3

Chapter 3

Communication

In Chapter 1, we touched briefly on how to communicate with your application's windows using `SendMessage()` and `PostMessage()`. But what if you wanted to communicate with another application entirely or even an application on another system?

In this chapter, we will review six methods available to your application for communicating with the outside world, including applications on other systems, on the Internet, or over a serial line. We will see how this communication link will allow us to control another application or even use its functionality. We will also discover a couple of ways to share large amounts of data with another application.

Interprocess Communication

Communication between applications, on the same system, or over a network is called Interprocess Communication (IPC). Your MFC application has the following six avenues available for communicating with another application.

Windows messaging allows you to communicate with the windows of any other application. This is the same windows messaging we used to communicate with our own application's windows.

Dynamic Data Exchange (DDE) allows you to pass large amounts of data between applications by maintaining it in globally allocated memory. You could do this yourself by manually putting data in a globally allocated chunk of memory and then passing that pointer using windows messaging. However, DDE provides a standard that allows any application that conforms to it to play.

Message Pipes can be used to set up a permanent communication channel between applications through which data can be read and written as if your applications were accessing a flat file. You give up the speed of DDE data transfers, but Message Pipes allow you to seamlessly send data to applications on other systems.

Window Sockets have all the functionality of a Message Pipe, but follow a communication standard that allows you to communicate with applications on non-Windows systems, such as UNIX. You should, in fact, use Window Sockets in favor of Message Pipes and DDE in any new development.

Serial/Parallel communication allows your application to talk to an application or device through a serial or parallel port.

Internet communication allows your application to upload or download a file from an Internet address.

Communicating Strategy

Although each of these communication methods is invoked with a different Windows API or MFC class, the procedure for using them is almost identical.

1. Use the Windows API or a MFC class to open a communication link with the other application.
2. Read and write to that other application. With some methods, that can mean sending and receiving messages. With other methods, the process is not that much different from reading and writing to a flat file.
3. Then close the link.

Synchronous vs. Asynchronous Communication

Each of these communication methods allows you to communicate synchronously or asynchronously. *Synchronous communication* causes your application to pause until finished reading or writing to the other application. *Asynchronous communication* allows your application to continue while the system finishes the read or write operation for you and then informs you when it's done with an event flag or by calling a function you specify.

Although asynchronous communication sounds preferable, especially in an application that will be talking to several other applications at once, it isn't very object-oriented. You need to either step out of an object to provide a static callback function or you need to worry about processing an event flag. Setting up an asynchronous link is also more complicated and, therefore, more prone to bugs.

The solution is to use synchronous communication, but to put each read or write operation in its very own thread so that it doesn't stop your application from communicating with other applications. When the operation is finished, the thread can automatically store any read data back into your application. The thread can also set an event flag, but only in the more hospitable environment of MFC classes. Examples 51 and 52 show how to do this.

Note: Your application's ability to create its own thread has only just been available with Windows 95 and Windows NT. If you will be developing for a Windows 3.1 platform, you will need to use asynchronous communication. With asynchronous communication, the operating system itself is putting the read or write operation in its own special thread.

Note: In Window Socket jargon, an application that communicates synchronously is considered to be "blocking".

We will now present each communication method, including how to open and close a communication link and how to read and write over it.

Windows Messaging

The same messaging system that allows your application to control its windows and process command messages from the user can also be used to control and process commands from other applications. What makes this all possible is that each window object is globally allocated such that any application can send a message to any other application's windows. The trick is knowing which window to talk.

Opening and Closing

To open a communication link with another application, simply determine the appropriate window handle to which you should send messages. However, you don't know this handle at compile or link time because each handle is created when the window is created and can be different each time. You can't ask the other window for its handle because you need its handle to make the request. There are three other techniques for getting that window handle.

1. When a parent application creates a child application, it passes the handle of one of its windows as a creation argument. The child application can then send that handle the window handle of one of its own windows and communication can proceed. Typically, the handles that these applications exchange are to plain (but always hidden) windows created simply to communicate through. Closing communication is then simply a matter of closing these windows.
2. If the target application is already running, you can instead look for the appropriate window using the Windows API call `FindWindow()`. This function will return a handle to any window that matches a particular name or uses a particular Window Class. To help `FindWindow()` find the right window, you can, therefore, create a plain messaging window as discussed previously using a highly unique caption or Windows Class name.
3. You can also use the Windows API call `BroadcastSystemMessage()`, which will send a message to every application currently running. You can use `BroadcastSystemMessage()` to broadcast a special message that contains your application's own window handle. Applications that process this message can then reply with their own window handles, thus completing the circuit. To see this last method in action, please see Example 49.

Reading and Writing

To now communicate with the other application is simply a matter of plugging this window handle into `::SendMessage()` or `::PostMessage()`. You can also wrap the window handle in a `CWnd` class using `Attach()`, and then use `CWnd`'s other methods to send and post messages. Sending a message provides your application with synchronous communication; posting provides asynchronous communication.

Although you send the standard window messages (e.g., `WM_CLOSE`, `WM_MOVE`, etc.) to the target window, you will typically want to create your own window message to perform some custom task.

You could create your window message ID by simply defining it in the range above `WM_USER`, as seen here.

```
#define WM_MYMESSAGE WM_USER+1
```

An arguably more robust method of creating a new message ID is to register your message with the system by using

```
#define IDString "MyMessage"
MsgID = ::RegisterWindowMessage( IDString );
```

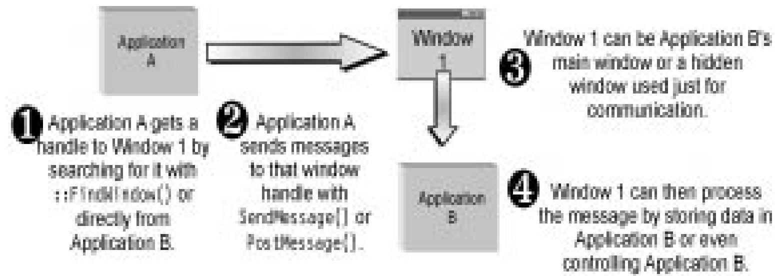
where `IDString` is a unique text string of which every application is aware, and `MsgID` is the created message ID. Therefore, rather than assigning a numeric value for your message ID that might get mismatched among applications, you can use a descriptive text string, such as `OpenFile`, and `::RegisterWindowMessage()` creates a message ID for you. If one application registers `OpenFile` first, subsequent applications's calls to `::RegisterWindowMessage()` using `OpenFile` returns that same message ID so that there's never a danger of mismatch.

To see how these functions are used, please refer to Example 49.

Review

Sending messages to another application using windows messaging is then simply a matter of getting a window handle that belongs to the other application and then sending it a message. For an overview of this process, please see Figure 3.1.

Figure 3.1 Windows Messaging



Dynamic Data Exchange (DDE)

`SendMessage()` and `PostMessage()` allow you to send two integer values at a time to the other application. To send one thousand bytes would, therefore, take 250 messages. To send more at a time, you can stuff your data in globally allocated memory and pass its handle as one of these parameters. This approach, however, will only allow you to exchange data with other applications that you design. To communicate data with applications you don't design, you should follow the DDE standard, which has been implemented in the Dynamic Data Exchange Library (DDEML).

Client/Server

With DDE we are also introduced to the concept of a Client Application and a Server Application. In the case of DDE, almost all data resides in the Server Application, where it is accessed by the Client Application. Communication is then a matter of the Server putting out an "Open for Business" sign and a Client reading or writing data to the Server.

Client/Server is discussed in more detail in the section "Client/Server" on page 100.

Opening and Closing

To open a communication link, the Server Application starts by advertising itself with

```
// initialize DDE services
UINT DdeInitialize(
    LPDWORD pidInst,           // a pointer to the instance
    PFNCALLBACK pfnCallback,   // a callback function (see below)
```

```
    DWORD afCmd,                // command and filter flags
    DWORD ulRes                 // reserved
    );
// create a string handle to the name of the service we  are creating
HSZ DdeCreateStringHandle(
    DWORD idInst,    // returned by DdeInitialize() above
    LPTSTR psz,      // pointer to service name string
    int iCodePage    // CP_WINANSI or CP_WINUNICODE for Unicode
    );
// register the service
HDDEDATA DdeNameService(
    DWORD idInst,    // returned by DdeInitialize() above
    HSZ hsz1,        // string handle to service name
    OL,              // reserved
    UINT afCmd       // service name flags
    );
```

A Client Application can then connect to the Server with

```
UINT DdeInitialize( ... );    // as seen above
HCONV DdeConnect(
    DWORD idInst,       // returned by DdeInitialize() above
    HSZ hszService,     // handle to service name string
    HSZ hszTopic,       // handle to topic name string (optional)
    PCONVCONTEXT pCC    // pointer to structure with context data
    );
```

To handle this connection in the Server Application, the Server's callback routine must handle an `XTYP_CONNECT` message.

```
HDDEDATA CALLBACK DdeCallback( type ... )
{
    switch( type )
    case XTYP_CONNECT:
        return TRUE;    // return FALSE to reject connection
}
```

The Client can then close this link with

```
BOOL DdeDisconnect(
    HCONV hConv    // handle returned by DdeConnect() above
    );
```

Reading and Writing

To request data from the Server, the Client can then use

```
HDDEDATA DdeClientTransaction(
    NULL, 0,
    HCONV hConv,          // handle returned by DdeConnect() above
    HSZ hszItem,          // string handle of name of data to read
                          //     and return
    UINT wFmt,            // format of data based on clipboard formats
    XTYP_REQUEST,         // <<<<<<<<<<<<<<transaction type
    DWORD dwTimeout,      // should request time-out
    LPDWORD pdwResult     // pointer to result
    );
```

The data that's returned is actually a handle to a DDE data object, which can either be accessed with

```
LPBYTE DdeAccessData(
    HDDEDATA hData,        // handle returned by DdeClientTransaction()
    LPDWORD pcbDataSize    // pointer to data length
    );
```

or copied automatically into a local buffer with

```
DWORD DdeGetData(
    HDDEDATA hData,    // handle returned by DdeClientTransaction()
    LPBYTE pDst,       // pointer to destination buffer
    DWORD cbMax,       // amount of data to copy
    DWORD cbOff        // offset to beginning of data in hData
    );
```

The Server processes requests for data in its callback routine, as seen here. A data handle to requested data is created using `DdeCreateDataHandle()`.

```
case XTYP_ADVREQ:
{
    HDDEDATA hData =
        DdeCreateDataHandle(
            DWORD idInst,    // returned by server's DdeInitialize()
            LPBYTE pSrc,     // pointer to source buffer
            DWORD cb,        // length of source buffer
            DWORD cbOff,     // offset from beginning of source buffer
            HSZ hszItem,     // handle to item name string
            UINT wFmt,       // format of data in clipboard formats
            UINT afCmd       // creation flags
            );
            return hData;
}
```

For the Client to write data to the Server, it can either put the data into another data handle or pass the data directly with

```
HDDEDATA DdeClientTransaction(
    LPBYTE pData,        // pointer to data to pass to server
                         //     or data handle
    DWORD cbData,        // length of data
                         //     or Offffffffh if pData is a data handle
    HCONV hConv,         // handle returned by DdeConnect() above
    HSZ hszItem,         // string handle of name of data to read
                         //     and return
    UINT wFmt,           // data format based on clipboard formats
    XTYP_POKE,           // <<<<<<<<<<<<<<transaction type
    DWORD dwTimeout,     // should request time-out
    LPDWORD pdwResult    // pointer to result
    );
```

The Server must then accept this data in its callback routine with

```
case XTYP_POKE:
    DdeGetData();    // as before
```

Other DDE Functions

The DDEML provides a lot more functionality than what's presented here. For example, you can set up an advise loop that allows both Server and Client to be immediately advised if the data they share is modified.

The DDEML also supports a standard that allows a Client to ask the Server to perform a command. Which command to execute is expressed in a text string, such as `Open file.dat`, where `Open` is the command which the Server parses and executes if it's in its repertoire of commands.

MFC Support

Your SDI and MDI applications automatically support three DDE command strings.

`[open( "file.dat" )]` tells your application to open the specified file.

`[print( "file.dat" )]` tells your application to print the specified file.

`[printto( "file.dat","printer","driver","port" )]` tells your application to print the specified file to the specified device.

Your MFC application also has built-in support for the command line flag `/dde`, which simply hides your application initially so that your application operates in the background when printing files.

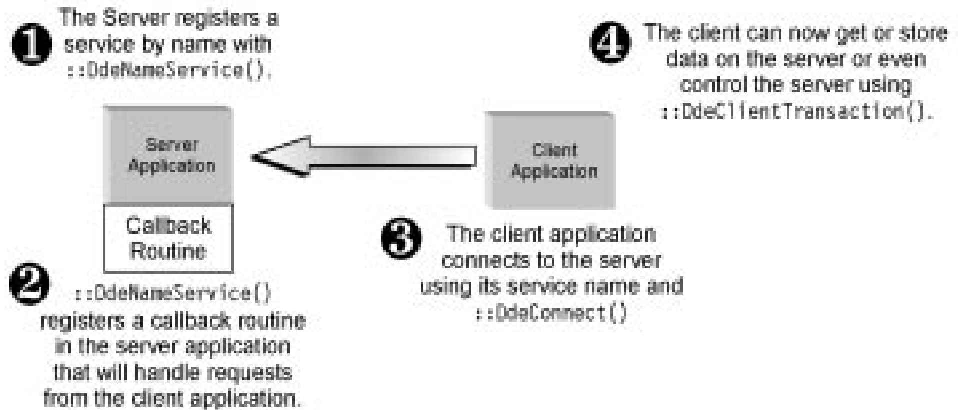
Other than this limited support, MFC does not support DDE. Why not? For one reason—applications have moved away from running on single systems to having multiple parts that can run on multiple systems. The shared global memory of one system is not accessible outside of that system. In other words, DDE is out of fashion.

Window messaging also can't make the leap between machines. In fact, Message Pipes were created to make that leap.

Review

DDE allows larger amounts of data to be shared among applications by orchestrating the use of global memory. This is also its downfall, since global memory can't be shared over a network. For an overview of this process, please see Figure 3.2.

Figure 3.2 Dynamic Data Exchange



Message Pipes

To allow continuous communication between two applications, the Windows API provides *Message Pipes*, which are a system of buffers and handles that even allow your application to talk to an application on another system. There are two types of message pipes: anonymous and named. *Anonymous pipes* are usually used to communicate between a parent "shell" application and its child processes. *Named pipes* can talk to any application, including those on another system. We will be reviewing only named pipes in this chapter.

Note: Message Pipes are actually outdated just like DDE. We present them here for the sake of completeness. For new development, you should consider using Window Sockets instead, which are presented in the section "Window Sockets" on page 86.

Opening and Closing

For your application to create a named pipe to which other applications can connect, you would use

```
HANDLE CreateNamedPipe(
    LPCTSTR lpName,            // name of pipe in the form:
                               // where you provide
                               // servername and pipename
```

```
    DWORD dwOpenMode,          // open mode:
                               // PIPE_ACCESS_DUPLEX means
                               //     pipe is read/write
                               // PIPE_ACCESS_INBOUND means read only
                               // PIPE_ACCESS_OUTBOUND means write only
    DWORD dwPipeMode,          //     pipe-specific modes:
                               // PIPE_TYPE_BYTE means data is
                               //     written as a stream of
                               //     potentially unrelated bytes.
                               // PIPE_TYPE_MESSAGE means each time
                               //     a block of data is written, it's
                               //     treated like a message packet.
    DWORD nMaxInstances,       // max number of instances of this pipe
    DWORD nOutBufferSize,      // output buffer size, in bytes
    DWORD nInBufferSize,       // input buffer size, in bytes
    DWORD nDefaultTimeOut,     // time-out time, in milliseconds
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
                               // pointer to security attributes
    );
```

If another application wanted to connect to this pipe, it would call

```
HANDLE hPipe = CreateFile(
    LPCTSTR lpFileName,              // pipe name created above
    DWORD dwDesiredAccess,           // GENERIC_READ and/or
                                     //     GENERIC_WRITE
    DWORD dwShareMode,               // FILE_SHARE_READ and/or
                                     //    FILE_SHARE_WRITE
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                     // security
    DWORD dwCreationDisposition,     // always OPEN_EXISTING
    DWORD dwFlagsAndAttributes,      // file attributes:
                                     // FILE_FLAG_OVERLAPPED makes this
                                     //     an asynchronous connection
    HANDLE hTemplateFile             // copy attributes from this file
    );
```

To close a pipe, you can call

```
CloseHandle(
    HANDLE hPipe
    );
```

Reading and Writing

Once a pipe connection has been established, you can read and write to it using the following two functions.

```
BOOL ReadFile(
    HANDLE hPipe,                   // pipe handle created above
    LPVOID lpBuffer,                // buffer that receives data
    DWORD nNumberOfBytesToRead,     // number of bytes to read
    LPDWORD lpNumberOfBytesRead,    // number of bytes actually read
    LPOVERLAPPED lpOverlapped       // used with asynchronous
                                    //     communication
    );

BOOL WriteFile(
    HANDLE hPipe,                      // pipe handle
    LPCVOID lpBuffer,                  // data to write
    DWORD nNumberOfBytesToWrite,       // number of bytes to write
    LPDWORD lpNumberOfBytesWritten,    // number of bytes actually
                                       //     written
    LPOVERLAPPED lpOverlapped          // used with asynchronous
                                       //     communication
    );
```

Although the Message Pipe API contains a lot of additional functionality to support asynchronous communication, you need not be concerned with it as long as you use `ReadFile()` and `WriteFile()` from within their own threads, as discussed previously. Synchronous communication is always much simpler and, therefore, less prone to bugs.

Review

Message Pipes allow you to set up a permanent connection between applications, which allows you to communicate as if you were reading and writing from a flat disk file. Please see Figure 3.3.

Figure 3.3 Message Pipes

Although Message Pipes represent a quantum leap over simple messaging, you can probably forget all about them. MFC doesn't provide any classes for them and you won't find any examples in this book. Message Pipes were presented for background only because the Windows API provides a much more flexible solution to communication through Window Sockets.

Window Sockets

Window Sockets are essentially Message Pipes that conform to the UNIX sockets implementation of the Berkeley Software Distribution (v4.3) and, therefore, allow your application to talk to an application on any system that supports this standard (including, of course, the many flavors of UNIX).

Even though Window Sockets are typically used to communicate over a network, you can still use them to communicate between applications on the same system. Therefore, you have lots of flexibility when configuring your applications. You can install them all on the same system, but still have the option of putting another application on another system.

Your application has three levels of Window Socket support to choose from, including direct Windows API access, an MFC class that simply wraps that API called `CAsyncSocket`, and an upper level MFC class called `CSocket`. For the sake of simplicity, you should only use `CSocket`. Since `CSocket` is derived from `CAsyncSocket`, you still have access to all of the lower class's functionality, and `CSocket` provides some functionality that makes communicating with a non-Windows system easier.

Communication with Window Sockets is accomplished with three sockets. In addition to the socket that each application creates to talk through, a third socket is created to "listen" for new connections. In other words, a Server application creates this third socket to look for new connections from another application.

Opening and Closing

To allow your application to listen for a connection request, you would use

```
CListenSocket listenSock;
listenSock.Create(
    UINT nPort          // between 1025 and 0xffffffff set by you to
                        //     identify this listener to your other apps
    );
listenSock.Listen()     // start listening
```

The `CListenSocket` class used here is your own derivation of the `CSocket` class. We don't use `CSocket` directly because we need to override one of its member functions, as we will soon see.

If another application wanted to connect to this socket, it would call

```
CSocket sock;
sock.Create();                // take the defaults
sock.Connect(
    LPCTSTR lpszHostAddress,  // system address of application
                              //     with listening socket specified as:
                              //     "" or "128.23.1.22" or
                              //     "localhost" to talk to an
                              //     application server on
                              //     the same system
    UINT nHostPort            // the port number specified when creating
                              //     the listening socket
    );
```

In this example, we created the sock object on the stack for clarity. Normally, you would either make this a member variable of a class or allocate it in the heap. Once the sock object is destroyed, the connection is closed.

To complete this connection, we need to override a member function of our `CListenSock` class called `OnAccept()`. This function is called whenever `CListenSocket` senses that another application is knocking on the door. In

`OnAccept()`, we create the third and final socket, which talks directly to the other application like so.

```
CListenSocket::OnAccept()
{
    CSocket sock;
    listenSock.Accept( sock );
}
```

Again, make sure in real life to create the sock where it won't get destroyed. To close a connection on purpose, you can call

```
sock.Close();
```

Reading and Writing

Once a Window Socket connection has been established, you can read and write to it using the following two functions.

```
int numBytesReceived =          // number of bytes received
    Receive(
        void* lpBuf,            // buffer to contain data
        int nBufLen,            // length of buffer
        int nFlags = 0          // if set to MSG_PEEK, will cause
                                //     Receive() to leave received
                                //     data in socket queue, but data
                                //     is still copied to lpBuf
        );

int numBytesSent =              // actual number of bytes sent
    Send(
        const void* lpBuf,      // bytes to send
        int nBufLen,            // number of bytes to send
        int nFlags = 0
        );
```

`Receive()` and `Send()` are both synchronous functions. Therefore, you should execute them in their own threads, as seen in Example 51. Another hazard in using `Receive()` is that if you request more bytes than will be sent, it will never process the message. To remedy this, you should construct your messages so that they have a fixed size header that contains a total message

size variable. `Receive()` can then be set up to receive the header size and then use that size variable to receive any more data.

To see a real example of this, please refer to Example 51.

Serializing Over a Window Socket

When using `CSocket`'s `Send()` and `Receive()` member functions (actually inherited from `CAsyncSocket`), you are responsible for stuffing your message into any data structures and for straightening out byte orders and string formats in communications from non-Windows MFC systems such as the Mac. However, if you use another MFC class, `CSocketFile`, this tedium will be taken care of for you—but only if both applications are created with MFC. `CSocketFile` accomplishes this using the same technique you use when saving a document: Serialization. For more on Serialization please refer to your MFC documentation or my previous book, ***Visual C++ MFC Programming by Example***.

You can read from an application using serialization with

```
CSocketFile sockFile( &sock );    // where sock is your socket object
CArchive ar( &sockFile,CArchive::load );

ar >> data;                       // and any other data
```

You can write to an application using

```
CSocketFile sockFile( &sock );
CArchive ar( &sockFile,CArchive::store );

ar << data;
```

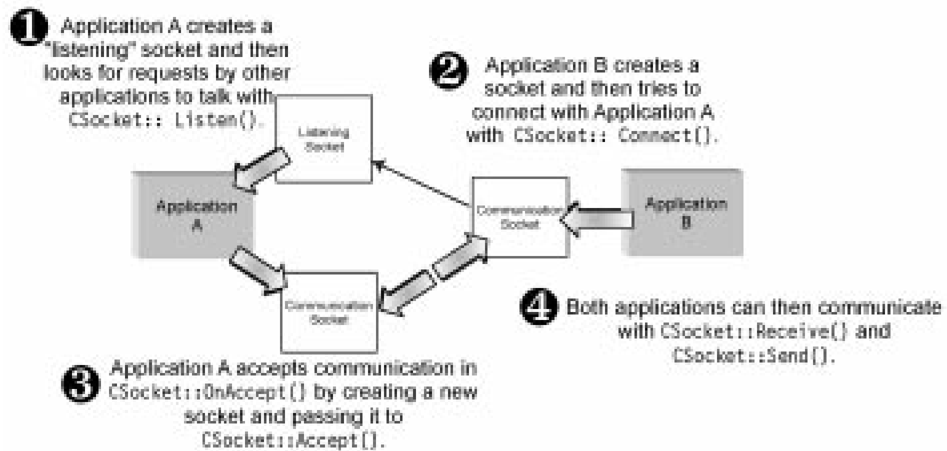
Streams vs. Datagrams

A Window Socket can be in one of two modes: stream or datagram. All of the examples presented in this book use the stream mode. Only a handful of applications use datagrams, which require less overhead but can be very unreliable. What kind of application can use something that's unreliable? One example is an application that tries to synchronize the clocks of all the systems on a network. Since this type of application is continually sending out messages, perhaps once an hour, it doesn't matter that one or two messages get lost.

Review

Window Sockets allow two or more applications to communicate, even on non-Windows platforms. A special "listening" socket is created first by an application to listen for a request to talk from another application. This listening application then creates another socket to talk through. Please see Figure 3.4.

Figure 3.4 Window Sockets



Serial/Parallel Communication

Serial and parallel communication are usually reserved for talking to an embedded device, such as a network node, a printer, or a telephone switch, through the ports on the back of your system. Communicating with these devices, however, isn't that much different from accessing a flat file on a disk. In fact, you use the same MFC `CFile` class. Again, the only difference has to do with the nature of communication: performing synchronous reads and writes in a thread and not trying to read more bytes than are available. Please see Example 52.

Opening and Closing

To open a serial or parallel port for communication you would use

```
CFile file;
CFileException e;
file.Open(
    portName,    // examples "COM1", "COM2", "LPT2"
    CFile::modeReadWrite,
    &e
    );
```

Reading and Writing

To read or write from this port you would use

```
UINT nBytes =           // actual number of bytes read
    Read(
        void* lpBuf,    // buffer to store bytes
        UINT nCount     // number of bytes to read
        );

file.Write(
    void* lpBuf,        // buffer to write
    UINT nCount         // number of bytes to write
    );
```

Configuring the Port

Although all parallel ports are created equal, you must set up a serial port to match the device to which it's talking. That means you need the same baud rate, parity, stop bits, etc. Although you can set these values using the operating system, you can also programmatically set them using the Windows API with `SetCommState()`. Typically, you will start by getting the current configuration.

```
DCB dcb;
::GetCommState( file.m_hFile, &dcb );
dcb.BaudRate = 1200, ...;
```

```
dcb.ByteSize = 7 or 8;
dcb.StopBits = 0,1,2 = 1, 1.5, 2;
dcb.Parity = 0-4 = no,odd,even,mark,space;
::SetCommState( file.m_hFile, &dcb );
```

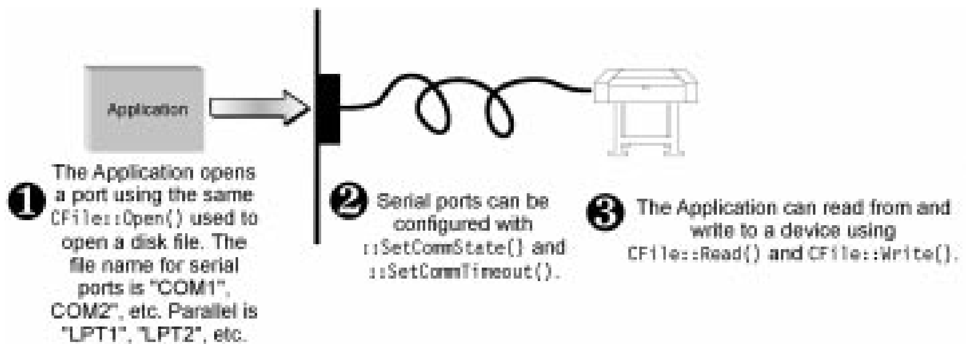
Both serial and parallel communications can timeout, which can be undesirable—especially if you perform a synchronized read within a thread that should continue to read until a message comes in. To turn off this timeout, you can use `::SetCommTimeout()`, like so.

```
COMMTIMEOUTS cto;
::GetCommTimeouts( file.m_hFile, &cto );
cto.ReadIntervalTimeout = 0;
cto.WriteTotalTimeoutMultiplier = 0;
cto.WriteTotalTimeoutConstant = 0;
::SetCommTimeouts( file.m_hFile, &sto );
```

Review

Serial and parallel communication, then, is achieved with the same API syntax as accessing a flat file—the systems virtual drivers take care of the specifics. Only serial communication involves extra consideration to match its characteristics (e.g., baud rate, etc.) with the device on the other side. Please see Figure 3.5.

Figure 3.5 Serial and Parallel Communication



Internet Communication

Several MFC classes encapsulate the Windows API's Internet Extensions (WinInet), allowing you C++ access to the Internet using one of four protocols: File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), gopher, or file. You connect to an Internet website using MFC's `CInternetSession` class. You can then access a file directly with one of four MFC classes: `CStdioFile`, `CHttpFile`, `CGopherFile`, or `CInternetFile`. You can also control a website with one of three connection classes: `CFtpConnection`, `CHttpConnection`, or `CGopherConnection`. Both the file classes and connection classes are created by the `CInternetSession` class.

Opening and Closing Files

To open an Internet file, you first open an Internet session with

```
CInternetSession();    // there are several arguments but you
                       //     will typically use the defaults
```

You can then open a particular type of file by using the `OpenURL()` member function of `CInternetSession`.

```
CStdioFile*pFile =         // see Table 3.1 for returned object type
    session.OpenURL(
        LPCTSTR pstrURL    // see Table 3.1 for sample values
        );
// There are several other arguments, however you will typically
//     use the default values.
```

The type of URL that you open determines the class object that `CInternetSession` creates for you, based on Table 3.1

Table 3.1 `OpenURL()` **Argument Specification**

URL Type	Internet Class Pointer Returned
file://	CStdioFile*
http://	CHttpFile*
gopher://	CGopherFile*
ftp://	CInternetFile*

File objects created this way can only be read. To write files and control the website, refer to the section "Opening and Closing Connections" on page 94.

Reading Files

Reading from any of these file types is similar to reading from a flat file. In fact, `CHttpFile`, `CGopherFile`, and `CInternetFile` are derived from `CStdioFile`, which allows you to read a file by number of bytes or by string.

```
UINT pFile->Read( void* lpBuf, UINT nCount );

pFile->ReadString( CString str );
```

`CHttpFile` also allows you to access an HTTP file by object and verb.

Opening and Closing Connections

The `CInternetSession` class also has three member functions that allow you to open a connection to a website. With a connection, not only can you read and write files, but you can also control the site. For example, an FTP connection allows you to programmatically execute FTP commands on an FTP site.

FTP Connection

To open an FTP connection, you would use

```
CFtpConnection* ftp = GetFtpConnection();
CInternetFile* pFile =
    ftp.OpenFile(
        LPCTSTR pstrFileName,
        DWORD dwAccess = GENERIC_READ,              // and/or
                                                    // GENERIC_WRITE
        DWORD dwFlags = FTP_TRANSFER_TYPE_BINARY,   // use default
        DWORD dwContext = 1                         // use default
        );
```

To then open a file on that site, you can use

```
CFtpFile *ftpFile = ftp.OpenFile(
    // same as CStdioFile open
    );
```

With this file object, you can read and write.

Gopher Connection

To open a gopher connection, you would use

```
CGopherConnection* gopher = GetGopherConnection();
```

To then open a file on that site, you can use

```
CGopherFile *gopherFile = gopher.OpenFile(
    // same as CStdioFile Open()
    );
```

HTTP Connection

To open an HTTP connection, you would use

```
CHttpConnection* http = GetHttpConnection();
```

To then open a file on that site, you can use

```
CHttpFile* http.OpenRequest(
    // please refer to your MFC documentation for these arguments
    );
```

Other Internet Classes

Three other MFC Internet client classes of interest are `CFtpFileFind`, `CGopherFileFind`, and `CGopherLocator`. `CFtpFileFind` and `CGopherFileFind` are derived from the `CFindFile` class and can locate files over the Internet. The `CGopherLocator` class gets a gopher “locator” from a gopher server and makes the locator available to `CGopherFileFind`.

Communication Summary

The communication method you choose depends on your application. For simple, limited communication, you should use windows messaging. For extended or sophisticated communication, you should use Window Sockets. If you will be communicating with an embedded device, such as a printer, you should use serial/parallel communication. You really have no choice for Internet communication.

You shouldn’t use DDE or Message Pipes at all in new development. Window Sockets are a much more flexible solution to interprocess communication and have class support in the foundation classes.

Although DDE represented a faster way to communicate data than Window Sockets, there really is no alternative to sharing data over a network other than to send one byte through the wire at a time.

Sharing Data

So far, we have covered active communication between applications in which both applications had to participate. Your application also has two ways to communicate passively by allowing other applications to access its data.

Shared Memory File uses the same globally allocated memory that DDE uses and, therefore, can't be accessed from another machine.

File Mapping allows your application to share a file, and even memory, with another application—even over the network to another Windows machine.

Note: File Mapping is still not an option if you want to share data with a non-Windows platform (e.g., UNIX). In this case, you are still stuck with Window Sockets. But don't worry, File Mapping doesn't afford your application any faster data access then Window Sockets—it still shares its data one byte at a time over the network.

Note: Your application can also share data through the clipboard. However, the clipboard is typically reserved for user interaction.

Shared Memory File

If you don't need to follow the DDE standard and you don't need to share your data across the network, you can use your own globally allocated memory to share data with other applications. In fact, MFC's `CSharedFile` class makes creating and accessing global memory just as easy as creating a flat file.

Creating and Destroying

Just creating and destroying an instance of the `CSharedFile` class will create and destroy your global memory.

```
CSharedFile file;
```

Reading and Writing

Reading and writing to this file is identical to reading and writing bytes from a flat file. Therefore, for a complete transfer of data scenario, you would write with

```
CSharedFile file;
file.Write( buffer, nBytes );
// extract global memory handle from CFile object to send another
application
HGLOBAL hgbl = file.Detach();
SendMessage( hWnd, WM_MYMESSAGE, ( WPARAM ) hgbl,0 );
```

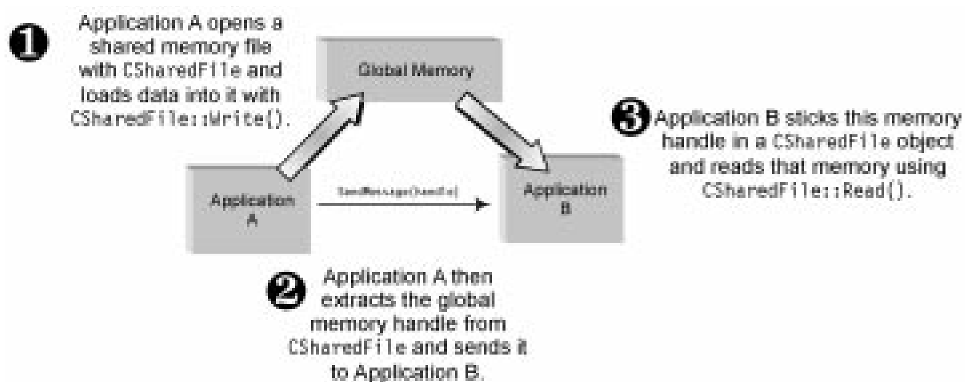
You would read in the receiving program with

```
HRESULT OnMyMessage( WPARAM wParam, LPARAM lParam )
{
    CSharedFile file;
    // encapsulate global memory handle in this CFile object
    file.SetHandle( ( HGLOBAL ) wParam );
    file.Read( buffer,nBytes );
}
```

Review

Data can be shared through global memory using the `CSharedFile` class using the same function syntax as you would use accessing a flat file. Please see Figure 3.6.

Figure 3.6 Shared Memory Files



File Mapping

If you need to share data among applications that can potentially be on other Windows platforms, you can use File Mapping. Unlike globally allocated memory, which can't be accessed by more than one application at once, File Mapping allows two or more applications to simultaneously share memory. For example, you can have an array called `Bob[100]` in two applications, such that storing `34` in `Bob[23]` in Application A causes `34` to appear in `Bob[23]` in Application B.

This sharing actually takes place within a file—thus, the name File Mapping. When sharing data on just one system, you will typically use the default file, which is actually the swap file used to provide your system with virtual memory. If you will be sharing over a network, you will need to open the file yourself and provide its handle.

Opening and Closing

To open a segment of the swap file for shared memory, you would use

```
m_hMap = ::CreateFileMapping(
    ( HANDLE ) 0xffffffff,    // or can be an open file handle
    0,                        // security
    PAGE_READWRITE,           // or PAGE_READONLY or PAGE_WRITECOPY
    0,                        // size -- high order
                              //     (required if no file handle)
    0x1000,                   // size -- low order
                              //     (required if no file handle)
    MAP_ID                    // unique id--required if no file handle
    );
```

This returned map handle can be turned into a memory pointer with

```
m_pSharedData = ::MapViewOfFile( m_hMap,
    FILE_MAP_WRITE,    // or FILE_MAP_READ, FILE_MAP_COPY
                       //     (FILE_MAP_WRITE is read/write)
    0,                 // offset --high order
    0,                 // offset -- low order
    0                  // number of bytes (zero maps entire file)
    );
// When using swap file, offset must be zero(0)
```

To close this shared memory, you would use

```
::UnmapViewOfFile(m_pSharedData);
::CloseHandle(m_hMap);
```

Reading and Writing

If, as mentioned previously, we were sharing an array called `Bob`, we could simply use the following in any application

```
int *Bob = ( int * ) pSharedData;
Bob[23] = 34;
```

Data Synchronization

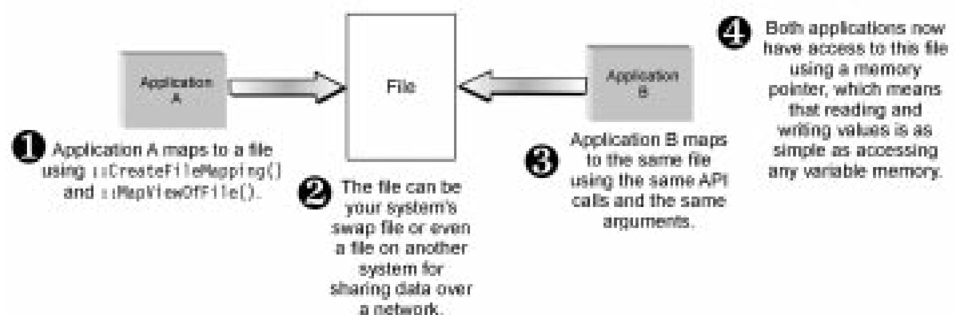
All access to mapped memory on the same system is synchronized. In other words, two applications can't access the same memory location at the same time. This prevents, for instance, one application from reading from an area of memory at the same time another application is writing to it.

Mapping a file over a network, however, can run into trouble. If the file through which you are sharing is on another system, your application will actually be writing to a network buffer that, over time, will be sent to the other system. In the meantime, another application on another system could also be writing to that file. In this case, you will need to manually make sure your applications aren't clobbering each other.

Review

Memory sharing can be achieved through a file. There is no other way for two applications to share the same address space. Please see Figure 3.7.

Figure 3.7 File Mapping



Client/Server

Not only can you send a message to another application or share data with it, but you can even access its functionality by calling its member functions indirectly. An application that shares its functionality this way is called a Server—it's there to serve you. Applications that access this functionality are called Clients—they're the customers.

Allowing one application to share its functionality with others can be accomplished with any of the communication methods presented previously with essentially the same steps.

1. Create a new command message to represent each function you want to access (e.g., `IDC_CALL_HOME` for the `Home()` method).
2. Send this message to the application that's sharing its functionality (the Server) and pass along in the same message any arguments required by the desired function.

```
wParam = x;
lParam = y;
SendMessage( hWnd,IDC_CALL_HOME,wParam,lParam );
```

3. The Server will then convert your message and arguments into an actual function call to the desired function.

```
case IDC_CALL_HOME:
    x = wParam;
    y = lParam;
    res = Home( x,y );
    :    :    :
```

4. Any values returned by the function are returned to your application, either with this message

```
return res;
```

or with a new message

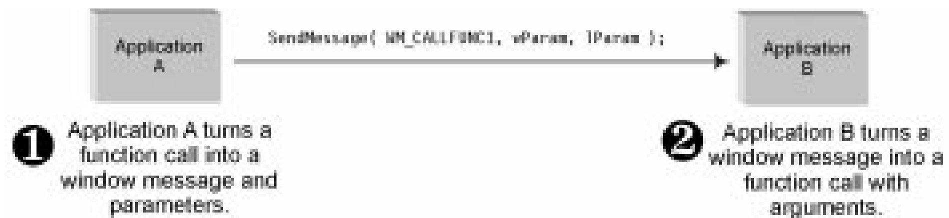
```
wParam = y;
SendMessage( IDC_CALL_HOME_REPLY,wParam,lParam );
```

5. The Client can then respond to a reply message by sticking the following values into local memory.

```
case IDC_CALL_HOME_REPLY:
    y = wParam;
```

Please see Figure 3.8 for an overview of this technique.

Figure 3.8 Indirectly Calling an External Function



Passing Calling Arguments

The method you use to pass your arguments will depend on the method you're using to communicate. As seen previously, Windows Messaging allows you to pass two arguments with the message. To pass additional arguments, you can stick them in a structure and pass that structure using DDE, File Mapping, or even some globally shared memory.

You can't just simply pass a pointer to this structure to the other application because the structure exists in a different address space than the other application (a pointer in one application wouldn't point to the same data in another application). This also means you can't stick pointers in your message structure—the contents of entire arrays and argument data must be contained in this structure for it to be accessible to the other application.

When the Server Application is on another system, you can't even use DDE or global memory to pass your arguments—there's no way for the other system to access it. You can still use File Mapping to contain the arguments or you can pass them in the message itself. If you pass the arguments in the communication, any return values must also be communicated back to the Client Application. That means if the Server can potentially modify an array, the entire array must be sent back to the Client. File Mapping accomplishes the same thing, except in the background. Either way, getting large or numerous arguments back and forth can be slow—so you should plan accordingly.

Remote Procedure Calling

Arguments are most commonly passed between systems in the communication itself. In fact, there's a Windows API that will handle a lot of the tedium of accessing a Server Application over a network called Remote Procedure Call (RPC) and it sends the arguments in the message. RPC is used extensively by Microsoft's Component Object Model (COM) components, but it falls outside of the scope of this book.

Summary

In this chapter, we discovered six methods available to your application to communicate with its environment.

- **Windows messaging** for simple interprocess communication
- **Dynamic Data Exchange** for sending large amounts of data
- **Message Pipes** for communication over a network
- **Window Sockets** for communication over a network to non-Windows platforms
- **Internet Classes** for communicating over the Internet
- **Serial/Parallel Communication** for talking to devices connected to your system's serial or parallel port

We also found that Window Sockets supersede DDE and Message Pipes for the ability to distribute your application over a network.

For sharing data between applications, we reviewed two methods.

- **Share Memory Files** are created by the MFC `CShareFile` class to wrap globally allocated memory for easy C++ access.
- **File Mapping** allows sharing data, even across a network—but not to non-Windows platforms.

This concludes the pure text portion of this book. In the following sections, we will look at MFC and Visual C++ from the other angle. Actual features you might want to add to your own application will be presented, along with a step-by-step guide to implementing them. And yes, there will be more notes explaining how these features are implemented so that they might be broadened and enhanced.