Win32 Inter-Process Communication

by Brian Long

Applications often need to exchange data with each other. They may well do it indirectly, using a database system, and this approach is commonly used. However, applications may also need to communicate directly with one another. Since Win32 partitions applications into their own separate address spaces, this is rather more difficult than it was in 16-bit Windows.

This article looks at several options for Inter-Process Communication (or IPC) in the Win32 environment and implementing them in Delphi applications.

There are other mechanisms we won't have space to look into, or have been covered before in *The Delphi Magazine*. For example, the clipboard provides an easy way for applications to communicate, although programmatic use of the clipboard can destroy any data the user stores there. The Delphi `ClipBoard` object supports text, pictures and components and can readily deal with other formats.

Dynamic Data Exchange, or DDE, is showing its age now, but if an application supports no better IPC mechanism than DDE, it's easy to implement using the components on the `System` page of the palette.

COM allows an application's objects (and their data) to be accessed by other applications, through well-defined sets of functions called interfaces (supported in Delphi since version 3). In particular, to get an arbitrary block of data from one application to another, you can use a `Variant` byte array as shown by Steve Teixeira in his *COM Corner* column in Issue 44. Also, OLE (a COM technology) allows an application to support compound documents containing linked or embedded data. When selected, the appropriate application automatically starts to allow editing of the data (see the `TOleContainer` component). COM application development has been discussed in many past articles, so we won't go into the subject here.

RPC (Remote Procedure Calls) and Windows Sockets also provide mechanisms for talking to applications on other machines (potentially running a different OS).

Having (b)rushed through these IPC options, let's look in rather more detail at some more.

Windows Messages

Back in Issue 45, David Baer discussed Windows messages primarily from the perspective of being used in a single application. That article provides an excellent overview of Delphi's message support, including window procedures in objects and message handlers: I recommend you read it. However, the Windows API message support is quite large and messages are a common IPC solution, so we will have a look at additional areas not covered in the earlier article.

You can send a message to any window in an application running in the Windows environment on the same machine using a variety of Win32 API calls. All you need to know is the window handle of the target window. The window handle is a 32-bit value that uniquely identifies the window apart from all other windows in the system. A window's handle can be found in various ways, as we will see later. The message itself can be accompanied by two 32-bit integer parameters (`WParam` and `LParam`) which can contain whatever information you choose to give to the target window. The most commonly used APIs for delivering a message to a window are `PostMessage` and `SendMessage`.

If the message is being sent to another process (which will live in its own separate address space), you must ensure that you do not set either of these parameters to contain pointers, object or interface references, or Pascal long string references. Any memory address valid for one process will be *invalid* in any other process running on a Win32 platform.

Ultimately, the *window procedure* for the target window will get the message sent to that window. In the case of a Delphi control or form, the window procedure is the `WndProc` method, which may well farm the message out to a dedicated message handler if one exists. The two 32-bit values that come with the message may well contain useful information and are typically examined or used when the message is handled (meaning received and processed).

How the message arrives at the window procedure depends upon how it was sent. Each application has a message queue where messages (for any window in the application) can be deposited. Such messages are called *queued messages*. When the application is not busy executing code, a message loop checks the message queue to see if anything has arrived. If it has, the message is dispatched off to the window procedure of the target window. Since the application is often busy drawing itself, or servicing events, queued messages are not handled immediately.

Delphi programs that spend a non-negligible amount of time doing some processing, causing the message loop to remain unchecked and possibly brimming with unhandled messages, are advised to periodically call `Application.ProcessMessages` to process any pending messages. This generally makes the application not look like it has hung, by allowing paint and other messages to be processed.

Queued messages are sent with `PostMessage` and, since they have no definite time frame for being handled, are by definition non-urgent. Messages with a higher degree of urgency are sent

with SendMessage (and related APIs) and are called *non-queued messages*. SendMessage causes the message to be delivered directly to the window procedure of the target window. The SendMessage call does not return until the target window procedure has processed the message and so gives synchronous message processing. Any value returned by the window procedure is returned by the SendMessage function.

SendMessage can be used to broadcast a given message to all top level windows. When SendMessage is called with a window handle value of HWnd_Broadcast, it sends it to the window procedure of all top-level windows in the system. Any values returned by the various window procedures will be ignored. BroadcastSystemMessage can be used to perform a similar job, but there is a problem using it under Windows 95. Delphi maps BroadcastSystemMessage onto the underlying API called BroadcastSystemMessageA, which exists in both Windows 98 and NT. Unfortunately, the API is actually called BroadcastSystemMessage in Windows 95. If an application calls this routine on Windows 95, the incorrect API mapping causes the application to fail whilst loading.

If you plan to broadcast arbitrary messages around the system, you are advised to register a custom message for this purpose. Whenever a unique string is passed to the RegisterWindowMessage API, you get a unique window message value back. If another application uses the same string, it gets the same value back. So if two applications register the same string, they both have the same message number to use. One application can broadcast this message and the other application can pick up the message and process it as needed.

Normal message handling in Delphi is implemented using message handling methods, where the message to be handled is specified as a constant value. Because custom messages (registered with RegisterWindowMessage) have unknown values until runtime, you cannot use message handling methods to pick these messages up. Instead you must override the WndProc method.

The FindWindow API can be used to get the window handle of a given window so long as you know the window class name (which happens to be the form class in a Delphi application) and/or window caption. The WinSight32 tool that comes with Delphi can be used at development time to examine the windows in running applications and identify their window classes, for those cases where you do not know their names.

If you are developing both the applications which are communicating information, you can manufacture other ways for one application to find the window handle of the relevant window in the other application. For example, if application A launches application B, it can pass its own main form handle as a command-line parameter. Application B can then send a message to application A, passing its own window handle (which equates to its main form's Handle property) as a parameter to the message.

Alternatively, application B can define a Windows *atom*. This is a string registered in the Windows atom table with appropriate API calls. When you register an atom

```
unit CustomMsgU;
interface
uses Windows;
type
  TBroadcastSystemMsgFunc = function (Flags: DWORD; Recipients: PDWORD;
    uiMessage: UINT; wParam: WPARAM; lParam: LPARAM): Longint; stdcall;
var
  wm_ClinicMessage: UInt;
  BroadcastSystemMsg: TBroadcastSystemMsgFunc;
implementation
uses SysUtils;
procedure InitBroadcastMsgPtr;
const
  APIName: array[Boolean] of PChar =
    ('BroadcastSystemMessageA', 'BroadcastSystemMessage');
var InWin95: Boolean;
begin
  //Running under Windows 95?
  InWin95 := (Win32Platform = VER_PLATFORM_WIN32_WINDOWS) and
             (Win32MajorVersion = 4) and (Win32MinorVersion = 0);
  @BroadcastSystemMsg :=
    GetProcAddress(GetModuleHandle(Windows.User32), APIName[InWin95])
end;
initialization
  wm_ClinicMessage := RegisterWindowMessage('MyUniqueString!!!');
  InitBroadcastMsgPtr
end.
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
procedure TMsgSendForm.btnBroadcastClick(Sender: TObject);
begin
  WinExec('MsgRcv1.Exe', sw_ShowNormal);
  SendMessage(HWnd_Broadcast, wm_ClinicMessage, updWParam.Position,
    updLParam.Position)
end;
procedure TMsgSendForm.btnBroadcast2Click(Sender: TObject);
var Recipients: DWord;
begin
  WinExec('MsgRcv1.Exe', sw_ShowNormal);
  Recipients := BSM_APPLICATIONS;
  //Can make message be sent to all applications except this one
  BroadcastSystemMsg(BSF_IGNORECURRENTTASK, @Recipients,
    wm_ClinicMessage, updWParam.Position, updLParam.Position)
end;
```

```
TMsgReceiveForm = class(TForm)
protected
  procedure WndProc(var Message: TMessage); override;
end;
...
procedure TMsgReceiveForm.WndProc(var Message: TMessage);
begin
  if Message.Msg = wm_ClinicMessage then
    ShowMessageFmt('Received a message (WParam = %d, LParam = %d)',
      [Message.WParam, Message.LParam])
  else
    inherited
end;
```

➤ *Listing 3*

you get a unique identifying number back that could be sent. Application B could store the text version of its window handle as an atom, so application A can easily find its window handle value. For more information, look up *Atoms* in the Win32 help file optionally installed with any 32-bit Delphi.

If you look in the Messages subdirectory of the directory where this article's files are located on this month's CD-ROM you will find two projects that demonstrate simple inter-process communication via messages. One application (MsgSend.Dpr) broadcasts a custom message both with `SendMessage` and `BroadcastSystemMessage`. The other application (MsgRcv.Dpr) picks up the message and displays the values of `WParam` and `LParam` that were passed. A helper unit, `CustomMsgU`, registers the custom Windows message and also deals with the problem of `BroadcastSystemMessage` not working on Windows 95.

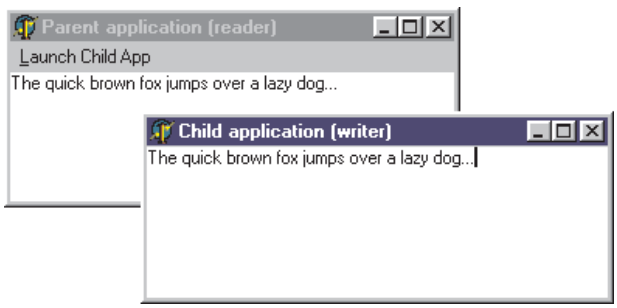
The initialisation section of the unit checks which platform the program is running on, and then locates the address of the relevant routine (whose name depends on the platform). A global function pointer called `BroadcastSystemMsg` is then set to point at the address of the routine. Listing 1 shows this unit, whilst Listing 2 shows the two code snippets that broadcast the message around the system. Finally, Listing 3 shows the target application's overridden `WndProc` method.

The wm_CopyData Message

Windows messages can be useful, but they generally only allow 8 bytes of data to be sent. To communicate more data, you can use the dedicated `wm_CopyData` message. This allows you to set up a block of data (which must be pure data, with no addresses or pointers contained within) and send it to a target window. It must be *sent*, as opposed to *posted*, to the target, since the data block is only valid in the target process until the call to `SendMessage` returns. If the message was posted with `PostMessage`, the data block would be invalid by the time the message was extracted from the queue and passed to the window procedure.

When you send the `wm_CopyData` message, the `WParam` value must be set to the window handle of the window sending the message (to give an easy way for the target window to reply) and the `LParam` value must be set to the address of a `TCopyDataStruct` record. This record contains the address and size of your data block and also one other `DWord` field that can be used for any additional information you wish to pass.

In the Messages subdirectory off the main CD-ROM directory for this article, you will find a pair of projects that give an example of `wm_CopyData` in action. The two applications both have memo components on the form. As the user types into the memo on the second application (Child.Dpr), which is launched by the first application (Parent.Dpr), each key press causes the memo contents to be read, packaged up and sent off as a `wm_CopyData` message. The parent picks up the message, containing only character data, reads the contents and writes it into its own memo. Thus we can see the data being sent from one program to another.



➤ *Figure 1: One app sending textual data to another.*

```
procedure TChildForm.Memo1Change(Sender: TObject);
var
  CopyData: TCopyDataStruct;
  TextMsg: String;
begin
  TextMsg := Memo1.Text;
  //Make the data pointer point at the first character of the string
  CopyData.lpData := @TextMsg[1];
  //Set the length field to indicate the number of characters in the memo
  CopyData.cbData := Length(TextMsg);
  //Send the message to the parent's window handle, as sent on the command-line
  SendMessage(StrToInt(ParamStr(1)), wm_CopyData, Handle, Integer(@CopyData))
end;
TParentForm = class(TForm)
...
public
  procedure WMCopyData(var Msg: TWMCopyData);
    message wm_CopyData;
end;
...
procedure TParentForm.WMCopyData(var Msg: TWMCopyData);
var TextMsg: String;
begin
  //Set the string variable length and content to match what was sent
  SetString(TextMsg, PChar(Msg.CopyDataStruct^.lpData),
Msg.CopyDataStruct^.cbData);
  //Put string into memo
  Memo1.Text := TextMsg
end;
```

➤ *Listing 4*

Before sending a `wm_CopyData` message, the child has to find the parent's window handle. The approach taken is for the parent to send its window handle as a command-line parameter to the child.

Listing 4 contains the important snippets from the two projects, the memo's `OnChange` event handler in the child project and the parent's `wm_CopyData` message handling method. Figure 1 shows the two programs working at runtime.

Mailslots

A *mailslot* is not a particularly well known IPC mechanism among Delphi programmers. Whilst many programmers have heard of, say, named pipes (covered later), mailslots don't usually sound familiar.

The job of a mailslot is to make it easy to send messages between applications and also to broadcast messages throughout a network domain. A mailslot provides one-way communication and it is

implemented as a memory-based file. Normal Windows file routines are used to access it. One application, the *mailslot server*, creates and owns the mailslot and can retrieve messages stored in the mailslot by other applications, *mailslot clients*. When the mailslot server creates the mailslot, it gets a mailslot handle, which is used for reading messages. When every server handle to the mailslot is closed, the mailslot and all data contained in it are deleted.

Any application that has the name of a mailslot can be a mailslot client and write messages to the mailslot. The mailslot name could be passed as a command-line parameter. Newer messages are placed in the mailslot after older messages. Since the mailslot is a pseudofile, messages stored in it may consist of data in any format.

A mailslot can be accessed across a network. Also, mailslots can broadcast messages within a domain. If several mailslot servers in a domain create mailslots with the same name, any message for the mailslot which is sent to the domain will be delivered to each mailslot server's mailslot. While there is no limit on the number of messages that can be sent to a mailslot, messages that are to be broadcast to a domain must be no bigger than 400 bytes. Messages that are not broadcast should be no bigger than 64Kb.

To create a mailslot, the mailslot server process calls `CreateMailslot`. This API takes a case-insensitive mailslot name which must be of the form:

```
\\.\mailslot\[path]name
```

The `name` part must be unique on the current machine, and the optional `path` represents pseudo-directories you can specify to further identify the mailslot, or allow several mailslots to appear related. For example, the following mailslot names all include optional path specifiers:

```
\\.\mailslot\sales\1997
\\.\mailslot\sales\1998
\\.\mailslot\sales\1999
```

The `CreateMailslot` routine also takes three other parameters. One indicates the maximum message size allowed (0 means the default maximum as specified before). The next parameter specifies the amount of time (in milliseconds) that a read operation will wait for a message to be written to the mailslot before a timeout occurs. The constant `MAILSLOT_WAIT_FOREVER` indicates that there is no timeout, but Delphi 2 does not define this symbol (it has a value of -1). This timeout can be changed with a call to `SetMailslotInfo`. The last parameter is a security specifier that indicates (on NT) whether child process will also be able to use the mailslot handle that `CreateMailslot` will return.

`GetMailslotInfo` returns several bits of information about the mailslot, including the maximum message size, the read timeout, the number of messages waiting to be read and the size of the next message. If there are no messages waiting, the next message size parameter will be `MAILSLOT_NO_MESSAGE` which, again, Delphi 2 does not define (it should be -1).

To get a message from a mailslot, the mailslot server calls the Windows API `ReadFile` (or the Delphi version of it, which is `FileRead` from the `SysUtils` unit).

The mailslot client accesses the mailslot by passing the name of an existing mailslot to the Windows `CreateFile` function (or Delphi's `FileOpen`). To write to a mailslot on the same machine, the mailslot name is passed exactly as it was by the mailslot server. To write to a mailslot on a specific machine, the mailslot needs to be in this form:

```
\\computername\mailslot\
  [path]name
```

To broadcast messages to all mailslots with a given name in a specified domain, the mailslot name must be in this form:

```
\\domainname\mailslot\
  [path]name
```

To broadcast messages to all mailslots with a given name in the system's primary domain, this form should be used:

```
\\*\mailslot\[path]name
```

Messages are written to the mailslot with `WriteFile` (or `FileWrite`) and the client side of the mailslot is closed with `CloseHandle` (or Delphi's `FileClose`).

Since you have the choice of using raw API calls or Delphi wrappers for them, two sample project groups are supplied to demonstrate simple mailslot usage, one for each set of calls. The Mailslots subdirectory has both API and Delphi subdirectories, which have identical projects, each using a different set of calls. In each case, the mailslot server has code to launch the mailslot client (the examples work with local mailslots).

The two applications operate (to the user) in exactly the same way as with the `wm_CopyData` samples described above. As the user types into the mailslot client memo, each key press causes the memo contents to be written to the mailslot as a new message. The mailslot server uses a timer to regularly check for new messages. When one arrives, it reads the message into a string and then writes it into its own memo. Listing 5 contains the pertinent code from the mailslot server project (called ServerMailslot.Dpr) and Listing 6 shows the corresponding code from the mailslot client (called ClientMailslot.Dpr).

The prime advantage of a mailslot is that it can broadcast a message to many PCs in a domain. The primary disadvantage is that the mailslot client has no guarantee that the message sent through the mailslot is received by the mailslot server, and is not alerted to a failed message send.

Pipes

A pipe is an IPC mechanism with two ends, each of which is defined by a handle. The pipe can be created for one-way communication only, or for duplex (two-way) communication. If a program has a handle to one end of a pipe, it can communicate with the process

```
const
  MailslotNameFixedReadPrefix = '\\.\mailslot\';
  MailslotName = 'SampleMailslot';
  MailslotReadName = MailslotNameFixedReadPrefix +
    MailslotName;
  MailslotMaxSize = 0; //any size
  MailslotTimeout = 0; //don't wait
var Mailslot: THandle;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Mailslot := CreateMailslot(MailslotReadName,
    MailslotMaxSize, MailslotTimeout, nil);
  if Mailslot = Invalid_Handle_Value then
    raise EWin32Error.Create('Unable to create mailslot');
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FileClose(Mailslot);
end;
procedure TForm1.Timer1Timer(Sender: TObject);
var
  NextMsgSize: DWord;
  Msg: String;
begin
  if not GetMailslotInfo(Mailslot, nil, NextMsgSize, nil,
    nil) then begin
    Timer1.Enabled := False;
    raise EWin32Error.Create(
      'Cannot get mailslot information')
  end;
  //Check there is a message to read
  if NextMsgSize <> Mailslot_No_Message then begin
    //Allocate string size as required and set length
    SetLength(Msg, NextMsgSize);
    //Read the data into the string variable
    if FileRead(Mailslot, Msg[1], NextMsgSize) =
      Integer(HFile_Error) then begin
      Timer1.Enabled := False;
      raise EWin32Error.Create('Cannot read from mailslot');
    end;
    Memo1.Text := Msg
  end
end;
```

➤ *Listing 5*

```
const
  MailslotNameLocalWritePrefix = '\\.\mailslot\';
  MailslotName = 'SampleMailslot';
  MailslotWriteName = MailslotNameLocalWritePrefix + MailslotName;
var
  Mailslot: THandle;
procedure TForm1.FormCreate(Sender: TObject);
begin
  Mailslot := FileOpen(MailslotWriteName, fmOpenWrite or fmShareDenyWrite);
  if DWord(Mailslot) = Invalid_Handle_Value then
    raise EWin32Error.Create('Cannot open client side of mailslot');
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FileClose(Mailslot);
end;
procedure TForm1.Memo1Change(Sender: TObject);
var Msg: String;
begin
  Msg := Memo1.Text;
  if FileWrite(Mailslot, Msg[1], Length(Msg)) = Integer(HFile_Error) then
    raise EWin32Error.Create('Cannot write to mailslot');
end;
```

➤ *Listing 6*

that has a handle to the other end of the pipe. The Win32 API supports two kinds of pipe: anonymous and named. Unlike mailslots, you know if the data gets to the other end of a pipe, as the API fails if the pipe is broken in some way.

Anonymous Pipes

An *anonymous pipe* is an unnamed, one-way pipe that supports communication (again with file writing functions) between a parent process and a child process (a process launched by the parent), or between two child processes of a parent process. Because of this, anonymous pipes operate on a single machine and do not support networked communications. After creation, the pipe will exist until the handles to both ends of the pipe are closed.

The `CreatePipe` API creates an anonymous pipe and supplies the program with a read-only handle to the read end of the pipe and a write-only handle to the write end of the pipe. To get parent/child process communication, one of the handles must be given to the child process. For the child process to be able to use the handle, the handle must be inheritable. This can be ensured by following these steps.

Firstly, the `pPipeAttributes` parameter to `CreatePipe` can have the `bInheritHandle` field set to `True`. However, even if this is not done, a call to `DuplicateHandle` can toggle the inheritability of a given handle. Next, the child process is launched with `CreateProcess`, where the `bInheritHandles` parameter is set to `True`, which means the child process will be able to successfully use the same handle value as the parent process, for any of the parent's inheritable handles.

To actually give the handle to the child, apart from simply passing it as a command-line parameter, a common approach is to use one of the standard handles (standard input, standard output and standard error). Since the child process will inherit the parent's standard handles, the parent can call `SetStdHandle` to set one of them to be the pipe handle. The child can then use `GetStdHandle` to pick the handle up.

Again, there are a pair of project groups supplied in subdirectories off the AnonymousPipes subdirectory, containing both a raw API approach and one using Delphi RTL calls. The parent process in each case (the project called ParentAnonymousPipe.Dpr) creates an anonymous pipe. The intention is for the child process (ChildAnonymousPipe.Dpr) to write to it and the parent to read from it, so the pipe handles are created as inheritable. To get the pipe write handle to the child, the standard output handle is replaced by it before launching the child. To keep things tidy, the parent then ensures that the only write handle belongs to the child process, by closing its own handle after launching the child. Also, its read handle is made non-inheritable (with a call to `DuplicateHandle`) to ensure the only pipe read handle belongs to the parent.

Having got things set up, the parent application then spawns a thread to keep checking the pipe for messages. If the thread ever reads anything from the pipe (which in this case is assumed to be text), it uses a call to `Synchronize` to get the text added to a memo on the form. Listing 7 shows the important bits of code.

The child process upon starting up gets the pipe write handle with a call to `GetStdHandle` and then

```
function LaunchChildApp: THandle;
const
  ChildApp = 'ChildAnonymousPipe.Exe';
var
  SI: TStartupInfo;
  PI: TProcessInformation;
begin
  GetStartupInfo(SI);
  if not CreateProcess(nil, ChildApp, nil, nil,
    //Make sure child inherits our inheritable handles
    //as it will need to refer to them to use the pipes
    True, 0, nil, nil, SI, PI) then
    raise EWin32Error.Create('Unable to launch '+ChildApp);
  Result := PI.HProcess
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  Security: TSecurityAttributes;
  ThisProcess, PipeReadTmp, PipeWrite: THandle;
const
  PipeBufferSize = 0; //default size
begin
  with Security do begin
    nLength := SizeOf(Security);
    bInheritHandle := True; //create inheritable handles
    lpSecurityDescriptor := nil;
  end;
  Win32Check(CreatePipe(PipeRead, PipeWrite, @Security,
    PipeBufferSize));
  //Turn STDOUT into the write pipe handle
  Win32Check(SetStdHandle(Std_Output_Handle, PipeWrite));
  //Ensure pipe read handle is not inheritable
  ThisProcess := GetCurrentProcess();
  Win32Check(DuplicateHandle(ThisProcess, PipeRead,
    ThisProcess, @PipeReadTmp, 0, False,
    Duplicate_Same_Access));
  FileClose(PipeRead);
  PipeRead := PipeReadTmp;
  //This launches the child process and waits for it to
  //settle down before moving on to the next statement
  WaitForInputIdle(LaunchChildApp, Infinite);
  //Ensure only the child has an open write pipe handle
  FileClose(PipeWrite);
  //Start reader thread off
  with TTestAnonymousPipe.Create(PipeRead) do
    FreeOnTerminate := True
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FileClose(PipeRead);
end;
{ Thread object }
TTestAnonymousPipe = class(TThread)
private
  FUIString: String;
  FPipeHandle: Integer;
  //Two routines to simplify showing our progress
  procedure UpdateUI;
  procedure WriteString(const S: String);
protected
  //Body of the thread
  procedure Execute; override;
public
  constructor Create(PipeRead: Integer);
end;
constructor TTestAnonymousPipe.Create(PipeRead: Integer);
begin
  inherited Create(False);
  FPipeHandle := PipeRead;
end;
procedure TTestAnonymousPipe.UpdateUI;
begin
  with Form1.Memo1 do
    Text := Text + FUIString;
end;
procedure TTestAnonymousPipe.WriteString(const S: String);
begin
  FUIString := S;
  Synchronize(UpdateUI);
end;
procedure TTestAnonymousPipe.Execute;
const
  BytesToRead = 1000;
var
  BytesRead: Integer;
  DataBuf: array[0..BytesToRead] of Char;
begin
  repeat
    BytesRead :=
      FileRead(FPipeHandle, DataBuf, BytesToRead);
    if BytesRead <> -1 then begin
      SetString(FUIString, DataBuf, BytesRead);
      Synchronize(UpdateUI);
    end
  until Terminated or (BytesRead = -1);
  if not Terminated then
    WriteString('Pipe is broken: ' +
      SysErrorMessage(GetLastError))
end;
```

➤ *Listing 7*

allows the user to add things to the pipe with an edit control and a button. When the button is pushed, the edit control contents are written to the pipe. Listing 8 shows the details in brief.

It is also common in Windows apps for anonymous pipe handles to be left as the standard handles. A parent process can set its standard input to the pipe read handle and the child process can have its standard output handle set as the pipe write handle. However, communication through standard handles isn't usal in Delphi programs, except console mode applications.

Named Pipes

After all that messing around with handles, it is nice to find that named pipes are simpler to deal with (although they do have a lot more creation options). A *named pipe* can be one-way or two-way and is used to communicate between a *named-pipe server* process (which creates the named pipe) and one or more *named-pipe client* processes.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  PipeWrite := GetStdHandle(Std_Output_Handle)
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FileClose(PipeWrite);
end;
procedure TForm1.Button1Click(Sender: TObject);
var Msg: String;
begin
  Msg := Edit1.Text + #13#10;
  if FileWrite(PipeWrite, Msg[1], Length(Msg)) = Integer(HFile_Error) then
    raise EWin32Error.Create('Cannot write to anonymous pipe');
end;
```

➤ *Listing 8*

The call to `CreateNamedPipe` used by the server process to create the pipe can request more than one instance of the pipe. Each instance has the same name, but its own storage space and handles, and multiple pipe instances allow multiple clients to simultaneously connect to the pipe. An instance of a named pipe will be destroyed when the last handle to that instance is closed.

A client process only needs to know the name of the pipe to use it, and so unrelated processes can communicate with it. Also, named pipes allow communication across a network. The primary restriction with named pipes is that the server process must be running on Windows NT (Windows 95/98 do not support `CreateNamedPipe`).

When creating a named pipe, the server process must use a name in the following format:

```
\\.\pipe\pipename
```

The client process specifies the name of the pipe to connect to in this format:

```
\\servername\pipe\pipename
```

If the pipe is local, servername can be replaced with a period, as in the server's pipe name format. The pipename part of the name is case-insensitive and can have as many as 256 characters.

CreateNamedPipe specifies many aspects of the pipe's communication support by its various parameters. The dwOpenMode parameter specifies the access mode (communication direction). An access mode of PIPE_ACCESS_INBOUND means the pipe server can only read and the pipe client can only write to the pipe. PIPE_ACCESS_OUTBOUND is the reverse where the server can only write and the client can only read. PIPE_ACCESS_DUPLEX means both the server and client can read and write to the pipe. Other flags allow you to customise how quickly write operations return when the amount of data being written is large, and when the pipe client and server are on different machines.

Another parameter specifies the pipe's type mode, either byte-type (PIPE_TYPE_BYTE for arbitrary data) or message-type (PIPE_TYPE_MESSAGE, where each write is considered a message unit). You also specify the requested input and output buffer sizes, although these are advisory only. The help for this API in the Win32 SDK discusses the full set of flags.

The same set-up of an API version and a Delphi version of a pair of projects can be found in the Named Pipes subdirectory. The projects attempt to do the same job as the anonymous pipe versions, in order to highlight the alternative API calls needed for the job. Listings 9 and 10 show the code that differs for named pipes.

File Mappings And Shared Memory

A *file mapping*, or memory-mapped file, allows one or more processes to associate a portion of their virtual address space (called a *file view*) with a section of a file on disk. The association is made by an operating system file-mapping object. This allows potentially many processes to access the same file in an efficient way, however there is a requirement to synchronise the access by each process. Stefan Boether briefly talked about the advantages of file mappings in the *Tips & Tricks* column back in Issue 16.

So a file mapping could allow communication of data from one application to another, via the disk file that they both have a file view on. However, the file in question that gets mapped into memory can be the operating system paging file, if you wish. The net effect of this is to provide a shared memory mechanism between multiple processes, into which you can place whatever data you like.

A shared memory file mapping is created with CreateFileMapping, which takes a file handle that indicates the file to map. A file handle value of $FFFFFFFF indicates the Windows paging file, which gives you *named shared memory* between processes on the same machine.

One of the demonstration projects supplied with Delphi since version 2 is designed to show IPC techniques and has a TSharedMem class (in the IPCThrd unit) that simplifies the business of calling CreateFileMapping, MapViewOfFile and any other calls that are required. As long as you set the entry for Search Path in the Directories/Conditionals page of the project options dialog to include Delphi's Demos\IPCDemos directory, you can add IPCThrd to your unit's uses clause.

The projects on the disk (in the File Mappings subdirectory) are again supplied twice. One project group uses TSharedMem (don't forget to set the unit search path for the IPCThrd unit), whilst the other uses API calls to deal with the file mapping. Clearly, the non-API versions are simpler, so we will focus on them. Again, the projects are designed to emulate the mailslot and pipe projects from earlier. To see how they work when implemented with shared memory file mappings, Listing 11 has some code from the parent process that displays what was written to the file mapping. Listing 12 shows the details of the child process that writes information to the file mapping.

It's worth noting at this point that many of the other IPC mechanisms available under Win32 are actually implemented by memory-mapped files.

It should also be made clear that when several processes are using a memory-mapped file, some form of synchronisation should be employed to ensure coherent views of the data. In other words,

```
const
  PipeNameFixedPrefix = '\\.\pipe\';
  PipeName = PipeNameFixedPrefix + 'SampleNamedPipe';
  PipeMaxInstances = 1; //only allow 1 pipe
  PipeOutBufferSize = 0; //any size
  PipeInBufferSize = 0; //any size
  PipeTimeout = 0; //don't wait
procedure TForm1.FormCreate(Sender: TObject);
const PipeBufferSize = 0; //default size
begin
  PipeRead := CreateNamedPipe(PipeName, Pipe_Access_Inbound, Pipe_Type_Byte,
    PipeMaxInstances, PipeOutBufferSize, PipeInBufferSize, PipeTimeOut, nil);
  if PipeRead = Invalid_Handle_Value then
    raise EWin32Error.Create('Cannot create named pipe');
  //This launches the child process and waits for it to
  //settle down before moving on to the next statement
  WaitForInputIdle(LaunchChildApp, Infinite);
  //Start reader thread off
  with TTestNamedPipe.Create(PipeRead) do
    FreeOnTerminate := True
end;
```

➤ *Above: Listing 9*

➤ *Below: Listing 10*

```
const
  PipeNameFixedPrefix = '\\.\pipe\';
  PipeName = PipeNameFixedPrefix + 'SampleNamedPipe';
procedure TForm1.FormCreate(Sender: TObject);
begin
  PipeWrite := FileOpen(PipeName, fmOpenWrite);
  if PipeWrite = Invalid_Handle_Value then
    raise EWin32Error.Create('Cannot open pipe for writing');
end;
```

```
MemMapFile: TSharedMem;
...
const
  MemMapFileName = 'SampleMemoryMappedFile';
  MemMapSize = 1000;
procedure TForm1.FormCreate(Sender: TObject);
begin
  MemMapFile := TSharedMem.Create(MemMapFileName, MemMapSize);
  LaunchChildApp;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  MemMapFile.Free
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  //If there is a PChar, then extract it
  if PChar(MemMapFile.Buffer)[0] <> #0 then
    Memo1.Text := StrPas(PChar(MemMapFile.Buffer))
end;
```

➤ *Above: Listing 11*

➤ *Below: Listing 12*

```
MemMapFile: TSharedMem;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  MemMapFile := TSharedMem.Create(MemMapFileName, MemMapSize);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  MemMapFile.Free
end;
procedure TForm1.Memo1Change(Sender: TObject);
var Msg: String;
begin
  Msg := Memo1.Text;
  //Copy memo contents (as PChar, with trailing 0 character) into shared memory
  Move(PChar(Msg)^, MemMapFile.Buffer^, Length(Msg) + 1)
end;
```

```
Mutex: TMutex;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  Mutex := TMutex.Create('FileMappingMutex');
  MemMapFile := TSharedMem.Create(MemMapFileName, MemMapSize);
end;
procedure TForm1.Memo1Change(Sender: TObject);
var Msg: String;
begin
  Mutex.Get(Integer(Infinite));
  Msg := Memo1.Text;
  Move(PChar(Msg)^, MemMapFile.Buffer^, Length(Msg) + 1);
  Mutex.Release;
                                    end;
```

➤ *Listing 13*

one process should gain exclusive access to the file when writing, and then the other process should also gain exclusive access to the file for reading. A suitable synchronisation mechanism would be a *mutex*, specifically designed for enforcing mutually exclusive access. You can find an easy-to-use `TMutex` class in the `IPCThrd` unit.

To show briefly how the mutex would be incorporated into the code, Listing 13 is a modified version of Listing 12. Listing 11 should also be modified in a similar way to ensure only one process is accessing the file mapping at a time. If one program tries to get the mutex and another process already has it, the calling thread is blocked until the mutex is released.

The only problem to watch out for is the `TMutex.Get` parameter, used to specify how long to wait for the mutex. It is defined as an `Integer` (a signed type), and the value is then passed to the Win32 API `WaitForSingleObject` which takes a `DWord` (unsigned type). One value Windows does let you pass is the constant `Infinite`, defined as a `DWord` with a value of `$FFFFFFFF`), but you need to take care. Firstly, you need to typecast this to an `Integer` to make the `Get` method accept it, which turns the value into -1. This integer is then passed to `WaitForSingleObject`, but if range-checking is enabled, you will get an error, due to -1 being lower than the lowest `DWord` (0). You should either re-code the `Get` method to take a `DWord` parameter, or turn range-checking off in the `IPCThrd` unit.

Check out John Chaytor's article on sharing data back in Issue 17 (January 1997) for information on various synchronisation objects available from Win32.

Summary

There are many Win32 inter-process communication mechanisms available to take advantage of with different capabilities, which will dictate the one you will use. Some support network communications and some don't. Some support communication between unrelated processes and some don't. Some rely on a particular version of Windows.

If you need two applications to communicate with each other, hopefully you will have picked up some useful ideas from this article.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com