Under Construction: Talking DDE To ProgMan

by Bob Swart

When writing installation scripts or programs, one of the things we often need to do is create a new group or folder in the Windows environment and add new items to it. In this article we will develop and port to 32-bits a simple `TProgMan` component for just this purpose.

Dynamic Data Exchange, DDE in short, takes place between a client and a server. The DDE client starts the exchange by initiating a conversation with the DDE server, so that transactions for data and/or services can be sent to the server. Usually, the client terminates the conversation when it no longer needs the server's data or services, although sometimes the server can terminate the conversation as well (if the server is shut down).

A DDE server usually uses a three-level identification hierarchy, starting with the application or service name (the name with which the client starts the server), a topic name (the string which identifies a logical data context) and finally an item name (a string which identifies a particular unit of data or service requested by the client).

DDE in the Windows API is based on messages sent back and forth between the client and the server. In the early days of Windows 3.x, DDE was said to be complex, buggy and unreliable. This is why the DDE Management Library (DDEML) was invented: to shield the programmer from the trouble of the low-level Windows DDE messages. However, in my humble opinion, the original DDE concept consists of a much lower level API and far fewer primitives, so we'll be using this good old message based DDE protocol for the remainder of this article. And no, we're not going to use the native Delphi VCL DDE components either.

At the end of this article, we'll see that this old stuff still works in 32-bits, now who would've thunk that!

16-Bit ProgMan

ProgMan is a DDE server and can only talk about *itself* as a topic (it must find itself very important, since it doesn't understand any other topics). Using ProgMan for both the name of the application and the topic, we can execute a number of DDE macro commands using the `WM_DDE_EXECUTE` message. We have the following commands available:

- Create a new group,
- Delete an existing group,
- Activate, minimize, maximize or iconize the existing group,
- Create a new item in an existing group,
- Remove an existing item from an existing group.

We can also request information from ProgMan if we post it a `WM_DDE_REQUEST` message. ProgMan is able to tell us the names of the existing groups, as well as item information in the active group. In this article, we're mainly interested in a list of existing groups.

But first, let's see how we can make a DDE connection between a simple Delphi component and ProgMan itself.

DDE-Connection

In order to make a DDE connection we need to send a `WM_DDE_INITIATE` message to a window.

We specify that we send it to any window, by using `HWnd(-1)`, but we know only ProgMan will respond since we specifically ask for only the `PROGMAN` application and topic. ProgMan will respond to this message by sending one back. To do that, it needs to know the Windows handle of our initiating control. The control, in its turn, must listen for this `WM_DDE_ACK` acknowledgement message from ProgMan. The `WM_DDE_ACK` message sent by ProgMan can be used to obtain the Windows handle for ProgMan, so we have a specific return address to use for the remainder of the connection, instead of plain `HWnd(-1)` which allows every application to

➤ *Listing 1*

```
procedure TProgMan.InitiateConversation;
var ApplicationName, Topic: TAtom;
    LParamLo, LParamHi: Word;
begin
  ApplicationName := GlobalAddAtom('PROGMAN');
  Topic := GlobalAddAtom('PROGMAN');
  LParamLo := ApplicationName;
  LParamHi := Topic;
  if SendMessage(HWnd(-1), wm_DDE_Initiate, Handle, MakeLong(LParamLo,
    LParamHi)) = 0 then begin
    { success }
    GlobalDeleteAtom(ApplicationName);
    GlobalDeleteAtom(Topic)
  end
end {InitiateConversation};

procedure TProgMan.WMDDEAck(var Msg: TMessage);
{ respond to a DDE acknowledgement message }
begin
  if not Connected then begin
    { first time connection, get ProgMan HWnd }
    Connected := True;
    PMWindow := Msg.WParam;
    GetGroups; { explained later... }
  end
end {WMDDEAck};
```

eavesdrop on our conversation. The two routines in Listing 1 (on the previous page) will initiate the DDE conversation with ProgMan and receive the acknowledge message, after which the connection is said to be active.

The need to have a Windows Handle poses the first (minor) problem. For our TProgMan component to have a handle it must be derived from TWinComponent or one of its descendants.

It is a bit strange to have a non-visual component derived from TWinControl, and if we drop this TProgMan component onto a form it looks almost invisible. To fix that, we can use the component palette bitmap to show as a picture of the otherwise non-visual component. A bit like the TTable and other non-visual components. You are of course free to simply override the Paint method of the TProgMan component and provide your own interface.

Terminating an active DDE connection between our Delphi component and ProgMan can be done by posting a WM_DDE_TERMINATE message. This message can be posted by either our component or ProgMan itself and we should reply with a similar WM_DDE_TERMINATE message to indicate the connection is also terminated on the client side (so neither will send a DDE message over the connection again).

Note that the Windows API help states clearly that the WM_DDE_TERMINATE message must be posted using PostMessage, instead of sent using SendMessage, since we can get into a deadlock situation otherwise (if both ends *sent* a WM_DDE_TERMINATE message at the same time).

Hence, the code to terminate the DDE conversation between TProgMan and ProgMan is as shown in procedure TerminateConversation in Listing 2. Alternatively, ProgMan could post us a WM_DDE_TERMINATE message, so we need to respond to that as shown in the procedure WMDDETerminate in Listing 2.

Note that we need to post the WM_DDE_TERMINATE message in response to ProgMan's message if we didn't initiate the termination ourselves.

DDE Macro Command	**Syntax**
CreateGroup	[CreateGroup(NewGroupName)]
ShowGroup	[ShowGroup(GroupName, ShowCommand)]
DeleteGroup	[DeleteGroup(GroupName)]
AddItem	[AddItem(CommandLine, ProgName, Path, Icon)]
DeleteItem	[DeleteItem(ItemName)]

➤ *Table 1: DDE macro commands*

```
procedure TProgMan.TerminateConversation;
begin
  PostMessage(PMWindow, wm_DDE_Terminate, Handle, LongInt(0));
  PMWindow := 0;
end {Terminate};
procedure TProgMan.WMDDETerminate(var Msg: TMessage);
{ respond to a DDE terminate message }
begin
  if (PMWindow <> 0) and not ClosedByPM then begin
    { we're not already closing }
    ClosedByPM := True;
    PostMessage(PMWindow, wm_DDE_Terminate, Handle, LongInt(0))
  end;
  Connected := False
end {WMDDETerminate};
```

➤ *Listing 2*

```
procedure TProgMan.SendMacroString(macro: PChar; size: Byte);
var
  LParamLo, LParamHi: Word;
  MacroHandle: Word;
  MacroPtr: Pointer;
  MacroPChar: PChar;
begin
  MacroHandle := GlobalAlloc(gmem_moveable OR gmem_DDEShare, size+1);
  if MacroHandle <> 0 then begin
    MacroPtr := PChar(GlobalLock(MacroHandle));
    if MacroPtr <> nil then begin
      MacroPChar := MacroPtr;
      StrCopy(MacroPChar, macro);
      GlobalUnLock(MacroHandle);
      LParamHi := MacroHandle;
      LParamLo := 0;
      if not PostMessage(PMWindow, wm_DDE_Execute, Handle,
        MakeLong(LParamLo, LParamHi)) then begin
        GlobalFree(MacroHandle);
        MessageBeep($FFFF)
      end
    end else
      GlobalFree(MacroHandle)
  end
end {SendMacroString};
```

➤ *Listing 3*

Executing DDE Macros

As soon as we have a connection established between TProgMan and ProgMan we can execute DDE macro commands. Table 1 shows the commands which can be issued.

Note that the macros must be embedded in square brackets. The ShowCommands parameter of the ShowGroup macro can be 1 to restore or activate, 2 to iconize, 3 to maximize and 6 to minimize. For further values, refer to the Windows help.

If a DDE connection between TProgMan and ProgMan is established, executing a DDE macro is done using the SendMacroString method, see Listing 3.

Now that we have a method to send DDE macros to ProgMan to be executed, the rest is easy. For each action, we need to compose the right macro and call SendMacroString. For example, to create a new group, we need the name of the new group, compose the corresponding macro and call the procedure SendMacroString as shown in Listing 4.

Note that I use a dirty trick here to convert a String into a PChar (null terminated string): I add a #0 (null) character to the string and pass a pointer to the first character into SendMacroString. Using the byte absolute trick ensures that Len is mapped on the Length byte of a 16-bit String.

This code will not port as-is to Delphi 2. However, since I started this component months before Delphi 2 was even available, there may be more code around using these dirty non-portable tricks. We'll see how to port them later (the simple solution is to change all String definitions to ShortString).

Deleting an existing group is as easy as creating one: all we need to know is the name of the group to be deleted, and we can construct the DDE macro (Listing 4).

The ShowGroup macro has abilities to activate, maximize, minimize and iconize a given group. Activating an existing group means in fact restoring it to its proper size (ie not minimized, maximized or iconized). It will not really activate the group, but the group will get focus. See Listing 5.

To iconize, maximize or minimize a group we need a similar macro for SendMacroString, with a different final parameter (1 for activate, 2 for iconize, 3 for maximize and 6 for minimize). See Listing 5.

Adding groups or manipulating groups is nice, but it would be helpful to be able to add new program items to groups, or delete existing ones. This is done with the last two macros implemented in TProgMan, see Listing 6.

Note that AddItemToActiveGroup has two arguments: one for the CommandLine (the path to the executable file) and one for the logical name of the item. We could have added two more arguments: one for the working directory path and one for the icon. These items need to be added in the CommandLine separated by commas (as the CommandLine and Name currently are). This is left as an exercise for you, dear reader...

Requesting Group Information

Now that we're able to add and delete groups and items, it's time to let ProgMan do some talking back to us. How about a list of existing groups? (so we won't need to create a group that already exists). For this we need to send requests for data using the WM_DDE_REQUEST message, with the handle to the item Groups as LParamHi (Listing 7).

Note that we only try to request information if a DDE connection exists. Right after we've sent the request for information, ProgMan

➤ Listing 4

```
procedure TProgMan.CreateNewGroup(Name: String);
var Len: Byte absolute Name;
begin
  Name := '[CreateGroup(' + Name + ')]'#0;
  SendMacroString(@Name[1],Len)
end {CreateNewGroup};
procedure TProgMan.DeleteGroup(Name: String);
var Len: Byte absolute Name;
begin
  Name := '[DeleteGroup(' + Name + ')]'#0;
  SendMacroString(@Name[1],Len)
end {DeleteGroup};
```

➤ Listing 5

```
procedure TProgMan.Activate(Group: String);
var Len: Byte absolute Group;
begin
  Group := '[ShowGroup(' + Group + ',1)]'#0;
  SendMacroString(@Group[1],Len)
end {Activate};
procedure TProgMan.Iconize(Group: String);
var Len: Byte absolute Group;
begin
  Group := '[ShowGroup(' + Group +',2)]'#0;
  SendMacroString(@Group[1],Len)
end {Iconize};
procedure TProgMan.Maximize(Group: String);
var Len: Byte absolute Group;
begin
  Group := '[ShowGroup(' + Group +',3)]'#0;
  SendMacroString(@Group[1],Len)
end {Maximize};
procedure TProgMan.Minimize(Group: String);
var Len: Byte absolute Group;
begin
  Group := '[ShowGroup(' + Group +',6)]'#0;
  SendMacroString(@Group[1],Len)
end {Minimize};
```

➤ Listing 6

```
procedure TProgMan.AddItemToActiveGroup(CommandLine, Name: String);
var Len: Byte absolute CommandLine;
begin
  if Name <> '' then
    CommandLine := '[AddItem(' + CommandLine + ',' + Name +')]'#0
  else { command-line }
    CommandLine := '[AddItem(' + CommandLine +')]'#0;
 SendMacroString(@CommandLine[1],Len)
end {AddItemToActiveGroup};
procedure TProgMan.DeleteItemFromActiveGroup(Item: String);
var Len: Byte absolute Item;
begin
  Item := '[DeleteItem(' + Item +')]'#0;
  SendMacroString(@Item[1],Len)
end {DeleteItemFromActiveGroup};
```

will send us a WM_DDE_DATA message with the information we've requested. We need to listen for this message as well, we obtain the requested data as a pointer to a TDDEData record from the Msg.LParamLo parameter (Listing 7).

FGroups is a property of type TStringList which is used to hold the names of the groups (each separated by a carriage return when ProgMan sends them to us). Of course, it would be nice it we could somehow notify the user of the component that the data has been received, and that's what we'll be doing later in this article, when we assign an FOnDDEdata notify event to TProgMan.

Final 16-Bit Declaration

Now that we have the most important methods defined, it's time to take a look at the final declaration of the 16-bit version of TProgMan, in particular at three special properties: Active, Groups and OnDDEdata (Listing 8).

In the constructor we need to set the private fields to their initial value (Listing 9).

Active Properties

TProgMan has two data properties (Active and Groups) and one event handler property. We can toggle the value of Active, which will result in a call to the SetActive procedure, calling either BeginConversation or EndConversation as appropriate.

If the Active property is True then BeginConversation, which calls InitiateConversation to ensure that the connection is made. The names of the existing groups are requested right after the DDE connection is made, in the WMDDEAck procedure (Listing 1).

So, if you drop a TProgMan component onto a form and set the Active property to True, then click on the Groups property in the Object Inspector you will see something like Figure 1.

Note you must not change any of these groups in the *String list editor*. Indeed, when the OK button is clicked, the SetGroups procedure is called, which puts up a message dialog to tell you this is not supported. I've made SetGroups virtual, so you could add the functionality of removing, adding and modifying groups here if you wish.

Example Program

The example program (Figure 2) calls just about every method from TProgMan except for the AddItem and DeleteItem macros. To make sure the existing groups are added to the listbox at startup of the form, the following code is included in the FormCreate event handler:

```
ProgMan1.Active := True;
```

When the data is received, the

➤ *Listing 7*

```
procedure TProgMan.GetGroups;
var
  LParamLo, LParamHi: Word;
  Item: TAtom;
begin
  if Connected then begin
    Item := GlobalAddAtom('Groups');
    LParamHi := Item;
    LParamLo := CF_TEXT;
    if not PostMessage(PMWindow, wm_DDE_Request, Handle,
      MakeLong(LParamLo, LParamHi)) then begin
      GlobalDeleteAtom(Item);
      MessageBeep($FFFF) { failed }
    end
  end
end {GetGroups};

procedure TProgMan.WMDDEData(var Msg: TMessage);
{ respond to a DDE data delivery message }
var Data: PDDEData;
begin
  Data := PDDEData(GlobalLock(Msg.LParamLo));
  FGroups.Clear;
  FGroups.SetText(Data^.Value);
end {WMDDEData};
```

➤ *Listing 8*

```
Type
  TProgMan = class(TWinControl)
    constructor Create(AOwner: TComponent); override;
    destructor  Destroy; override;
    procedure BeginConversation;
    procedure EndConversation;
    procedure GetGroups;
    procedure CreateNewGroup(Name: String);
    procedure DeleteGroup(Name: String);
    procedure Activate(Group: String);
    procedure Iconize(Group: String);
    procedure Maximize(Group: String);
    procedure Minimize(Group: String);
    procedure AddItemToActiveGroup(CommandLine, Name: String);
    procedure DeleteItemFromActiveGroup(Item: String);
  protected
    FActive: Boolean;
    FGroups: TStringList;
    FOnDDEdata: TNotifyEvent;
    procedure SetActive(Value: Boolean); virtual;
    procedure SetGroups(Value: TStringList); virtual;
  published
    property Active: Boolean read FActive write SetActive;
    property Groups: TStringList read FGroups write SetGroups;
    property OnDDEdata: TNotifyEvent read FOnDDEdata write FOnDDEdata;
  private
    PMWindow: HWnd;
    Connected: Boolean;
    ClosedByPM: Boolean;
    procedure InitiateConversation;
    procedure TerminateConversation;
    procedure WMDDEData(var Msg: TMessage); message wm_DDE_Data;
    procedure WMDDEAck(var Msg: TMessage); message wm_DDE_Ack;
    procedure WMDDETerminate(var Msg: TMessage); message wm_DDE_Terminate;
    procedure SendMacroString(macro: PChar; size: Byte);
  end {TProgMan};
```

OnDDEdata event handler will be fired, which contains just the following line of code:

```
ListBox1.Items := ProgMan1.Groups;
```

That's all there is to using the TProgMan component. The assignment to the Active property will enable the connection and get the group names, so the listbox can get its items from the Groups property.

Note that 16-bit applications developed using TProgMan will work on Windows 95 and NT to communicate with the 32-bit version of ProgMan and add groups and items into the Windows environment. So, if you want to install either a 16-bit or a 32-bit version of your application depending on the customer's platform (Win 3.1 or Win95/NT), you can make one 16-bit install program to run on all platforms.

However, it would be nice to be able to use TProgMan from a 32-bit Delphi 2 program as well...

Porting To 32-Bit

Porting a component to 32-bit involves finding out which parts of the component won't work in 32-bit and replacing them. Of course, we should always try to end up with a single component that compiles in both environments, to avoid having to maintain two separate sets of source code (which would be an excellent way to introduce bugs!). To do that, we need to use conditional compilations, for example to include a 16-bit or 32-bit component palette bitmap:

```
{$IFDEF WIN32}
  {$R PROGMAN.D32}
{$ELSE}
  {$R PROGMAN.D16}
{$ENDIF}
```

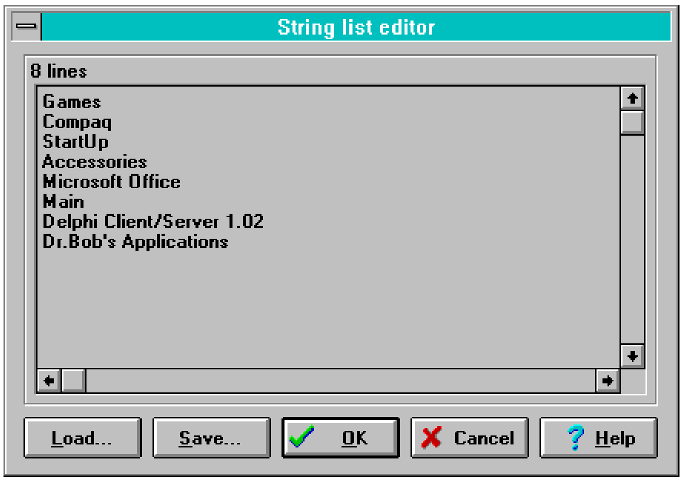
One of the first things we need to do is change every occurrence of String into ShortString. We've really been misusing the String type in our code so far, and the TProgMan code surely won't work in Delphi 2. If we change every String into a ShortString the code will compile and work in both Delphi 1 and 2. Of course, Delphi 1 doesn't know about the ShortString type, so we need to define that as well:

```
{$IFNDEF WIN32}
Type
  ShortString = String;
{$ENDIF}
```

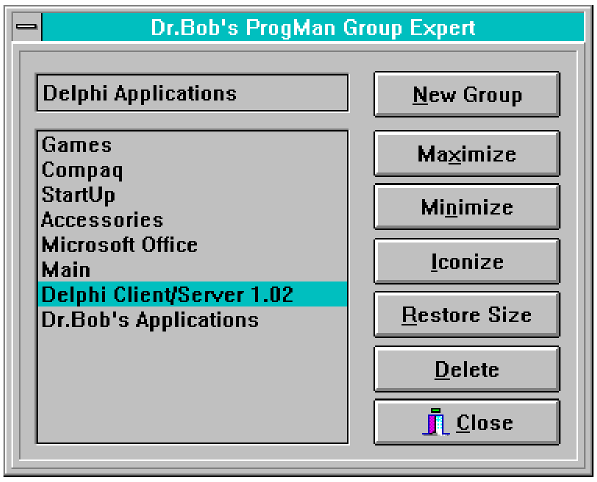
Using explicit ShortString definitions is safer than compiling with the {$H-} flag, as it is much clearer in the code what type of string is used. Using the Length byte of a Delphi 2 (long) String is a big no-no, but the new ShortString type is exactly the same as the Delphi 1 String type (255 bytes of string data, with the length in byte 0) so there's no problem.

We could rewrite the code to use only long strings, but then again, this code would only work with Delphi 2, so we need to write two sets of code. I'm not convinced that Windows 3.x will disappear any time soon and so I need to support both 16-bit and 32-bit code.

➤ *Figure 1*



➤ *Figure 2*



```
constructor TProgMan.Create;
begin
  inherited Create(AOwner);
  Height := 10;
  Width := 10;
  PMWindow := -1; { any Window }
  FActive := False;
  Connected := False;
  ClosedByPM := False;
  FGroups := TStringList.Create
end {Create};
destructor TProgMan.Destroy;
begin
  FGroups.Free;
  inherited Destroy
end {Destroy};
```

➤ *Listing 9*

Another issue that comes up is the bitmap. In 16-bit Windows I can use the LoadBitMap Windows API to load a bitmap from the resource (which is linked in with the component itself) and assign that to the TBitmap.Picture.Bitmap.Handle. In 32-bit Windows this no longer seems to work and I need to call the LoadFromResourceName method of the Bitmap property itself. See Listing 10.

I'm not sure why this has changed. It sure breaks a lot of otherwise perfectly legal code and when porting a DDE component to 32-bit this was the last place I expected problems.

32-Bit DDE

So far, we've seen a few minor porting issues. Now to the real work of porting the Windows DDE messages to 32-bit begins. In the old 16-bit world, one process can access the memory of another. DDE uses that to communicate from process to process. Behind the scenes, 16-bit DDE can be considered to be just using the clipboard to copy data from the server to the client.

In the 32-bit world of Windows 95 and NT this is no longer possible. A process cannot simply share the data and memory of another process *[See John Chaytor's article in Issue 17 for details of sharing data between applications on a 32-bit system. Editor]*. Sending messages from one process to another is still possible, but the items need to be global atoms to the entire systems, that's why we need to use GlobalAlloc and GlobalFree for our atoms.

The big problem shows its face when we look at the parameter sizes: in 16-bit DDE the lParam of the WM_DDE_DATA message contains a 16-bit data handle and a 16 bits atom. For Windows 3.x we can simply use the MakeLong function to put these two 16-bit values into one 32-bit Long value. However, in 32-bit Windows the data handle is a 32-bit value. We can't stuff more than 32 bits into a 32-bit Long, so there's no more room for the atom.

To solve this, and ease the porting pain of message-based DDE, the guys from Microsoft invented the concept of DDE parameter packing. To work correctly, a Win32 application must use the PackDDElParam function to pack the 32-bit handle and atom into an lParam parameter and the UnpackDDElParam function to remove the values.

The Win32 API also includes ReuseDDElParam, to reduce the number of memory allocations during a DDE conversation, and FreeDDElParam to free the memory which is associated with a data handle.

Packing DDE Data

So, we need to use PackDDElParam and supply the message itself as well (probably so the kernel can keep track of which packed parameters are used for which message). Luckily, this means only a single change in the call to SendMessage in the InitiateConversation procedure. Likewise, for GetGroups and SendMacroString we need to change the MakeLong call inside the PostMessage to a call to PackDDElParam as well. SendMacroString gets the same treatment. Listing 11 shows the changes in these three routines.

Allocating the atoms is still done using GlobalAddAtom and, allocating the data uses GlobalAlloc (to make it available in the global scope of the system). Deleting the Atoms is done with GlobalDeleteAtom, deleting data with GlobalFree.

Unpacking DDE Data

Packing and sending data to the DDE server is one thing. Receiving data, for example a list of existing groups, is yet another. As outlined before, we need to call UnpackDDElParam to translate the lParam into a data handle and atom we can use. For that, we need to take a close look at the TWMDDE_Data message type (defined in the Messages unit):

```
TWMDDE_Data = record { right }
  Msg: Cardinal;
  PostingApp: HWND;
  PackedVal: Longint;
  Result: Longint;
end;
```

By the way, don't trust the Delphi 2 help on this one! For some reason

➤ *Listing 10*

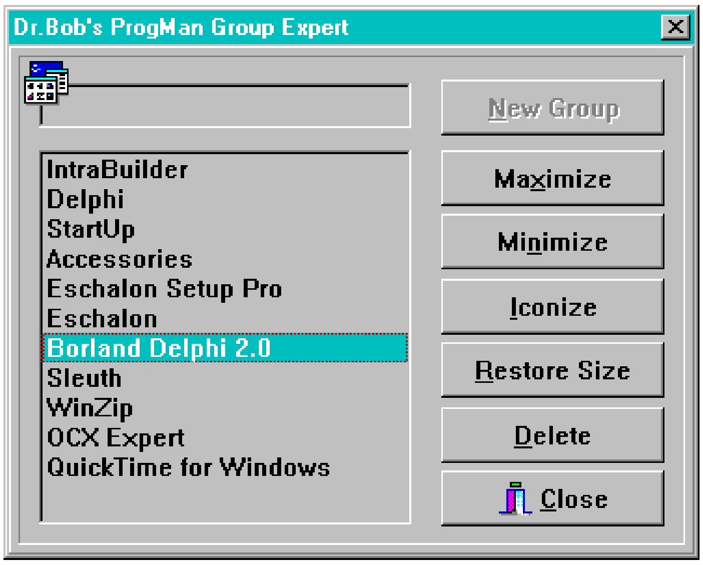
```
{$IFDEF WIN32}
 FBitmap.Picture.Bitmap.LoadFromResourceName(HInstance, 'TPROGMAN');
{$ELSE}
 FBitmap.Picture.Bitmap.Handle := LoadBitmap(HInstance,'TPROGMAN');
{$ENDIF}
```

➤ *Listing 11*

```
{ change to InitiateConversation: }
if SendMessage(HWnd(-1), wm_DDE_Initiate, Handle,
{$IFDEF WIN32}
  PackDDElParam(wm_DDE_Initiate, ApplicationName, Topic)) = 0
{$ELSE}
  MakeLong(ApplicationName, Topic)) = 0
{$ENDIF}

{ change to PostMessage: }
if not PostMessage(PMWindow, wm_DDE_Request, Handle,
{$IFDEF WIN32}
  PackDDElParam(wm_DDE_Request, CF_TEXT, Item))
{$ELSE}
  MakeLong(CF_TEXT, Item))
{$ENDIF}

{ change to SendMacroString: }
if not PostMessage(PMWindow, wm_DDE_Execute, Handle,
{$IFDEF WIN32}
  PackDDElParam(wm_DDE_Execute, 0, MacroHandle))
{$ELSE}
  MakeLong(0, MacroHandle))
{$ENDIF}
```



IntraBuilder
Delphi
StartUp
Accessories
Eschalon Setup Pro
Eschalon
Borland Delphi 2.0
Sleuth
WinZip
OCX Expert
QuickTime for Windows

➤ *Figure 3*

```
procedure TProgMan.WMDDEData(var Msg: TWMDDE_Data);
{$IFDEF WIN32}
 var
   DataHandle,DataTopic: PUINT;
{$ENDIF}
 var
   Data: PDDEData;
begin
  inherited;
 {$IFDEF WIN32}
  if UnpackDDElParam(Msg.Msg, Msg.PackedVal, DataHandle, DataTopic)
    and (DataHandle <> nil) then
    Data := PDDEData(GlobalLock(DataHandle^))
  else
    Data := nil; { in Win32 design mode... }
 {$ELSE}
  Data := PDDEData(GlobalLock(Msg.Data));
 {$ENDIF}
  FGroups.Clear;
  if Data <> nil then
    FGroups.SetText(Data^.Value);
 {$IFDEF WIN32}
  if not PostMessage(PMWindow, wm_DDE_Ack, Handle, Msg.PackedVal) then
    FreeDDElParam(Msg.Msg, Msg.PackedVal);
 {$ENDIF}
  if Assigned(FOnDDEdata) then
    FOnDDEdata(Self)
end {WMDDEData};
```

➤ *Listing 12*

it reports another layout as follows:

```
 TWMDDE_Data = record { wrong }
     Msg: Cardinal;
     PostingApp: HWND;
     Data: THANDLE;
     Item: Word;
     Result: Longint;
   end;
```

This is clearly wrong! The problem is precisely what the Microsoft DDE team had to solve: a 32-bit Thandle and Item do not fit into one 32-bit PackedVal long integer.

First, we need to use UnpackDDElParam to unpack the Msg.PackedVal into the two separate data handles and data topic. When we have a data handle, we need to make sure that it is not nil. If we have indeed received and unpacked a valid data handle, we need to call GlobalLock to be able to re-access the data in the handle (pointer) itself.

After the GlobalLock, we are free to de-reference the data and get our information from it (in this case the list of existing groups). It should be clear that this code is very delicate and one mistake is enough to get that big blue screen in Windows 95 (the exception 0D, otherwise known as the good old General Protection Fault).

The code, including three sections that depend on conditional compilation, is shown in Listing 12.

For some reason, the code above works fine, *except* in design mode in Delphi 2. I'm not sure why it doesn't work, but it looks like the DDE message with the requested data is already eaten (unpacked and freed) by someone else, probably the Delphi 2 IDE itself. When running outside the Delphi IDE, the component behaves correctly, as can be seen in the 32-bit version of the example application shown in Figure 3.

Conclusion

We've seen that using Delphi we can create a simple yet powerful component to encapsulate the DDE connection and macros with ProgMan. Using inheritance, this component could be made even more powerful by adding missing features such as manipulation of new items and group names at design time. While this component only works correctly in design time with Delphi 1, at run-time it can be used with all versions of Delphi.

The DDE principles used for the TProgMan component, including parameter packing, can be used to write a DDE interface to any DDE server, so as usual the end (of this article) is only the beginning...

Full source code for TProgMan and the example program is on this month's disk.

Bob Swart (home.pi.net/~drbob/) is a professional knowledge engineer technical consultant using Delphi and C++ for Bolesian (www.bolesian.com), free-lance technical author for The Delphi Magazine, and co-author of *The Revolutionary Guide to Delphi 2*. Bob is now co-working on a new book about Delphi and the internet (*Delphi Internet Solutions*). In his spare time, Bob likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 2.5 year old son Erik Mark Pascal and his newborn daughter Natasha Louise Delphine.