Custom Shell Verbs Using DDE

by Brian Long

This is another in the occasional series of articles featuring questions sent to *The Delphi Clinic* which warrant an answer too long and involved for the normal column. The relevant reader's question was:

I have been trying for a while now to set up an (admittedly old-fashioned) DDE conversation between Windows Explorer and my application. I am trying to use Explorer's `File Types` *dialog page (from the* `View | Options...` *menu item) to add a DDE handler for a custom file extension. However, I can't get it to work and I cannot find any examples that show how to use the DDE components.*

The core problem is that I want to double-click a file with my extension and open it into my application, and maybe add a right-click menu item for printing like Microsoft Word does with .DOC files (which I notice it does using DDE). Any suggestions?

My first reaction as I read this question was: 'DDE? No one uses that any more, do they? Surely there's a better solution to whatever the problem is...' However, by the time I had finished reading the question, I had changed my attitude and decided this is a great question.

The currently fashionable way to add items to a file's context menu in Windows Explorer is to write a shell extension object. We have had various articles about COM shell extensions in past issues of *The Delphi Magazine* (for example, Dave Jewell's article from Issue 15, which was superseded by a later one in Issue 30). There is also an example shell extension COM object in the ContMenu.dpr project in Delphi's Demos\ActiveX\ShellExt directory (Delphi 4 and later).

Whilst this is all fine, it strikes me that writing a shell extension seems to be rather more involved than writing a DDE handler (often called a DDE server). Implementing all those interfaces in order to get a menu item is an awful lot of work, and then you pass a variety of command lines to the application. If you want documents opened into a running copy of the program, if it is present, you have to invent some clever scheme where the freshly invoked copy communicates the requested document name to the original copy.

Also, if massive, commercially successful, applications such as Microsoft Word and Excel are happy to use DDE to implement their Explorer context menu items, there is no reason why we should shy away from using it too. So let's get started.

How Does DDE Work?

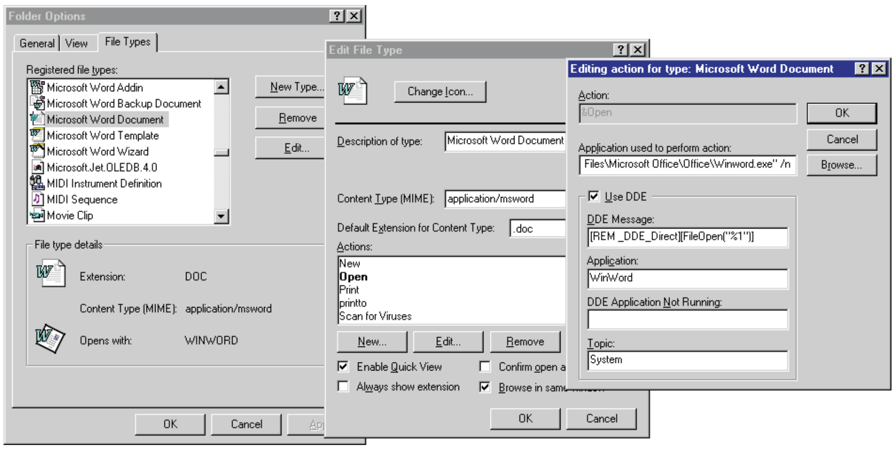
Firstly, a quick overview of DDE as it relates to this problem (which means that I'll be leaving out any DDE stuff that has no bearing on this question). DDE is a means of communicating information between two consenting applications. The application that initiates this communication (or DDE conversation) is called the DDE client, and the DDE client wants something. The other application in the conversation is the DDE server, and it has something to offer, in this case one or more ways of processing a file.

To establish the communication, the DDE client must know which DDE service it needs to communicate with. Each DDE server implements a DDE service, and the service often has the same name as the executable's base name. In fact, a Delphi DDE server makes sure the DDE service defaults to the executable name (although I outlined how this could be changed way back in February 1996, in the *Service Not Good Enough* entry from *The Delphi Clinic* in Issue 6). Programmatically, a DDE server could report its own DDE service by displaying `ddeMgr.AppName` (`ddeMgr` is defined in the `DDEMan` unit).

When trying to establish the conversation, the DDE client will first see if the service is already available (meaning the DDE server is running) and if not will launch the DDE server to get the service. Within any DDE service, the DDE server implements one or more topics (you can think of them as different DDE topics of conversation). Microsoft recommend that you implement a `System` topic, but this is up to you.

The client must choose an appropriate topic, and can then execute macros within that topic. A macro is just a command formed as a string of characters. The DDE server will try and interpret this macro in order to execute some

➤ *Figure 1: The settings for opening a Word document.*



Registered file types:
Microsoft Word Addin
Microsoft Word Backup Document
Microsoft Word Document
Microsoft Word Template
Microsoft Word Wizard
Microsoft.Jet.OLEDB.4.0
MIDI Instrument Definition
MIDI Sequence
Movie Clip

File type details
Extension: DOC
Content Type (MIME): application/msword
Opens with: WINWORD

Description of type: Microsoft Word Document
Content Type (MIME): application/msword
Default Extension for Content Type: .doc
Actions:
New
Open
Print
printto
Scan for Viruses

[✓] Enable Quick View
[ ] Confirm open a
[ ] Always show extension
[✓] Browse in same

Editing action for type: Microsoft Word Document
Action: &Open
Application used to perform action: Files\Microsoft Office\Office\Winword.exe" /n
[✓] Use DDE
DDE Message: [REM _DDE_Direct][FileOpen("%1")]
Application: WinWord
DDE Application Not Running:
Topic: System

appropriate code. Other DDE servers may also have many data items within each topic that can be linked to, but that is outside the scope of this article.

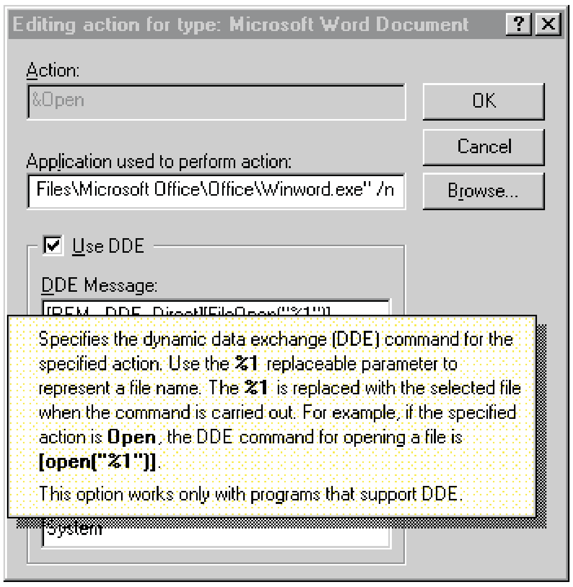
Our goal is to set up a file association with some custom verbs (commands on the Windows Explorer context menu), each of which will execute a DDE macro within a specified topic in a specified service, implemented by a Delphi application.

The ultimate goal is to have such an application that can be installed and will have details of its file association set up during installation, but let's not get ahead of ourselves. For now, let's look at how to manually set up a file association that describes supported DDE macros.

File Associations And DDE

The best thing to do to start with is to look at an existing file association and see what it has set. From a copy of Windows Explorer, drop down the `View` menu and choose either `Options...` or `Folder Options...`, depending which is available. Select the `File Types` page, scroll down the list of registered file types and locate *Microsoft Word Document*.

Press the `Edit...` button to see what settings it has and you will see there are several actions listed. The default one is in bold (`Open`) and if you select it and press `Edit...` you can see the settings for it (see Figure 1).



Editing action for type: Microsoft Word Document

Action: &Open

Application used to perform action: Files\Microsoft Office\Office\Winword.exe" /n

☑ Use DDE

DDE Message: [REM _DDE_Direct][FileOpen("%1")]

Specifies the dynamic data exchange (DDE) command for the specified action. Use the **%1** replaceable parameter to represent a file name. The **%1** is replaced with the selected file when the command is carried out. For example, if the specified action is **Open**, the DDE command for opening a file is **[open("%1")]**.

This option works only with programs that support DDE.

System

➤ *Figure 2: Windows suggesting how to write DDE messages.*

These settings state that when a .DOC file needs to be opened, WinWord.exe (which is specified with a full path) will be used to do the job. The checkbox says that a DDE conversation will cause the file to be opened, rather than simply passing the file name on the command-line.

Taking the DDE group box fields out of order, the second one, marked `Application:`, specifies the DDE service. If you leave this field blank, it assumes the service is the application name (from the `Application used to perform action:` field above) without the path or extension.

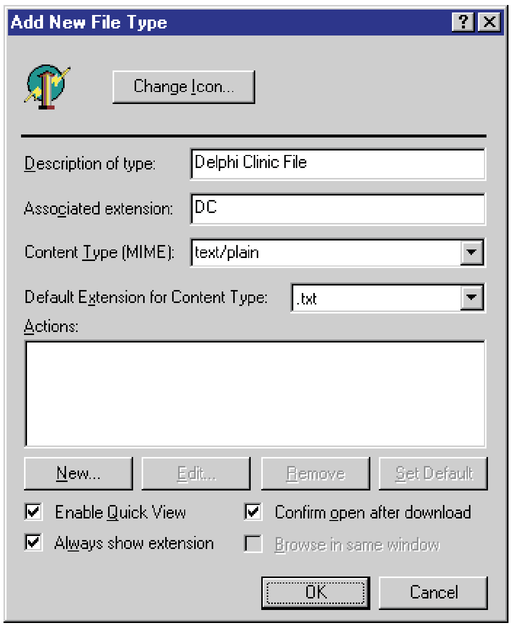
The fourth field specifies the DDE topic which, if left blank, is assumed to be `System`.

The first field in the group contains the DDE macro (referred to as a DDE message) that will be executed to perform the action (of opening the document in this case). In the message, the string `%1` is substituted by Explorer (or by the `ShellExecute` and `ShellExecuteEx` APIs, which actually do the work with file associations) for the selected file.

The macro may potentially be made up of multiple individual commands to the program and there is a convention where each command looks like a pseudo-C function call and is surrounded by square brackets. For example, Figure 1 shows that Word understands a `FileOpen` command in the DDE message that takes a file name in double quotes.

It should be stressed that this is just a convention, not a rule, despite the context help for these edit boxes suggesting the general syntax is fixed (see Figure 2). The DDE macro format is entirely up to you, as your installation will set up all these details (so the user won't have to worry about them) and it will eventually be down to your application code to parse the macro and get the appropriate information out. In fact, as Figure 2 shows, the suggested DDE macro command for opening a file is `open`, but Word uses `FileOpen`, which is instantly inconsistent.

The third and final field allows you to specify a different DDE macro to execute in the case where the DDE server is not already running. For example, the normal macro used by Word for a `print` verb is a set of three commands: `FileOpen`, `FilePrint` and `DocClose`. These commands open the specified document, print it and close the document leaving Word open. However, the `DDE Application Not Running:` field uses these slightly different commands: `FileOpen`, `FilePrint` and `FileExit`. These open the document, print it and then terminate the Word session. If you leave the field blank, the macro from the `DDE Message:` field will be used.



Add New File Type

Description of type: Delphi Clinic File

Associated extension: DC

Content Type (MIME): text/plain

Default Extension for Content Type: .txt

Actions:

☑ Enable Quick View

☑ Confirm open after download

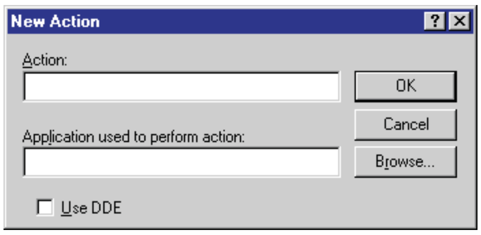
☑ Always show extension

☐ Browse in same window

➤ *Figure 3: Setting up a file association.*

Making A DDE Server

Now we know the underlying DDE principles, let's start work on an application that can act as a DDE server. Make a new project and save it (the sample project on the disk is called FileAssoc.dpr).



➤ *Figure 4: Preparing to set up a new context menu verb.*

Unless we change it, this means the application's DDE service will be `FileAssoc`.

To set up a DDE topic, all you need to do is drop a `TDdeServerConv` component on the form from the `System` page of the Component Palette. If you prefer, you can make a data module and drop the DDE component there to save cluttering up your form.

The name of the component will determine the topic name, so, to make a `System` topic, set the component's name to `System`. The topic must be able to support macro execution, so we need to make an `OnExecuteMacro` event handler for it. The event handler is passed two parameters, `Sender` and `Msg`. `Sender` will be the component itself and `Msg` will be the DDE macro passed as a `TStrings` object.

Unfortunately, the component does absolutely no parsing of the DDE macro that comes through from the client. We get exactly what was sent without any change and can do with it what we like. To test the general idea, just put a simple statement in the event handler that displays the DDE macro in a message box.

A Custom File Association

With this test code in place, let's set up a file association for the application. For this test the extension .DC will be used for our application files, and a file with this extension will be assumed to contain simple text. From the `File Types` page of the Explorer options dialog we saw before, choose `New Type...` and then fill in the dialog. Figure 3 shows an example with a custom icon for the file type, a description, and an indication that .DC files contain plain text information.

Set the checkboxes at the bottom of the dialog as you like. In my case I have specified that Quick View will be available on the context menu for one of these files (Quick View can show the content of a text file, after all). I have also made sure that the file extension will be displayed, even when Explorer is set to hide extensions of known file types (the default setting).

The `New...` button can now be used to add new verbs (or actions or commands, depending which piece of Microsoft terminology you want to use). When you press the button, the dialog starts life with only two fields plus an unchecked checkbox (see Figure 4).

The `Action:` field is where you specify the verb you want added to the file's Explorer context menu (and also available to `ShellExecute` and `ShellExecuteEx`). You can enter the menu item's caption here, including ampersand characters as you would in a Delphi menu item's `Caption` property, but beware: if you deploy your application to other countries, you may need to cater for translating these strings accordingly.

Because of this, Microsoft set up a number of *canonical verbs*. If you enter the name of one of these verbs, the context menu item caption will be automatically set up by Windows, no matter what country the application runs in. The canonical verbs include `open`, `print`, `explore`, `find`, `openas` and `properties`. Most of these are self-explanatory except `openas`, which corresponds to the `Open With...` menu item.

We should use the `open` and `print` canonical verbs to answer the question. In both cases, browse to find the compiled version of the application and check the `Use DDE` checkbox to expand the dialog, making it look more like the one in Figure 2. Specify the DDE service and topic in the appropriate fields, along with a DDE macro of your choice to represent the action.

➤ *Figure 5: A custom context menu verb.*

For the purposes of consistency I'll try using the convention suggested by the Windows help, as you can see in Figure 5. However, at this stage it does not matter what you put in there, since FileAssoc.exe only shows the macro string in a message box.

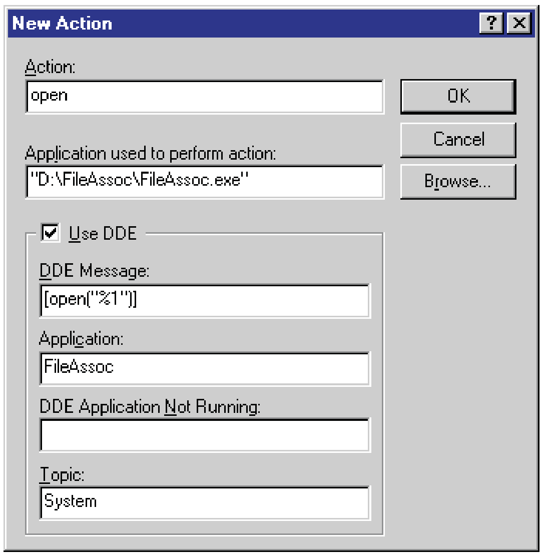
When you have added the two actions, make sure the `open` action is set as the default with the `Set Default` button, and so displays as bold in the list (like Word's `open` action in Figure 1). This means that when someone double-clicks your file, or selects it and presses `Enter`, the file will be opened (or the `open` verb will be executed, which will amount to the same thing by the time we have finished).

Does DDE Work?

And now to test the program. Create a text file in any directory, then change its extension from .TXT to .DC. Next, right-click the file in Explorer and note that the option for Quick View is visible along with our two custom verbs (see Figure 6), but beware of a

```
procedure
  TForm1.SystemExecuteMacro(
  Sender: TObject; Msg: TStrings);
begin
  ShowMessage(Msg.Text)
end;
```

➤ *Listing 1: Showing which macros your DDE clients execute.*



Action: open

Application used to perform action: "D:\FileAssoc\FileAssoc.exe"

☑ Use DDE

DDE Message: [open("%1")]

Application: FileAssoc

DDE Application Not Running:

Topic: System

possible problem when choosing them.

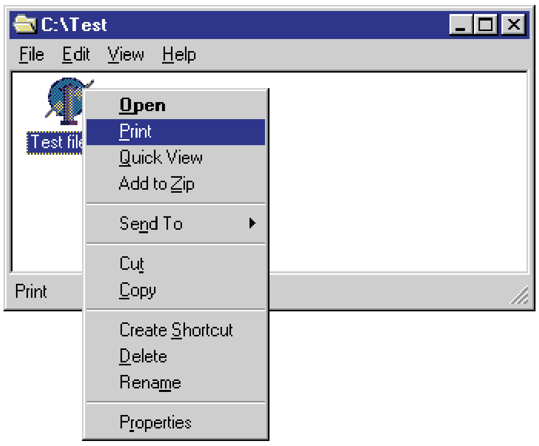
There is a flaw in the implementation of `TDdeServerConv` in that it does not distinguish between runtime and design-time. The upshot of this is that the design-time instance of the component that is sitting on your form or data module in the Delphi IDE is quite likely to take part in the conversation from Windows Explorer. This means that your application possibly won't even be executed (instead, Delphi will be brought to the foreground). You won't get very far like this, so if you get the problem you must close the offending designer. Select the relevant module and choose `File | Close` (remember you can get any closed project unit back with `Ctrl+F12`, and any form and unit back with `Shift+F12`).

Now you can test the new verbs. Figure 7 shows the message box I get when double-clicking my test file, or right-clicking and pressing `Enter`. As you can see, the `%1` placeholder has been replaced by the fully qualified file name.

Implementing The Application

To be able to support useful macros we need an application that does something, so our very simplistic application needs something of a makeover. The questioner referred to an MDI application, so that's what we will use. Rather than go through all the details of what gets added to the project, I'll give an overview. The finished FileAssoc.dpr on the disk has an MDI form with a menu and toolbar both of whose functionality are implemented through a number of actions.

The application revolves around opening text files (with .DC extensions) and the MDI child form has a rich edit control on to display the text file content. The main form `File` menu has options for opening a file, printing the current file, closing the current file and exiting. These options will all be supported by corresponding DDE commands and so Explorer will ultimately be able to invoke their functionality.

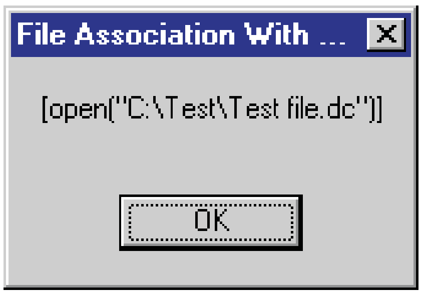


➤ *Figure 6: Custom context menu verbs at last.*

Distinguishing Macros

The next job is to enhance the DDE macro code to try and understand what the program is being asked to do. Parsing strings into understandable chunks is not something I've had a need to do before, so bear with me. My approach may not be the most appropriate or optimal. In fact, maybe I should have read up on the subject in past issues of *The Delphi Magazine* before embarking on implementing the code. A quick search on *The Delphi Magazine Collection 2000* CD for the strings *parser* and *parsing* yielded many past articles.

The code in the FileAssoc.dpr project tries to mimic what Word does with its DDE commands. Whilst I pointed out that the DDE macro format as laid out in Figure 1 was just a convention, I have decided to follow the convention. Supported commands must be placed in square brackets and a macro can contain any number of such commands. Each command consists of a command name followed by a pair of parentheses. These can either contain nothing, or a file name surrounded by double quote characters. This could be extended in a more ambitious version of the application to support other parameters.



[open("C:\Test\Test file.dc")]

➤ *Figure 7: The DDE macro being passed through to the server.*

In order to interpret a given macro string, the `OnExecuteMacro` event handler first passes the `TStrings` object it gets to another routine called `MassageCmds` which massages it into a more usable form.

Recall that the `TStrings` object seems to get an unparsed single string. This means that even if multiple commands are passed in from Explorer, they will all be passed in as the first string in the `TStrings` object because each command is specified in the same string without carriage returns. `MassageCmds`,

➤ *Listing 2: MassageCmds places each DDE command on a separate line.*

```
//Turn string containing possibly many commands in square brackets into multiple
//strings, each containing one command, without square brackets
procedure MassageCmds(Cmds: TStrings);
var
  S: String;
  OpenCmd, CloseCmd: Integer;
begin
  S := Trim(Cmds.Text);
  Cmds.Clear;
  while Length(S) > 0 do begin
    OpenCmd := Pos('[', S);
    CloseCmd := Pos(']', S);
    if (OpenCmd < CloseCmd) and (OpenCmd >= 1) then begin
      Cmds.Add(Trim(Copy(S, OpenCmd+1, CloseCmd-OpenCmd-1)));
      Delete(S, OpenCmd, CloseCmd-OpenCmd+1);
      S := Trim(S)
    end else
      Break
  end;
end;
```

```
type
  TCommandType = (ctNone, ctOpen, ctPrint, ctClose, ctExit);
var
  Commands: array[TCommandType] of String = ('', 'Open',
    'Print', 'Close', 'Exit');
//Return a TCommandType value corresponding to a specified
// command string
function StrToCommand(const Cmd: String): TCommandType;
var
  Idx: TCommandType;
begin
  for Idx := Succ(Low(TCommandType)) to High(TCommandType) do
    if CompareText(Cmd, Commands[Idx]) = 0 then begin
      Result := Idx;
      Exit
    end;
  Result := ctNone;
  MessageDlg(Format('Unknown DDE command "%s"', [Cmd]),
    mtError, [mbCancel], 0)
end;
//For a given command string, return the command
//type and specified filename, if present
function GetCommandAndParameter(CmdText: String; var Command:
  TCommandType; var Parameter: String): Boolean;
var
  OpenParens, CloseParens: Integer;
begin
  Result := True;
  OpenParens := Pos('(', CmdText);
  CloseParens := Pos(')', CmdText);
  if (OpenParens < CloseParens) and
   (OpenParens > 1) then begin
    Command :=
      StrToCommand(Trim(Copy(CmdText, 1, OpenParens-1)));
    if Command = ctNone then
      Result:= False
    else begin
      Parameter := Copy(CmdText, OpenParens+1,
        CloseParens-OpenParens-1);
      Parameter := Trim(StringReplace(Parameter, '"', '',
        [rfReplaceAll]));
    end
  end
end;
```

➤ *Listing 3: Identifying the supported DDE commands.*

which is shown in Listing 2, takes a `TStrings` object and searches for strings surrounded by square brackets. Each one found is added as a separate string stripped of its square brackets. So all the passed DDE commands are placed on separate lines.

With a more manageable collection of DDE commands, the next task is to identify each individual command we support and respond to it if it arrives (with its parameter, if it has one). Listing 3 shows the code that does the job. An enumerated type lists a token (identifier) for each command (along with an error value of `ctNone`) and an array holds the textual version of each DDE command as it will be sent from a DDE client.

The `GetCommandAndParameter` function takes a DDE command string as input and has two `var` parameters for output, one for the command token, and one for any parameter passed in brackets. The command name is deemed to be anything to the left of the open parenthesis, and is passed to a helper routine `StrToCommand` to translate between the textual name and a command token. The parameter is taken as anything between the brackets.

The `OnExecuteMacro` handler now has to call `MassageCmds` to tidy up the `TStrings` object, and then call `GetCommandAndParameter` once for each string in the `TStrings` object. Once it has the command token it uses a `case` statement to decide how to proceed (see Listing 4).

I should mention that the code in the `actOpen` action will only invoke the file open dialog (`dlgOpenFile`) if its `FileName` property is blank. Since the code in Listing 4 sets `FileName` to the file that needs to be opened, no dialog will be displayed when the DDE command is executed.

I should perhaps also mention that the printing option does not really print the file. Instead it produces a message box saying that feature is not implemented.

```
procedure TMainForm.SystemExecuteMacro(Sender: TObject; Msg: TStrings);
var
  I: Integer;
  Cmd: TCommandType;
  Parameter: String;
begin
  //Here is where we parse the DDE Message (or macro) and act on its commands
  MassageCmds(Msg);
  for I := 0 to Msg.Count - 1 do
    if GetCommandAndParameter(Msg[I], Cmd, Parameter) then begin
      case Cmd of
        ctOpen:
          begin
            dlgOpenFile.FileName := Parameter;
            actOpen.Execute
          end;
        ctPrint: actPrint.Execute;
        ctClose: actClose.Execute;
        ctExit: actExit.Execute
      end
    end
end;
```

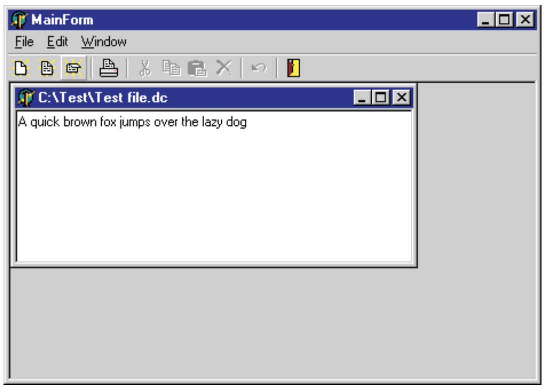
➤ *Listing 4: Acting on the DDE commands.*

Testing The Macros

Now we can test this properly. The `open` verb from Explorer should successfully open the file into a new MDI child window now (as shown in Figure 8). However, for the `print` verb, I recommend going back to the options in Explorer. Since the application supports several DDE commands, we can tailor the behaviour to different scenarios, like Word does.

If the DDE server is running and a file needs printing, it can be loaded, printed and closed. If the server is not running, the file can be loaded into the newly executed program, printed, and then the server can be terminated. You can get this behaviour by using the DDE commands available as shown in Figure 9. Notice that `open`,

➤ *Figure 8: A file opened in the DDE server.*



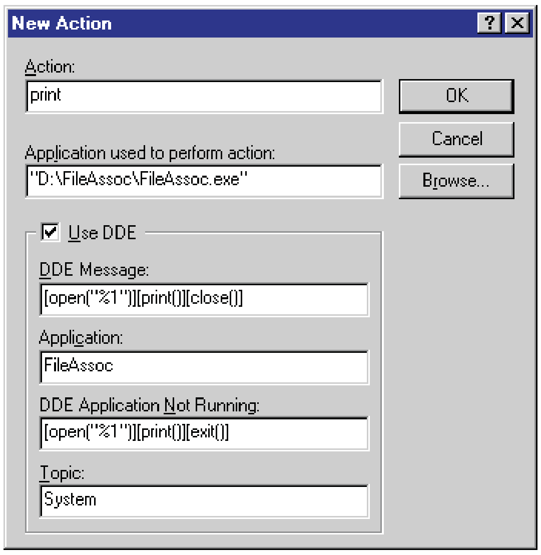
A quick brown fox jumps over the lazy dog `print` and `close` are used if the server is already running, but `open`, `print` and `exit` are used otherwise.

Testing Verbs Programmatically

As well as testing from Windows Explorer, we can test that the verbs are accessible through the Shell programming interface (the routines in the `ShlAPI` import unit). Both `ShellExecute` and `ShellExecuteEx` allow you to specify a verb to apply to the file (where `nil` means the default `open` verb). `ShellExecute` has an `lpOperation` parameter and `ShellExecuteEx` has a field in the record that gets passed to it called `lpVerb`. Listing 5 shows some simple code that tests the `print` verb against a test .DC file. Both APIs worked well in my testing.

Custom File Associations And Setup

Now we are at a point where the file association works just fine and the DDE commands are successfully passed to the program and acted upon by it. The final challenge is to get the file association set up programmatically, so we do not have to rely on the user to do it through Explorer as we did.


➤ *Figure 9: Setting up a custom print verb.*

New Action

Action: print

Application used to perform action: "D:\FileAssoc\FileAssoc.exe"

OK | Cancel | Browse...

[✓] Use DDE

DDE Message: [open("%1")][print()][close()]

Application: FileAssoc

DDE Application Not Running: [open("%1")][print()][exit()]

Topic: System

The details regarding file associations and their storage in the Windows registry are given in articles by Dave Jewell in Issues 36 and 37, so I won't go into the subject here. You can also find information in the Platform SDK documentation on the Microsoft Developer Network Library CD or available through Microsoft's website. The path through the documentation goes: *User Interface services, Windows Shell, Shell Programmer's Guide, Shell Basics, Extending The Shell, Extending Context Menus.*

In order to set up file associations and associated verbs you can use the two routines shown in Listing 6 (from the project MakeFileAssoc.dpr). `MakeAssoc` sets up the basic file association. It takes a file extension (which must include the full stop prefix) and a file class. The file class is a custom identifier used in the registry to hold information about the file association, for example the file class for .DC files could be `DC_File`.

The file extension is used as the name of a registry key under `HKEY_CLASSES_ROOT` which specifies the file class. The file class is then used as the name of another registry key, which has all the command information stored below it.

Other parameters to `MakeAssoc` include the file description and a reference to the icon that should be used for these files. This icon reference is a string that contains an executable file followed by a comma and a zero-based number

➤ *Listing 5: Testing the verbs programmatically.*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShellExecute(Handle, 'print', 'c:\Test\Test file.dc', nil, nil, SW_SHOWNORMAL)
end;
procedure TForm1.Button2Click(Sender: TObject);
var
  SEI: SHELLEXECUTEINFO;
begin
  FillChar(SEI, SizeOf(SEI), 0);
  SEI.cbSize := SizeOf(SEI);
  SEI.fMask := SEE_MASK_FLAG_DDEWAIT;
  SEI.Wnd := Handle;
  SEI.lpVerb := 'print';
  SEI.lpFile := 'c:\Test\Test file.dc';
  SEI.nShow := SW_SHOWNORMAL;
  ShellExecuteEx(@SEI);
end;
```

➤ *Listing 6: Setting up file associations.*

```
uses
  ComObj, ShlObj;
procedure MakeAssoc(const FileExt, FileClass, Description,
  DefaultIcon: String; AlwaysShowExt, QuickView: Boolean);
begin
  if (Length(FileExt) = 0) or (FileExt[1] <> '.') then
    raise Exception.Create('Invalid file extension');
  CreateRegKey(FileExt, '', FileClass);
  CreateRegKey(FileClass, '', Description);
  if DefaultIcon <> '' then
    CreateRegKey(FileClass + '\DefaultIcon', '',
      DefaultIcon);
  if AlwaysShowExt then
    CreateRegKey(FileClass, 'AlwaysShowExt', '');
  if QuickView then
    CreateRegKey(FileClass + '\QuickView', '', '*');
  SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, nil, nil);
end;
procedure MakeAssocVerb(const FileExt, Verb, VerbCaption,
  VerbCommand: String; UseDDE: Boolean; const Service, Topic,
  Macro, MacroNotRunning: String);
var
  FileClass: String;
begin
  if (Length(FileExt) = 0) or (FileExt[1] <> '.') then
    raise Exception.Create('Invalid file extension');
  FileClass := GetRegStringValue(FileExt, '');
  if FileClass = '' then
    raise Exception.Create('File extension not registered');
  CreateRegKey(FileClass + '\shell\' + Verb, '',
    VerbCaption);
  CreateRegKey(FileClass + '\shell\' + Verb + '\command', '',
   VerbCommand);
  if UseDDE then begin
    CreateRegKey(FileClass + '\shell\' + Verb + '\ddeexec',
      '', Macro);
    CreateRegKey(FileClass + '\shell\' + Verb +
      '\ddeexec\Application', '', Service);
    CreateRegKey(FileClass + '\shell\' + Verb +
      '\ddeexec\Topic', '', Topic);
    if MacroNotRunning <> '' then
      CreateRegKey(FileClass + '\shell\' + Verb +
        '\ddeexec\ifexec', '', MacroNotRunning);
  end;
  SHChangeNotify(SHCNE_ASSOCCHANGED, SHCNF_IDLIST, nil, nil);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MakeAssoc('.DC', 'DC_File', 'Delphi Clinic File',
    'D:\FileAssoc\FileAssoc.exe,0', True, True);
  MakeAssocVerb('.DC', 'open', '',
    '"D:\FileAssoc\FileAssoc.exe" %1', True, 'FileAssoc',
    'System', '[open("%1")]', '');
  MakeAssocVerb('.DC', 'print',  '',
    '"D:\FileAssoc\FileAssoc.exe" %1', True, 'FileAssoc',
    'System',
    '[open("%1")][print()][close()]',
    '[open("%1")][print()][exit()]');
end;
```

➤ *Listing 7: Setting up our file association.*

that indicates which icon in the executable to use. Two `Boolean` parameters are also expected that specify whether the file extension should always be displayed and whether the Quick View context menu should be available.

`MakeAssocVerb` is used to set up individual verbs for the file type. It takes the file extension for the verb, the verb name and caption (which should be blank to indicate either a canonical verb or one whose menu item caption matches the verb name). The command for the verb is then passed, followed by a `Boolean` parameter that dictates whether DDE will be used. Assuming this parameter is `True`, you can then specify the DDE service, topic and macro along with the macro to use if the server is not running (if any).

Some code that sets up the file association described in this article is shown in Listing 7.

Summary

After all this hard work, we now have knowledge of how to add DDE server capabilities into your application to support custom verbs, displayed as custom context menu items in Windows Explorer. We also now know how to programmatically set up a custom file association in the Windows registry that defines such custom verbs, along with their associated DDE commands. And all of this without a sniff of COM!

Brian Long is a freelance trainer and problem solver specialising in Delphi and C++Builder work. Visit www.blong.com or email him on brian@blong.com