Beating The System: Wrapping The RAS Services API, Part 2

by Dave Jewell

Well, here we are running `TRASPhoneBookManager` under a combination of Delphi 5 and Windows 2000 and, to my great surprise, everything seems to work just fine. As I mentioned last month, I wasn't able to try out the code under Windows 2000 before going to press, one of the problems being that the modem on that machine had been taken out by a lightning strike! If you've ever tried setting up Dial Up Networking on a Windows 2000 system, you'll find that it's impossible to do so without a working modem, the system insists on dialling some number to see what special offers, etc, are available in your area and it won't allow you to create normal phonebook entries until that's been done! Such is life.

With the `TRASPhoneBookManager` component sat on a design-time form under NT or Win2000, you can point the `PhoneBookFileName` at a valid phonebook and the other properties of the component will instantly change to reflect the contents of the file. One minor oversight is that with last month's code it was impossible to point the `PhoneBookFileName` at a valid phonebook and subsequently set the property back to an empty string in order to reference the default phonebook. This has been fixed in this month's code, which is on the disk as always.

As promised last month, we're going to continue enhancing the phonebook component and presenting a few other goodies, but first...

Time For A Little Anarchy

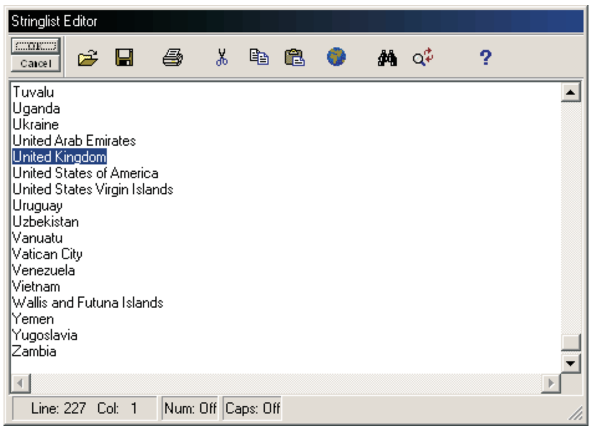
One of the things I wanted to do during this exploration of the RAS subsystem was come up with a component that gives you the list of country names and dialling codes. In order to do this, you have to issue a call to `RasGetCountryInfo`. If you examine the MSDN documentation relating to this call, you'll see that it requires the OSR2 release of Windows 95 or better, and on NT it needs version 4.0 or later. Moreover, you'll notice that the documentation makes frequent reference to TAPI because, as you're probably aware, RAS is itself built on top of TAPI, Microsoft's Telephony API. The `RasGetCountryInfo` call is certainly better designed than the `RasEnumEntries` routine that we looked at last time, because it gives you country information one item at a time, rather than requiring the client to provide an enormous buffer in which to dump everything. Nevertheless, I decided to indulge in a little bra-burning anarchy and 'hit the metal' directly, bypassing RAS and TAPI altogether.

This disgraceful behaviour was prompted through spending some time sleuthing around in the system registry and discovering that Windows 2000 and Windows 98 use exactly the same registry location in which to store the country list. Nah. I just couldn't resist. To see the country list under Windows 9x and NT, check out the following registry location using RegEdit:

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  Microsoft\Windows\
  CurrentVersion\Telephony\
  Country List
```

You'll notice there's a string value, `CountryListVersion`, which has the value 25 under Windows 2000 and 12 under NT. Despite this, the actual format of each entry appears to be identical, and my code will work quite happily under both platforms. The code for the new country list component, `TRASCountryList`, is given in Listing 1. As you can see, it's not based on the `TRASBaseComponent` base class that I presented last month. That's because, since it doesn't use the RAS API, it doesn't have to concern itself with loading RAS DLLs, ensuring that specific routines exist in the various DLLs and so forth. Life is *so* much simpler when you leave Microsoft APIs behind. 'Nuff said ☺.

As you'll see from the code, the component stores the list of available country names in the `Countries` property, which is a string list. This property is populated by calling `Refresh` from the control's constructor. The `Refresh` routine is public so that you can call it for yourself if it's been some time since you created the control, but in practice I'm not aware of anything that's likely to modify TAPI's list of countries. Things can sometimes get a bit volatile in the Balkans, but Microsoft haven't yet built a real-time country list update facility into Windows.



```
Tuvalu
Uganda
Ukraine
United Arab Emirates
United Kingdom
United States of America
United States Virgin Islands
Uruguay
Uzbekistan
Vanuatu
Vatican City
Venezuela
Vietnam
Wallis and Futuna Islands
Yemen
Yugoslavia
Zambia

Line: 227  Col: 1   Num: Off  Caps: Off
```

➤ *Figure 1: The deeply patriotic TRASCountryList component sorts the country list alphabetically, rather than by dialling code, which has an interesting effect on the state of play.*

The CountryName property stores the currently 'selected' country. If you try and set this property to something that isn't in the list of country names, you'll be politely ignored. Contrariwise, when you do set the property to a different country, the corresponding CountryID and CountryDialCode properties will be automatically updated.

So what's all that about then? Basically, every country known to TAPI has a unique country ID. However, it's possible for more than one country to share the same dialling code. In other words, there's a potential one to many relationship when mapping from dialling code to country and country ID. Typically, the dialling code is the same as the country ID, but it doesn't have to be. Thus, for example, the United Kingdom has an international dialling code of 44, and also has a unique country ID of 44. Russia, likewise, has a dialling code of 7 and a country ID of 7. However, this same dialling code is shared with Kazakhstan (country ID 705) and with Tajikistan (country ID 708). For these reasons, you can set a new value for the CountryID property because this is an unambiguous operation, but you're not allowed to set a new value for the CountryDialCode property because several different countries might potentially map to the same dial code.

With these deliberations out of the way, the code itself is quite straightforward. The Refresh method creates a TRegistry object and uses the GetKeyNames method to retrieve the list of all currently defined country IDs from the aforementioned registry location. It then steps through the list retrieving the actual name of each country which is then added to the fCountries list. In order to avoid having to re-read the registry later, the code also obtains the dialling code from the registry and packs it into the Objects array of the string list, alongside the country ID. You'll also notice that I decided to sort the list of country names alphabetically, which places the United States into its rightful place just after the United Kingdom (see Figure 1). Ooops, did I say that... As a final bit of unashamed patriotism, the TRASCountryList.Create constructor sets the country name property to the United Kingdom by default.

The SetCountryName and SetCountryID property 'setters' are both very simple. The former uses the passed country name to obtain the corresponding index into the country name list, using this

➤ *Listing 1*

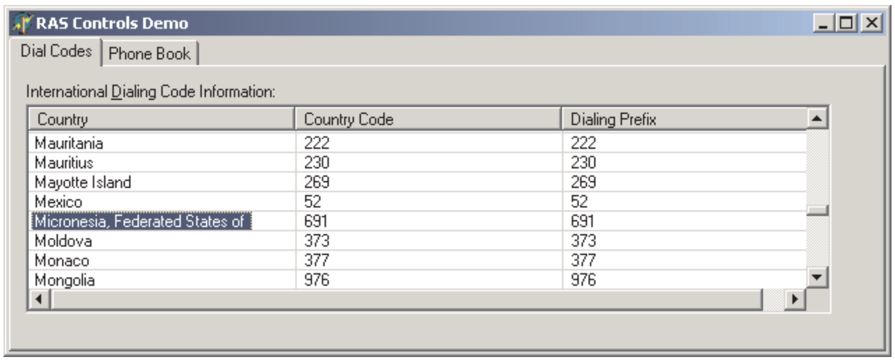
```
unit RASCountryList;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TRASCountryList = class (TComponent)
  private
    fCountries, fDummy1: TStrings;
    fCountryName: String;
    fCountryDialCode, fCountryID, fDummy2: Integer;
    procedure SetCountryID (Value: Integer);
    procedure SetCountryName (const Value: String);
  protected
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Refresh;
  published
    property Countries: TStrings read fCountries
      write fDummy1 stored False;
    property CountryName: String read fCountryName
      write SetCountryName;
    property CountryDialCode: Integer
      read fCountryDialCode write fDummy2;
    property CountryID: Integer
      read fCountryID write SetCountryID;
  end;
procedure Register;
implementation
uses
   Registry;
constructor TRASCountryList.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fCountries := TStringList.Create;
  Refresh;
  // Why should the US have everything it's way? :-)
  SetCountryName ('United Kingdom');
end;
destructor TRASCountryList.Destroy;
begin
  fCountries.Free;
  Inherited Destroy;
end;
procedure TRASCountryList.Refresh;
const
  CListReg = '\SOFTWARE\Microsoft\Windows\CurrentVersion\'+
             'Telephony\Country List';
var
  Idx: Integer;
  Reg: TRegistry;
  ObjectData: Integer;
  IDNames: TStringList;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := hKey_Local_Machine;
    if Reg.OpenKey (CListReg, False) then begin
      fCountries.Clear;
      // Set Sorted = False for speed....
      TStringList (fCountries).Sorted := False;
      IDNames := TStringList.Create;
      try
        Reg.GetKeyNames (IDNames);
        for Idx := 0 to IDNames.Count - 1 do
          if Reg.OpenKey(CListReg + '\' + IDNames[Idx],
            False) then begin
            ObjectData :=
              MakeLong(Reg.ReadInteger('CountryCode'),
              StrToInt(IDNames[Idx]));
            fCountries.AddObject(Reg.ReadString('Name'),
              TObject(ObjectData));
          end;
      finally
        IDNames.Free;
        TStringList(fCountries).Sorted := True;
      end;
    end;
  finally
    Reg.Free;
  end;
end;
procedure TRASCountryList.SetCountryName(
  const Value: String);
var
  Idx: Integer;
begin
  Idx := fCountries.IndexOf (Value);
  if Idx <> -1 then begin
    fCountryName := fCountries [Idx];
    fCountryDialCode :=
      Integer (fCountries.Objects [Idx]) and $ffff;
    fCountryID := Integer (fCountries.Objects [Idx]) shr 16;
  end;
end;
procedure TRASCountryList.SetCountryID (Value: Integer);
var
  Idx: Integer;
begin
  for Idx := 0 to fCountries.Count - 1 do
    if Value = Integer (fCountries.Objects [Idx]) shr 16
      then begin
      SetCountryName (fCountries [Idx]);
      Exit;
    end;
end;
procedure Register;
begin
  RegisterComponents ('DelphiMag', [TRASCountryList]);
end;
end.
```

information to retrieve the 'packed' country ID and dial codes from the `Objects` array. Similarly, the `SetCountryID` routine iterates through the string list until it finds a matching country ID, whereupon it calls `SetCountryName` to do the business.

Well, so much for the `TRASCountryList` component. Before someone writes to complain about me riding rough-shod over Microsoft's APIs, let me just point out that the name of this column is *Beating the System*! To me, it seems pointless to waste time mucking around with Microsoft's cumbersome APIs when all we're actually doing is retrieving a simple list of values from the registry. In order to obtain the same information in the official manner, the RAS code has to call the `lineGetCountry` routine (part of the TAPI API) which calls... well, you get the idea. If Microsoft were to properly document the way in which TAPI, RAS, etc, use the registry, then much of this nonsense could be eliminated. Harrumph!

A RAS Demo

At this point, I decided that it was time to build a testbed program for the components built so far. I also decided to rationalise things somewhat, and so I renamed last month's RASControls.Pas file to RASPhoneBook.Pas, placing the low-level ancestor code into RASBase.Pas.


RAS Controls Demo

Dial Codes | Phone Book

International Dialing Code Information:

Country	Country Code	Dialing Prefix
Mauritania	222	222
Mauritius	230	230
Mayotte Island	269	269
Mexico	52	52
Micronesia, Federated States of	691	691
Moldova	373	373
Monaco	377	377
Mongolia	976	976

➤ *Figure 2: Here's the TRASCountryList component installed as part of the testbed program included on this month's disk.*

You can see the Dial Codes page of the testbed program in Figure 2. While browsing through the list of countries, I was intrigued to find an entry marked 'Federated States of Micronesia'. *Micro*nesia? Hmmm... this has to be one of Bill's little jokes, right? Well, I tracked down the FSM on the Internet and it's for real. I guess you learn something new every day *[Even though our esteemed columnist didn't know you existed, our mailing house does, and all you residents of the FSM are very welcome indeed to become subscribers! Ed]*.

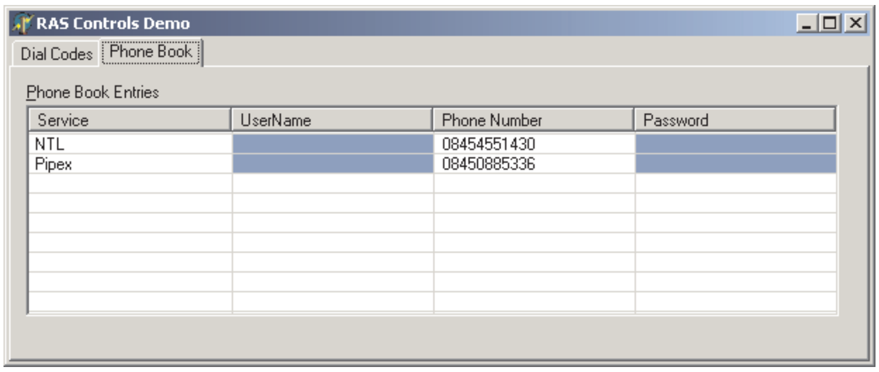
You can see the phonebook page of the testbed in Figure 3. For reasons that should be somewhat obvious, I've removed my passwords from this illustration. I trust you all implicitly, but...

As I promised last time, I wanted to enhance the phonebook component by making it possible to modify some important fields, such as username, password, etc, rather than just implementing the component as a read-only browser. Inevitably, this requires a certain amount of caution, especially for folks like me whose livelihood depends on their internet connection! The bottom line is that if you want to play around with the existing code, you should back up your current phonebook entries before experimenting. On NT, this is easy, because you can simply make a backup copy of the .PBK file that you're going to be playing around with. Under Windows 95/98, you should use something like RegEdit to export the registry sub-tree corresponding to your current phonebook, or at the very least write down the username and password settings corresponding to each of your ISP accounts.

Last month, we discussed the `TRASDialParams` and `TRASDialParamsNT4_2000` data structures whose contents are retrieved via a call to the `RASGetEntryDialParams` routine. You may have noticed that in last month's code I separated out the actual call to this routine into a method called `InternalGetDialParameters` rather than placing it inside the `GetDialParameters` routine. I did this in preparation for this month's instalment because it makes it possible for the new `SetDialParameters` routine (see Listing 2) to easily retrieve the current state of play, alter only the field that needs to be changed and then write the record back via `RASSetEntryDialParams`. When you are working with complicated Microsoft data structures, it's best to do things this way rather than trying to build a whole new data structure from scratch, which would be tedious, error-prone and probably require a call to `RASGetEntryDialParams` at some point anyway.

As with last month's code, it's easy to implement both `UserName`

➤ *Figure 3: And here are the phonebook entries on my Windows 2000 machine, courtesy of the phonebook component. Passwords and user names have been hidden to protect the innocent.*


RAS Controls Demo

Dial Codes | Phone Book

Phone Book Entries

Service	UserName	Phone Number	Password
NTL	[redacted]	08454551430	[redacted]
Pipex	[redacted]	08450885336	[redacted]

and Password as writeable properties, simply by pointing the 'getter' clause of their property declarations at the new SetDialParameters routine. This is also shown in Listing 2, which is a sort of 'delta' of last month's code. As ever, complete source code is included on the disk.

There are few surprises in the actual implementation of SetDialParameters. There's an explicit check to make sure that an empty string isn't passed as the username. According to the Microsoft documentation, setting an empty username would be disallowed within RASSetEntryDialParams, but maybe it's better to make doubly sure. Similarly, if you set the password to an empty string, this will cause the RemovePassword flag to be set on the call into the RAS API.

Note: If you do remove a password in this way, then TRASPhoneBookManager will then report the password as --not available-- because of the code in GetDialParameters. This may or may not be what you want. If it isn't, then it might be better to modify GetDialParameters to simply return an empty string.

In order to alter the telephone number, device name, etc, we need to use the RASSetEntryProperties routine which corresponds to the RASGetEntryProperties API which we looked at last time. Both of these routines use the TRASEntry data structure which is complicated by the fact that it can be followed by an arbitrary number of alternative phone numbers formatted as C-style strings. Here again, I've adopted the strategy of reading the entire data structure into memory (a 10,000 byte buffer ought to be big enough!) just modifying the wanted fields, and then writing the thing back out again via RASSetEntryProperties.

Listing 2 includes the source code for the new SetEntryProperties routine, which I've tied into the PhoneNumber, DeviceType and DeviceName properties using the revised property declarations in Listing 2. You'll notice that the code allocates a 10,000 byte buffer (yes, I know you're not likely to have that many alternative phone numbers, but you never know when Microsoft are going to add something else to the end of the structure) and passes the buffer size to the new InternalSetProperties routine. This returns the current state of play, including any alternative phone number information, and the code then updates the requisite field of the data structure before calling RASSetEntryProperties.

You might perhaps raise an eyebrow at the fact that I don't perform any input parameter validation here. If you try specifying a non-existent device name or device type, the RASSetEntryProperties routine is smart enough to politely ignore you. Try fiddling with these property values in the Object Inspector and you'll see them snap back to their previous values if you enter something invalid. Currently, TRASPhoneBookManager allows you to specify an empty phone number.

One thing you might wish to do is create an entirely new phonebook entry programmatically, as opposed to calling the Add function, which simply brings up Microsoft's dialog. Again, this is pretty easy. You could do it the hard way by filling in a TRASEntry data structure from scratch, but a simpler approach is to use RASGetEntryProperties to retrieve the data structure of an *existing* phonebook entry, edit the phone number as appropriate, and then write the data structure back with RASSetEntryProperties, using an entry name that *doesn't already exist*. This will create a new entry in the phonebook. Obviously, this is only appropriate for cases where there's already at least one existing entry, and when you're running on a system that has only one RAS-compatible device.

A RAS Device Manager

Did I mention devices? We've got a drop-in component for examining and modifying phonebook entries,

➤ *Listing 2*

```
property UserName: String index 0 read GetDialParameters
  write SetDialParameters stored False;
property Password: String index 1 read GetDialParameters
  write SetDialParameters stored False;
property PhoneNumber: String index 0 read GetEntryProperties
  write SetEntryProperties stored False;
property DeviceType: String index 1 read GetEntryProperties
  write SetEntryProperties stored False;
property DeviceName: String index 2 read GetEntryProperties
  write SetEntryProperties stored False;

procedure TRASPhoneBookManager.SetDialParameters (Index:
  Integer; const Value: String);
var
  GotPassword: Bool;
  Params: TRasDialParamsNT4_2000;
  RasSetEntryDialParams: function(Phonebook: PChar;
   var RasDialParams: TRasDialParamsNT4_2000;
   RemovePassword: Bool): Integer; stdcall;
begin
  if InternalGetDialParameters (Params, GotPassword) > 0
    then begin
    // Can't set username to an empty string.
    if (Index = 0) and (Value = '') then
      Exit;
    if Index = 0 then
      StrPCopy (Params.UserName, Value)
    else
      StrPCopy (Params.PassWord, Value);
    RasSetEntryDialParams :=
      GetProc('RasSetEntryDialParamsA');
    if Assigned (RasSetEntryDialParams) then
      CallProc (RasSetEntryDialParams(PhoneBookNameAsPChar,
        Params, (Index = 1) and (Value = '')));
  end;
end;

procedure TRASPhoneBookManager.SetEntryProperties (Index:
  Integer; const Value: String);
var
  Props: TRASEntry;
  EntrySize, DevInfoSize: Integer;
  Buffer: array [0..10000] of Char absolute Props;
  RasSetEntryProperties: function(
    Phonebook, EntryName: PChar; var Entry: TRASEntry;
    var EntrySize: Integer; DevInfo: Pointer;
    var DevInfoSize: Integer): Integer; stdcall;
begin
  EntrySize :=
    InternalGetEntryProperties(Props, sizeof(Buffer));
  if EntrySize > 0 then begin
    case Index of
      0 : StrPCopy(Props.LocalPhoneNumber, Value);
      1 : StrPCopy(Props.DeviceType, Value);
      2 : StrPCopy(Props.DeviceName, Value);
    end;
    DevInfoSize := 0;
    RasSetEntryProperties :=
      GetProc('RasSetEntryPropertiesA');
    if Assigned (RasSetEntryProperties) then
      CallProc(RasSetEntryProperties(PhoneBookNameAsPChar,
        PChar(fEntries[fItemIndex]), Props, EntrySize,
        Nil, DevInfoSize));
  end;
end;
```

```
unit RASDevices;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, RASBase;
type
  TRASDeviceManager = class (TRASBaseComponent)
  private
    fDeviceNames, fDeviceTypes, fDummy2: TStrings;
  protected
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Refresh;
  published
    property DeviceNames: TStrings read fDeviceNames
      write fDummy2 stored False;
    property DeviceTypes: TStrings read fDeviceTypes
      write fDummy2 stored False;
  end;
procedure Register;
implementation
constructor TRASDeviceManager.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fDeviceNames := TStringList.Create;
  fDeviceTypes := TStringList.Create;
  Refresh;
end;
destructor TRASDeviceManager.Destroy;
begin
  fDeviceNames.Free;
  fDeviceTypes.Free;
  Inherited;
end;
procedure TRASDeviceManager.Refresh;
type
  TRasDevInfo = record
  dwSize: DWORD;
  DeviceType: array [0..16] of Char;
  DeviceName: array [0..128] of Char;
  end;
var
  CurDev: ^TRasDevInfo;
  Buffer: array [0..10000] of Char;
  Idx, BufSize, NumDevices: Integer;
  RasEnumDevices: function (Buffer: PChar; var BufSize,
    NumDevices: Integer): Integer; stdcall;
begin
  if Available then begin
    // First off, refresh the entries list
    fDeviceNames.Clear;
    fDeviceTypes.Clear;
    RasEnumDevices := GetProc ('RasEnumDevicesA');
    if Assigned (RasEnumDevices) then begin
      CurDev := @Buffer;
      CurDev^.dwSize := sizeof (TRasDevInfo);
      BufSize := sizeof (Buffer);
      if CallProc(RasEnumDevices(Buffer, BufSize,
        NumDevices)) then
        for Idx := 0 to NumDevices - 1 do begin
        fDeviceTypes.Add (CurDev^.DeviceType);
        fDeviceNames.Add (CurDev^.DeviceName);
        Inc (CurDev);
      end;
    end;
  end;
end;
procedure Register;
begin
  RegisterComponents ('DelphiMag', [TRASDeviceManager]);
end;
```

➤ *Listing 3*

and another for viewing country list information. How about another component for displaying the list of RAS- compatible devices on a system?

The `TRASDeviceManager` is the simplest RAS component presented so far, the code for which is shown in Listing 3. This component merely 'exports' a couple of string list properties, `DeviceNames` and `DeviceTypes`, which list the name of each RAS-compatible device and its corresponding type. You can see the sort of information that's retrieved by referring to Figure 4.

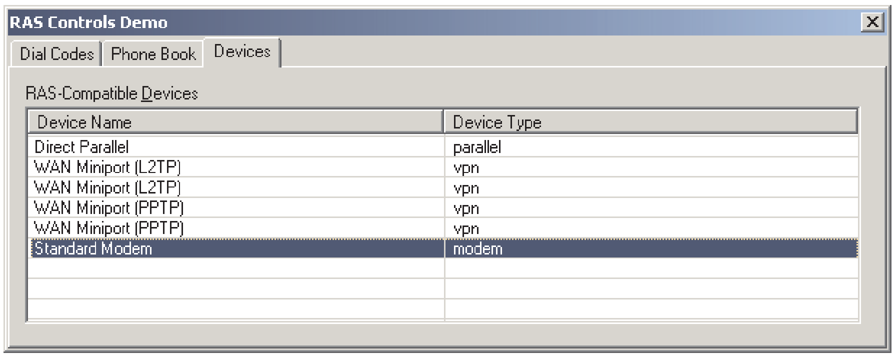
Here again, I was sorely tempted to bypass the RAS API altogether and simply retrieve the information directly from the registry, but I decided to be a good boy this time round! In this case, we don't have to contend with variable-sized data structures and buffers that may or may not be followed by arbitrary amounts of other stuff so, in reality, this is one of the simpler API calls to work with.

As with the phonebook component, `TRASDeviceManager` is derived from `TRASBaseComponent`. The constructor and destructor simply manage creation and deletion of the two internal string lists, and all the real work is done inside the `Refresh` routine. This simply allocates a (very!) generous sized buffer, and calls the `RasEnumDevices` API call to fill the buffer with a series of `TRASDevInfo` data structures, the layout of which can be seen from the code listing. At the same time, `RasEnumDevices` returns an integer which indicates how many device info structures have been copied to the buffer.

Using this information, we can simply loop through the returned data, extracting the device type and device name information and storing it in our string lists. This particular call exists on NT 4.0 or later, and on Windows 95 OSR2 or later, but as ever, the `Refresh` method is written in such a way as to fail gracefully if the call isn't implemented.

Since writing last month's code, it occured to me that a better implementation of the `Available` property in `TRASBaseComponent` would be to write it in such a way as to call a virtual abstract method, `GetAvailable`. This could then be overridden in derived classes to provide a more accurate indication of whether or not a service is available. In the current implementation, we are setting `Available` to `True` if the RAS DLL is present, whereas we should really be setting it according to whether or not a specific RAS API is available. This enhancement is left as an exercise for the reader!

➤ *Figure 4: Under Windows 2000, quite a generous array of RAS-compatible devices are available. Under Windows 95 or 98, you will typically only see your modem listed on a standalone machine.*



Device Name	Device Type
Direct Parallel	parallel
WAN Miniport (L2TP)	vpn
WAN Miniport (L2TP)	vpn
WAN Miniport (PPTP)	vpn
WAN Miniport (PPTP)	vpn
Standard Modem	modem

The Connection Manager

Figure 5 shows my connection manager component, TRASConnectionManager running as part of the RAS testbed program. The implementation of this component involves some extra issues in addition to what we've discussed so far. First and foremost, the status of a connection can obviously change 'under the feet' of an application, whereas things like country lists and available devices are far less 'volatile'. Thus, in Figure 5, you can sit in the testbed program and watch the status of a connection change as the user gets authenticated and the connection is finally established. This is a deeply cool feature (in my humble opinion!) and I'm quite pleased with it.

My original intention in implementing the connection manager was to make use of the RasConnectionNotification routine. This makes use of a kernel-level 'event object' and requires that the calling thread block on one of the special wait functions such as WaitForSingleObject, and so on. This, in turn, requires that a thread is dedicated to the business of waiting for a connection status change to take place. You'll remember that this is exactly the technique that I used a few months back when implementing my TFileSystem class.

Although this approach works well, and isn't especially difficult to do, there's a fundamental problem in that the RasConnectionNotification routine is only implemented under NT 4.0 (or later) and Windows 98. In other words, it doesn't exist under Windows 95 at all. Now, I can live with a RAS API routine that requires W95 OSR2, but I think that cutting Windows 95 out of the equation altogether is far too rigorous a requirement. Speaking personally, I still know an awful lot of people using Windows 95, and I think it would be foolhardy to write code that won't run on this platform, especially when there are various ways around the difficulty. In the end, I decided to use a standard Windows timer to provide a periodic update of the connection status. In the code presented here, I've set this to one second, which represents a fairly leisurely update, but you can obviously change this if you want.

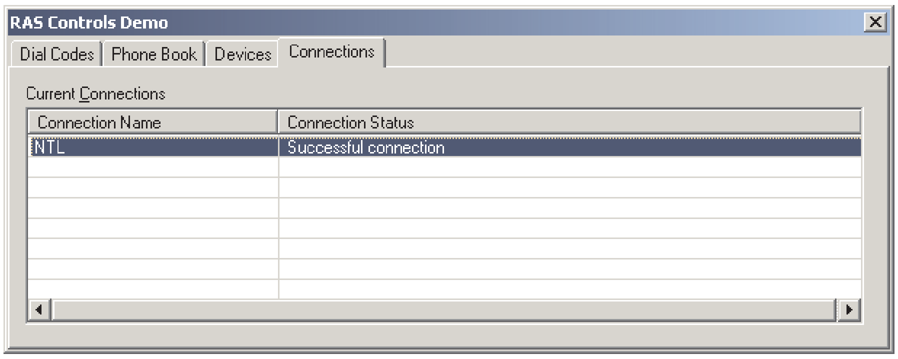
The code for the connection manager is given in Listing 4. As you can see, I create an ordinary timer by calling SetTimer in the component constructor, storing the timer identifier into fTimer. For some reason, even seasoned Windows programmers seem to think that you can't have a Windows-level timer unless you've created a window to receive wm_Timer messages, and that's simply not true. Even Borland's own developers don't seem to realise that no window is necessary. Don't believe me? If you take a peek at the implementation of TTimer (in EXTCTRLS.PAS) you'll see that a hidden window is created specifically for the purpose of receiving those timer messages. A careful reading of the SDK documentation shows this to be completely unnecessary, and the TTimer implementation would indeed be a lot simpler without it.

The secret is to specify zero as the window handle, and supply a pointer to a callback routine that's called directly by the kernel, rather than posting wm_Timer messages to your application. In this case, the callback routine is named TimerTickProc and, as with any API-level callback, this must be declared using stdcall.

There is one fly in the ointment, this being that the Windows API doesn't make provision for passing an application-supplied 32-bit value to the callback routine, thus making it impossible to recover the value of Self from within the callback. The documentation states that if the supplied window handle is zero, SetTimer ignores the supplied timer identifier parameter, which is certainly true. However, if Microsoft had had their brain cells engaged when they designed SetTimer, the timer identifier would have been passed as the third parameter to the callback routine.

Because we haven't got access to Self, we have to cheat and use a global variable (pause for screams of outrage!) which I've called InstanceHack. This global allows us to recover the instance handle for our connection manager, but it does mean that we can only have one instance of TRASConnectionManager at a time. In their defence, Borland might argue that it's because of such problems that they decided to implement TTimer with a hidden window. I guess there's some truth in that, but it would have been far more efficient (in terms of system resource usage) to use a single window shared by all TTimer instances rather than giving each timer its own window.

Be that as it may, you'll see that the component constructor starts off by calling the Refresh method, and it's this method which also gets called on each timer tick. Even though the connection manager makes a 'reality check' every second, the application doesn't really want to know unless some status change has actually taken



Connection Name	Connection Status
NTL	Successful connection

➤ *Figure 5: The connection manager component keeps track of all current connections, and will automatically send an update event to the application whenever the status of any of the connections is changed.*

place. Consequently, I've added an `OnStatusChange` event handler which is triggered whenever the component detects that the state of play has changed.

The connection manager can support multiple simultaneous connections and so, for each connection in existence, the phone-book name of the connection can be obtained from the `ConnectionNames` property, and the corresponding status from

➤ *Listing 4*

```
unit RASConnection;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, RASBase;
type
  TRASConnectionManager = class (TRASBaseComponent)
  private
    fTimer: Integer;
    fConnectionNames, fDummy1: TStrings;
    fOnStatusChange: TNotifyEvent;
    function GetStatus (Index: Integer): String;
    function StatusChanged (NewStatus: TStringList):
      Boolean;
  protected
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Refresh;
    property Status [Index: Integer]: String read GetStatus;
  published
    property ConnectionNames: TStrings read fConnectionNames
      write fDummy1 stored False;
    property OnStatusChange: TNotifyEvent
      read fOnStatusChange write fOnStatusChange;
    end;
procedure Register;
implementation
var
  InstanceHack: TRASConnectionManager;
procedure TimerTickProc(
  Wnd: hWnd; Msg, Event, Time: Integer); stdcall;
begin
  InstanceHack.Refresh;
end;
function ConStateToString(State: Integer): String;
begin
  case State of
    $0000: Result := 'Com Port Opening';
    $0001: Result := 'Com Port Opened';
    $0002: Result := 'Connecting Device';
    $0003: Result := 'Device Connected';
    $0004: Result := 'All Connected';
    $0005: Result := 'Starting Authenticate';
    $0006: Result := 'Authentication Event';
    $0007: Result := 'Retrying Authentication';
    $0008: Result := 'Callback number requested';
    $0009: Result := 'Password change request';
    $000A: Result := 'Projection Starting';
    $000B: Result := 'Calculating Link Speed';
    $000C: Result := 'Acknowledging Authentication Request';
    $000D: Result := 'Starting Reauthentication';
    $000E: Result := 'Successfully Authenticated';
    $000F: Result := 'Disconnecting for callback';
    $0010: Result := 'Resetting modem for callback';
    $0011: Result := 'Waiting for callback';
    $0012: Result := 'Projection complete';
    $0013: Result := 'Authenticating user';
    $0014: Result := 'Callback complete';
    $0015: Result := 'Logging onto network';
    $0016: Result := 'Subentry connected';
    $0017: Result := 'Subentry disconnected';
    $1000: Result := 'Terminal State';
    $1001: Result := 'Retrying Authentication';
    $1002: Result := 'Callback set by user';
    $1003: Result := 'Password has expired';
    $1004: Result := 'Paused for EAPUI';
    $2000: Result := 'Successful connection';
    $2001: Result := 'Failed connection';
    else   Result := 'Unknown';
  end;
end;
constructor TRASConnectionManager.Create(
  AOwner: TComponent);
begin
  // Yes, it's dirty, but it's also quick. ;-)
  if InstanceHack <> Nil then
    Exception.Create(
      'Only one TRASConnectionManager allowed');
  Inherited Create(AOwner);
  InstanceHack := Self;
  fConnectionNames := TStringList.Create;
  Refresh;
  fTimer := SetTimer(0, 0, 1000, @TimerTickProc);
end;
destructor TRASConnectionManager.Destroy;
begin
  if fTimer <> 0 then
    KillTimer(0, fTimer);
  fConnectionNames.Free;
  Inherited;
  InstanceHack := Nil;
end;
function TRASConnectionManager.StatusChanged(
  NewStatus: TStringList): Boolean;
var Idx: Integer;
begin
  Result := fConnectionNames.Count <> NewStatus.Count;
  if not Result then begin
    Result := True;
    for Idx := 0 to fConnectionNames.Count - 1 do begin
      if fConnectionNames [Idx] <> NewStatus [Idx] then
        Exit;
      if fConnectionNames.Objects[Idx] <>
        NewStatus.Objects[Idx] then
        Exit;
    end;
    Result := False;
  end;
end;
procedure TRASConnectionManager.Refresh;
type
  TRasConn = record
    dwSize: DWord;
    hConn: THandle;
    EntryName: array [0..20] of Char;
  end;
  TRasConStatus = record
    dwSize: DWord;
    ConState: Integer;
    Error: Integer;
    DeviceType: array [0..16] of Char;
    DeviceName: array [0..32] of Char;
  end;
var
  CurCon: ^TRasConn;
  NewStatus: TStringList;
  Status: TRasConStatus;
  Buffer: array [0..10000] of Char;
  Idx, BufSize, NumConnections: Integer;
  RasEnumConnections: function(Buffer: PChar; var BufSize,
    NumConnections: Integer): Integer; stdcall;
  RasGetConnectStatus: function(hConn: THandle; var Status:
    TRasConStatus): Integer; stdcall;
begin
  if Available then begin
    RasEnumConnections := GetProc('RasEnumConnectionsA');
    RasGetConnectStatus := GetProc('RasGetConnectStatusA');
    if Assigned (RasEnumConnections) and
      Assigned(RasGetConnectStatus) then begin
      NewStatus := TStringList.Create;
      try
        CurCon := @Buffer;
        CurCon^.dwSize := sizeof (TRasConn);
        BufSize := sizeof (Buffer);
        if CallProc (RasEnumConnections (Buffer, BufSize,
          NumConnections)) then
          for Idx := 0 to NumConnections - 1 do begin
          Status.dwSize := sizeof (Status);
          if CallProc(RasGetConnectStatus(
            CurCon^.hConn, Status)) then
            NewStatus.AddObject(CurCon^.EntryName,
              TObject(Status.ConState));
          Inc (CurCon);
        end;
        // Now see if anything has changed
        if StatusChanged (NewStatus) then begin
          fConnectionNames.Assign (NewStatus);
          if Assigned (OnStatusChange) then
            OnStatusChange(Self);
        end;
      finally
        NewStatus.Free;
      end;
    end;
  end;
end;
function TRASConnectionManager.GetStatus(
  Index: Integer): String;
begin
  Result := '';
  if Index < fConnectionNames.Count then
    Result := ConStateToString(
      Integer(fConnectionNames.Objects[Index]));
end;
procedure Register;
begin
  RegisterComponents('DelphiMag', [TRASConnectionManager]);
end;
end.
```

the Status property array. When a connection terminates, it automatically disappears from the list of available connections on the next timer tick.

The real 'guts' of the connection manager are contained within the Refresh routine which actually requires two distinct RAS routines. First, it's necessary to call RASEnumConnections, which enumerates all the current connections that RAS knows about. In Microsoft's time-honoured fashion, this routine uses a data structure, TRasConn, which comes in no less than four different flavours according to which platform you're running on and whether it's a leap year or not (OK, I lied about that last part).

On the positive side, Microsoft have at least implemented the correct 'fallback' behaviour for the RASEnumConnections routine. In other words, if you set up the dwSize field so as to request an early version of the data structure, later versions of the code will recognise the format of data structure that's required, and return only the fields that the older code expects. This is great news as far as we're concerned, because quite frankly we don't give a monkey's armpit what device type, device name and phonebook path are being used for the connection. In a sense, all this information is redundant anyway, because knowing the phonebook entry name, we could determine everything else from the phonebook.

Accordingly, the TRASConnectionManager.Refresh method asks for (and gets) the earliest possible version of TRasConn, thereby eliminating most of the deeply messy code that would otherwise ensue. The returned list of current connections also includes a set of connection handles in the hConn field of the data structure. These are important because we need them in order to get the current status of each connection. Armed with a connection handle, we call the RASGetConnectStatus to obtain the current status of a particular connection. Notice that the TRASConStatus once again tries to furnish us with information that we know already (the device type and device name being used for the connection) but cunningly encodes the much more interesting connection status as an integer.

In order to determine whether things have changed since the last timer tick, the phonebook names and corresponding connection statuses (*stati*?) are stored in a TStringList object which is passed to the StatusChanged function. This checks the two string lists to determine whether anything has changed. If it has, the OnStatusChange event handler is fired. The code presented here is relatively unsophisticated, but with a little more work you could pass all the pertinent information back through the OnStatusChange event handler, such as the name of the phonebook entry for which the status has changed, and the new status as a descriptive string.

In order to convert Microsoft's status codes into something meaningful, I wrote the ConStateToString function. This is called by the GetStatus method which implements the Status property array. To the casual eye, many of these description strings look vaguely similar to one another, but one may as well provide a string for all the status codes currently defined. If you write an application which makes use of TRASConnectionManager (or something similar) I'd strongly caution you against writing code which expects to see certain specific status codes in a certain order. For example, depending on the timer tick frequency you use, you might miss a certain status transition and then lock up your application, waiting for something that's already happened. Also, comparing a Windows 2000 and a Windows 98 system, I've noticed that you see things on one platform that you might not see on the other. This is undoubtedly due to differences in the RAS internal implementation on the different platforms.

Caveats And Conclusions

The only real caveat is that I've tried out the various components under Windows 98 and Windows 2000, but not under Windows 95. As discussed elsewhere, I've made a conscious effort to support '95, but I don't myself have a live Windows 95 system any more. In theory, things should work as advertised, but if you find any Windows 95 related problems, then let me know and I'll pass them on.

I don't doubt that one or two folks will berate me for a certain lack of efficiency. After all, when you update a phonebook entry in the testbed program, the three property assignments to UserName, PhoneNumber and Password each require the phonebook entry properties to be read, updated, and then written out again. Fair enough, but like I said, it's much easier to do things that way than it is to generate Microsoft's idiosyncratic data structures from scratch and besides, editing phonebook entries isn't something your average PC needs to do hundreds of times a second!

There's also the issue of rampant 'featureitis'. If you study the various RAS data structures in Microsoft's specifications, you'll see that there's a huge amount of extra functionality in there (especially TRASEntry!) which I haven't 'surfaced' as component properties. At the end of the day, you'll have different ideas about what level of flexibility you want, I'm just trying to give you a foundation to start from.

In a similar vein, you might be surprised that I haven't provided code to encapsulate the important business of dialling and hanging up a connection, (ie wrappers for RASDial and RASHangup). Again, this is because you'll probably have your own ideas about how you want things to work. Part of my motivation in creating the connection manager was to build a utility that could programmatically monitor connections created by other applications and periodically warn if a connection was still present after a certain time. This is because I've often walked away from a machine then come back a few hours later to discover that it's still sat there eating away at the phone bill! In terms of dialling, RASDial will allow you to dial synchronously or asynchronously. In other words, you can have everything done inside RASDial, with the routine not returning until the connection is established, or you can provide a window to which notification messages are sent during the dial process. Unless you have a requirement to capture each and every notification message that's going, I suspect the easiest option would be to simply dial asynchronously, passing Application.Handle as the window to receive notification messages. Your application will then receive (and ignore) status notifications sent from RAS, but you'll be able to keep track of the current state of play via the connection manager component described earlier.

RAS aside, the aim of these two articles has been to demonstrate how to wrap up a Microsoft API in such a way as to fail gracefully if the API (or rather, the DLL that implements the API!) doesn't exist on the target machine. By using the base functionality provided by TRASBaseComponent, the amount of logic required in the various derived classes is greatly reduced. We don't care whether a specific API routine is implemented in RASAPI32.DLL, or in RNAPH.DLL, it all just happens automatically. Here again, the efficiency pundits might complain but, trust me, GetProcAddress is *very* fast.

Finally, I said I'd mention the mysterious PBK files used by NT to store phonebook entries. Since last time, I've been playing around with some real PBK files and, yes, just as I suspected, these are just plain ordinary .INI files with a different file extension. What's really bizarre here is the way in which Microsoft appear to have re-implemented the various .INI file handing routines within RASAPI32.DLL. Why not just use the existing routines? If you feel like tinkering around with PBK files, bear in mind that they don't actually contain the passwords associated with specific phonebook entries, although Is suspect those mysterious GUID entries have something to do with that!

For reasons of space, the source code to the testbed program isn't listed below, but everything can be found on this month's disk.

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave at TechEditor@itecuk.com