

# Chapter 19 Mailslots

## Chapter 19 Mailslots

### 19.1 Mailslot Implementation Details

#### 19.1.1 Mailslot Names

#### 19.1.2 Message Sizing

#### 19.1.3 Compiling Applications

#### 19.1.4 Error Codes

### 19.2 Basic Client/Server

#### 19.2.1 Mailslot Server Details

#### 19.2.2 Mailslot Client Details

### 19.3 Additional Mailslot APIs

### 19.4 Platform and Performance Considerations

#### 19.4.1 8.3-Character Name Limits

#### 19.4.2 Inability to Cancel Blocking I/O Requests

#### 19.4.3 Timeout Memory Leaks

### 19.5 Conclusion

Microsoft Windows NT, Windows 95, Windows 98, and Windows Me platforms (but not Windows CE) include a simple one-way interprocess communication (IPC) mechanism known as mailslots. In simplest terms, mailslots allow a client process to transmit or broadcast messages to one or more server processes. Mailslots can assist transmission of messages among processes on the same computer or among processes on different computers across a network. Developing applications using mailslots is simple, requiring no formal knowledge of underlying network transport protocols such as TCP/IP or IPX. Because mailslots are designed around a broadcast architecture, you can't expect reliable data transmissions using mailslots. They can be useful, nevertheless, in certain types of network programming situations in which delivery of data isn't mission-critical.

One possible scenario for using mailslots is developing a messaging system that includes everyone in your office. Imagine that your office environment has a large number of workstations. The office is suffering from a soda shortage, and every workstation user in your office is interested in knowing every few minutes how many sodas are available in the vending machine. Mailslots lend themselves well to this type of situation. You can easily implement a mailslot client application that monitors the soda count and broadcasts to every interested workstation user the total number of available sodas at five-minute intervals. Because mailslots don't guarantee delivery of a broadcast message, some workstation users might not receive all updates. A few transmission failures won't be a problem in this case because messages sent at

five-minute intervals with occasional misses are still frequent enough to keep the workstation users well informed.

The major limitation of mailslots is that they permit only unreliable one-way data communication from a client to a server. The biggest advantage of mailslots is that they allow a client application to easily send broadcast messages to one or more server applications.

This chapter explains how to develop a mailslot client/server application. We'll describe mailslot naming conventions before we address the message sizing considerations that control the overall behavior of mailslots. Next we'll show you the details of developing a basic client/server application. At the end of this chapter, we'll tell you about known problems and limitations of mailslots and offer workaround solutions.

## 19.1 Mailslot Implementation Details

Mailslots are designed around the Windows file system interface. Client and server applications use standard Win32 file system I/O functions, such as ReadFile and WriteFile, to send and receive data on a mailslot and take advantage of Win32 file system naming conventions. Mailslots rely on the Windows redirector to create and identify mailslots using a file system named the Mailslot File System (MSFS). Chapter 18 described the Windows redirector in greater detail.

### 19.1.1 Mailslot Names

Mailslots use the following naming convention for identification:

`\\server\Mailslot\[path]name`

This string is divided into three portions: `\\server`, `\Mailslot`, and `\[path]name`. The first string portion, `\\server`, represents the name of the server on which a mailslot is created and on which a server application is running. The second portion, `\Mailslot`, is a hard-coded mandatory string for notifying the system that this filename belongs to MSFS. The third portion, `\[path]name`, allows applications to uniquely define and identify a mailslot name; the path portion might specify multiple levels of directories. For example, the following types of names are legal for identifying a mailslot:

`\\Oreo\Mailslot\Mymailslot`

`\\Testserver\Mailslot\Cooldirectory\Funtest\Anothermailslot`

`\\.\Mailslot\Easymailslot`

`\\*\Mailslot\Myslot`

The server string portion can be represented as a dot (.), an asterisk (\*), a domain name, or a server name. A domain is simply a group of workstations and servers that share a common group name. We'll examine mailslot names in greater detail later in this chapter, when we cover implementation details of a simple client.

Because mailslots rely on the Windows file system services for creation and transferring data over a network, the interface protocol is independent. When creating your application, you don't have to worry about the details of underlying network transport protocols to form communications among processes across a network. When mailslots communicate remotely to computers across a network, the Windows file system services rely on the Windows redirector to send data from a client to a server using the SMB protocol. Messages are typically sent via connectionless transfers, but you can force the Windows redirector to use connection-oriented transfers on the Windows NT platform, depending on the size of your message.


### 19.1.2 Message Sizing

Mailslots normally use datagrams to transmit messages over a network. Datagrams are small packets of data that are transmitted over a network in a connectionless manner. Connectionless transmission means that each data packet is sent to a recipient without packet acknowledgment. This is unreliable data transmission, so you cannot guarantee message delivery. However, connectionless transmission does give you the capability to broadcast a message from one client to many servers. The exception to this occurs on Windows NT platforms when messages exceed 424 bytes.

On Windows NT platforms, messages larger than 426 bytes are transferred using a connection-oriented protocol over an SMB session instead of using datagrams. This allows large messages to be transferred reliably and efficiently. However, you lose the ability to broadcast a message from a client to many servers. Connection-oriented transfers are limited to one-to-one communication: one client to one server. Connection-oriented transfers normally provide reliable guaranteed delivery of data between processes, but the mailslot interface on Windows NT platforms does not guarantee that a message will actually be written to a mailslot. For example, if you send a large message from a client to a server that does not exist on a network, the mailslot interface does not tell your client application that it failed to submit data to the server. Because Windows NT platforms change their transmission method based on message size, an interoperability problem occurs when you send large messages between a machine running Windows NT and a machine running Windows 95, Windows 98, or Windows Me.

Windows 95, Windows 98, and Windows Me platforms deliver messages using datagrams only, regardless of message size. If a client running one of these operating systems attempts to send a message larger than 424 bytes to a Windows NT platform, Windows NT will accept the first 424 bytes and truncate the remaining data.

Windows NT expects larger messages to be sent over a connection-oriented SMB session. A similar problem exists in transferring messages from a Windows NT client to a Windows 95, Windows 98, or Windows Me server. Remember that Windows 95, Windows 98, and Windows Me receive data via datagrams only. Because Windows NT transfers data via datagrams for messages 426 bytes or smaller, Windows 95, Windows 98, and Windows Me cannot receive messages larger than 426 bytes from such clients. Table 19-1 outlines these message size limitations in detail.

 Windows CE was intentionally left out of Table 19-1 because the mailslot-programming interface is not available. Also note that messages sized 425 to 426 bytes are not listed in this table due to a Windows NT redirector limitation.

**Table 19-1** *Mailslot Message Size Limitations*

Transfer Direction	Connectionless Transfer Using Datagrams	Connection- Oriented Transfer
Windows 95, Windows 98, Windows Me -> Windows 95, Windows 98, Windows Me	Message size up to 64 KB.	Not supported.
Windows NT -> Windows NT	Messages must be 424 bytes or less.	Messages must be greater than 426 bytes.
Windows NT -> Windows 95, Windows 98, Windows Me	Messages must be 424 bytes or less.	Not supported.
Windows 95, Windows 98, Windows Me -> Windows NT	Messages must be 424 bytes or less; otherwise, the message is truncated.	Not supported.

Another limitation of Windows NT platforms is worth discussion because it affects datagram data transmissions. The Windows NT redirector cannot send or receive a complete datagram message of 425 or 426 bytes. For example, if you send out a message from a Windows NT client to a Windows 95, Windows 98, Windows Me, or Windows NT server, the Windows NT redirector truncates the message to 424 bytes before sending it to the destination server.

To accomplish total interoperability among all Windows platforms, we strongly recommend limiting message sizes to 424 bytes or less. If you are looking for

connection-oriented transfers, consider using named pipes instead of mailslots. Named pipes are covered in Chapter 20.

### 19.1.3 Compiling Applications

When you build a mailslot client or server application using Microsoft Visual C++, your application must include the WINBASE.H include file in your program files. If you include WINDOWS.H (as most applications do) you can omit WINBASE.H. Your application is also responsible for linking with KERNEL32.LIB, which is typically configured with the Visual C++ linker flags.

### 19.1.4 Error Codes

All Win32 API functions that are used in developing mailslot client and server applications (except for CreateFile and CreateMailslot) return the value 0 when they fail. The CreateFile and CreateMailslot API functions return INVALID\_HANDLE\_VALUE. When these API functions fail, applications should call the GetLastError function to retrieve specific information about the failure. For a complete list of error codes, see the standard Windows error codes in Chapter 21 or consult the header file WINERROR.H.

## 19.2 Basic Client/Server

As we mentioned earlier, mailslots feature a simple client/server design architecture in which data can flow only from a client to a server. The data communication model is one-way, or unidirectional. The server is responsible for creating a mailslot and is the only process that can read data from it. Mailslot clients are processes that open instances of mailslots and are the only processes that can write data to them.

### 19.2.1 Mailslot Server Details

Implementing a mailslot requires developing a server application to create a mailslot. The following steps describe how to write a basic server application:

1. Create a mailslot handle using the CreateMailslot API function.
2. Receive data from any client by calling the ReadFile API function using the mailslot handle.
3. Close the mailslot handle using the CloseHandle API function.

As you can see, very few API calls are needed to develop a mailslot server application.

Server processes create mailslots using the CreateMailslot API call, which is defined as follows:

```
HANDLE CreateMailslot(  
    LPCTSTR lpName,  
    DWORD nMaxMessageSize,  
    DWORD lReadTimeout,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes  
);
```

The first parameter, `lpName`, specifies the name of the mailslot. The name must have the following form:

`\\.\Mailslot\[path]name`

Notice that the server name is represented as a dot, which represents the local machine. This is required because you cannot create a mailslot on a remote computer. In the `lpName` parameter, `name` must represent a unique name. This might simply be a name, or a full directory path might precede it.

The `nMaxMessageSize` parameter defines the maximum size—in bytes—of a message that can be written to a mailslot. If a client writes more than `nMaxMessageSize` bytes, the server doesn't see the message. Specifying the value 0 allows the server to accept a message of any size.

Read operations can operate in blocking or nonblocking mode on a mailslot, depending on the `lReadTimeout` parameter, which determines the amount of time in milliseconds that read operations wait for incoming messages. Specifying the value `MAILSLOT_WAIT_FOREVER` allows read operations to block and wait indefinitely until incoming data is available to be read. If you specify 0, read operations return immediately. We discuss details of reading later in this chapter. The `lpSecurityAttributes` parameter determines access control rights to a mailslot. Using Windows 95, Windows 98, or Windows Me, this parameter must be NULL because you cannot apply security to objects. Using the Windows NT platform, this parameter is only partially implemented, so you should also specify a NULL parameter. The only security that you can enforce on a mailslot is for local I/O, in which a client attempts to open a mailslot with a dot (.) for the server name. A client can get around this security by specifying the server's actual name instead of a dot (.), as when making a remote I/O call. The `lpSecurityAttributes` parameter is not implemented for remote I/O on the Windows NT platform because of the extreme inefficiency of forming an authenticated session between the client and the server every time a message is sent. Mailslots, therefore, only partially follow the Windows NT security model found in the standard file systems. As a consequence, any mailslot client on your network can send data to your server.

After a mailslot is created with a valid handle, you can begin reading data. The server is the only process that can read data from a mailslot. The server should use the Win32 ReadFile function to accomplish this. ReadFile is defined as follows:

```
BOOL ReadFile(  
    HANDLE hFile,  
    LPVOID lpBuffer,  
    DWORD nNumberOfBytesToRead,  
    LPDWORD lpNumberOfBytesRead,  
    LPOVERLAPPED lpOverlapped  
);
```

CreateMailslot returns the handle hFile. The lpBuffer and nNumberOfBytesToRead parameters determine how much data can be read off a mailslot. It is important to make the size of this buffer greater than the nMaxMessageSize parameter from the CreateMailslot API call. Additionally, the buffer must be larger than incoming messages on the mailslot; if it is not larger, ReadFile will fail with the error ERROR\_INSUFFICIENT\_BUFFER. The lpNumberOfBytesRead parameter reports the actual number of bytes read when the ReadFile operation completes.

The lpOverlapped parameter provides a way to read data asynchronously off a mailslot. This parameter uses the Win32 overlapped I/O mechanism, which we describe in greater detail in Chapter 20. By default, the ReadFile operation blocks (waits) on I/O until data is available for reading. Overlapped I/O can be accomplished only on the Windows NT platform; you should specify NULL for this parameter when using Windows 95, Windows 98, or Windows Me. The following code further demonstrates how to write a simple mailslot server application.

```
// Server1.cpp
```

```
#include <windows.h>  
#include <stdio.h>
```

```
void main(void)  
{
```

```
    HANDLE Mailslot;  
    char buffer[256];  
    DWORD NumberOfBytesRead;
```

```
    // Create the mailslot
```

```
    if ((Mailslot = CreateMailslot("\\\\.\\Mailslot\\Myslot", 0,  
        MAILSLOT_WAIT_FOREVER, NULL)) ==  
        INVALID_HANDLE_VALUE)
```

```
    {  
        printf("Failed to create a mailslot %d\n", GetLastError());
```

```

        return;
    }

    // Read data from the mailslot forever!
    while(ReadFile(Mailslot, buffer, 256, &NumberOfBytesRead,
        NULL) != 0)
    {
        printf("%. *s\n", NumberOfBytesRead, buffer);
    }
}

```

## 19.2.2 Mailslot Client Details

Implementing a client requires developing an application to reference and write to an existing mailslot. The following steps describe how to write a basic client application:

1. Open a reference handle to the mailslot we want to send data to using the CreateFile API.
2. Write data to the mailslot by calling the WriteFile API.
3. Once you are finished writing data, close the mailslot handle using the CloseHandle API.

As we described earlier, mailslot clients communicate to mailslot servers in a connectionless manner. When a client opens a reference handle to a mailslot, the client does not form a connection to the mailslot server. Mailslots are referenced using the CreateFile API call, which is defined as follows:

```

HANDLE CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);

```

The lpFileName parameter describes one or more mailslots that can be written to using the mailslot name format described earlier. Table 19-2 describes mailslot naming conventions in greater detail. The dwDesiredAccess parameter must be set to GENERIC\_WRITE because a client can only write data to the server. The dwShareMode parameter must be set to FILE\_SHARE\_READ, allowing the server to open and perform read operations on the mailslot. The lpSecurityAttributes parameter has no effect on mailslots and should be set to NULL. The dwCreationDisposition



flag should be set to OPEN\_EXISTING. This setting is useful when a client and a server are operating on the same machine: If the server has not created the mailslot, the CreateFile API function fails. The dwCreationDisposition parameter has no effect if the server is operating remotely. The dwFlagsAndAttributes parameter should be defined as FILE\_ATTRIBUTE\_NORMAL. The hTemplateFile parameter should be set to NULL.

**Table 19-2** *Mailslot Name Types*

Name Format	Description
\\.\mailslot\name	Identifies a local mailslot on the same machine
\\servername\mailslot\name	Identifies a remote mailslot server named servername
\\domainname\mailslot\name	Identifies all mailslots of a particular name in the specified domain
\\*\mailslot\name	Identifies all mailslots of a particular name in the system's primary domain

After a handle has been successfully created, you can begin writing data to a mailslot. Remember, a client can only write data to the mailslot. This can be accomplished using the Win32 WriteFile function, defined as follows:

```

BOOL WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped
);

```

The hFile parameter is the reference handle that CreateFile returns. The lpBuffer and nNumberOfBytesToWrite parameters determine how many bytes will be sent from the client to the server. The maximum size of a message is 64 KB. If the mailslot handle was created using a domain or asterisk format, the message size is limited to 424 bytes on Windows NT and 64 KB on Windows 95, Windows 98, and Windows Me. If a client attempts to send a message that exceeds those limits, the WriteFile function fails and the GetLastError function returns ERROR\_BAD\_NETPATH. This happens because the message is sent as a broadcast datagram to all servers on the network. The lpNumberOfBytesWritten parameter returns the number of bytes sent to a server when the WriteFile operation completes.

The lpOverlapped parameter provides a way to write data asynchronously to a mailslot. Because mailslots feature connectionless data transfer, the WriteFile function is not subject to blocking on I/O calls. This parameter should be set to NULL on the client. The following code further demonstrates how to write a simple mailslot client application.

```

// Client.cpp

#include <windows.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
    HANDLE Mailslot;
    DWORD BytesWritten;
    CHAR ServerName[256];

    // Accept a command line argument for the server to send
    // a message to
    if (argc < 2)
    {
        printf("Usage: client <server name>\n");
        return;
    }
    sprintf(ServerName, "\\\\"%s\\Mailslot\\Myslot", argv[1]);

    if ((Mailslot = CreateFile(ServerName, GENERIC_WRITE,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL,
        NULL)) == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }

    if (WriteFile(Mailslot, "This is a test", 14, &BytesWritten,
        NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }

    printf("Wrote %d bytes\n", BytesWritten);

    CloseHandle(Mailslot);
}

```

## 19.3 Additional Mailslot APIs

A mailslot server application can use two additional API functions to interact with a mailslot: `GetMailslotInfo` and `SetMailslotInfo`. The `GetMailslotInfo` function retrieves message sizing information when messages become available on a mailslot.

Applications can use this to dynamically adjust their buffers for incoming messages of varying length. `GetMailslotInfo` can also be used to poll for incoming data.

`GetMailslotInfo` is defined as follows:

```
BOOL GetMailslotInfo(  
    HANDLE hMailslot,  
    LPDWORD lpMaxMessageSize,  
    LPDWORD lpNextSize,  
    LPDWORD lpMessageCount,  
    LPDWORD lpReadTimeout  
);
```

The `hMailslot` parameter identifies a mailslot returned from the `CreateMailslot` API call. The `lpMaxMessageSize` parameter points to how large a message (in bytes) can be written to the mailslot. The `lpNextSize` parameter points to the size in bytes of the next message. `GetMailslotInfo` might return the value `MAILSLOT_NO_MESSAGE`, indicating that no message is currently waiting to be received on the mailslot. A server can potentially use this parameter to poll the mailslot for incoming data, preventing your application from blocking on a `ReadFile` function call. Polling for data using this mechanism is not a good programming approach. Your application will continuously use the computer's CPU to check for incoming data—even when no messages are being processed—resulting in a slower overall performance by the computer. If you want to prevent the `ReadFile` function from blocking, we recommend using Win32 overlapped I/O. The `lpMessageCount` parameter points to a buffer that receives the total number of messages waiting to be read. You can use this parameter for polling purposes, too. The `lpReadTimeout` parameter points to a buffer that returns the amount of time in milliseconds that a read operation can wait for a message to be written to the mailslot before a timeout occurs.

The `SetMailslotInfo` API function sets the timeout values on a mailslot for how long read operations wait for incoming messages. Thus the application has the ability to change the read behavior from blocking to nonblocking mode or vice versa.

`SetMailslotInfo` is defined as follows:

```
BOOL SetMailslotInfo(  
    HANDLE hMailslot,  
    DWORD lReadTimeout  
);
```

The `hMailslot` parameter identifies a mailslot that is returned from the `CreateMailslot` API call. The `lReadTimeout` parameter specifies the amount of time in milliseconds that a read operation can wait for a message to be written to the mailslot before a timeout occurs. If you specify 0, read operations will return immediately if no message is present. If you specify `MAILSLOT_WAIT_FOREVER`, read operations will wait forever.

## 19.4 Platform and Performance Considerations

Mailslots on Windows 95, Windows 98, and Windows Me platforms have three limitations that you should be aware of: 8.3-character name limits, inability to cancel blocking I/O requests, and timeout memory leaks.

### 19.4.1 8.3-Character Name Limits

Windows 95, Windows 98, and Windows Me platforms silently limit mailslot names to an 8.3-character name format. This causes interoperability problems between Windows 95, Windows 98, Windows Me, and Windows NT. For example, if you create or open a mailslot with the name `\\.\Mailslot\Mymailslot`, Windows 95, Windows 98, and Windows Me will actually create and reference the mailslot as `\\.\Mailslot\Mymailsl`. The `CreateMailslot` and `CreateFile` functions succeed even though name truncation occurs. If a message is sent from Windows NT to Windows 95, Windows 98, or Windows Me, or vice versa, the message will not be received because the mailslot names do not match. If both the client and the server are running on Windows 95, Windows 98, or Windows Me machines, there isn't a problem—the name is truncated on both the client and the server. An easy way to prevent interoperability problems is to limit mailslot names to eight characters or less.

### 19.4.2 Inability to Cancel Blocking I/O Requests

Windows 95, Windows 98, and Windows Me platforms also have a problem with canceling blocking I/O requests. Mailslot servers use the `ReadFile` function to receive data. If a mailslot is created with the `MAILSLOT_WAIT_FOREVER` flag, read requests block indefinitely until data is available. If a server application is terminated when there is an outstanding `ReadFile` request, the application hangs forever. The only way to cancel the application is to reboot Windows. A possible solution is to have the server open a handle to its own mailslot in a separate thread and send data to break the blocking read request. The following code demonstrates this solution in detail:

```

// Server2.cpp

#include <windows.h>
#include <stdio.h>
#include <conio.h>

BOOL StopProcessing;

DWORD WINAPI ServeMailslot(LPVOID lpParameter);
void SendMessageToMailslot(void);

void main(void) {

    DWORD ThreadId;
    HANDLE MailslotThread;

    StopProcessing = FALSE;
    MailslotThread = CreateThread(NULL, 0, ServeMailslot, NULL,
        0, &ThreadId);

    printf("Press a key to stop the server\n");
    _getch();

    // Mark the StopProcessing flag to TRUE so that when ReadFile
    // breaks, our server thread will end
    StopProcessing = TRUE;

    // Send a message to our mailslot to break the ReadFile call
    // in our server
    SendMessageToMailslot();

    // Wait for our server thread to complete
    if (WaitForSingleObject(MailslotThread, INFINITE) == WAIT_FAILED)
    {
        printf("WaitForSingleObject failed with error %d\n",
            GetLastError());
        return;
    }
}

//
// Function: ServeMailslot
//
// Description:

```

```

//      This function is the mailslot server worker function to
//      process all incoming mailslot I/O
//
DWORD WINAPI ServeMailslot(LPVOID lpParameter)
{
    char buffer[2048];
    DWORD NumberOfBytesRead;
    DWORD Ret;
    HANDLE Mailslot;

    if ((Mailslot = CreateMailslot("\\\\.\\mailslot\\myslot", 2048,
        MAILSLOT_WAIT_FOREVER, NULL)) ==
INVALID_HANDLE_VALUE)
    {
        printf("Failed to create a MailSlot %d\n", GetLastError());
        return 0;
    }

    while((Ret = ReadFile(Mailslot, buffer, 2048,
        &NumberOfBytesRead, NULL)) != 0)
    {
        if (StopProcessing)
            break;

        printf("Received %d bytes\n", NumberOfBytesRead);
    }

    CloseHandle(Mailslot);

    return 0;
}

//
// Function: SendMessageToMailslot
//
// Description:
//      The SendMessageToMailslot function is designed to send a
//      simple message to our server so we can break the blocking
//      ReadFile API call
//
void SendMessageToMailslot(void)
{
    HANDLE Mailslot;
    DWORD BytesWritten;

```

```

if ((Mailslot = CreateFile("\\\\.\\mailslot\\myslot",
    GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL)) ==
INVALID_HANDLE_VALUE)
{
    printf("CreateFile failed with error %d\n", GetLastError());
    return;
}

if (WriteFile(Mailslot, "STOP", 4, &BytesWritten, NULL) == 0)
{
    printf("WriteFile failed with error %d\n", GetLastError());
    return;
}

CloseHandle(Mailslot);
}

```

### 19.4.3 Timeout Memory Leaks

The final problem with Windows 95, Windows 98, and Windows Me platforms worth mentioning is memory leaks, which can occur when you're using timeout values on mailslots. When you create a mailslot using the `CreateMailslot` function with a timeout value greater than 0, the `ReadFile` function leaks memory when the timeout expires and the function returns `FALSE`. After many calls to the `ReadFile` function, the system becomes unstable and subsequent `ReadFile` calls with timers that expire start returning `TRUE`. As a result, the system is no longer able to execute other MS-DOS applications. To work around this, create the mailslot with a timeout value of either 0 or `MAILSLOT_WAIT_FOREVER`. This prevents an application from using the timeout mechanism, which causes the actual memory leak.

The Microsoft knowledge base documents the following problems and limitations. You can access the knowledge base at <http://support.microsoft.com/support/search>. We briefly describe each issue here.

- **Q139715 ReadFile Returns Wrong Error Code for Mailslots**

If a server opens a mailslot using `CreateMailslot`, specifies a timeout, and then uses `ReadFile` to receive data, the `ReadFile` fails if no data is available. `GetLastError` returns an error code of 5 (access denied).

- **Q192276 GetMailslotInfo Returns Incorrect lpNextSize Value**

If you call the API function `GetMailslotInfo` under Windows 95 OEM Service Release 2 (OSR2) or Windows 98 without a network client component installed, you receive an incorrect value (usually in the millions) or a negative number for the `lpNextSize` parameter. If you repeatedly call the function, it usually returns the correct value.

- **Q170581 Mailslot Created on Win95 Allows Only 4093 Bytes**

If you call the `WriteFile` API function to write more than 4093 bytes to a mailslot that has been created on a Windows 95 workstation, it fails.

- **Q131493 CreateFile and Mailslots**

The documentation for the `CreateFile` API function incorrectly describes the possible values that `CreateFile` returns when opening a client end of a mailslot.

## 19.5 Conclusion

This chapter introduced the mailslot networking technology, which provides an application with simple one-way interprocess data communication using the Windows redirector. One of the most useful features of mailslots is that they allow you to broadcast a message to one or more computers over a network. However, because of the broadcast capability, mailslots do not provide reliable data transmission. If you want reliable data communication using the Windows redirector, consider using named pipes—the focus of our next chapter.