19.

Matrix1Util_28 tutorial

Now that we have a connected client, we will loop so that we can send back any data that the client sends to us.

Line 20.

We wait for a string to be sent by the client and then read it. We use the same `RECV SOCKET STRING$()` function that you have already seen.

Lines 22-29.

We then check to see if an error occurred using the `SOCKET ERROR()` function. If there was no error, then we send the string that we read back to the client.

Lines 30-34.

If the error occurred was not error code 1, then we abort. The reason that we check for an error code greater than 1 is that this code has been reserved to signal that the client has closed the socket and that there is no data left to read. This is the equivalent of end-of-file for TCP type sockets.

Line 36.

We loop until the end-of-file condition is detected on the client socket.

Lines 38-42.

The client has closed down their end of the socket, we have dealt with all the data, so now is the time to delete the socket that we hold by using the `DELETE SOCKET` command.

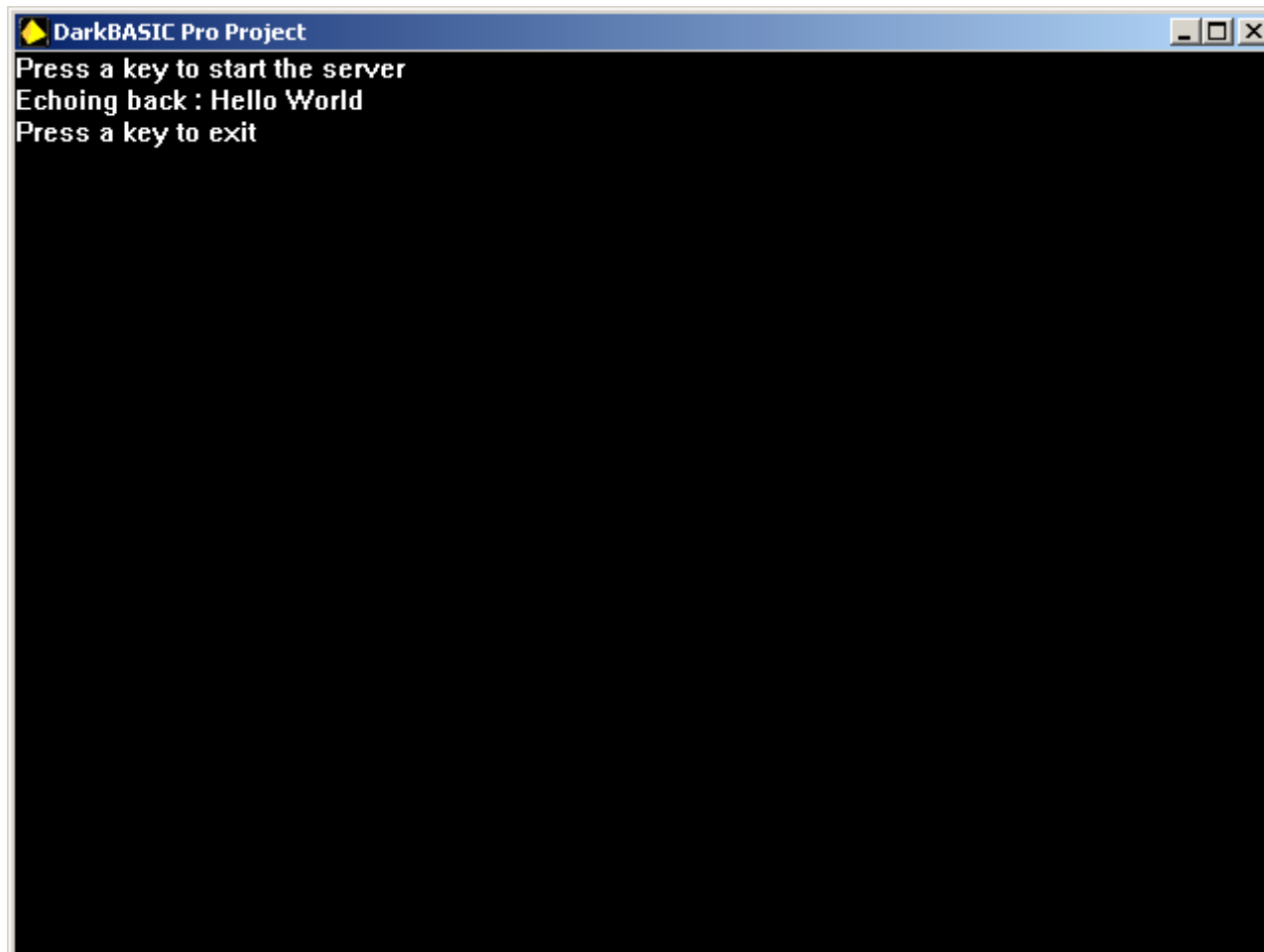
Lines 44-49.

The same `Abort()` function that you saw in the client program.

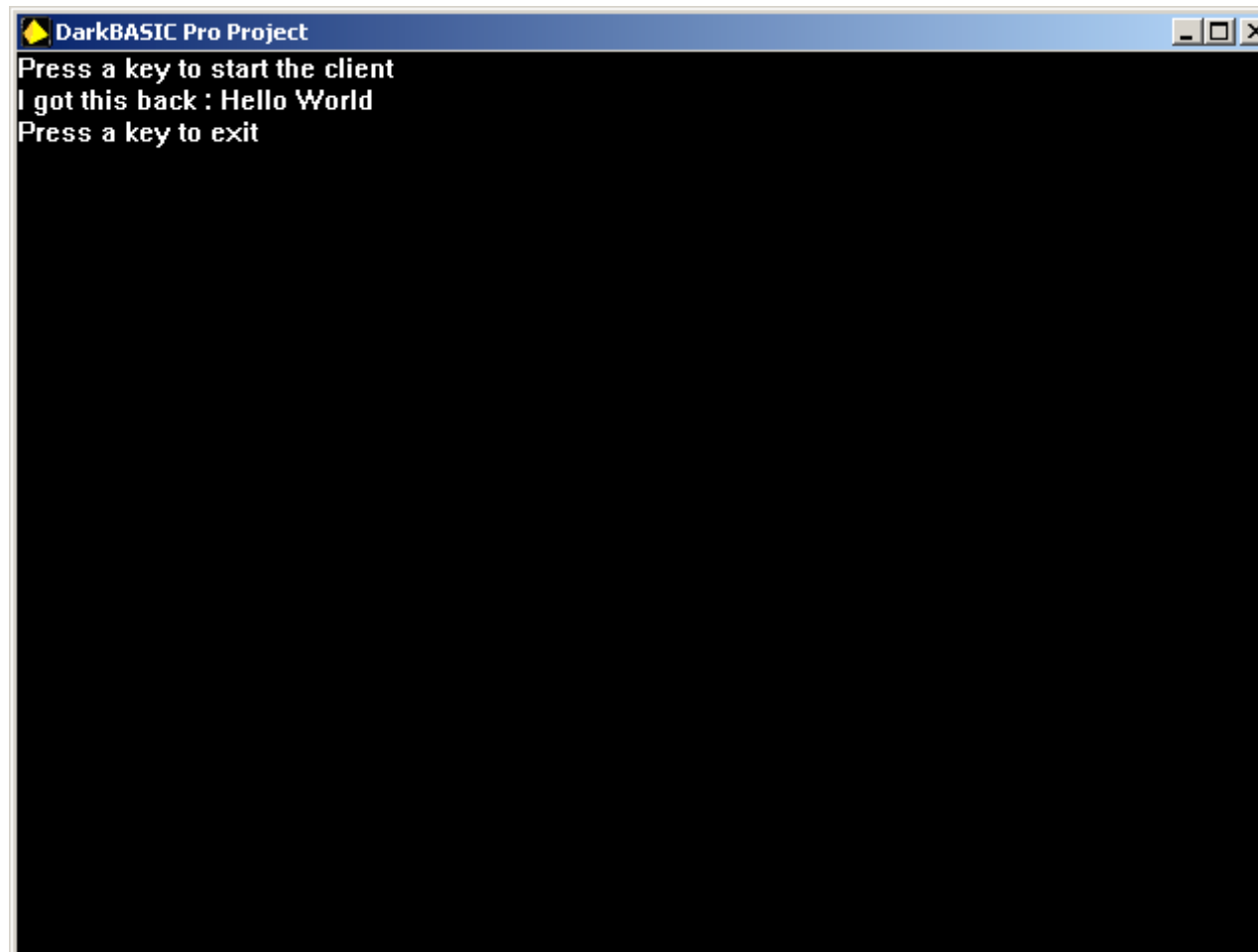
When you run this code, I suggest that you ensure that it has been compiled to run in a standard window. You should also compile the client program in the same mode.

Run both the server and the client programs and position them so that you can see both windows. Then select the server program and hit a key to start it, and then do the same with the client program. You should see the following in each of the server and client windows.

Matrix1Util_28 tutorial



Matrix1Util_28 tutorial



As it stands, there are many problems with our current server program:

- It only deals with a single client (not very good in a game environment)
- It waits for data to be sent to it before continuing (again, not very good in a game environment)

Matrix1Util_28 tutorial

- It is easy to get it to lock up by sending non-string data (good for sore losers attempting to crash your game, but not for anyone else)

We deal with each of these issues in the next tutorial chapter.

Matrix1Util_28 tutorial

Tutorial 3 : Improving the server

Things are now going to get a little more complicated with our server. The server is usually the more complex part of the networking system. It has to be able to deal with many connections at once while the client usually has only one connection. It also needs to deal with unexpected data and disconnections without failing.

Luckily for us at the moment, all we want to implement is a server that returns everything that it receives, so the data itself doesn't have to make any sense at all. We also don't need to treat the incoming data as a string. We can treat it as a sequence of bytes and store the data in a memblock instead.

This time, before we go through the code, we will cover some of the new ideas, commands and functions that we are going to use.

Firstly, we need to be able accept incoming connections without sitting there and waiting for them. To do this, we will create a listening socket by using the `NEW LISTEN SOCKET()` function. A listening socket is simply a socket that sits and waits for new connections. It does not allow communication with a connecting socket, but it does allow you to use it to create a new socket that will do that by using the `ACCEPT SOCKET()` function.

We can detect a new connection using the `SOCKET POLL READ()` function. This function returns a 1 when the socket checked either has an incoming connection ready to be accepted (for a listening socket) or when data is ready to read (for a data connection). For the listening socket, this is good enough. Every time that we receive a new connection we will accept it so that we can communicate with it when needed.

Every once in a while, we will scan the open sockets for any incoming data or to see if the remote clients have closed them. When we get data we will read directly into a buffer using the `RECV SOCKET()` function. There is a potential problem here though. One of the arguments for this function is the number of characters we want to read. If we set this number to a value that is higher than the number of characters currently available, the function will wait until there is enough data to fill our request, or the remote client closes the connection. Luckily, there is another function that we can use called `SOCKET BYTES AVAILABLE()` that we can use to find out how much data is actually available to read.

We also want to detect when an error has occurred on a socket, and when the client signals an end-of-file condition. This is a little beyond the `SOCKET POLL READ()` function that we covered earlier. Instead, we will use the `SOCKET POLL()` function, which is a little harder to use, but is far more flexible. When an error or end-of-file is detected for a socket, we will delete the socket, and remove it from the array.

The only part not covered at this point is where all of the socket handles for the communication are going to be held, and how to check each of them in turn to ensure that each gets its data returned to it. This is where the `PERFORM CHECKLIST FOR SOCKETS` command comes in. This will allow you to get a list of the sockets available, and use the standard checklist functions to identify the type of socket (listening and data sockets) so that the appropriate action can be taken against each one.

Matrix1Util_28 tutorial

Here is the code that we will be using for our improved server.

```
EchoServer02.dba
1.  #constant ECHO_PORT      7
2.  #constant SERVER_NAME    "localhost"
3.
4.  IPAddress as dword
5.  Listener as dword
6.
7.  print "Initialising the server"
8.
9.  IPAddress = HOSTNAME TO IP( SERVER_NAME )
10. if IPAddress = 0 then Abort("Unable to determine the IP address")
11.
12. Listener = NEW LISTEN SOCKET( IPAddress, ECHO_PORT )
13. if Listener = 0 then Abort("Unable to create a new listening socket")
14.
15. make memblock 1, 1024
16.
17. print "Waiting for new connections"
18.
19. repeat
20.
21.     PERFORM CHECKLIST FOR SOCKETS
22.
23.     for i=1 to checklist quantity()
24.         select checklist value c(i)
25.             case 1
26.                 AcceptNewConnection( checklist value a(i) )
27.             endcase
28.             case 2
29.                 ProcessClientConnection( checklist value a(i) )
30.             endcase
31.         endselect
32.     next i
33.
```

Matrix1Util_28 tutorial

```
34.      wait 1
35.
36. until spacekey() > 0
37.
38. delete memblock 1
39.
40. print "Closing all sockets"
41. PERFORM CHECKLIST FOR SOCKETS
42. for i=1 to checklist quantity()
43.     DELETE SOCKET checklist value a(i)
44. next i
45.
46. print "Done - Press a key to exit"
47.
48. while scancode() > 0
49. endwhile
50.
51. wait key
52. end
53.
54.
55. function AcceptNewConnection( Listener as dword )
56.     local NewClient as dword = 0
57.
58.     if SOCKET POLL READ( Listener ) > 0
59.         NewClient = ACCEPT SOCKET( Listener )
60.         if NewClient <> 0
61.             print "A new connection has been accepted from "; SocketID( NewClient )
62.         endif
63.     endif
64.
65. endfunction
66.
67.
68. function ProcessClientConnection( Socket as dword )
69.     local Status as dword
```

Matrix1Util_28 tutorial

```
70.
71.     Status = SOCKET POLL( Socket )
72.
73.     if (Status && 1) <> 0
74.
75.         print "Data received on socket "; SocketID( Socket )
76.
77.         BytesAvailable = SOCKET BYTES AVAILABLE( Socket )
78.         if BytesAvailable > get memblock size(1) then BytesAvailable = get memblock size(1)
79.
80.         BytesRead = RECV SOCKET( Socket, get memblock ptr(1), BytesAvailable )
81.         BytesWritten = SEND SOCKET( Socket, get memblock ptr(1), BytesRead )
82.
83.     endif
84.     if (Status && 4)
85.
86.         ReportError("Oops - an error occurred on socket " + SocketID( Socket ) )
87.         DELETE SOCKET Socket
88.
89.     endif
90.     if (Status && 8)
91.
92.         print "Socket " + SocketID( Socket ) + " closed by client"
93.         DELETE SOCKET Socket
94.
95.     endif
96.
97. endfunction
98.
99.
100. function SocketID( Socket as dword )
101.     local IPAddress as dword
102.     local PortNo as dword
103.     local Result as string
104.
105.     IPAddress = SOCKET REMOTE IP( Socket )
```

Matrix1Util_28 tutorial

```
106.      PortNo = SOCKET_REMOTE_PORT( Socket )
107.
108.      Result = IP_TO_STRING$( IPAddress ) + ":" + str$( PortNo )
109. endfunction Result
110.
111.
112. function Abort(Msg as string)
113.     ReportError( MSG )
114.     wait key
115. end
116. endfunction
117.
118.
119. function ReportError(MSG as string)
120.     print "Error : "; MSG
121.     print "Error : ("; SOCKET_ERROR(); ") "; SOCKET_ERROR$()
122. endfunction
```

Lines 1-2.

Define the hostname and port number that the server is available on.

Lines 9-10.

Get the IP Address for the hostname.

Lines 12-13.

Create a listening socket on port 7.

Line 15.

Create a memblock to use as the read/write buffer.

Line 19-36.

Loop until the spacebar is pressed.

Line 21.

Build the checklist of all currently open sockets.

Matrix1Util_28 tutorial

Lines 23-32.

Process each socket within the checklist in turn.

Lines 25-29.

This select statement checks the usage of each socket (i.e. whether it is a listening socket or a data socket).

If the value in checklist value c is 1, then this is a listening socket. We check for a new incoming connection, using a function called AcceptNewConnection() that will be described later.

If the value is 2 instead, then this is a data socket. We call the function ProcessClientConnection() to deal with the actual data transfers. Again, this function will be described later.

Lines 38-44.

For cleanup, the memblock that we created earlier is deleted, all sockets including the listening socket are closed, and then the program is ended.

Lines 55-65.

This is the function AcceptNewConnection(). It uses the SOCKET POLL READ() function to detect if any new connections have been made. If so, a data socket is created by using the ACCEPT SOCKET() function.

Lines 68-97.

This is the function ProcessClientConnection(). This one is a little more complex, so I'll break it down.

Line 71.

We read the status of the socket with the SOCKET POLL() function. The format of the number returned is a little complex, so you should definitely read the help for this command for a breakdown of how the status value is formatted.

Lines 73-83.

We test the read bit of the socket status, and if it is set, then we get the number of bytes ready for reading using the SOCKET BYTES AVAILABLE() function, limiting this number to the size of our memblock buffer. The data is read into the memblock using the RECV SOCKET() function while taking care to record the actual number of bytes read. This is important, because the socket system is not guaranteed to return the number of characters that you ask for. Finally, the data is retransmitted to the client using the SEND SOCKET() function.

Lines 84-89.

We test the error bit of the socket status. If it is set, we close the socket using the DELETE SOCKET command.

Lines 90-95.

Matrix1Util_28 tutorial

We test the end-of-file bit of the socket status. If it is set, then we are finished with this socket, and close it using the DELETE SOCKET command.

Lines 100-109.

This function isn't really needed by this program. It has been included to show you how to retrieve information about an existing socket. Using the functions SOCKET REMOTE IP() and SOCKET REMOTE PORT() we can find out where the connection has come from and uniquely identify the connection.

Lines 112-122.

The Abort() function has now been broken down into two functions. You still use the Abort() function to report an error and terminate your application, but now you can use the ReportError() function to display error messages without halting the program.

As before, you should compile and run this code in a standard window. When it is running, you can run the client program as many times as you wish to see everything working. Press the spacebar on the server program to halt it. For a slightly better demonstration of the server code, there is another program (**EchoClient02.dba**) that I have written that will connect to the server, accept input and send/retrieve the data via the server. You can end the program simply by typing 'quit'. You should attempt to run two or more of these client programs and watch how the server deals with them all.

Matrix1Util_28 tutorial

Using UDP - Expanding the Echo Server

Tutorial 4: A UDP Client program

UDP operates on a completely different level to TCP, as already explained in the Networking Concepts section of this document. There are no actual connections involved, no reliability, only sending and receiving of data. You should review the concepts of UDP prior to continuing with this tutorial and the next.

The client program that is shown below simply sends a packets of data once every second, and also checks for returned data. Note that there is no guarantee provided by UDP to deliver or receive the packets of data transmitted, or any guarantee of the order they will be received in, so we have to plan the code accordingly.

There is also another thing that needs to be taken into account - when sending a UDP packet to a machine that is running but does not have a listening socket open on the port you are transmitting to, a message is sent back to notify you of this fact. This is reported back to your code as data being received, but when you attempt to read this data, an error is reported instead. The code following needs to detect and deal with this in some way rather than just reporting the error and aborting the program as we have been doing so far.

Despite using UDP, there is only one new command that you need to learn - the NEW UDP SOCKET() function that will open the socket for you to use.

The following client program will transmit a new message to the server once per second. When a return message is received, it will be displayed immediately.

```
EchoClient03.dba
1. #constant ECHO_PORT      7
2. #constant SERVER_NAME    "127.0.0.1"
3.
4. #constant SOCK_CONNRESET 10054
5.
6. sync off
7.
8. IPAddress as dword
9. Connection as dword
```

Matrix1Util_28 tutorial

```
10. ReturnString as string
11. SendString as string
12. SendMessage as string
13.
14. IPAddress = HOSTNAME TO IP( SERVER_NAME )
15. if IPAddress = 0 then Abort("Unable to determine the IP address")
16.
17. input "    Your text -> "; SendString
18. print "Starting transmission to host"
19.
20. Connection = NEW UDP SOCKET()
21. if Connection = 0
22.     Abort("Unable to get a connection to the server")
23. endif
24.
25.
26. MyTimer = timer()
27. Count = 0
28. repeat
29.     if MyTimer - timer() <= 0
30.         MyTimer = MyTimer + 1000
31.         inc Count
32.         SendMessage = str$(Count) + "-" + SendString
33.         if SEND SOCKET STRING TO( Connection, SendMessage, IPAddress, ECHO_PORT ) = 0
34.             Abort("Unable to transmit data to the server")
35.         endif
36.     endif
37.
38.     repeat
39.         PollStatus = SOCKET POLL( Connection, 0 )
40.         if PollStatus > 0
41.             print "Poll Status: "; PollStatus
42.             ReturnString = RECV SOCKET STRING$( Connection )
43.
44.             if SOCKET ERROR() > 0
45.                 if SOCKET ERROR() <> SOCK_CONNRESET
```

Matrix1Util_28 tutorial

```
46.         Abort("Receive Error occurred")
47.     else
48.         print "There is no server running on the remote machine"
49.     endif
50. endif
51.
52.     print "Received from "; SocketID( Connection ); " -> "; ReturnString
53. endif
54.     until PollStatus = 0
55. until spacekey() = 1
56.
57. DELETE SOCKET Connection
58.
59. print "Press a key to exit"
60. wait key
61. end
62.
63. function SocketID( Socket as dword )
64.     local IPAddress as dword
65.     local PortNo as dword
66.     local Result as string
67.
68.     IPAddress = SOCKET REMOTE IP( Socket )
69.     PortNo = SOCKET REMOTE PORT( Socket )
70.
71.     Result = IP TO STRING$( IPAddress ) + ":" + str$( PortNo )
72. endfunction Result
73.
74. function Abort(Msg as string)
75.     print "Error : "; MSG
76.     print "Error : ("; SOCKET ERROR(); ") "; SOCKET ERROR$()
77.     wait key
78. end
79. endfunction
```

Lines 1-2

Matrix1Util_28 tutorial

Here we define the hostname and port number that the server is available on.

Line 4

This is the error number that will be returned if we attempt to transmit to a host that does not have a server process running.

Line 6.

Due to the nature of this client program and the fact that we may want to actually run other programs alongside it, we need to ensure that we don't hog the processor to ourselves. The SYNC OFF command accomplishes this for us simply and safely.

Lines 14-15

Get the IP Address for the hostname.

Lines 17-18

Accept a line of text that will be used as a part of the message that we transmit to the server.

Lines 20-23

Create a new UDP socket for communicating with the outside world by using the NEW UDP SOCKET() function.

Lines 26-27

Initialise a timer and a counter. This will be used to check when each second has expired, and so when to transmit the next message to the server.

Lines 28-55

Loop until the spacebar is pressed.

Lines 29-36

This section checks to see if the timer has expired. If it has, the timer is reset, the counter is increased by one, and a string message is sent to the server.

Lines 30-32

Reset the timer, increase the count and get the message ready to send.

Line 31

This is the actual line that does the transmission. The IP address of the host and the port number are provided as extra arguments to the SEND SOCKET STRING TO() function. As you can see, this function is almost identical to the SEND SOCKET STRING() function. If no data is transmitted, then an error has occurred.

Lines 38-54

Matrix1Util_28 tutorial

Loop until all received messages are dealt with.

Line 39

We use the `SOCKET POLL()` command to check the status of the socket and to see if there is anything to read. A time-out value of zero is used because we cannot afford for the program to pause for new data - we still need to transmit the outgoing message once per second.

Line 42

Read the returned message into a string. We just use the standard `RECV SOCKET STRING$()` function to read the returned string as we do when reading a TCP socket.

Line 44-50

Here is where we check for errors. If an error occurred, but it was not the 'Connection Reset' error code then we will abort in the normal way. If it was the 'Connection Reset' error code then we just display a message to this effect and allow processing to continue.

Line 52

Print the received message to the screen

Lines 57-61

Dispose of the socket using `DELETE SOCKET` and exit the program.

Lines 63-72

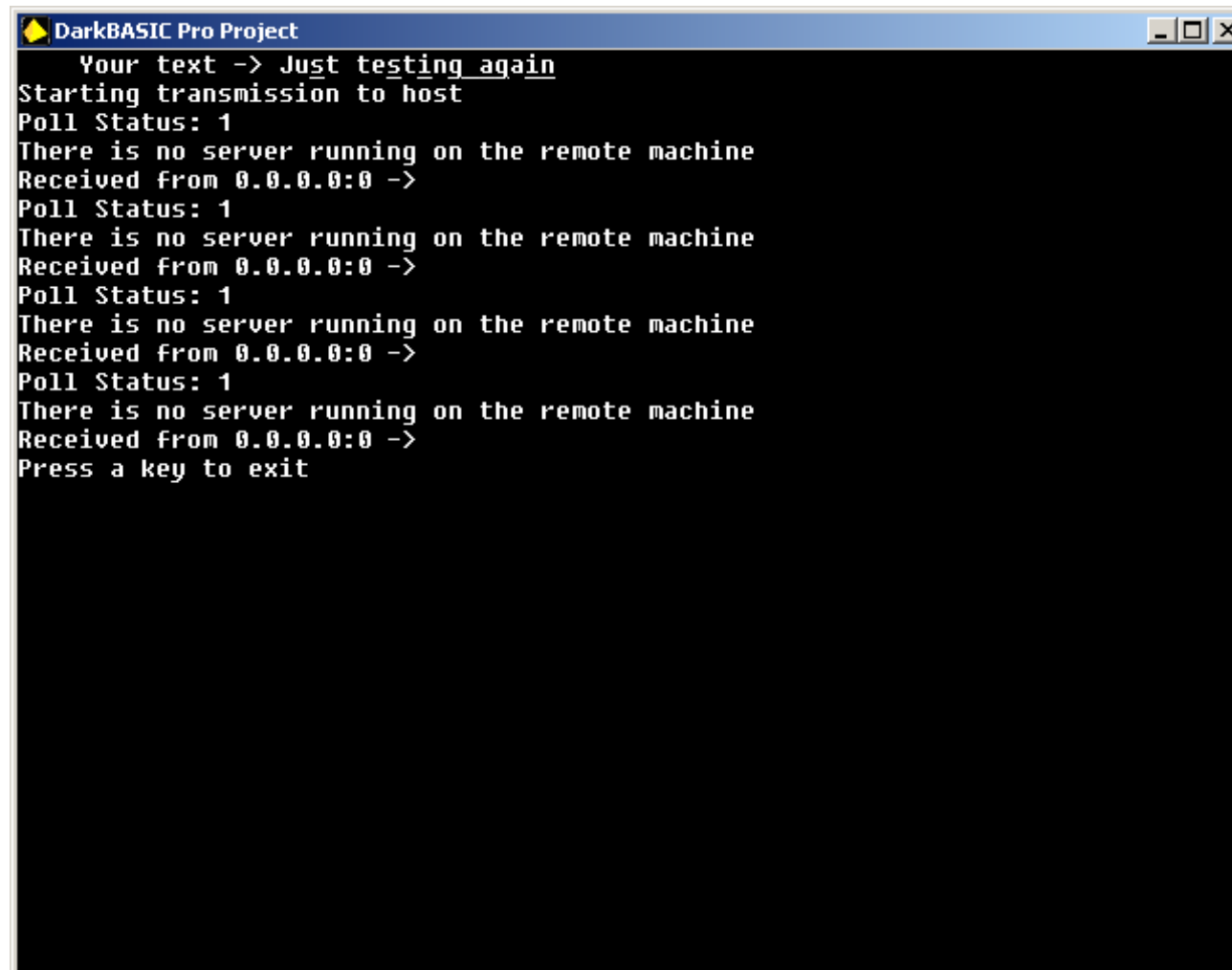
This is the function you've seen before that tells you the remote IP address and port number for the transmitted data. This works almost exactly the same for UDP sockets as it did for TCP sockets.

Lines 74-79

The standard Abort function that we have used before.

If you run this program now without running the server you will get the following reported on the screen (I pressed space to stop the program after 4 seconds):

Matrix1Util_28 tutorial



```
DarkBASIC Pro Project
Your text -> Just testing again
Starting transmission to host
Poll Status: 1
There is no server running on the remote machine
Received from 0.0.0.0:0 ->
Poll Status: 1
There is no server running on the remote machine
Received from 0.0.0.0:0 ->
Poll Status: 1
There is no server running on the remote machine
Received from 0.0.0.0:0 ->
Poll Status: 1
There is no server running on the remote machine
Received from 0.0.0.0:0 ->
Press a key to exit
```

There are several things to note about the code:

Matrix1Util_28 tutorial

- There is no connection command.

This socket can be reused to transmit and receive from anyone. This means that anyone can transmit data to your machine at any time. If you are not careful, this could cause bad information to be used by your application. You can use a connection command if you wish, but this will restrict who you can communicate with using that connection. This will be explained when we refine the client in Tutorial 6.

- It is not possible to see which machine caused a 'Connection Reset' error.

This is not a problem if the socket has only been used for transmission to a single server, as there is then only a single possibility for which server was being communicated with. The SOCKET REMOTE IP() and SOCKET REMOTE PORT() functions will only report the details of the last successfully received transmission.

The next thing that we need to do is expand the existing server program to handle UDP connections.

Matrix1Util_28 tutorial

Tutorial 5 : Adding UDP to the Server

Rather than create a new program for the UDP server, it makes more sense to expand the existing server program to handle UDP connections as well as TCP connections. The changes needed are actually quite simple to make, so the explanation of the following program will only cover the new lines of code.

```
EchoServer03.dba
1.  #constant ECHO_PORT      7
2.  #constant SERVER_NAME    "localhost"
3.
4.  IPAddress as dword
5.  Listener as dword
6.  UDPSocket as dword
7.
8.  print "Initialising the server"
9.
10.  IPAddress = HOSTNAME TO IP( SERVER_NAME )
11.  if IPAddress = 0 then Abort("Unable to determine the IP address")
12.
13.  Listener = NEW LISTEN SOCKET( IPAddress, ECHO_PORT )
14.  if Listener = 0 then Abort("Unable to create a new listening socket")
15.
16.  UDPSocket = NEW UDP SOCKET()
17.  if UDPSocket = 0 then Abort("Unable to create a new UDP socket")
18.
19.  if BIND SOCKET( UDPSocket, IPAddress, ECHO_PORT ) = 0
20.      Abort("Failed to bind")
21.  endif
22.
23.  make memblock 1, 1024
24.
25.  print "Waiting for new connections"
26.
27.  repeat
28.
```


Matrix1Util_28 tutorial

```
29.     PERFORM CHECKLIST FOR SOCKETS
30.
31.     for i=1 to checklist quantity()
32.         if checklist value b(i) = 1
33.             select checklist value c(i)
34.                 case 1
35.                     AcceptNewConnection( checklist value a(i) )
36.                 endcase
37.                 case 2
38.                     ProcessClientConnection( checklist value a(i) )
39.                 endcase
40.             endselect
41.         else
42.             ProcessUDPConnection( checklist value a(i) )
43.         endif
44.     next i
45.
46.     wait 1
47.
48. until spacekey() > 0
49.
50. delete memblock 1
51.
52. print "Closing all sockets"
53. PERFORM CHECKLIST FOR SOCKETS
54. for i=1 to checklist quantity()
55.     DELETE SOCKET checklist value a(i)
56. next i
57.
58. print "Done - Press a key to exit"
59.
60. while scancode() > 0
61. endwhile
62.
63. wait key
64. end
```

Matrix1Util_28 tutorial

```
65.  
66.  
67. function AcceptNewConnection( Listener as dword )  
68.     local NewClient as dword = 0  
69.  
70.     if SOCKET POLL READ( Listener ) > 0  
71.         NewClient = ACCEPT SOCKET( Listener )  
72.         if NewClient <> 0  
73.             print "A new connection has been accepted from "; SocketID( NewClient )  
74.         endif  
75.     endif  
76.  
77. endfunction  
78.  
79.  
80. function ProcessClientConnection( Socket as dword )  
81.     local Status as dword  
82.  
83.     Status = SOCKET POLL( Socket )  
84.  
85.     if (Status && 1) <> 0  
86.  
87.         print "Data received on socket "; SocketID( Socket )  
88.  
89.         BytesAvailable = SOCKET BYTES AVAILABLE( Socket )  
90.         if BytesAvailable > get memblock size(1) then BytesAvailable = get memblock size(1)  
91.  
92.         BytesRead = RECV SOCKET( Socket, get memblock ptr(1), BytesAvailable )  
93.         BytesWritten = SEND SOCKET( Socket, get memblock ptr(1), BytesRead )  
94.  
95.     endif  
96.     if (Status && 4)  
97.  
98.         ReportError("Oops - an error occurred on socket " + SocketID( Socket ) )  
99.         DELETE SOCKET Socket  
100.
```

Matrix1Util_28 tutorial

```
101.     endif
102.     if (Status && 8)
103.
104.         print "Socket " + SocketID( Socket ) + " closed by client"
105.         DELETE SOCKET Socket
106.
107.     endif
108.
109. endfunction
110.
111.
112. function ProcessUDPConnection( Connection as dword )
113.     local RemoteIP as dword
114.     local RemotePort as dword
115.
116.     if SOCKET POLL READ( Connection ) > 0
117.         Bytes = RECV SOCKET MEMBLOCK( Connection, 2 )
118.         if Bytes > 0
119.             print "UDP packet received from "; SocketID( Connection )
120.             RemoteIP = SOCKET REMOTE IP( Connection )
121.             RemotePort = SOCKET REMOTE PORT( Connection )
122.             Bytes = SEND SOCKET MEMBLOCK TO( Connection, 2, RemoteIP, RemotePort)
123.         endif
124.     endif
125. endfunction
126.
127.
128. function SocketID( Socket as dword )
129.     local IPAddress as dword
130.     local PortNo as dword
131.     local Result as string
132.
133.     IPAddress = SOCKET REMOTE IP( Socket )
134.     PortNo = SOCKET REMOTE PORT( Socket )
135.
136.     Result = IP TO STRING$( IPAddress ) + ":" + str$( PortNo )
```

Matrix1Util_28 tutorial

```
137. endfunction Result
138.
139.
140. function Abort(Msg as string)
141.     ReportError( MSG )
142.     wait key
143.     end
144. endfunction
145.
146.
147. function ReportError(MSG as string)
148.     print "Error : "; MSG
149.     print "Error : ("; SOCKET ERROR(); ") "; SOCKET ERROR$()
150. endfunction
```

Lines 16-17

Here is where we open the new UDP socket using the NEW UDP SOCKET() function.

Lines 19-21

We then bind the UDP socket to a specific port using the standard BIND SOCKET() function. If we didn't bind the socket to a port, the system would choose a random port for us, making a connection from a client impossible to make.

Line 31-44

Within the loop that checks each socket for new data, we add an extra check for the socket type. When it identifies a UDP socket type, the code calls a new UDP specific function (described later). Otherwise, it is dealt with as before.

Line 53-58

There are no changes to this section of code. I just wanted to point out that the UDP socket is also deleted at this point alongside all the other open sockets.

Lines 112-125

This is the ProcessUDPConnection function that will handle all UDP data transmissions.

Line 116

Check to see if new data has been received using the SOCKET POLL READ() function.

Matrix1Util_28 tutorial

Lines 117-123

Attempt to read the data into a new memblock using the standard `RECV SOCKET MEMBLOCK()` function. If the number of bytes is zero then the code will do nothing, as one of two things has happened. We were either sent an empty packet (in which case there is no data to echo), or we returned a packet to a client that is no longer running. Neither of these is life-threatening for the server, so we'll ignore it.

Lines 120-122

Transmit the data back to where it came from. We first get the clients address data using the `SOCKET REMOTE IP()` and `SOCKET REMOTE PORT()` functions and then transmit it using the `SEND SOCKET MEMBLOCK TO()` function. Again, we will ignore any errors by simply not checking for them.

That covers everything for the server. As you can see, it didn't take much to implement it. If we were running continuous sessions over UDP such as during a game, then some error detection would be required, but as this is a simple echo server, the code does everything it needs to do.

Matrix1Util_28 tutorial

Tutorial 6 : Making the Client safer

As noted previously, there is a potential problem with our UDP client and server code - the data we receive can come from anyone at any time. With the server, this may not be what we want. It is definitely not what we want for the client.

There is already one technique that you can use to check where the data has come from - the SOCKET REMOTE IP() function can be used to get the remote IP address and then it can be checked against a list of valid IP Addresses held within an array. In fact, this is a possible way that you would want to deal with this problem for the server, as you would normally need to associate the IP address with the client-specific data held on the server.

For clients though, this would just add unnecessary overhead to the code. There is another way to ensure that data only comes from the required place and that all other attempts at communication are silently ignored, and that is to use the CONNECT SOCKET() function. This allows you to specify the IP Address and port number that you are expecting communication from, but it does not actually make a connection to the remote machine as a TCP socket would do. There are not many changes required to the existing code to implement this.

```
EchoClient04.dba
1. #constant ECHO_PORT      7
2. #constant SERVER_NAME    "localhost"
3.
4. #constant SOCK_CONNRESET 10054
5.
6. sync off
7.
8. IPAddress as dword
9. Connection as dword
10. ReturnString as string
11. SendString as string
12. SendMessage as string
13.
14. IPAddress = HOSTNAME TO IP( SERVER_NAME )
15. if IPAddress = 0 then Abort("Unable to determine the IP address")
16.
17. input "    Your text -> "; SendString
18. print "Starting transmission to host"
19.
```

Matrix1Util_28 tutorial

```
20. Connection = NEW UDP SOCKET()
21. if Connection = 0
22.     Abort("Unable to get a connection to the server")
23. endif
24.
25. if CONNECT SOCKET( Connection, IPAddress, ECHO_PORT ) = 0
26.     Abort("Unable to set a permanent target for the socket")
27. endif
28.
29. MyTimer = timer()
30. Count = 0
31. repeat
32.     if timer() >= MyTimer
33.         MyTimer = MyTimer + 1000
34.         inc Count
35.         SendMessage = str$(Count) + "-" + SendString
36.         if SEND SOCKET STRING( Connection, SendMessage ) = 0
37.             Abort("Unable to transmit data to the server")
38.         endif
39.     endif
40.
41.     repeat
42.         PollStatus = SOCKET POLL( Connection, 0 )
43.         if PollStatus > 0
44.             print "Poll Status: "; PollStatus
45.             ReturnString = RECV SOCKET STRING$( Connection )
46.
47.             if SOCKET ERROR() > 0
48.                 if SOCKET ERROR() <> SOCK_CONNRESET
49.                     Abort("Receive Error occurred")
50.                 else
51.                     print "There is no server running on the remote machine"
52.                 endif
53.             endif
54.
55.             print "Received from "; SocketID( Connection ); " -> "; ReturnString
```

Matrix1Util_28 tutorial

```
56.         endif
57.         until PollStatus = 0
58. until spacekey() = 1
59.
60. DELETE SOCKET Connection
61.
62. print "Press a key to exit"
63. wait key
64. end
65.
66. function SocketID( Socket as dword )
67.     local IPAddress as dword
68.     local PortNo as dword
69.     local Result as string
70.
71.     IPAddress = SOCKET REMOTE IP( Socket )
72.     PortNo = SOCKET REMOTE PORT( Socket )
73.
74.     Result = IP TO STRING$( IPAddress ) + ":" + str$( PortNo )
75. endfunction Result
76.
77. function Abort(Msg as string)
78.     print "Error : "; MSG
79.     print "Error : ("; SOCKET ERROR(); ") "; SOCKET ERROR$()
80.     wait key
81.     end
82. endfunction
```

Again, we'll just cover the changes to the existing code.

Lines 25-27

We use the `CONNECT SOCKET()` function to set the remote IP Address and port number. Remember that this does not actually make a connection to the remote machine. It just specifies the remote IP Address and port number that will be used for all communication on this socket.

Line 36

Matrix1Util_28 tutorial

Because we have already specified the IP Address and the port number that we wish to communicate with, the SEND SOCKET STRING() is now used instead of the SEND SOCKET STRING TO().

... and that's it. This program will only send data to the specified host, and more importantly, will only receive data from that host.

The server process has not been amended in this way - this does not mean that you can't do it, but just that it isn't a suitable thing to do for this server. For your own servers, you may decide that this is a good thing to do, and create a connected UDP socket to each of your clients (e.g. for a multiplayer game, you might have a general port for new connections. Once a connection attempt has been accepted, you could create a socket specifically for that client, and pass the port number back to the client).

Matrix1Util_28 tutorial

Glossary

Echo Server

A process (usually listening on port 7) that simply returns the data that you send to it.

Hostname

The name assigned to a machine on the network

IP Address

An IP Address is an identifier for a computer or device on a TCP/IP network. It is made up using a 32 bit number, and is usually written as 4 numbers separated by periods. In this library they should be held in dword values.

Localhost

A name used to identify the current machine. This is always configured with an IP Address of 127.0.0.1 and is not available to external machines.

Listening Socket

A Socket that is configured for accepting incoming connection attempts from a remote host

Port

A port number is a 16 bit number that identifies a specific process to which an Internet or other network message is to be forwarded when it arrives at a server. In this library they can be held as either a dword or integer value.

Socket

A socket is one end-point of a two-way communication link between two programs running on the network. By connecting a socket on your machine to a socket on another machine via the network, you can pass information between the machines.

Socket handle

Socket handles are used within the plug-in library to identify each socket in use by your program.

TCP or TCP/IP

This is a connected network stream protocol. It treats the data that you transmit as a stream of characters that are guaranteed to reach the remote machine in the same sequence that they were transmitted. Because of this (and a number of other guarantees it provides), the TCP/IP protocol is usually

Matrix1Util_28 tutorial

seen as slower than UDP.

UDP or UDP/IP

This is an unconnected network datagram protocol. It treats the data that you transmit as a single chunk of data. It provides no guarantees for either delivering the data, or the sequence that each chunk of data will be received in. Because of this, it is the faster way to get data to a remote machine ... if the data makes it intact.