

maXbox



# maXbox Starter 26

## Start with Sockets

### 1.1 From Socket to Interprocess Communication

This Tutorial is based on an article by Chad Z. Hower which he wrote in the days of Indy 8.0 published on [swissdelphicenter.ch](http://www.swissdelphicenter.ch) years ago:

<http://www.swissdelphicenter.ch/en/showarticle.php?id=4>

I did also inherit a bit from the Starter 5 which deals with Internet Programming. Most of the above mentioned article from Chad still applies and is very useful for newer versions of Indy. It shows the implementation of an Indy Server which communicates with a client over TCP and Sockets. I made it scriptable to demonstrate the fascination of 2 boxes communicate with each other at the same time.

Indy uses blocking socket calls. Blocking calls are much like reading and writing to a file. When you read data, or write data, the function will not return until the operation is complete. The difference from working with files is that the call may take much longer as data may not be immediately ready for reading or writing (It can only operate as fast as the network or the modem can handle the data).

For example, to connect simply call the connect method and wait for it to return. If it succeeds, it will return when it does. If it fails, it will raise an exception.

If a maXbox script or app is programmed to assume the host standard, it is always started relative to the path where the maXbox3.exe as the host itself is:

```
playMP3(ExePath+'examples\maxbox.mp3');
```

So for example you want to run with two boxes, all your external files (we need one file below mentioned) or resources in a script will be found relative to `ExePath()`:

```
E:\Program Files\maxbox\maxbox3\examples\ZipCodes.txt'
```

In this case `ExePath` is `E:\Program Files\maxbox\maxbox3`.

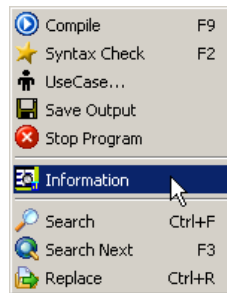
`ExePath` is a useful function where you always get the path of maXbox.

If someone tries to start (install) the script or the app to or from a different drive for space or organizational reasons, it may fail to (install) or to run the script after installation<sup>1</sup>.

---

<sup>1</sup> You don't have to install maXbox or a script anyway, just unzip or copy the file

You find all info concerning run environment of the app and script in menu  
../Program/Information/.....



When Winsock was "ported" to Windows, a problem quickly arose. In Unix it was common to fork (kind of like multi threading, but with separate processes instead of threads). Unix clients and daemons would fork processes, which would run, and use blocking sockets. Windows 3.x could not fork and did not support multi threading.

Using the blocking interface "locked" user interfaces and made programs unresponsive. So asynchronous extensions were added to WinSock to allow Windows 3.x with its shortcomings to use Winsock without "locking" the main and only thread of a program. This however required a different way of programming., and MS and others vilified blocking vehemently so as to mask the shortcomings of Windows 3.x.

In reality, blocking sockets are the ONLY way Unix does sockets. Blocking sockets also offer other advantages, and are much better for threading, security, and other aspects. Some extensions have been added for non-blocking sockets in Unix. However they work quite differently than in Windows. They also are not standard, and not in wide use. Blocking sockets under Unix still are used in almost every case, and will continue to be so.

And Blocking has more advantages:

- Easy to program - Blocking is very easy to program. All user code can exist in one place, and in a sequential order.
- Easy to port to Unix - Since Unix uses blocking sockets, portable code can be written easily. Indy uses this fact to achieve its single source solution.
- Work well in threads - Since blocking sockets are sequential they are inherently encapsulated and therefore very easily used in threads.

Hope you did already read Starters 1 till 24 at:

<http://sourceforge.net/apps/mediawiki/maxbox/>

First we start with the TCP Server and his functionality.  
You need the following file ready to download:

<http://www.softwareschule.ch/examples/tcpserversocks.zip>

After unpacking the namespace you get:

```
23.05.2013 19:54 <DIR>      examples
17.03.2013 13:14 <DIR>      web
29.07.2013 23:59          7'370 388_TCPServerSockClient.TXT
28.07.2013 00:39          3'866 388_TCPServerSockClient.uc
11.07.2013 21:04          9'464 388_TCPServerSock.TXT
11.07.2013 19:13          80'096 ZipCodes.txt
```

The first project has been designed to be as simple as possible. ZipCode Lookup will allow a client to ask a server what city and state a zip code is for.

☞ For those of you outside the US who may not know what a zip code is, a zip code is a US postal code that specifies a postal delivery area. Zip codes are numeric and 5 digits long.

The first step in building a client or server is to understand the **protocol**. For standard protocols this is done by reading the appropriate RFC. For ZipCode Lookup a protocol has been defined and is below.

☞ Most protocols are conversational and plain text. Conversational means that a command is given, a status response follows, and possibly data. Protocols that are very limited are often not conversational, but are still plain text. ZipCode Lookup is plain text, but not conversational. Plain text makes protocols much easier to debug, and also to interface to using different programming languages and operating systems.

Upon connection the server will respond with a welcome message, then accept a command. That command can be "ZipCode x" (Where x is the zip code) or "Quit". A ZipCode command will be responded to with a single line response, or an empty line if no entry exists. Quit will cause the server to disconnect the connection. The server will accept multiple commands, until a Quit command is received.

First we create and configure some 2 objects:

```
procedure TCPServerMain_Create(Sender: TObject);
begin
    ZipCodeList:= TStringList.Create;
    ZipCodeList.LoadFromFile(ExePath+'Examples\'+'SFILE');
    IdTCPServer1:= TIdTCPServer.Create(self);
    with IdTCPServer1 do begin
        defaultPort:= TCPPOrt;
        Active:= true;
        onConnect:= @TCPServer_IdTCPServer1Connect;
        OnExecute:= @TCPServer_IdTCPServer1Execute;
        Printf('Listening TCPServer on
                %s:%d.',[getIP(getHostName),Bindings[0].Port]);
        ShowmessageBig('Close OK to shutdown TCP Server listen on:
                '+intToStr(Bindings[0].Port));
        Active:= false;
        Free;
        TCPServer_Destroy(self);
    end;
    Writeln('Server Stopped at '+DateTimeToInternetStr(Now, true))
end;
```

The only parts that are Indy specific are the IdTCPServer1 component, IdTCPServer1Connect method, and the IdTCPServer1Execute method.

The String list represents a data Container as a key – value dictionary.

IdTCPServer1 is a TIdTCPServer and is a component on the form. The following properties were altered from the default:

- Active = True - Set the server to listen when the application is run.
- DefaultPort = 6000 - An arbitrary number for this demo.

This is the port the listener will listen on for incoming client requests.

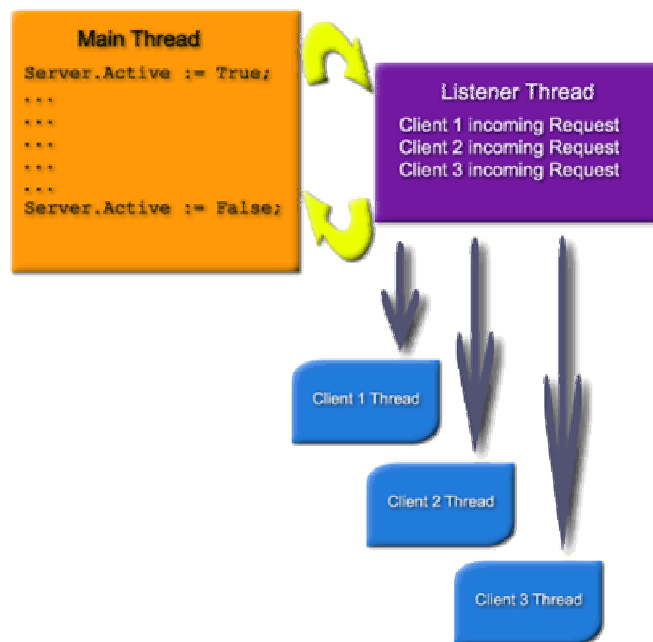
The `IdTCPServer1Execute` method is hooked to the `OnExecute` event of the server. The `OnExecute` event is fired by the server after a client connection has been accepted. The `OnExecute` event is uniquely different from other events you may be familiar with. `OnExecute` is executed in the context of a thread.

👉 The thread the event is called from is passed in the `AThread` argument of the method. This is important as many `OnExecute` events may be executing at the same time. This was done with an event so that a server could be built without the requirement of building a new component. There are also methods that can be overridden when descendant components are created.

The `OnConnect` is called after a connection has been accepted, and a thread created for it. In this server it is used to send the welcome message to the client. This could also be done in the `OnExecute` event if desired.

The `OnExecute` event will be called repeatedly until the connection is disconnected or broken. This eliminates the need to check the connection and loop inside the event.

Indy server components create a listener thread that is separate from the main thread of the program. The listener thread listens for incoming client requests. For each client that it answers, it then spawns a new thread to service that client. The appropriate events are then fired within the context of that thread.



Many Listener Threads Step by Step

`IdTCPServer1Execute` uses two basic Indy functions, `ReadLn` and `WriteLn`. `ReadLn` reads a line from the connection and `WriteLn` writes a line to the socket stream connection.

```
sCommand := ReadLn('', 1000, 80);
```

The above line reads the command with 3 specific optimisations from the client and puts the input into the local string variable `sCommand`.

```

if SameText(sCommand, 'QUIT') then begin
    Disconnect;
end else if SameText(Copy(sCommand,1,8), 'ZipCode ') then begin
    WriteLn(ZipCodeList.Values[Copy(sCommand,9,MaxInt)]);
end;

```

Next the input in `sCommand` is parsed to see which command the client issued.

If the command is "Quit" the connection is disconnected. No more reading or writing of the connection is permitted after a disconnect call. When the event is exited after this, the listener will not call it again. The listener will clean up the thread and the connection.


If the command is "zipCode" the parameter after the command is extracted and used to look up the city and state. The city and state is then written to the connection, or an empty string if one a match for the parameter is not found.

Finally the method is exited. The server will recall the event again as long as the connection is connected, allowing the client to issue multiple commands.

In this lesson we deal with multiple instances of `maxbox` and his creation of a `TCPClient` and a `TCPServer` instance. You can create multiple instances of the same app to execute parallel code, just type <F4>. For example, you can launch a new instance within a script of the box in response to some user action, allowing each script to perform the expected response.

```
ShellExecute3(ExePath+'maxbox3.exe',ExePath+'examples\'+ascript,secmdopen);
```

So creating multiple instances results in launching multiple instances of the application and the scripts too. There's no good way to launch one application (`maxbox`) with multiple scripts in it. Maybe with OLE Automation in that sense you open Office programs (word, excel) or other external shell objects.

 Try also the example of OLE Objects `318_excel_export3.TXT` and the tutorial 19.

An external script or a least a second one could also be a test case to compare the behaviour. Each test case and test project is reusable and rerunnable, and can be automated through the use of shell scripts or console commands.

*Per* Lets now take a look at the Client.

`Button1Click` is a method that is hooked to the `OnClick` event of `Button1`. When the button is clicked it executes this method. The `Indy` portion of this method can be reduced to the following:

1. Connect to Server ( `Connect;` ) – Instance1 - Server
2. Read welcome message from the server.
3. For each line the user entered in the `TMemo`:
4. Send request to server ( `WriteLn('ZipCode ' + memoInput.Lines[i]);` ) Instance2 - Client
5. Read response from server ( `s:= ReadLn;` )
6. Send Quit command ( `WriteLn('Quit');` )
7. Disconnect ( `Disconnect;` ) from Sockets:

```

procedure TCPServer_IdTCPServer1Execute(AThread: TIdPeerThread);
var
    sCommand: string;
begin
    with AThread.Connection do begin
        sCommand:= ReadLn(' ',1000,80);
        if SameText(sCommand, 'QUIT') then begin

```

```

Disconnect;
end else if SameText(Copy(sCommand,1,8),'ZipCode ') then begin
  WriteLn(ZipCodeList.Values[Copy(sCommand,9,MaxInt)]);
end;
PrintF('Command %s at %-10s received from %s:%d',[sCommand,
  DateTimeToStr(Now), socket.binding.PeerIP,
  Socket.binding.PeerPort]);
end; {with}
end;
{try [trye | try except | Borland.EditOptions.Pascal]
| except
end;}

```

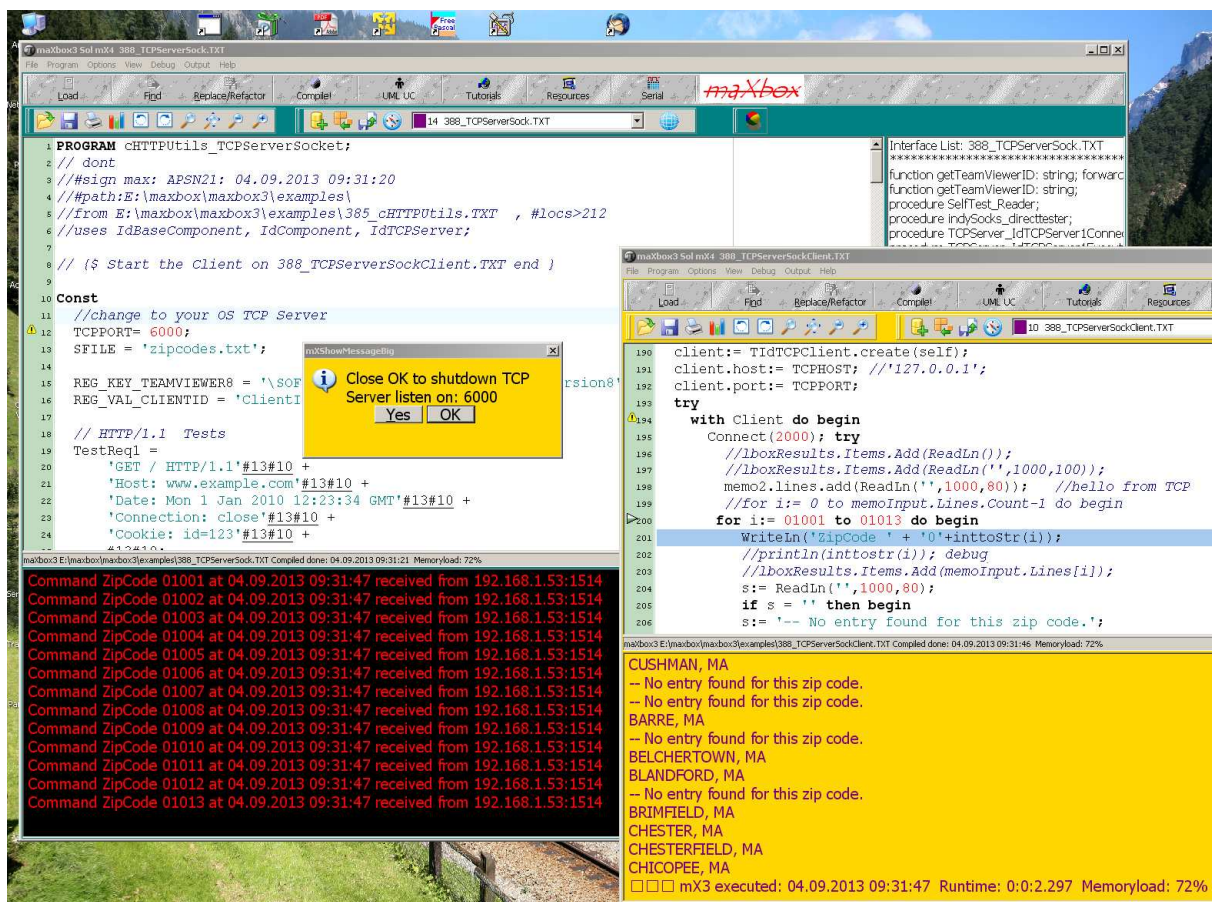
The Function ReadLn (not to confuse with the normal one) takes 3 parameters:

```

//('Function ReadLn(ATerminator: string; const ATimeout,
  AMaxLineLength: Integer): string');

```

👉 The only parts that are Indy specific are the Client component and the Button1Click method.



## 2: TCP Client/Server Script Loaded

👉 Could be that the Server runs forever (no Quit). A standard approach to break a running loop in a script or configuration is the well known KeyPress or IsKeyPressed function you can check:

```

procedure LoopTest;
begin
  Randomize;
  REPEAT
    Writeln(intToStr(Random(256*256)));
  UNTIL isKeyPressed; //on memo2 output
  if isKeyPressed then writeln(Key has been pressed!');
end;

```

As you know the memo2 is the output window as the shell, so the `keypress` is related to memo2; by the way memo1 is still the editor!

With another function `KeyPressed(VK: Integer): Boolean`; returns True, if key VK has been pressed.

Client is a `TIdTCPClient` and is a component in the app. The following properties were altered from the default:

- Host = 127.0.0.1 - Host was set to contact a server on the same machine as the client (or set by 192.168.1.53).
- Port = 6000 - An arbitrary number for this demo. This is the port that the client will contact the server with.

```

procedure TCPMain_Connect1Click(Sender: TObject);
begin
  //butnLookup.Enabled:= true;
  client:= TIdTCPClient.create(self);
  client.host:= TCPHOST; //'127.0.0.1';
  client.port:= TCPPORT;
  try
    with Client do begin
      Connect(2000); try
        //lboxResults.Items.Add(ReadLn());
        memo2.lines.add(ReadLn('',1000,80)); //hello from TCP
        //for i:= 0 to memoInput.Lines.Count-1 do begin
          for i:= 01001 to 01013 do begin
            WriteLn('ZipCode ' + '0'+inttoStr(i));
          end;
        end;
      finally
        writeln('inifile sysdata2: '+boot_conf+':'+intToStr(ip_port));
        Free;
      end;
    end;
end;


```

This process is handled directly, through an object so each time it changes `ReadLn` of the connection also and not on demand.



An Internet socket is characterized by a unique combination of the following:

- Local socket address: Local IP address and port number
- Remote socket address: Only for established TCP sockets. As discussed in the client-server section below, this is necessary since a TCP server may serve several clients concurrently. The server creates one socket for each client, and these sockets share the same local socket address from the point of view of the TCP server.
- Protocol: A transport protocol (e.g., TCP, UDP, raw IP, or others). TCP port 53 and UDP port 53 are consequently different, distinct sockets.

 In other words within the operating system and the application that created a socket, a socket is referred to by a unique integer number called socket identifier or socket number. The operating system forwards the payload of incoming IP packets to the corresponding application by extracting the socket address information from the IP and transport protocol headers and stripping the headers from the application data.

## 1.2 Telnet Testing

These demos were pre-tested and will work as long as TCP/IP is installed and active on your system. You can change this to run across the network from one computer to another by running the server on another computer and changing the host property of the client to the IP or TCP/IP name of the machine the server is running on. Otherwise it will look for the server on the same computer as the client.

To test the projects, compile and run the server script. Then compile and run the client script. Enter a zip code(s) into the memo field or direct in code on the left and click lookup. Plain text protocols can be debugged easily because they can be tested using a telnet session. To do this you merely need to know what port the server is running on. Zip Code Lookup Server listens on port 6000.

Run ZipCode Lookup Server Script again. Next open a command window (a.k.a Dos Box Window). Now type on the shell (picture 4):

```
telnet 127.0.0.1 6000 <enter>
or telnet 192.168.1.53 6000 <enter>
```

You are now connected to the Zip Code Lookup Server. Some servers will greet you with a welcome message. This one does not at first time then after Echo it does:

```
>>> Indy Hanoi Zip Code mXServer Ready.
```

You will not see your keystrokes. Most servers do not echo the commands as it would be a waste of bandwidth.

You can however change your telnet settings by setting "Echo On". Different telnet clients will call this feature different things. A few do not even have this option. Now type:

```
>>zipCode 37642 <enter>
```

You will see the server respond with:

```
CHURCH HILL, TN
```

To disconnect from the server enter:

```
>>Quit <enter>
```

Telnet is a network protocol used on the Internet or local area networks to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. User data is



interspersed in-band with Telnet control information in an 8-bit byte oriented data connection over the Transmission Control Protocol (TCP).

👉 In maXbox you can also start with read only mode (Options/Save before Compile), so nothing will be write on the disk or you can set your webconfig before.

```
/**/ Definitions for maXbox mX3 **/  
[WEB]  
IPPORT=8080 //for internal webserver - menu /Options/Add Ons/WebServer2  
IHOST=192.168.1.53  
ROOTCERT='filepathY' //for use of HTTPS and certificates...  
SCERT='filepathY'  
RSAKEY='filepathY'  
VERSIONCHECK=Y //checks over web the version
```

//V 3.9.8.9 also expand macros (see right below) in code e.g. #path or #file: and will cover below.

🖱️ Try to find out how you get the time from TCP Server:?

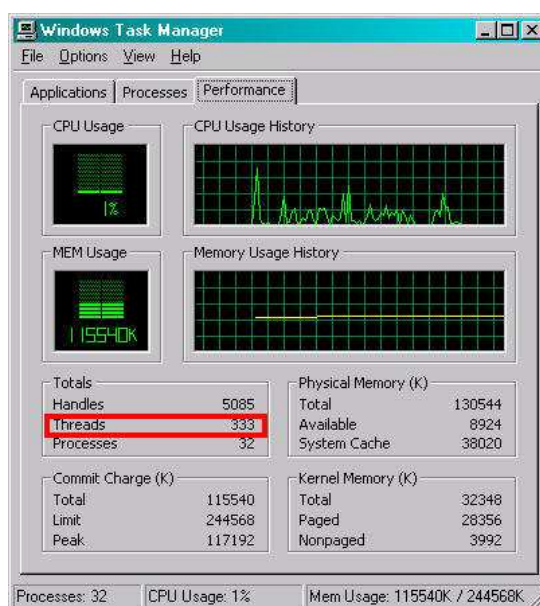
👉 Now let's take a look at the code of the solution:

```
if SameText(sCommand, 'TIME') then begin  
    WriteLn(DateTimeToInternetStr(Now, true));  
end
```

Another idea will be to compare with FTP the whole standard directory with your directory and check if something on the configuration is missing or damaged. But I think that will going to far. Something to keep in mind - there are literally hundreds of platform-specific directory listing formats still being used by FTP servers on the Internet today.

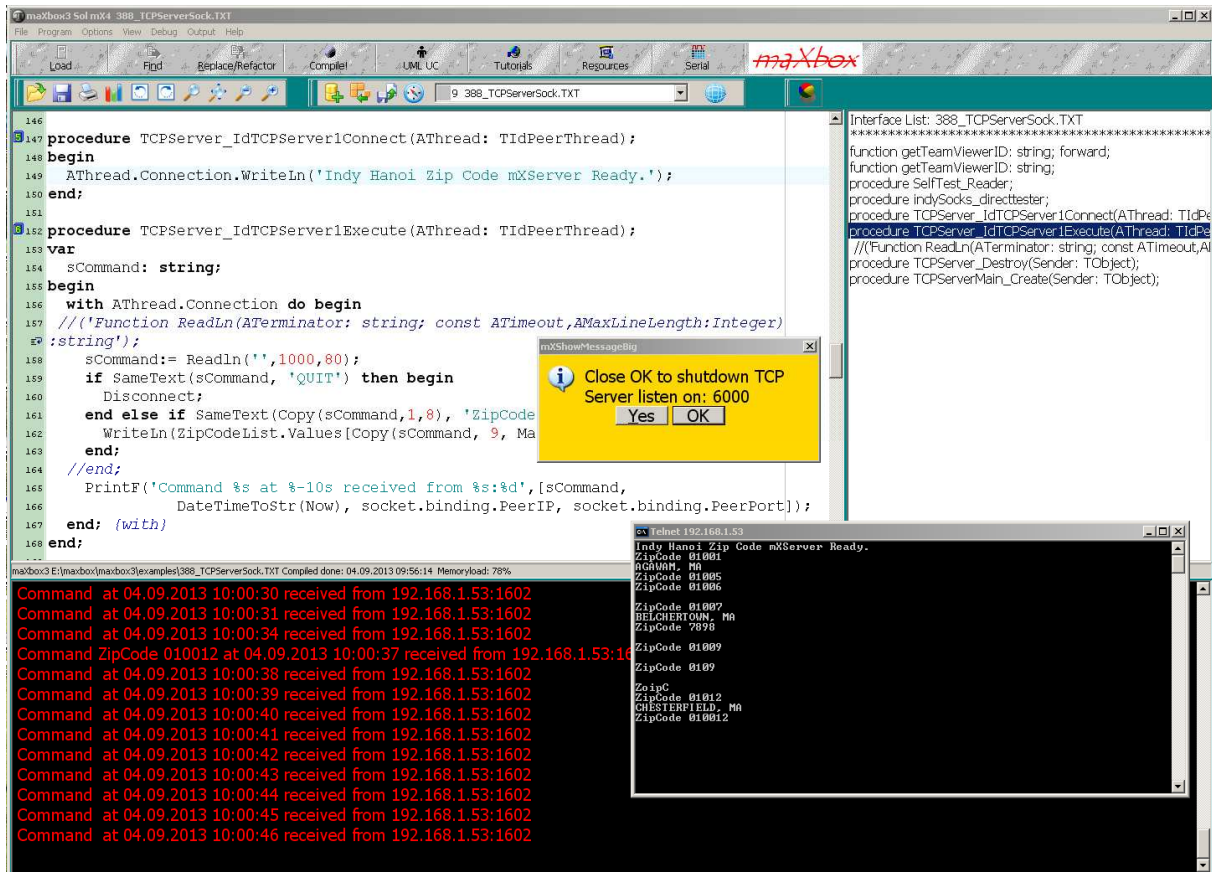
The LIST command outlined in the original FTP specification, RFC 959, did not define any kind of formatting to be used for listings, so systems were free to use whatever they wanted to use, and they did do exactly that over the years. Windows and Unix formats are common, but they are not required.

A formal listing format was not defined until RFC 3659 in the MLSD extension to FTP, which replaces the old LIST command (TIdFTP.List() does use MLSD if the server supports it).



3: Indy Threads on a TCPServer runs

Historically, Telnet provided access to a command-line interface (usually, of an operating system) on a remote host. Most network equipment and operating systems with a TCP/IP stack support a Telnet service for remote configuration (including systems based on Win NT). However, because of serious security issues when using Telnet over an open network such as the Internet, its use for this purpose has waned significantly in favour of SSH.



#### 4: Telnet in Action

Telnet is a client-server protocol, based on a reliable connection-oriented transport. Typically this protocol is used to establish a connection to Transmission Control Protocol (TCP) port number 23, where a Telnet server application (`telnetd`) is listening. Telnet, however, predates TCP/IP and was originally run over Network Control Program (NCP) protocols.



Check your system environment with `GetEnvironmentString`:

```
SaveString(ExePath+'\\Examples\\envinfo.txt',GetEnvironmentString);
OpenFile(ExePath+'\\Examples\\envinfo.txt');
```

### 1.3 Add The Macro

The only thing you need to know is to set the macros like `#host`: in your header or elsewhere in a line, but not two or more on the same line when it expands with content:

Let's have a look at the demo `369_macro_demo.txt`

```
{
*****
* Project   : Macro Demo
* App Name: #file:369_macro_demo.txt
* Purpose  : Demonstrates functions of macros in header
* Date    : 21/09/2010 - 14:56 - #date:01.06.2013 16:38:20
}
```

```

* #path E:\maxbox\maxbox3\examples\
* #file 369_macro_demo.txt
* #perf-50:0:4.484
* History : translate/implement to maXbox June 2013, #name@max
*          : system demo for mX3, enhanced with macros, #locs:149
*****}

```

All macros are marked with red. One of my favour is #locs means lines of code and you get always the certainty if something has changed by the numbers of line. So the editor has a programmatic macro system which allows the pre compiler to be extended by user code I would say user tags.

Below an internal extract from the help file All Functions List maxbox\_functions\_all.pdf:

```

//-----
10181: //*****mX4 Macro Tags *****
10182: //-----
10183:
10184: #name, #date, #host, #path, #file, #head, #sign, #tech
10185:
10186: SearchAndCopy(memol.lines, '#name', getUsernameWin, 11);
10187: SearchAndCopy(memol.lines, '#date', datetimetoStr(now), 11);
10188: SearchAndCopy(memol.lines, '#host', getComputernameWin, 11);
10189: SearchAndCopy(memol.lines, '#path', fpath, 11);
10190: SearchAndCopy(memol.lines, '#file', fname, 11);
10191: SearchAndCopy(memol.lines, '#locs', intToStr(getCodeEnd), 11);
10192: SearchAndCopy(memol.lines, '#perf', perftime, 11);
10193: SearchAndCopy(memol.lines, '#head',Format('%s: %s: %s %s ',
10194: [getUsernameWin, getComputernameWin, datetimetoStr(now), Act_Filename]),11);
10195: SearchAndCopy(memol.lines, '#sign',Format('%s: %s: %s ',
    [getUsernameWin, getComputernameWin, datetimetoStr(now)]), 11);
10196: SearchAndCopy(memol.lines, '#tech',Format('perf: %s threads: %d %s %s',
    [perftime, numprocessthreads, getIPAddress(getComputerNameWin),
    timetoStr(time)]), 11);

```

☞ Some macros produce simple combinations of one liner tags but at least they replace the content by reference in contrary to templates which just copy a content by value. You can also enhance the API with functions like the example above `GetEnvironmentString`:

```

function getEnvironmentString2: string;
var
    list: TStringList;
    i: Integer;
begin
    list:= TStringList.Create;
    try
        GetEnvironmentVars(list, False);
        for i:= 0 to list.Count-1 do
            result:= result + list[i]+#13#10;
        finally
            list.Free;
        end;
    end;

```

# maXbox

Feedback: [max@kleiner.com](mailto:max@kleiner.com)

Links of maXbox and Socket Programming:

<http://www.softwareschule.ch/maxbox.htm>

<http://sourceforge.net/projects/maxbox>

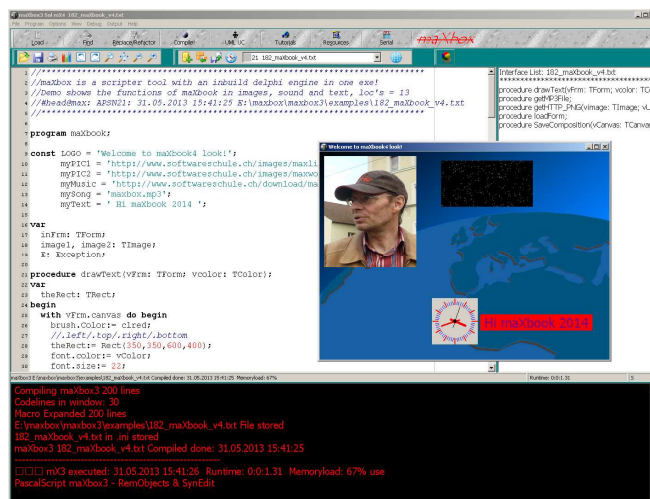
<http://sourceforge.net/apps/mediawiki/maxbox/>

<http://sourceforge.net/projects/delphiwebstart>

<http://devmentor.org/articles/network/Socket%20Programming%28v2%29.pdf>

<http://www.atozedsoftware.com/index.en.aspx>

[http://www.softwareschule.ch/maxbox\\_mainscreen.png](http://www.softwareschule.ch/maxbox_mainscreen.png)



## 1.4 Appendix

# EXAMPLE: List Objects

### Working with Lists

The VCL/RTL includes many classes that represents lists or collections of items. They vary depending on the types

of items they contain, what operations they support, and whether they are persistent.

The following table lists various list classes, and indicates the types of items they contain:

### Object Maintains

TList A list of pointers

TThreadList A thread-safe list of pointers

TBucketList A hashed list of pointers

TObjectBucketList A hashed list of object instances

TObjectList A memory-managed list of object instances

TComponentList A memory-managed list of components (that is, instances of classes descended from *TComponent*)

TClassList A list of class references

TInterfaceList A list of interface pointers.

TQueue A first-in first-out list of pointers

TStack A last-in first-out list of pointers

TObjectQueue A first-in first-out list of objects

TObjectStack A last-in first-out list of objects

TCollection Base class for many specialized classes of typed items.

TStringList A list of strings

THashedStringList A list of strings with the form Name=Value, hashed for performance.

The ProgID for each of the Shell objects is shown in the following table.

Object	Function
TList	A list of pointers
TThreadList	A thread-safe list of pointers
TBucketList	A hashed list of pointers
TObjectBucketList	A hashed list of object instances
TObjectList	A memory-managed list of object instances
TComponentList	A memory-managed list of components
TClassList	A list of class references
TInterfaceList	A list of interface pointers
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TCollection	Base class for many specialized classes of typed items
TStringList	A list of strings
THashedStringList	A list of strings with the form Name=Value, hashed for performance.