

STARTECH COMPUTER SERVICES

INTERNET COMPONENTS

FOR BORLAND DELPHI

SENDMAIL PACKAGE
GETMAIL PACKAGE
FTP PACKAGE
HTTP PACKAGE
IRC PACKAGE

**User's Guide
Version 1.4**

TABLE OF CONTENTS**INTRODUCTION** 7**THE SENDMAIL PACKAGE** 8

THE SENDMAIL COMPONENT	8
STEP 1: SETTING INITIAL PROPERTIES	8
The SMTP_Server property	9
The SMTP_Port property	9
The From_Name property	9
The From_Address property	9
The Subject property	9
Specifying TimeOut Values	10
Addressing Mail	10
Specifying the Message Text	10
Message Attachments	10
Adding Attachments to a message	11
Adding Custom Headers	14
STEP 2: SENDING THE MESSAGE	15
STEP 3: MONITORING THE PROGRESS OF THE TRANSACTION	15
Resolving the server name	15
Establishing the Connection	15
Processing recipients	15
Sending the message	16
Processing attachments	16
Transaction completion	16
Summary of OnMailInfo eventinfo values	17
Other OnMailInfo eventinfo values	17
Handling Errors	17
Checking for Transaction Success	19
Canceling the sending of mail	20
Advanced Topics	20
Sending several messages	20
THE SENDMAILDIALOG COMPONENT	21
THE TADDRESSBOOK COMPONENT	22
THE TADDRESSBOOKVIEWER COMPONENT	23

THE GETMAIL PACKAGE 25

THE GETMAIL COMPONENT	25
STEP 1: SET INITIAL PROPERTIES	25
The Mail_Server property	25

The Mail_Port property	26
The User_ID Property	26
The User_Password Property	26
Other Properties	26
STEP 2: INITIATING THE TRANSFER OF MAIL	26
STEP 3: HANDLING EVENTS	27
The OnMailInfo Event	27
Resolving the server name	27
Establishing the Connection	27
The Authorization phase	28
The Listing phase	28
The Retrieval phase	28
The OnMessage Loaded Event	28
The Msg_From property	28
The Msg_To property	29
The Msg_Size property	29
The Msg_Date property	29
The Msg_Subject property	29
The Msg_UIDL property	29
The Msg_ID property	30
The Mail_Text property	30
The Disconnection phase	32
Other OnMailInfo event <i>info</i> values	32
Handling Errors	32
Checking for Transaction Success	34
Handling OnMailInfo events	35
MESSAGE ATTACHMENTS	36
How the GetMail Component Handles Attachments	36
The TMIMEAttachment record type	36
How Attachments fit in the flow of events	38
The OnAttachmentGetLocation event	38
The OnAttachmentStored event	39
MESSAGE MANAGEMENT	39
Deleting Messages from the Server	39
Leaving Messages on the Server	40
The UIDLList property	41
Message Preview	42
Keeping track of already retrieved messages without UIDL support	43
The TUIDLManagerObject	43
Creating the TUIDLManager object	43
Querying the TUIDLManager object	44
Destroying the TUIDLManager object	44
Managing Retrieved Mail	45
THE TNMAILBOX COMPONENT	46
INITIALIZING THE TNMAILBOX COMPONENT	46
The Mailboxes property	46
The MailboxDirectory property	46
ADDING AND RETRIEVING MESSAGES	47
Selecting a mailbox	48
Selecting a Message	48

Adding a Message	49
Retrieving a Message	49
DELETING A MESSAGE	49
TRANSFERRING A MESSAGE TO ANOTHER MAILBOX	50
CREATING A NEW MAILBOX	50
COMPRESSING A MAILBOX	50
EMPTYING A MAILBOX	51
UPDATING A MESSAGE'S FLAG	51
STORING ATTACHMENTS	51
THE TATTACHMENTMANAGER COMPONENT	53
INITIALIZING THE COMPONENT	53
ADDING AN ATTACHMENT	53
Creating a New Entry	54
Updating an Attachment Entry	54
KEEPING TRACK OF ATTACHMENTS	55
RETRIEVING INFORMATION ABOUT AN ATTACHMENT	56
COPYING AN ATTACHMENT	56
DELETING AN ATTACHMENT	56
VIEWING OR EXECUTING ATTACHMENTS	56
THE NMAILBOXVIEWER COMPONENT	57
INITIALIZING THE NMAILBOXVIEWER COMPONENT	57
Selecting a Mailbox	58
CHANGING THE APPEARANCE OF THE CONTROL	58
The SectionNames Property	58
The SetHeaderSize Method	58
Setting Status Bitmaps	59
The Font property	60
The Align Property	60
WORKING WITH THE NMAILBOXVIEWER COMPONENT	60
AUTOMATIC OPERATIONS OF THE NMAILBOXVIEWER COMPONENT	61
Moving Messages	61
Message Count	61
Sorting messages	61
Finding selected messages	61
METHODS AVAILABLE IN MESSAGE VIEWER MANAGER OR MANUAL MODE	62
MANUAL MODE	63
Double Clicking on a Message	63
MESSAGE VIEWER MANAGER MODE	64
Designing the message viewer form	65
Handling Windows Messages	66
The MBV_PARENT message	67
The MBV_SETMESSAGE message	67
The MBV_UPDATEINFO message	68
The MBV_CLOSE message	68
Adding Functionality to the Message Viewer Form	69
The NextMessage and PreviousMessage methods	70
The ViewSelectedMessages Method	70
THE TSTRINGSVIEWER COMPONENT	70
TSTRINGSVIEWER PROPERTIES	71
Modifying the Control's Appearance	71

Modifying the Control's Operation	71
Modifying the Control's Contents	71
TStringsViewer Events	72
Clipboard Operations	72

THE FTP PACKAGE **73**

THE TFTP COMPONENT **73**

THE TFTP COMPONENT IN INTERACTIVE MODE **73**

INITIALIZING THEFTP COMPONENT 74

The FTPServer property 74

The FTPPort property 74

Logging in: the UserName, UserPassword and UserAccount properties 74

LOGGING IN TO THEFTP SERVER 75

AFTER LOGGING IN: DOING SOMETHING USEFUL 75

Listing Files and Directories 76

Deleting a File 77

Creating a Directory 77

Deleting a Directory 77

Renaming a File or Directory 77

Uploading a File 77

Downloading a File 78

Changing Directory 78

Ending a Session 78

Handling Errors 78

PUTTING IT TOGETHER A SIMPLE FILE TRANSFER 81

MONITORING TRANSACTION PROGRESS 82

Adding Status Notifications to the Example 84

ADVANCED USAGE 85

Techniques for Transferring Multiple Files. 87

Techniques for Transferring Multiple Directories 88

THE TFTP COMPONENT IN URL MODE **88**

INITIALIZING THEFTP COMPONENT FOR URL MODE. 89

STARTING THEFILE TRANSFER 89

MONITORING TRANSACTION PROGRESS AND COMPLETION 89

HANDLING ERRORS 90

OTHER METHODS, PROPERTIES AND EVENTS OF THEFTP COMPONENT 90

The IssueCommand method 90

The StopTransfer method 90

The TransferMode property 91

The WinsockStarted property 91

The Connected property 91

The OnFTPNeedInfo event 92

THE FTPURLDIALOG COMPONENT **92**

INITIALIZING THEFTPURLDIALOG COMPONENT 92

The LocalFile and URL properties 92

Specifying the direction and type of the transfer 93

Specifying Login information 93

STARTING THEFILE TRANSFER 93

STARTECH INTERNET COMPONENTS	6
<hr/>	
TRANSACTION COMPLETION	94
A SAMPLE FTPURLDIALOG PROGRAM	94
CUSTOMIZING THE APPEARANCE OF THE FTPURLDIALOG COMPONENT	94
Using the component editor to customize the FTPURLDialog component	95
Modifying the FTPURLDialog Programmatically	96
ADDITIONAL PROPERTIES OF THE FTPURLDIALOG COMPONENT	97
 THE HTTP PACKAGE	 98
<hr/>	
THE THttp COMPONENT	98
THE TStarImage COMPONENT	98
 THE IRC PACKAGE	 100
<hr/>	
 TTCPCClient AND TTCPServer	 102
<hr/>	
 APPENDIX A - INTERNET RFC	 104
<hr/>	
THE SMTP PROTOCOL	104
THE POP3 PROTOCOL	104
THE MIME PROTOCOL	104
THE FTP PROTOCOL	104
MUST READ IF YOU ARE DESIGNING A MAIL CLIENT	105
 APPENDIX B - ADDITIONAL MATERIAL	 106
<hr/>	
REGISTERED MIME TYPES	106

INTRODUCTION

This chapter discusses what the Internet is and how it works, and introduces the StarTech line of Internet components for Borland Delphi.

THE SENDMAIL PACKAGE

The SendMail package contains various components that greatly facilitate the sending of Internet mail using the Simple Mail Transfer Protocol (SMTP), as well as components to manage MIME attachments.

Component	Description
SendMail	SMTP client component. sends Internet mail
SendMailDialog	Facilitates status display while sending mail
TMimeManager	Handles classification of attachments
TAddressBook	Facilitates handling of e-mail addresses
TAddressBookViewer	Viewer for a TAddressBook component

The SendMail Component



The SendMail component is the basic component used to send Internet mail using the Simple Mail Transfer Protocol (SMTP). The general operation of the component is as follows:

- Set properties prior to initiating transfer of mail, which give the component necessary information, such as who to send a message to and the text of the message.
- Issue a command to initiate the transfer of mail.
- Respond to events which indicate the progress and completion of the mail transfer.

Step 1: Setting Initial Properties

Before sending a message using the *SendMail* component, we must specify some information, such as the recipient of the message and the server to be used to transfer mail. The following properties let you specify this information:

The SMTP_Server property

The *SMTP_Server* property is used to specify which server the component will connect to transfer mail. To do this, you will assign a string containing either the server's name or the server's address to this property.

Example:

```
SendMail1.SMTP_Server:='mail.acme.com';    {uses server name}  
or  
SendMail1.SMTP_Server:='123.54.6.78';    {uses server address}
```

Should you use the server name or server address in this property? If you have physical control of the mail server (i.e. you will know if its address changes), using the server address in the *SMTP_Server* property will save you the overhead of name resolution every time you connect. Most of the time, however, you will want to use the server name, as there is no guarantee that the server's address will remain the same over time.

The SMTP_Port property

The *SMTP_Port* property is used to specify which port to use on the server specified in the *SMTP_Server* property. The standard port for SMTP servers is 25.

Example:

```
SendMail1.SMTP_Port:=25;
```

The From_Name property

The *From_Name* property is used to specify the name of the sender of the message. This property is mandatory in version 1.3 and optional in version 1.4 and above.

Example:

```
SendMail1.From_Name:='John Smith';
```

The From_Address property

The *From_Address* property is used to specify the sender's e-mail address. This will be used by the recipient of the message to reply to the message. This property is mandatory in version 1.3 and optional in version 1.4 and above.

Example:

```
SendMail1.From_Address:='jsmith@acme.com';
```

The Subject property

The *Subject* property is used to specify the subject of the message. This will help the recipient distinguish the message from other messages in a list of messages. This property is optional.

Example:

```
SendMail1.Subject:='Product Announcement';
```

Specifying TimeOut Values

The *TimeOutArp* and *TimeOutConnect* properties can be used to specify how long the component should wait for address resolution and connection before timing out. If the property is assigned the value 0, no timeouts occur. Any other value indicates the timeout in seconds. Note that on 16 bit implementations, it is possible that a timer cannot not be allocated, as timers are limited resources. In this case, timeout monitoring will silently fail and no timeout will occur.

Addressing Mail

When addressing mail, there are three class of recipients:

- 'To:' recipients are the primary recipients of the message.
- 'cc:' recipients also receive the message and are listed as carbon copy recipients in the message header.
- 'Bcc:' recipients receive the message, but are not listed as carbon copy recipients in the message headers. It can be said they receive the message anonymously. For example, you would specify Bcc: recipients when sending a message to a long mailing list, to avoid hundreds of lines of headers listing the recipients and to maintain the privacy of the mailing list members.

There are three properties that are used to address a message, *ListTo*, *Listcc* and *ListBcc*, which are of type TStrings. To add a recipient to one of these lists, you will add a string containing the recipients e-mail address and optionally a bar character followed by the recipients name:

Example:

```
ToList.Add( 'janedoe@acme.com' );  
or  
ToList.Add( 'janedoe@acme.com|Jane Doe' );
```

The *To_Name* and *To_Address* properties are included for compatibility with previous version and should not be used, as they may be removed in future versions.

Specifying the Message Text

To specify the message text, you will use the *MailText* property, which is of type TStrings. The size of the message is limited to about 16000 lines of any length (previous versions had a line length limit of 253 characters). The following shows how to transfer text from a *Tmemo* component to the *MailText* property of a *SendMail* component:

Example:

```
SendMail1.MailText.Assign(Memo1.Lines);
```

Message Attachments

Most messages will consist of plain text. However it is possible for one or several files to be attached to the message. The standard manner of sending binary files

with a message (these files are referred to as *attachments*) is by using MIME (Multimedia Internet Mail Extensions). MIME provides the following facilities:

- the ability to encode binary files in a 7-bit format suitable for passage through various gateways on the Internet which might only support 7-bit text.
- the ability to package several attachments into one contiguous message, and to demarcate the boundary between attachments.
- the ability to provide additional information about the attachment, such as the type of information it contains (known as the MIME type) or the description of the content, for example.

Adding Attachments to a message

To add attachments to a message, you will use the *AddAttachment* method, passing a record of type *TSendMailAttachment*, which provides information about the file to be attached to the message. The *AddAttachment* method should be called once for every attachment. The *AddAttachment* method is prototyped as follows:

```
procedure AddAttachment(att: TSendMailAttachment);
```

The *TSendMailAttachment* record is prototyped as follows:

```
type TSendMailAttachment=record
    Name: string[255];
    Encoding: TMailEncoding;
    MimeType: string[80];
    MimeDisposition: string[30];
    MimeDescription: string[255];
    Location: string[255];
end;
```

The *TSendMailAttachment* record contains the following fields:

- The *Name* field is used to hold the name of the attachment. Note that the name of an attachment is not necessarily the original filename of the attachment. Additionally, note that the name of an attachment does not have to conform to DOS 8.3 filename length convention. For example, 'May1995SalesChart.gif' could very well be the name of an attachment originating from a machine running an OS with less stringent naming restrictions, such as UNIX, Windows 95 or Windows NT.
- The *Encoding* field is used to specify which type of encoding will be used on the attachment. Possible values of this field are:
 - *meDefault*: the attachment will be scanned and the best encoding method will be chosen.
 - *meMimeBase64*: the attachment is encoded using Base 64 encoding, which is most suitable for binary files.
 - *meMimeQuoted*: the attachment is encoded using Quoted Printable encoding, which is suitable for mostly text content with a few characters above ASCII 127 (such as ©, È, Ü, etc....).
 - *meMimePlain*: the attachment is not encoded, but sent as plain text. This type of encoding is suitable only on text files not containing any characters above ASCII 127.

- The *MimeType* field is used to hold the type of the attachment. A MIME type consists of two parts, the major type and the subtype, separated by a slash. For example, 'image/gif' has a major type of 'image' and a subtype of 'gif' and describe files containing images in CompuServe's Graphic Interchange Format. There are six major types which are defined in the MIME standard:
 - audio: files containing digitized audio.
 - image: files containing images
 - message: files containing a message or a collection of messages
 - text: files containing text
 - video: files containing digital video
 - application: used for applications, also a catch all type for any type that does not fit in other types.

Appendix C lists all registered MIME types. Additionally custom subtypes can be created by prepending an 'X' followed by a dash to the subtype. For example, there is no standard MIME type for Microsoft bitmaps. However, it is common to see them described with a MIME type of 'image/X-bmp' or 'image/X-bitmap'.

Common Mime Types

text/plain	plain text
image/gif	CompuServe GIF image
image/jpeg	JPEG image
audio/basic	AU audio file
video/mpeg	MPEG video
video/quicktime	Apple QuickTime video
message/rfc822	Standard Internet message
application/msword	Microsoft Word document
application/rtf	Rich Text Format document

Note: application/octet-stream is used for any attachment of any undetermined type. This is not recommended as no default viewers can be configured for this catch-all type. A complete list of registered MIME types can be found in appendix C.

- The *MimeDisposition* field is an experimental field which is sometimes included in MIME attachments. The two possible values of this field are:
 - 'attachment': the attachment should be displayed as an attachment to the message.
 - 'inline': the attachment should be displayed at the same time as the message. Since so few mail clients have the ability to display attachments concurrently with the message text, this value is seldom used.
- The *MimeDescription* field contains the plain text description of the attachment. For example, the *MimeDescription* field could contain 'Sales chart by region for May 1995'.
- The *Location* field specifies where the attachment file is stored.

Before adding attachments to a message, you should call the *ClearAttachments* method. Attachments are not cleared between messages and it is necessary to call the *ClearAttachments* method prior to calling the *AddAttachment* method.

The following example shows the sending of the 'config.sys' and 'autoexec.bat' file along with the message text.

```
procedure Form1.newMessage;
var
    att: TSendMailAttachment;
begin
    with SendMail 1 do
    begin
        SMTP_Server:='mail.acme.com';
        SMTP_Port:=25;
        From_Name:='John Smith';
        From_Address:='jsmith@acme.com';
        ToList.Add('support@microsoft.com|Product Support');
        MailText.Add('Here are the files you requested.');
```

{Add attachments}

ClearAttachments;

att.Name:='autoexec.bat';

att.Encoding:=meDefault;

att.MimeType:='text/plain';

att.MimeDescription:='my autoexec.bat file';

att.MimeDisposition:='attachment';

att.Location:='c:\autoexec.bat';

AddAttachment(att);

att.Name:='config.sys';

att.MimeDescription:='my config.sys file';

att.Location:='c:\config.sys';

{code to send message}

...

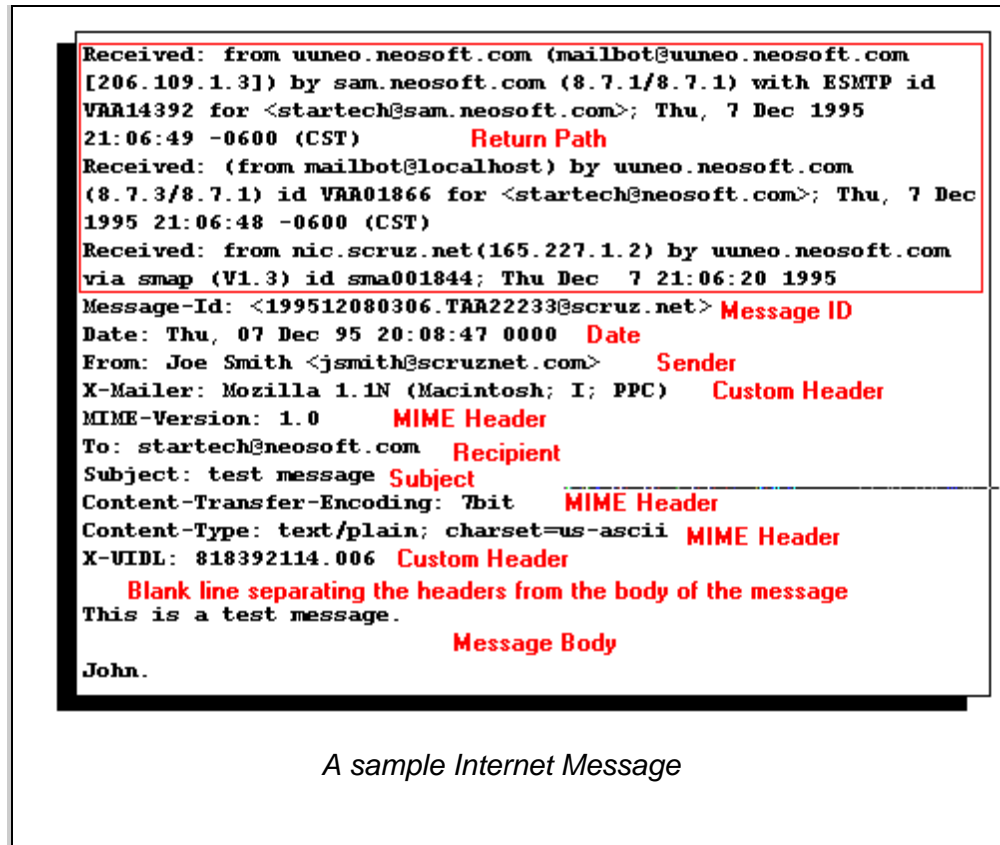
end;

end;

The *Attachments* property is included for compatibility with previous versions and should not be used as it may be removed in future versions.

Adding Custom Headers

Custom headers can be added to the message. Headers precede the message body and provide such information as who sent the message or the message of the subject.



The format of a custom header is as follows:

X-HeaderName: Custom Header goes here

For example, a common header is a header which indicates the priority of the message. It consists of the caption 'X-priority:' followed by a number between 1 and 5, with 1 having the highest priority and 5 having the lowest priority. (Note that the X-priority header has no effect on the way the message is sent, it is only for the benefit of the receiving program, which might chose to do something with the information). Another common custom header is the 'X-Mailer' header, which is used to indicate which mailing program was used to send the message.

To add custom headers to a message, you will use the *Headers* property, which is of type TStrings. The following example shows how to add the headers discussed above to a message.

Example:

```

SendMail1.Headers.Add('X-priority: 3');
SendMail1.Headers.Add('X-Mailer: My Program Version 1.0');

```

Step 2: Sending the Message

We have now set all the necessary properties and are ready to send the message. To send a message, you will set the *Action* property to the value *Send_Mail*.

Example:

```
SendMail1.Action:=Send_Mail;
```

Once you set the *Action* property, the component will connect to the server and will send the message. You will need to handle events if you want to receive notification of the progress of the transaction.

Step 3: Monitoring the progress of the transaction

To monitor the progress of the transaction after setting the *Action* property, you will need to handle the *OnMailInfo* event. The *OnMailInfo* event is prototyped as follows:

```
OnMailInfo=procedure(Sender: TObject; info: SendMailInfo; addinfo: string);
```

where *Sender* is the *SendMail* component which generated the notification, *info* is the type of notification being sent and *addinfo* is a string which may contain additional information, depending on the type of notification sent.

We will look at all the notifications sent, in the order in which they are likely to appear.

Resolving the server name

If you specified a host name (rather than an address) in the *SMTP_Server* property, you will receive an *OnMailInfo* event with an *info* value of *smResolvingAddress*. Once the server name is resolved into an address, you will receive an *OnMailInfo* event with an *info* value of *smAddressResolved*. The *addinfo* parameter will contain the address of the mail server.

Establishing the Connection

When the connection is established, you will receive an *OnMailInfo* event with an *info* value of *smServerConnected*. The *addinfo* parameter will contain the address of the server (for example, 198.64.6.34).

Processing recipients

When the *SendMail* component sends the recipients addresses to the server, you will receive an *OnMailInfo* event with an *info* value of *smRecipient*. The *addinfo* parameter will contain the e-mail address currently being processed.

If the server rejects an address, you will get an *OnMailInfo* event with an *info* value of *smBadAddress*. The *addinfo* parameter will contain the offending address. This will only happen in the following instances:

If user *jsmith* on server *acct.acme.com* wishes to address a message to user *janedoe* on the same server, he could address the mail as

'janedoe@acct.acme.com' or simply as 'janedoe'. As long as user *janedoe* has an account on server *acct.acme.com*, there will not be a problem with the second form of the address (without the @server). If user *janedoe* does not have an account on server *acct.acme.com*, you will get a *BadAddress* error.

To avoid this error, you should always specify the address in the format *user@server* and avoid using the shorter form consisting only of *user*.

Sending the message

The first notification you will receive when a message is sent is an *OnMailInfo* event with an *info* value of *smMessageSize*. The *addinfo* value contains the size of the message in bytes. The *smMessageSize* notification is new in version 1.4 and later.

Once the component starts sending the message, you will receive an *OnMailInfo* event with an *info* value of *smSendingMessage*. The *addinfo* value contains the size of the message sent so far in bytes. This will repeat until the entire message text has been sent. Note that in versions prior to 1.4, the *addinfo* parameter for this notification contained the number of lines sent.

Processing attachments

If the message contains attachments, you will receive various notifications as the attachments are being sent. For each attachment, you will first receive an *OnMailInfo* event with an *info* value of *smAttachmentName*. The *addinfo* parameter will contain the filename of the attachment.

You will next get an *OnMailInfo* event with an *info* value of *smAttachmentSize*. The *addinfo* parameter will contain the size of the attachment in bytes. Then, while the attachment is being sent, you will receive a series of *OnMailInfo* events with an *info* value of *smAttachmentBytes*. The *addinfo* parameter will contain the number of bytes sent so far.

If the attachment file could not be opened, you will get an *OnMailInfo* event with an *info* value of *smBadAttachment*. The *addinfo* parameter will contain the name of the offending attachment file. Note that unlike other errors, this error is not fatal. The message will continue to be sent, except for the bad attachment.

The steps outlined above will be repeated for each of the message's attachment.

Transaction completion

Once a message and any attachments included with the message have been sent, you will get an *OnMailInfo* event with an *info* value of *smMessageAccepted*, which means that the message has been accepted by the SMTP server and will be sent to the recipient(s). This will be followed by an *OnMailInfo* event with an *info* value of *smServerDisconnected*, which indicates that the component has disconnected from the SMTP server.

Finally, you will receive an *OnDone* event, which indicates that the component is ready for another transaction.

Summary of OnMailInfo event *info* values

<i>Connection Phase</i>	smResolvingAddress
	smAddressResolved
	smServerConnected
<i>Recipients Phase</i>	smRecipient (repeat for each recipient) (smBadAddress)
<i>Message Phase</i>	smMessageSize
	smSendingMessage (repeated)
<i>Attachment Phase</i>	smAttachmentName
	smAttachmentSize
	smAttachmentBytes
	(smBadAttachment)
<i>Completion Phase</i>	smMessageAccepted
	smServerDisconnected

Other OnMailInfo event *info* values

Every time the component receives a line from the server or sends a line to the server, you will receive an *OnMailInfo* event with an *info* value of *smTraceIn* or *smTraceOut*. When one of these notifications is received, the *addinfo* parameter will contain the line received or sent. These notifications can be used for debugging purposes, but should not be used normally as they will slow the sending of mail.

If a transaction is already in progress when you set the *Action* property to *Send_Mail*, you will receive an *OnMailInfo* event with an *info* value of *smAlreadyBusy*.

Handling Errors

The *OnMailError* event is used to monitor various *fatal* errors that might occur during a transaction. Note that non fatal errors that might occur during a transaction are reported by the *OnMailInfo* event. (such as *smAlreadyBusy*, *smBadAddress* or *smBadAttachment*).

After you receive an *OnMailError* event, the connection will be closed and you will receive an *OnDone* event, indicating that the component is ready for another transaction. The *Error* property will indicate what type of error occurred.

The *OnMailError* event is prototyped as follows:

```
TSendMailErrorEvent=procedure(Sender: TObject; error:
SendMailError; addinfo: string) of object;
```

where *error* is the error that was triggered. The following values of the *error* parameter are possible:

- smWinsockNotInitialized: The winsock interface could not be loaded. This can be caused by the lack of a winsock.dll (wssock32.dll on a 32 bit system) file, insufficient memory to load the winsock DLL, or lack of winsock resources (all available sockets are being used by other programs).

- **smNetworkDown:** The winsock interface has detected that the network is down,
- **smInvalidAddress:** The server name specified in *SMTP_Server* could not be resolved into an address.
- **smInternalError:** You should never get this error, which means that invalid parameters were passed to the Winsock interface.
- **smGeneralWinsockError:** This error indicates a general failure of the Winsock interface.
- **smConnAborted:** The connection was aborted due to a timeout or other failure.
- **smConnReset:** The remote server close the connection.
- **smConnectTimeOut:** The timeout value specified in the *TimeOutArp* or *TimeOutConnect* property has expired. (These properties are discussed in the *Advanced Topics* section).
- **smOutOfSockets:** A socket could not be created, as the winsock implementation has the maximum number of sockets open.
- **smNetworkUnreachable:** The winsock interface has detected that the network can't be reached from this host at this time.
- **smAddressNotAvailable:** The winsock interface has detected that the address specified by the *SMTP_Server* property is not available from this computer.
- **smConnectionRefused:** The server specified by the *SMTP_Server* property forcefully rejected the connection.
- **smProtocolError:** An unexpected response was received from the server.
- **smCanceled:** The user canceled the transaction.
- **smUnknown:** An unknown error occurred. You should never receive this error.
- **smAddressResolutionError:** An error occurred while trying to resolve the address specified in the *SMTP_Server* property.
- **smPrematureDisconnect:** The component was disconnected from the server before the transaction had completed.
- **smMailDestinationError:** A destination was not specified for the message.
- **smHostUnreachable:** The winsock interface has detected that the host is not reachable at this time.

Checking for Transaction Success

Once you receive an *OnDone* event, you can check for the success of the transaction using the *Success* property. If the value of the *Success* property is *True*, the message was successfully sent. If the value of the *Success* property is *False*, the message was not sent successfully and you can check the value of

the *Error* property to determine why the transaction failed. The *Error* property can take on one of the following values:

- *smNone*: no error occurred.
- *smWinsockNotInitialized*: The winsock interface could not be loaded. This can be caused by the lack of a winsock.dll (wsock32.dll on a 32 bit system) file, insufficient memory to load the winsock DLL, or lack of winsock resources (all available sockets are being used by other programs).
- *smNetworkDown*: The winsock interface has detected that the network is down,
- *smInvalidAddress*: The server name specified in *SMTP_Server* could not be resolved into an address.
- *smInternalError*: You should never get this error, which means that invalid parameters were passed to the Winsock interface.
- *smGeneralWinsockError*: This error indicates a general failure of the Winsock interface.
- *smConnAborted*: The connection was aborted due to a timeout or other failure.
- *smConnReset*: The remote server close the connection.
- *smConnectTimeout*: The timeout value specified in the *TimeOutArp* or *TimeOutConnect* property has expired. (These properties are discussed in the *Advanced Topics* section).
- *smOutOfSockets*: A socket could not be created, as the winsock implementation has the maximum number of sockets open.
- *smNetworkUnreachable*: The winsock interface has detected that the network can't be reached from this host at this time.
- *smAddressNotAvailable*: The winsock interface has detected that the address specified by the *SMTP_Server* property is not available from this computer.
- *smConnectionRefused*: The server specified by the *SMTP_Server* property forcefully rejected the connection.
- *smProtocolError*: An unexpected response was received from the server.
- *smCanceled*: The user canceled the transaction.
- *smUnknown*: An unknown error occurred. You should never receive this error.
- *smAddressResolutionError*: An error occurred while trying to resolve the address specified in the *SMTP_Server* property.
- *smPrematureDisconnect*: The component was disconnected from the server before the transaction had completed.
- *smMailDestinationError*: A destination was not specified for the message.

- `smHostUnreachable`: The winsock interface has detected that the host is not reachable at this time.

Therefore, if you are not interested in the progress of the transaction, you can simply wait for an *OnDone* event and then check the *Success* and *Error* property to determine if the message was sent successfully.

Canceling the sending of mail

To cancel the sending of message in progress, set the *Action* property to *Cancel_SendMail*. This will cause the connection with the server to be immediately closed. You will receive an *OnDone* event and the *Error* property will be set to the value *smCanceled*.

Advanced Topics

You now have all the information you need to make full use of the *SendMail* component. The following sections explore the following advanced topics:

- sending several messages in one transaction.
- sending data in a stream.
- checking for the availability of Winsock
- specifying connect timeouts

Sending several messages

When sending several messages at one time, you will want to keep the connection open until the last message is sent. This avoids the overhead of having to connect and disconnect with every message. The *KeepConnectionOpen* property, when set to *True*, keeps the component from disconnecting from the server when a message is sent successfully. The *KeepConnectionOpen* property should be set to *False* before the last message is sent.

The other issue to consider when sending several messages at one time is the timing involved in sending messages. The previous transaction must be completely finished before the next message is sent. To ensure this, we use the *OnDone* event to trigger the sending of messages. Each time the *OnDone* notification is received, you will prepare and send the next message. This process is best demonstrated with an example. The following example sends 10 messages to 10 different users (read in from a database from example). Note that if you wanted to send the same message to 10 users, it is much more efficient to send one message with the 10 recipients included in the *ToList*, *ccList* or *BccList* property. The technique described here is more useful for a form letter type of e-mail, with the message body being customized with the name of the recipient, or the product purchased, or some similar information.

The first step is to initialize some variables to keep track of how many messages to send and the current message number.

```
procedure TForm1.InitializeMail;  
begin
```

```

    MessagesToSend:=10;
    MessagesSent:=0;
    {add code to open database here}
    {setup SendMail component with server info}
    ...
end;

```

We will come back to this procedure later. We then need a procedure which prepares the next message to send.

```

Procedure TForm1.PrepareNextMessage;
begin
    Inc(MessagesSent);
    {check to see if we need to keep the connection open}
    if MessagesSent=MessagesToSend then
        SendMail1.KeepConnnectionOpen:=False
    else SendMail1.KeepConnnectionOpen:=True;
    {setup address of next recipient}
    {read next record}
    {set up body text, recipient, etc.. in SendMail component}
end;

```

We then need to handle the *OnDone* event, as discussed above and send the next message when this notification is received.

```

Procedure TForm1.SendMail1Done(Sender: TObject);
begin
    PrepareNextMessage;
    SendMail1.Action:=Send_Mail;
end;

```

Finally, we add the following line to the *InitializeMail* procedure discussed earlier:

```

procedure TForm1.InitializeMail;
begin
    MessagesToSend:=10;
    MessagesSent:=0;
    {add code to open database here}
    {setup SendMail component with server info}
    {Send first message}
    SendMail1Done(self); {<<<<<Line added<<<<<}
end;

```

The SendMailDialog component

<<<Note: this section to be completely rewritten>>>



The SendMailDialog component allows the user to send mail, without worrying about asynchronous event handling or status display.

Before sending mail, several properties must be set. First, the properties of the SMTP (Small Mail Transfer Protocol) server that will be used to send mail must be set, using the SMTP_Server and SMTP_Port properties.

Next, the recipient of the e-mail message should be specified using the TO_Name and TO_Address properties. You should also specify the name of the sender, using the FROM_Name property, as well as the reply address, using the FROM_Address property.

If you want your message to include a Subject: header, set the Subject property. Additional headers can be specified using the Headers Property. If you want copies of your message to be sent to more than one address, specify those addresses using the ccList property.

Enter your mail message using the MailText property or use the OnFeedData event if you wish to send a message with a size greater than 64K or lines longer than 253 characters. If you want to send any attachments with your message, using the Attachments property to specify the file names of these attachments. The Timeout property can be used to specify a timeout interval, after which the transaction will be canceled.

To start the process of sending mail, call the Execute function. A window will pop up showing the progress of the transaction. The function will return True if the transaction succeeded, or false if the transaction failed., in which case the Status property will contain the reason for the error.

The appearance of the dialog box can be edited at design time by double clicking the component, which will bring up a component editor or can be set using the Caption, Border, Color, Font, StatusBarColor, StatusBarBackground, StatusBarHeight and StatusBarWidth properties. The position of the window can be set using the Position, WindowTop and WindowLeft properties. You can select which objects appear in the window by setting the Options property. Finally, the text of the status messages can be set using the LanguageStrings property.

The TAddressBook component

This section has yet to be written, the following list of public methods, properties and events is included as minimal help:

```
type TAddressBookEntry=record
    LastName: string[40];
    FirstName: string[40];
    HomeMail: string[60];
    WorkMail: string[60];
    Nickname: string[20];
    Organization: string[60];
end;

type TAddressBookEntryPtr=^TAddressBookEntry;

type TAddressBook=class(TComponent)
public
    property Items[num: integer]: TAddressBookEntry read
GetEntry;
    procedure AddEntry(entry: TAddressBookEntry);
    procedure DeleteEntry(number: integer);
```

```

        procedure ModifyEntry(number: integer; a:
TAddressBookEntry);
        function FindEntry(firstname,lastname,address: string):
integer;
published
        property Directory: string read FDirectory write
SetDirectory;
        property NumEntries: integer read FNumEntries write
FDummyInt;
        property OnChange: TNotifyEvent read FOnChange write
FOnChange;
end;

end;

```

The TAddressBookViewer component



This section has yet to be written, the following list of public methods, properties and events is included as minimal help.

```

type TAddressBookView=(abvCompact,abvExpanded);
type TAddressBookViewerSort=(absLastName,absFirstName,absEMail);
type TAddressBookList=class(TListbox);

type TAddressBookViewer = class(TCustomControl)
public
        property ItemIndex: integer read GetIndex;
        procedure Update;
published
        property SelectedTab: integer read TabSelected write
TabChanged;
        property View: TAddressBookView read FView write SetView;
        property AddressBook: TAddressBook read GetAddressBook
write SetAddressBook;
        property SortOrder: TAddressBookViewerSort read FSortOrder
write SetSortOrder;
        property Font: TFont read FFont write SetFont;
        property Align;

```

THE GETMAIL PACKAGE

The GetMail package contains several components and objects which greatly facilitate the retrieval of mail from mail servers that use the Post Office Protocol Version 3, as well as components to store, manage and display messages and their attachments.

Component	Description
GetMail	POP3 client component. retrieves mail from server
TUIDL Manager	Assists in message retrieval for servers that do not support the UIDL command.
Nmailbox	Mailbox component, stores messages
TattachmentManager	Manages message attachments
TMailboxViewer	Multi column display of messages received.
TMIMEManager	Manages mime types and their viewer applications
GetMailDialog	Downloads and stores mail automatically

The GetMail Component



The *GetMail* component is the basic component used for retrieving mail.

The general operation of the component is as follows:

- Set properties prior to initiating transfer which give the component necessary information, such as the server to retrieve mail from.
- Issue a command to initiate the transfer of mail.
- Respond to events to assist the *GetMail* component when it needs additional information.

Step 1: Set Initial properties

Before initiating the transfer of mail, you must supply information to the GetMail component to direct it on which server to transfer mail from and which account to use. The following properties let you specify this information.

The Mail_Server property

The *Mail_Server* property is used to specify which server to connect to when mail retrieval begins. To do this, you will assign a string containing either the server's name or the server's address to this property.

Example:


```
GetMail1.Mail_Server:='mail.acme.com';    {uses server name}  
or  
GetMail1.Mail_Server:='123.54.6.78';    {uses server address}
```

Should you use the server name or server address in this property? If you have physical control of the mail server (i.e. you will know if its address changes), using the server address in the *Mail_Server* property will save you the overhead of name resolution every time you connect. Most of the time, however, you will want to use the server name, as there is no guarantee that the server's address will remain the same over time.

The Mail_Port property

The *Mail_Port* property is used to specify which port on the server to connect to. This will almost always be port 25 for POP3 servers, unless the server administrator has changed the default port assignment. The *Mail_Port* property is of type word.

Example:

```
GetMail1.Mail_Port:=25;
```

The User_ID Property

The *User_ID* property is used to specify which user to check mail for. To do this, assign a string containing the user ID to the *User_ID* property.

Example:

```
GetMail1.User_ID:='jsmith';
```

The User_Password Property

The *User_Password* property is used to specify the password for the user specified in the *User_ID* property. To do this, assign a string containing the password to the *User_Password* property.

Example:

```
GetMail1.User_Password:='mysecret';
```

Other Properties

The *UIDLList* and *Opt_Preview* properties are also set in step 1. These properties are described later in this chapter.

Step 2: Initiating the Transfer of Mail

Once the properties described in step 1 have been set, you are ready to initiate the transfer of mail. This is done by setting the *Mail_Action* property to *Get_Mail*. The *Mail_Action* property accepts other values which are described later in this chapter.

Example:

```
GetMail1.Mail_Action:=Get_Mail;
```

Once the *Action* property has been set, the *GetMail* component will attempt to connect to the server and will log in to the server. The component will then retrieve any messages waiting on the server, generating informational events and requesting any needed additional information. Finally, when all tasks have been completed, the component will disconnect from the server.

Step 3: Handling Events

As the component connects to the server, it generates several events. Events are used to accomplish the following tasks:

- provide information about the progress of the transaction.
- notification of any error condition
- request additional information from the user
- inform the user that requested information is available
- Notify the user that the transaction has completed and that the component is ready for another transaction.

The OnMailInfo Event

By far the most frequently generated event is the *OnMailInfo* event. This event is used to provide notification of the progress of the transaction. The prototype of the event is as follows:

```
procedure (Sender : TObject; info: GetMailInfo; addinfo: string);
```

The *Sender* parameter indicates which *GetMail* component triggered the event. The *info* parameter indicates what information is being conveyed and the *addinfo* parameter provides any additional information that may be needed. Let's look at the different values of *info* as they occur during the transaction.

Resolving the server name

If you specified a host name (rather than an address) in the *Mail_Server* property, you will receive an *OnMailInfo* event with an *info* value of *gmResolvingAddress*. Once the server name is resolved into an address, you will receive an *OnMailInfo* event with an *info* value of *gmAddressResolved*. The *addinfo* parameter will contain the address of the mail server.

Establishing the Connection

When the connection is established, you will receive an *OnMailInfo* event with an *info* value of *gmServerConnected*. The *addinfo* parameter will contain the address of the server (for example, 198.64.6.34).

The Authorization phase

Once connected to a server, the component enters the authorization phase. You will get an *OnMailInfo* event with an *info* value of *gmLogin*, which means the authorization phase is beginning. Once an *OnMailInfo* event with an *info* value of

gmLogin is received, you will receive another *OnMailInfo* event with an *info* value of *gmAccessGranted*, which means access was granted to the mail server.

The Listing phase

Once connected and authorized, the component will get a listing of the user's messages from the server. You will receive the *OnMailInfo* event with the following *info* values:

- *gmUIDLList*: the server is about to obtain a listing of messages on the server.
- *gmNumberMessage*: you will receive this message when the component determines the number of messages on the server. The *addinfo* parameter will contain the number of messages on the server waiting to be downloaded.

The Retrieval phase

The component is now ready to download any messages on the server. You will first receive some *OnMailInfo* events with the following *info* values:

- *gmMessageSize*: you will receive this event before every new message. The size of the message in bytes will be in the *addinfo* parameter.
- *gmGettingMessage*: as the message gets downloaded, you will receive this event. The number of bytes downloaded so far can be found in the *addinfo* parameter.

The OnMessage Loaded Event

Once the message has been completely downloaded, you will receive an *OnMessageLoaded* event. Once this message is received, you can use several properties to retrieve various headers and the text of the message. The prototype of the *OnMessageLoaded* event is as follows:

procedure (Sender: TObject);

where *Sender* indicates which GetMail component triggered the event.

The following properties can then be used to retrieve headers and the text of the message:

The Msg_From property

The *Msg_From* property will contain the sender of the message. There are three common formats for specifying E-Mail addresses, which are outlined in the *Format of Internet E-Mail addresses* inset.

The Msg_To property

The *Msg_To* property will contain the intended recipient of the message. There are three common formats for specifying E-Mail addresses, which are outlined in the *Format of Internet E-Mail addresses* inset.

Format of Internet E-Mail addresses

There are three widely used formats to express Internet E-Mail addresses:

- *E-Mail Address only*: only the recipient or sender's e-mail address is given.
For example: `jsmith@pobox.com`
- *E-Mail Address followed by name*: the recipient or sender's e-mail address is given, followed by their name in parentheses.
For example: `jsmith@pobox.com (John Smith)`
- *Name followed by e-mail address*: the recipient or sender's name is given, followed by the address, followed by the address in between angle brackets.
Sometimes the name is double quoted.
For example: `John Smith <jsmith@pobox.com>`
`"John Smith" <jsmith@pobox.com>`

The Msg_Size property

The *Msg_Size* property contains the size of the message in bytes.

The Msg_Date property

The *Msg_Date* property contains the date the message was sent, usually in Internet date format. The general format of an Internet date is as follows:

- 3 letter day of week abbreviation, followed by a comma and a space (Mon, Tue, Wed, Thu, Fri, Sat, Sun)
- day of the month followed by a space
- 3 letter month abbreviation , followed by a space (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)
- the year (four digits i.e. 1996 not 96) followed by a space
- the local time in hh:mm:ss format, followed by a space
- the difference from GMT time zone followed by a space. For example EST is -5, CST is -6.

For example, a message sent at 1:23 p.m. from the Pacific Standard Time zone on January 15th, 1996 would be formatted as follows:

`Mon, 15 Jan 1996 13:23: 10 (-0800)`

The Msg_Subject property

The *Msg_Subject* property contains the subject of the message.

The Msg_UIDL property

The *Msg_UIDL* property contains the content of the X-UIDL header field. A message UIDL's is a field that uniquely identifies a message on a POP3 servers. UIDL's are guaranteed to stay constant across sessions and to be unique for the lifetime of the user's account. For example, if you retrieve a message with a UIDL of '46828291.002', you will never receive a message with the same UIDL for that user's account. Additionally, if you leave the message on the server (explained later in the *Message Management* section), the message will have the same UIDL as it had the first time. A message's UIL is generated by the

POP3 server when it receives a message. Not all POP3 servers generate a message UIDL.

The **Msg_ID** property

The **Msg_ID** property contains the content of the message ID header field. The message ID is similar to the message UIDL, in respect to uniqueness, but it is generated by the mailing agent rather than by the receiving agent. All messages will have a message ID.

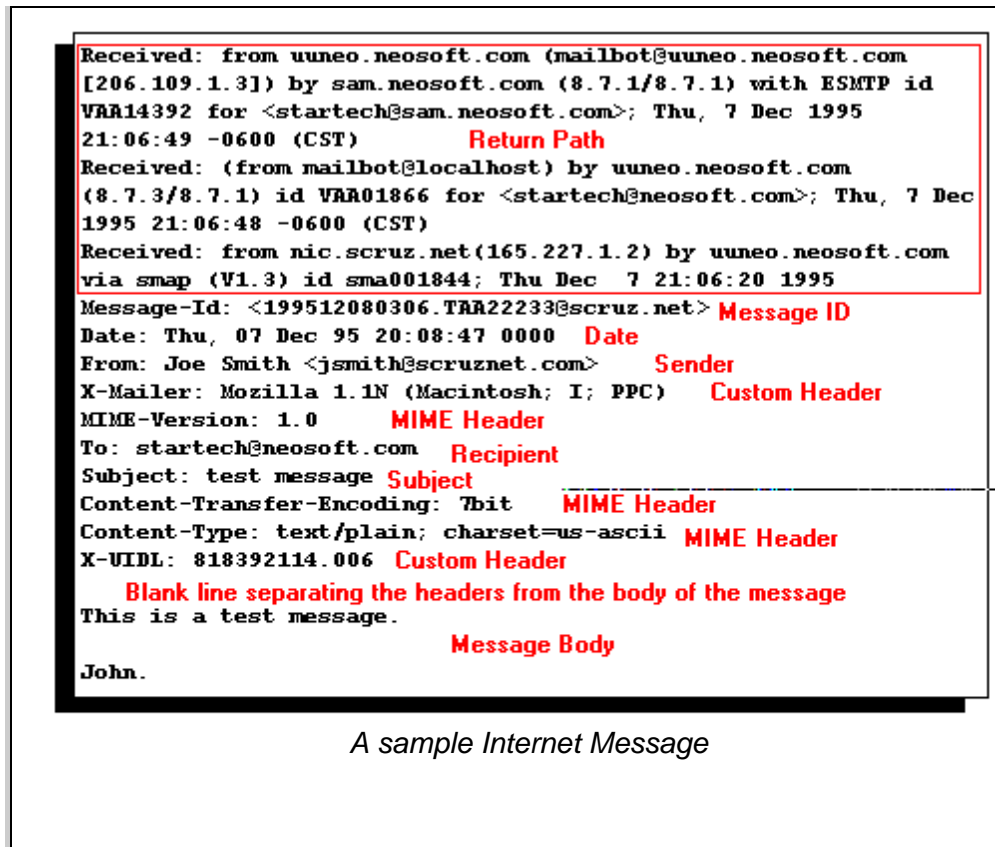
The **Mail_Text** property

The **Mail_Text** property (type **Tstrings**) contains the headers and the text of the message. The inset below shows a sample Internet mail message. Note that the headers and the body of the message are separated by a blank line. There is no limit on the size of the **Mail_Text** property. For this reason, caution should be used if displaying the message in a standard *Tmemo* or other standard Windows component limited to less than 32K of text. The *TStringsViewer* , discussed later in this chapter, can be used to display **Tstrings** of any size.

Q : How do I remove the headers from the message?

The following routine can be used to remove headers from the message and leave only the message text:

```
procedure RemoveHeaders(var Message: Tstrings);
begin
    while (Message.Count>1) and (Message[0]<>'') do
        Message.Delete(0);
    Message.Delete(0);
end;
```



Q : How can I easily extract other custom headers from the Mail_Text property?

The *GetMail* component provides a method called *ExtractHeader* to facilitate the extraction of a header field. The prototype of the *ExtractHeader* function is:

```
function GetMail.ExtractHeader(line: string; header: string; var stringvar: string):Boolean;
```

where *line* is a line from the header section of the message, *header* is the header to look for, including terminating colon and space, and *stringvar* returns the value of the header field. The function returns true if the header was found and false otherwise. The following routine can be used to search a message for a specific header. It takes the message and the header to be searched for and returns the header field if found or an empty string if not found.

```
function FindHeader(Message: Tstrings; header: string): string;
var
  i: integer;
  s: string;
begin
  i:=0;
  s:='';
  while (Message.Count>i) and (Message[i]<>'') and
    not ExtractHeader(Message[i],header,s) do Inc(i);
  Result:=s;
end;
```

For example calling `FindHeader(GetMail1.Mail_Text,'X-Mailer: ');`

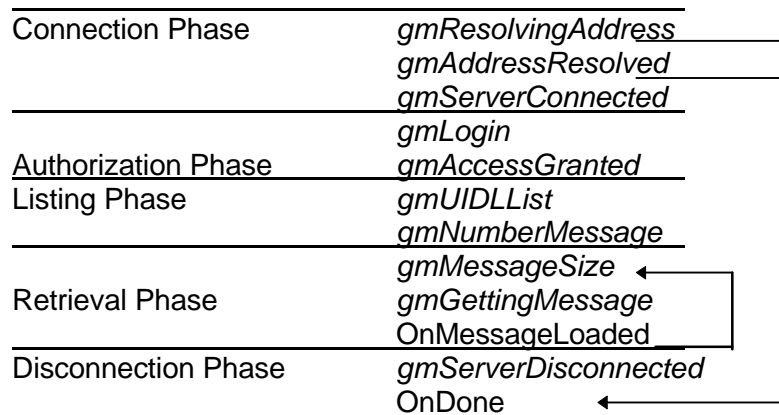
with the message shown in the inset in the preceding page would return the string 'Mozilla 1.1N (MacIntosh ; I; PPC) '.

The events described in the “*The Retrieval Phase*” section will be repeated until all messages have been downloaded.

The Disconnection phase

Once all messages have been downloaded, the component will disconnect from the server. You will get the *OnMailInfo* event an *info* parameter value of *gmServerDisconnected*, which means the component is about to disconnect from the server. Once the server has been disconnected from the server, you will receive an *OnDone* event, which means that the component has disconnected from the server and is now available for another transaction.

The following chart summarizes the events that one could receive during a normal transaction.



Other OnMailInfo event *info* values

Every time the component receives a line from the server or sends a line to the server, you will receive an *OnMailInfo* event with an *info* value of *gmTraceIn* or *gmTraceOut*. When one of these notifications is received, the *addinfo* parameter will contain the line received or sent. These notifications can be used for debugging purposes, but should not be used normally as they will slow the sending of mail.

If a transaction is already in progress when you set the *Mail_Action* property to *Get_Mail*, you will receive an *OnMailInfo* event with an *info* value of *gmAlreadyBusy*.

Handling Errors

The *OnMailError* event is used to monitor various *fatal* errors that might occur during a transaction. Note that non fatal errors that might occur during a transaction are reported by the *OnMailInfo* event. (such as *gmAlreadyBusy*)

After you receive an *OnMailError* event, the connection will be closed and you will receive an *OnDone* event, indicating that the component is ready for another transaction. The *Error* property will indicate what type of error occurred.

The *OnMailError* event is prototyped as follows:

```
TSendMailErrorEvent=procedure(Sender: TObject; error:
GetMailError; addinfo: string) of object;
```

where *error* is the error that was triggered. The following values of the *error* parameter are possible:

- *gmWinsockNotInitialized*: The winsock interface could not be loaded. This can be caused by the lack of a winsock.dll (wsock32.dll on a 32 bit system) file, insufficient memory to load the winsock DLL, or lack of winsock resources (all available sockets are being used by other programs).
- *gmNetworkDown*: The winsock interface has detected that the network is down,
- *gmInvalidAddress*: The server name specified in *SMTP_Server* could not be resolved into an address.
- *gmInternalError*: You should never get this error, which means that invalid parameters were passed to the Winsock interface.
- *gmGeneralWinsockError*: This error indicates a general failure of the Winsock interface.
- *gmConnAborted*: The connection was aborted due to a timeout or other failure.
- *gmConnReset*: The remote server close the connection.
- *gmConnectTimeOut*: The timeout value specified in the *TimeOutArp* or *TimeOutConnect* property has expired. (These properties are discussed in the *Advanced Topics* section).
- *gmOutOfSockets*: A socket could not be created, as the winsock implementation has the maximum number of sockets open.
- *gmNetworkUnreachable*: The winsock interface has detected that the network can't be reached from this host at this time.
- *gmAddressNotAvailable*: The winsock interface has detected that the address specified by the *SMTP_Server* property is not available from this computer.
- *gmConnectionRefused*: The server specified by the *SMTP_Server* property forcefully rejected the connection.
- *gmProtocolError*: An unexpected response was received from the server.
- *gmCanceled*: The user canceled the transaction.
- *gmUnknown*: An unknown error occurred. You should never receive this error.
- *gmAddressResolutionError*: An error occurred while trying to resolve the address specified in the *SMTP_Server* property.

- `gmPrematureDisconnect`: The component was disconnected from the server before the transaction had completed.
- `gmHostUnreachable`: The winsock interface has detected that the host is not reachable at this time.
- `gmAccessDenied`: The password or user ID supplied in the `User_Password` or `User_ID` property are not valid, or the mailbox was locked (this happens when a process attempts to access an account while another process is already accessing an account).

Checking for Transaction Success

Once you receive an *OnDone* event, you can check for the success of the transaction using the *Success* property. If the value of the *Success* property is *True*, the message was successfully sent. If the value of the *Success* property is *False*, the message was not sent successfully and you can check the value of the *Error* property to determine why the transaction failed. The *Error* property can take on one of the following values:

- `gmNone`: no error occurred.
- `gmWinsockNotInitialized`: The winsock interface could not be loaded. This can be caused by the lack of a winsock.dll (wsock32.dll on a 32 bit system) file, insufficient memory to load the winsock DLL, or lack of winsock resources (all available sockets are being used by other programs).
- `gmNetworkDown`: The winsock interface has detected that the network is down,
- `gmInvalidAddress`: The server name specified in `SMTP_Server` could not be resolved into an address.
- `gmInternalError`: You should never get this error, which means that invalid parameters were passed to the Winsock interface.
- `gmGeneralWinsockError`: This error indicates a general failure of the Winsock interface.
- `gmConnAborted`: The connection was aborted due to a timeout or other failure.
- `gmConnReset`: The remote server close the connection.
- `gmConnectTimeOut`: The timeout value specified in the `TimeOutArp` or `TimeOutConnect` property has expired. (These properties are discussed in the *Advanced Topics* section).
- `gmOutOfSockets`: A socket could not be created, as the winsock implementation has the maximum number of sockets open.
- `gmNetworkUnreachable`: The winsock interface has detected that the network can't be reached from this host at this time.
- `gmAddressNotAvailable`: The winsock interface has detected that the address specified by the `SMTP_Server` property is not available from this computer.

- `gmConnectionRefused`: The server specified by the `SMTP_Server` property forcefully rejected the connection.
- `gmProtocolError`: An unexpected response was received from the server.
- `gmCanceled`: The user canceled the transaction.
- `gmUnknown`: An unknown error occurred. You should never receive this error.
- `gmAddressResolutionError`: An error occurred while trying to resolve the address specified in the `SMTP_Server` property.
- `gmPrematureDisconnect`: The component was disconnected from the server before the transaction had completed.
- `gmHostUnreachable`: The winsock interface has detected that the host is not reachable at this time.
- `gmAccessDenied`: The password or user ID supplied in the `User_Password` or `User_ID` property are not valid, or the mailbox was locked (this happens when a process attempts to access an account while another process is already accessing an account).

Therefore, if you are not interested in the progress of the transaction, you can simply wait for an `OnDone` event and then check the `Success` and `Error` property to determine if the message was sent successfully.

Handling OnMailInfo events

While it is not necessary to handle any of the `OnMailInfo` events, you will usually want to handle most or all of them to give the user notification of the progress of the transaction. A `case` statement is usually best suited to handle the `OnMailInfo`, as this partial example code from the *StarMail* sample program demonstrates.

```
procedure TGetMailForm.GetMail1MailInfo(Sender: TObject; info:
GetMailInfo; addinfo: String);
begin
    case info of
        gmResolvingAddress: Status.Caption:= 'Resolving '+addinfo;
        gmAddressResolved: Status.Caption:= 'Connecting to'+addinfo;
        gmServerConnected: Status.Caption:= 'Connected to '+addinfo;
        gmServerDisconnected: Status.Caption:= 'Disconnected';
        gmAccessGranted: Status.Caption:= 'Logged in to server.';
        gmUIDLList: Status.Caption:= 'Getting message list';
        gmNumberMessage: MessagesLeft.Caption:= 'Messages left: '+
addinfo;
        gmLogin: Status.Caption:= 'Logging in';
    {
        rest of code deleted
    }
    end;
end;
```

Message Attachments

Most messages will consist of plain text. However it is possible for one or several files to be attached to the message. The standard manner of sending binary files with a message (these files are referred to as *attachments*) is by using MIME (Multimedia Internet Mail Extensions). MIME provides the following facilities:

- the ability to encode binary files in a 7-bit format suitable for passage through various gateways on the Internet which might only support 7-bit text.
- the ability to package several attachments into one contiguous message, and to demarcate the boundary between attachments.
- the ability to provide additional information about the attachment, such as the type of information it contains (known as the MIME type) or the description of the content, for example.

How the GetMail Component Handles Attachments

The GetMail component greatly simplifies the handling of attachments. When a message attachment is encountered within a message, the GetMail components triggers an event to request the location to store the attachment. The GetMail component then converts the attachment from its MIME encoding back to its original format. Once the attachment has been completely decoded and stored, the GetMail component triggers an event to inform the user that the attachment has been stored.

The TMIMEAttachment record type

The *TMIMEAttachment* record type is used to transfer information about an attachment between the component and the user program. The *TMimeAttachment* record type is prototyped as follows:

```
type TMIMEAttachment=record
    Name: string[255];
    MimeType: string[80];
    Disposition: string[30];
    Description: string[255];
    Size: LongInt;
    ContentID: string[40];
    Location: string[255];
    Stored: Boolean;
end;
```

The *TMIMEAttachment* record contains the following fields:

- The *Name* field is used to hold the name of the attachment. Note that the name of an attachment is not necessarily the original filename of the attachment. Additionally, note that the name of an attachment does not have to conform to DOS 8.3 filename length convention. For example, 'May1995SalesChart.gif' could very well be the name of an attachment originating from a machine running an OS with less stringent naming restrictions.
- The *MimeType* field is used to hold the type of the attachment. A MIME type consists of two parts, the major type and the subtype, separated by a slash. For example, 'image/gif' has a major type of 'image' and a subtype of 'gif' and describe files containing images in CompuServe's Graphic Interchange Format. There are six major types which are defined in the MIME standard:

- audio: files containing digitized audio.
- image: files containing images
- message: files containing a message or a collection of messages
- text: files containing text
- video: files containing digital video
- application: used for applications, also a catch all type for any type that does not fit in other types.

The Inset box below list common MIME types. Additionally custom subtypes can be created by prepending an 'X' followed by a dash to the subtype. For example, there is no standard MIME type for Microsoft bitmaps. However, it is common to see them described with a MIME type of 'image/X-bmp' or 'image/X-bitmap'.

- The *Disposition* field is an experimental field which is sometimes included in MIME attachments. The two possible values of this field are:
 - 'attachment': the attachment should be displayed as an attachment to the message.
 - 'inline': the attachment should be displayed at the same time as the message. Since so few mail clients have the ability to display attachments concurrently with the message text, this value is seldom used.
- The *Description* field contains the plain text description of the attachment. For example, the *Description* field could contain 'Sales chart by region for May 1995'.
- The *Size* field is filled in by the GetMail component with the size of the message in bytes once the entire message has been downloaded
- The *ContentID* field contains a unique ID for the attachment, which is generated by the mailing agent. It is similar to a Message ID, except that it applies to an attachment.
- The *Location* field is used to specify the location where the attachment should be stored.
- The *Stored* field is used by the *TAttachmentManager* component and is of no interest as far as the *GetMail* component is concerned.

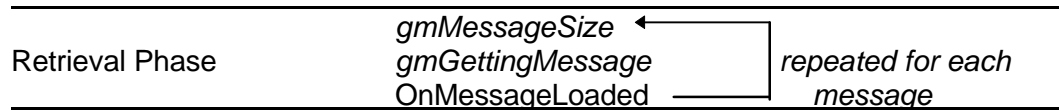
How Attachments fit in the flow of events

Attachments are processed during the Retrieval phase. Previously, we saw that the following notifications were received during the Retrieval phase:

Common Mime Types

text/plain	plain text
image/gif	CompuServe GIF image
image/jpeg	JPEG image
audio/basic	AU audio file
video/mpeg	MPEG video
video/quicktime	Apple QuickTime video
message/rfc822	Standard Internet message
application/msword	Microsoft Word document
application/rtf	Rich Text Format document

Note: application/octet-stream is used for any attachment of any undetermined type. This is not recommended as no default viewers can be configured for this catch-all type. A complete list of registered MIME types can be found in appendix C.



When attachments are part of a message, the following notifications will be received:



The `OnAttachmentGetLocation` event

The `OnAttachmentGetLocation` event is generated when the component has detected that an attachment follows. The prototype for this event is:

```
procedure(Sender: TObject; Attachment: TMIMEAttachmentPtr);
```

where *Sender* is the *GetMail* component which triggered the event and *Attachment* is a pointer to an aforementioned record of type *TMIMEAttachment*. When this event is received, the *Name*, *Description*, *MimeType*, *Disposition* and *ContentID* fields are filled in. When processing this event, the user is expected to fill in the *Location* field of the record with the desired location where the file should be stored.

The following example shows an event handler for the `OnAttachmentGetLocation` event where the user is prompted for a location to store the attachment:

```
procedure Form1.GetMailAttachmentGetLocation(Sender: TObject;
Attachment: TMIMEAttachment);
var
    d: TSaveDialog;
begin
    try
        d:=TSaveDialog.Create(self);
        d.Title:='Save attachment '+Attachment^.Name;
        if d.Execute then Attachment^.Location:=d.Filename
        else Attachment^.Location:='c:\junk';
    finally
        d.Free;
    end;
end;
```

The *TNMailbox* and *TAttachmentManager* components, described later in this chapter, provide a way to associate each attachment with a message and relieve the user from the chore of specifying a location for each attachment.

The OnAttachmentStored event

The *OnAttachmentStored* event occurs after an attachment has been completely decoded and stored. The prototype for the *OnAttachmentStored* event is:

```
procedure (Sender: TObject; Attachment: TMIMEAttachment);
```

where *Sender* is the *GetMail* component which triggered the event and *Attachment* is a pointer to an aforementioned record of type *TMIMEAttachment*. When this event is received, the attachment record will be filled in completely, including the size of the attachment. This event is of most use when use with the *TAttachmentManager* component, discussed later in this chapter.

Message Management

The mail retrieval procedure we have discussed so far do not mention what happens to the messages once they are retrieved. If they are not deleted from the server, the messages will remain on the server and will be downloaded again the next time mail is retrieved. Most of the time, we will want to remove a message from the server after it has been downloaded to the our computer. However, there are some cases when we will want to leave messages on the server until some later time. Consider the following example. You want to check your mail at work to see if any important messages have come in. However, you want to wait until you get home to download the messages to your machine, rather than downloading them to your office computer. There are several strategies that we will look at to handle all these possibilities.

Deleting Messages from the Server

The *Opt_Delete* property is used to specify whether a downloaded message should be kept on the server or deleted. You should set this property when you get an *OnMessageLoaded* event. Note that this property is set to *False* every time the *OnMessageLoaded* event is triggered. Therefore you must set this property to *True* inside the *OnMessageLoaded* event handler if you want a message to be deleted.

Example:

```
procedure Form1.GetMail1MessageLoaded(Sender: TObject);
begin
    {
    ...
    some code to store the Message
    ...
    }
    GetMail1.Opt_Delete:=True;    {delete messages from
server}
end;
```

Leaving Messages on the Server

The *GetMail* component supports leaving mail on the server. Additionally, the *GetMail* component can automatically skip messages that have already been downloaded, either by itself or with the help of the *TUIDLManager* component. It is easy to leave mail on the server. You simply set the *Opt_Delete* property to *False* inside the *OnMessageLoaded* event handler. The difficulty comes in determining which messages have been downloaded on the next transaction so that the same messages are not downloaded over and over. To further complicate matters, the UIDL command, which lists messages on the server and helps the component determine what new messages to download, is not supported on all POP3 servers.

The first step therefore, is to find out whether the UIDL command is supported on the target server. When the component determines whether the UIDL command is supported or not, it triggers an *OnMailInfo* event with an *info* value of *gmUIDLSupport*. This is done during the list phase, right after the *OnMailInfo* event with an *info* value of *gmUIDLList*.

<u>Authorization Phase</u>	
	<i>gmUIDLList</i>
Listing Phase	<i>gmUIDLSupport</i>
	<i>gmNumberMessage</i>
<u>Retrieval Phase</u>	

When you receive an *OnMailInfo* event with an *info* value of *gmUIDLSupport*, the *addinfo* will contain the string '1' or '0', with '1' meaning the UIDL command is supported and '0' meaning the command is not supported.

The UIDL command on a POP3 server retrieves a list of messages on the server, along with their UIDL. As mentioned previously, an UIDL is a string that uniquely identifies a message on a POP3 servers and that is guaranteed to stay constant across sessions and to be unique for the lifetime of the user's account. A UIDL listing might look like the following:

```
1 238765409.001
2 238765409.002
3 238765409.014
4 238765409.028
```

If the UIDL command is supported by the server that the *GetMail* is connecting to, the component will be able to use the previous listing to determine if there are any new messages to download and which messages to download. For example, given the previous listing above, if the component gets this listing the next time it gets mail:

```
1 238765409.001
2 238765409.002
3 238765409.014
4 238765409.028
5 240876537.003
```

the component will automatically skip messages 1 through 4 and only download message 5.

The UIDLList property

The *UIDLList* property is used to tell the component which messages have been previously downloaded. At the end of the transaction, the *UIDLList* property will contain an updated listing of messages on the server. The *UIDLList* property is of type *Tstrings*, so we can use the *LoadFromFile* and *SaveToFile* methods to load and save the UIDL listing between transactions.

The following example illustrates the use of the *UIDLList* property. Before initiating the checking of mail, we will load the *GetMail UIDLList* property from a file.

Example:

```
{determine application directory}
AppDirectory:=ExtractFilePath(Application.ExeName);
{load UIDL file into UIDL property}
try
    GetMail1.LoadFromFile(AppDirectory+'uidllist');
except
    ;
end;
GetMail1.Action:=Get_Mail;
```

When we get an *OnDone* event, we write the content of the *UIDLList* property back to the file we read it from.

Example:

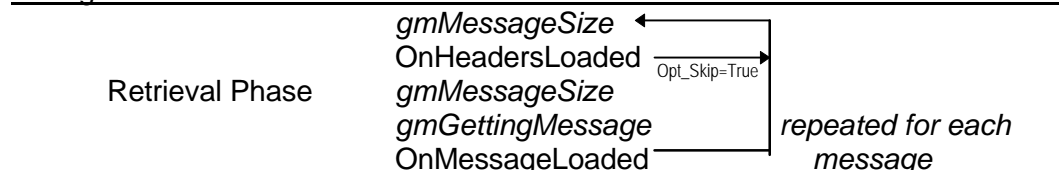
```
procedure Form1.GetMail1Done(Sender: TObject);
var
    AppDirectory: string;
begin
    case info of
        gmAvailable:
            begin
                {determine application directory}
                AppDirectory := ExtractFilePath(
Application.ExeName );
                {load UIDL file into UIDL property}
                try
                    GetMail1.SaveToFile( AppDirectory +
'uidllist' );
                except
                    ;
                end;
            end;
            {***** rest of code deleted ***** }
        end;
    end;
end;
```

This simple method of keeping track of already retrieved messages should work on most POP3 server. However, there are some older POP3 servers which do not support the UIDL command. An alternate, albeit slower and less efficient, method has been devised to handle servers which do not support the UIDL command. Before we examine this method, we must first learn about previewing messages on the server.

Message Preview

The *GetMail* component supports the ability to preview the headers of the messages, allowing the user to examine the headers and to decide whether to download or skip the message. The *Opt_Preview* and *Opt_Skip* properties and *OnHeadersLoaded* event are used for message preview. If the *Opt_Preview* property is set to *True* before starting the transfer of mail, the component will only download a message's header and trigger an *OnHeadersLoaded* event. The component will then examine the *Opt_Skip* property to determine what it should do with the message. If *Opt_Skip* is set to *True*, the component will not download the message and will move on the next message. If *Opt_Skip* is set to *False*, the component will download the message. Message preview alters the Retrieval Phase as follows:

Listing Phase



Disconnection Phase

The following code snippet shows how to download only messages containing the word 'order' in the subject:

```
procedure Form1.CheckMailBtnClick(Sender: Tobject);
begin
    {initialization code}
    ...
    {checking mail}
    GetMail1.Opt_Preview:=True;
    GetMail1.Action:=Get_Mail;
end;

procedure Form1.GetMail1HeadersLoaded(Sender: Tobject);
begin
    if Pos('order',Lowercase(GetMail1.Msg_Subject))>0 then
        GetMail1.Opt_Skip:=False
    else GetMail1.Opt_Skip:=True;
    {
        note: this could have been coded:
        GetMail1.Opt_Skip:=(Pos('order',Lowercase(GetMail1.Msg_Subje
ct))=0);
    }
end;
```

Keeping track of already retrieved messages without UIDL support

As mentioned previously, every Internet message has a Message-Id field which uniquely identifies the message. We can use the *Msg_ID* property in conjunction with the message preview feature to keep track of previously retrieved messages. The *TUIDLManager* object will also be used to keep track of previously retrieved messages and to help the *GetMail* component determine which messages should be downloaded and which messages should be skipped.

The TUIDLManagerObject

The TUIDLManager object keeps track of previously loaded messages. To use this object involves the following steps:

- Create the object.
- Query the object when the *OnHeadersLoaded* event is received.
- Destroy the object when the transaction is finished.

Creating the TUIDLManager object

The prototype for the TUIDLManager object's constructor is:

```
constructor Create(filename: string);
```

where *filename* is the name of a file used to store message IDs. The best time to create the *TUIDLManager* object is when an *OnMailInfo* event is received with an *info* value of *gmUIDLSupport* and an *addinfo* value of '0', which means the UIDL command is not supported. The following code snippet shows how one might create the *TUIDLManager* object:

```
type Form1=class(TForm)
...
private
    UIDLMgr: TUIDLManager;
    UIDLSupported: Boolean;
end;

implementation

procedure Form1.GetMail1MailInfo(Sender: TObject; info:
TGetMailInfo; addinfo: string);
var
    AppDir: string;
begin
    case info of
        gmUIDLSupport:
            begin
                UIDLSupported:=(addinfo='1');
                if not UIDLSupported then
                    begin
                        AppDir:=ExtractFilePath(Application.ExeName);

                        UIDLMgr:=TUIDLManager.Create(AppDir+'uidl1 ist');
                        GetMail1.Opt_Preview:=True;
                    end;
            end;
    ...
    end;
end;
```

Note that we also set the *Opt_Preview* property to *True* after we create the *TUIDLManager* object.

Querying the TUIDLManager object

When we receive the *OnHeadersLoaded* event, we query the *TUIDLManagerObject* using the *ProcessMessageID* method. The prototype for the *ProcessMessageID* method is:

```
function ProcessMessageID(id: string): Boolean;
```

where *id* is the message ID. This function returns *True* if the message with message ID "*id*" needs to be downloaded and *False* if the message has already been retrieved. The following code shows how to handle the *OnHeadersLoaded* event:

```
procedure Form1.GetMailHeadersLoaded(Sender: Tobject);
begin
    Opt_Skip:=not UIDLMgr.ProcessMessageID(GetMail1.Msg_ID);
end;
```

Destroying the TUIDLManager object

Once the transaction has completed, you must destroy the *TUIDLManager* object, so that the new list of message IDs can be saved back to disk. The following code snippet shows how this done:

```
procedure Form1.GetMailMailInfo(Sender: Tobject; info:
TGetMailInfo; addinfo: string);
begin
    case info of
        gmAvailable:
            begin
                if not UIDLSupported then UIDLMgr.Destroy;
                ...
            end;
        ...
    end;
end;
```

However, if we encounter an error before the transaction completes, we will usually not want to store a partial message list. The *Error* method of the *TUIDLManager* object is used to flag any errors. When this method is called, it signals the *TUIDLManager* object that an error has occurred so that when the object is destroyed, the old message list will not be overwritten. The following code snippet shows examples of when you would call the *Error* method:

```
procedure Form1.GetMailMailInfo(Sender: Tobject; info:
TGetMailInfo; addinfo: string);
begin
    case info of
        gmReadError:
            begin
                if not UIDLSupported then UIDLMgr.Error;
                ...
            end;
        gmWriteError:
            begin
                if not UIDLSupported then UIDLMgr.Error;
                ...
            end;
        gmPrematureDisconnect:
            begin
                if not UIDLSupported then UIDLMgr.Error;
                ...
            end;
    end;
end;
```

```
        end;  
        ...  
    end;  
end;
```

Managing Retrieved Mail

Now that we know how to retrieve mail from a POP3 server, let's look at how to store and display messages and their attachments. We can use the *TNMailbox* component to store messages and the *TAttachmentManager* to manage attachments. We will first look at the *TNMailBox* component.

The TNMailbox Component



The *TNMailbox* component is used to store messages as they are downloaded from the mail server. The user can configure several mailboxes to hold messages and can transfer messages between messages. Typically, a mail program will have three standard mailboxes:

- IN mailbox: used to receive incoming messages.
- OUT mailbox: used to store sent messages.
- TRASH mailbox: used to store deleted messages. Usually, all messages in this mailbox will be deleted when the program terminates.

Additionally, the user should be able to create mailboxes to sort incoming messages. For example, a user might create a mailbox named 'Letters from Judy' and another mailbox named 'Project X Files'.

Initializing the TNMailbox component

Before we can work with the *TNMailbox* component, we must initialize some properties to let the component know in which directory to create the mailbox files and which mailbox files to use.

The Mailboxes property

The *Mailboxes* property is used to specify the name of mailboxes and files. This property should only be set first, before setting the *MailboxDirectory* property. The *Mailboxes* property is of type *Tstrings* and should contain strings containing the filename for the mailbox (without an extension), followed by an equal sign character ('='), followed by the name of the mailbox.

The following shows what strings might be found in the *Mailboxes* property:

```
'in=Messages Received'  
'out=Messages Received'  
'trash=Trash Basket'  
'judy=Letters from Judy'  
'xfiles=Project X'
```

For each entry in the *Mailboxes* property, the component creates *.mbx and *.idx files in the mailbox directory, if the files do not exist. For example, the files *in.mbx* and *in.idx* will be created for the '*in=Messages Received*' entry.

The MailboxDirectory property

The *MailboxDirectory* property is used to specify the location of the mailbox files. For example, if you wanted to store the mailbox files in the 'mailbox' subdirectory of the application directory. The following code would accomplish this:

```
ApplicationDirectory:=ExtractFilePath(Application.ExeName);  
NMailbox1.MailboxDirectory:=ApplicationDirectory+'mailbox';
```

This property must be set after the *Mailboxes* property.

Adding and Retrieving Messages

In addition to saving the message text, the *TNMailbox* component stores certain key fields as the message, such as the sender, subject and date of the messages. The following properties are used to access the message and these key fields:

The Msg_Text property: The *Msg_Text* property is used to access the text of the message, including headers. It is of type *TStrings*.

The Msg_From property: The *Msg_From* property is used to access the sender of the message.

The Msg_To property: The *Msg_To* property is used to access the recipient of the message

The Msg_Date property: The *Msg_Date* property is used to access the date specifying when the message was sent.

The Msg_Subject property: The *Msg_Subject* property is used to access the subject of the message.

The Msg_Lines property: The *Msg_Lines* property is used to access the number of lines in the message text. This property is read only.

The Msg_Size property: The *Msg_Size* property is used to access the size of the message text in bytes. This property is read only.

The Msg_Flag property: The *Msg_Flag* property is used to access a flag for the message which indicates the status of the message. Although you can use any character to specify the status of the message, you should use the following values if you plan to use the *TMailboxViewer* component discussed later in this chapter:

‘U’: message is unread

‘V’: message was viewed

‘R’: a reply was made to this message.

The Msg_Part property: The *Msg_Part* property is used to specify the part of the message. Large messages can sometimes be split into smaller parts for transmission or storage.

The Msg_AttachStart property: The *Msg_AttachStart* property is used to specify the starting attachment number for the message. This property will be described fully when the *TAttachmentManager* property is described later in this chapter.

The Msg_AttachEnd property: The *Msg_AttachEnd* property is used to specify the ending attachment number for the message. This property will be described fully when the *TAttachmentManager* property is described later in this chapter.

Selecting a mailbox

Before we can add or retrieve a message to a mailbox, we must tell the component which mailbox to use and which message to retrieve. There are three properties which can be used to select a target mailbox and message.

The CurrentMailBoxNumber property: The CurrentMailboxNumber property is used to select a mailbox. This property should be set to a number between 1 and the number of mailboxes, as listed in the *Mailboxes* property. '1' refers to the first mailbox listed in the *Mailboxes* properties, '2' to the second message, and so on.

Example:

```
const IN_BOX=1;
...
CurrentMailboxNumber:=IN_BOX;
```

The CurrentMailboxName property: The CurrentMailboxName property is used to specify the base filename of a mailbox. It also provides an alternate means of selecting a mailbox, by specifying the base filename, rather than the mailbox number.

Example:

```
CurrentMailboxName:='inbox';
```

When the CurrentMailboxNumber or CurrentMailboxName property is set, the following properties are updated in addition to the CurrentMailboxNumber and CurrentMailboxName properties:

The CurrentMailBoxMessages property: The CurrentMailboxMessages property is used to indicate how many messages are stored in the selected mailbox. This property is read only.

The CurrentNMailboxSize property: The CurrentNMailboxSize property is used to specify the size, in bytes, of all the messages in the mailbox.

The CurrentNMailboxTrash property: The CurrentNMailboxTrash property is used to specify the amount of wasted space in bytes, contained in the mailbox. Wasted space increases as messages are deleted or transferred to another mailbox, and can be recouped by calling the *Compress* method, described late in this section.

Selecting a Message

The *Msg_Number* property is used to select a message. *Msg_Number* is a number between 1 and the number of messages in the mailbox, found in the *CurrentMailboxMessages* property. When a valid mailbox and message has been selected, all the *Msg_* properties are updated.

Adding a Message

There are three steps to adding a message to a mailbox:

- select a mailbox, using the *CurrentMailboxNumber* and *CurrentMailboxName* properties.
- set the *Msg_Text*, *Msg_From*, *Msg_To*, *Msg_Subject*, *Msg_Date*, *Msg_Flag*, *Msg_AttachStart* and *Msg_AttachEnd* properties.
- call the *AddMail* method.

The *AddMail* method appends the message to the selected mailbox. The *CurrentMailboxMessages* and *CurrentNMailboxSize* properties are also update.

The following example shows how one might store messages downloaded with the *GetMail* component (this example does not deal with the *Msg_AttachStart* and *Msg_AttachEnd* properties, which are discussed later in this chapter).

```
implementation
  const
    IN_BOX      =      1;

  ...
interface

procedure Form1.GetMailMessageLoaded(Sender: Tobject);
begin
  NMailbox1.CurrentMailboxNumber:=IN_BOX;
  NMailbox1.Msg_From:=GetMail1.Msg_From;
  NMailbox1.Msg_To:=GetMail1.Msg_To;
  NMailbox1.Msg_Date:=GetMail1.Msg_Date;
  NMailbox1.Msg_Subject:=GetMail1.Msg_Subject;
  NMailbox1.Flag:='U';
  NMailbox1.Msg_Text.Assign(GetMail1.Mail_Text);
  NMailbox1.AddMail;
end;
```

Retrieving a Message

To retrieve a message, set the *CurrentMailboxNumber* or *CurrentMailboxName* property with the target mailbox and the *Msg_Number* property with the target message. Once this is done, the *Msg_From*, *Msg_To*, *Msg_Date*, *Msg_Size*, *Msg_Subject*, *Msg_Lines*, *Msg_Flag*, *Msg_AttachStart* and *Msg_AttachEnd* properties will be updated. The *Msg_Text* property is not updated with the text of the message until the *LoadMail* method is called.

Example:

```
NMailbox1.CurrentMailboxNumber:=IN_BOX;
NMailbox1.Msg_Num:=3;
NMailbox1.LoadMail;
```

Deleting a Message

To delete a message, set the *CurrentMailboxNumber* or *CurrentMailboxName* property with the target mailbox and the *Msg_Number* property with the target message. To delete that message, use the *DeleteMail* method.

Example:

```
NMailbox1.CurrentMailboxNumber:=IN_BOX;
NMailbox1.Msg_Num:=3;
NMailbox1.DeleteMail;
```

Transferring a Message to Another Mailbox

To transfer a message, set the *CurrentMailboxNumber* or *CurrentMailboxName* property with the target mailbox and the *Msg_Number* property with the target message. To transfer that message, use the *TransferTo* method. The prototype for the *TransferTo* method is:

```
procedure TransferTo(name: string);
```


where *name* refers to the base filename for the mailbox. If you prefer to use a mailbox number, you can use the *MailboxFile* function. The prototype for the *MailboxFile* function is:

```
function MailBoxFile(i: integer): string;
```

which returns the base filename for mailbox *i*.

The following example shows how to transfer a message to a trash mailbox:

```
const
    IN_BOX      = 1;
    OUT_BOX     = 2;
    TRASHCAN    = 3;
...
NMailbox1.CurrentMailboxNumber:=IN_BOX;
NMailbox1.Msg_Num:=3;
NMailbox1.TransferTo(MailBoxFile(TRASHCAN));
```

Creating a New Mailbox

There are two steps involved in creating a new mailbox:

- Add an entry in the *Mailboxes* property for the mailbox to be created. For example:

```
NMailbox1.Mailboxes.Add('judy|'Letters from Judy');
```

- Call the *CreateMailbox* method to create the various mailbox files. The prototype for the *CreateMailbox* method is:

```
procedure CreateMailBox(name: string);
```

where *name* is the base filename of the mailbox.

Example:

```
NMailbox1.CreateMailbox('judy');
```

Compressing a Mailbox

When a message is deleted or transferred from a mailbox, wasted space develops in the mailbox file where the message was once stored. To compress a mailbox, set the *CurrentMailboxNumber* or *CurrentMailboxName* property with the target mailbox and call the *Compress* method. The following example demonstrates how to compress all mailboxes if the wasted space exceeds 10K.

```
with NMailbox1 do
begin
    for i:=1 to Mailboxes.Count do
    begin
        CurrentMailboxNumber:=i;
        if CurrentMailboxTrash>10240 then Compress;
    end;
end;
```

Emptying a Mailbox

To delete all messages from a mailbox, you will use the *EmptyMailbox* method.

The prototype for the *EmptyMailbox* method is:

```
procedure EmptyMailbox(num: integer);
```

where *num* is the mailbox number. The following example shows how to delete all messages from a trash mailbox when the program exits:

```
const
    TRASHCAN    = 3;
...
procedure Form1.Close(Sender: TObject);
```

```
begin
    NMailbox1.EmptyMailbox(TRASHCAN);
end;
```

Updating a Message's Flag

The *TNMailbox* component does not allow a message to be modified once it has been stored. However, it is sometimes to change a message's flag when its status changes, such as when it has been read. To change a message's flag, set the *CurrentMailboxNumber* or *CurrentMailboxName* property with the target mailbox and the *Msg_Number* property with the target message, then call the *UpdateFlag* method. The prototype for the *UpdateFlag* method is:

```
procedure UpdateFlag(c: char);
```

where *c* is the new message flag. The following example shows how to update a message flag:

```
NMailbox1.CurrentMailboxNumber:=IN_BOX;
NMailbox1.Msg_Number:=12;
NMailbox1.UpdateFlag('V'); {message has been read}
```

Storing Attachments

The *TNMailbox* component stores the text of Internet messages, but it does not store message Attachments. There are three ways of handling attachments:

- Prompt the user for a location when an attachment is processed. Sample code for this was given earlier in the description for the *OnAttachmentGetLocation* event of the *GetMail* component. The disadvantage of this method is that a user must be present when mail is retrieved, in case a message with an attachment is retrieved. This method is therefore inappropriate for automated mail retrieval.
- Automatically generate a filename from the *name* field of the MIME attachment. Unfortunately, an attachment name is optional in the MIME specifications. An attachment name will also probably not conform to the 8.3 DOS filename specifications. Additionally, all association between the message and the attachment is lost if either this method or the previous method is used.
- The *TAttachmentManager* component was designed to overcome the limitations of the first two methods. It allows attachments to remain associated with their original messages and relieves the user from having to specify a location for the attachment.

The TAttachmentManager component



The *TAttachmentManager* component works in the parallel with the *TNMailbox* component to facilitate the task of managing attachments.

Initializing the component

When the component is created, the component automatically detects the application directory and creates an 'attach' subdirectory to store attachments. For example, if the application directory is 'c:\mail', the component will store attachments on the 'c:\mail\attach' directory. To override this default behavior, set the *AttachmentDirectory* property with the name of the directory to use.

Adding an Attachment

There are two steps involved in bringing an attachment under the control of the *TAttachmentManager* component:

- Creating an entry for a new attachment and generating a filename for the attachment.
- Updating the entry for the attachment once it has been stored.

The record used for the entry is of type *TMIMEAttachment*, which was discussed earlier in the *Handling Attachments* section of the *GetMail* component. To recap, the *TMIMEAttachment* record type is prototyped as follows:

```
type TMIMEAttachment=record
    Name: string[255];
    MimeType: string[80];
    Disposition: string[30];
    Description: string[255];
    Size: LongInt;
    ContentID: string[40];
    Location: string[255];
    Stored: Boolean;
end;
```

The *TMIMEAttachment* record contains the following fields:

- The *Name* field is used to hold the name of the attachment.
- The *MimeType* field is used to hold the type of the attachment.
- The *Disposition* field specifies how the attachment should be displayed.
- The *Description* field contains the plain text description of the attachment.
- The *Size* field contains the size of the attachment.
- The *ContentID* field contains a unique ID for the attachment.
- The *Location* field contains the full path name where the attachment is stored.
- The *Stored* field specifies whether the file specified in *Location* is under the control of the *TAttachmentManager* component.

Creating a New Entry

To create a new entry and generate a unique filename to store an attachment, you will use the *NewAttachment* method. The *NewAttachment* method is prototyped as follows:

```
function NewAttachment(var AttachRec: TMIMEAttachment): LongInt;
```

where *AttachRec* is a record of type *TMIMEAttachment*. The function returns a number which can be used later to work with that attachment. Additionally, when this function is called, the *Location* field of the *AttRec* record will contain the location where the attachment should be stored.

The following example shows how the *OnAttachmentGetLocation* event of the *GetMail* component can be handled with a *TAttachmentManager* component.

```
var
    AttNumber: LongInt;

procedure Form1.GetMail1AttachmentGetLocation(Sender:TObject;
Attachment: TMIMEAttachment);
begin
    AttNumber:=AttachmentManager1.NewAttachment(Attachment);
end;
```

Updating an Attachment Entry

After the *GetMail* component has finished storing an attachment, it generates an *OnAttachmentStored* event. At this point, we should update the entry in the *TAttachmentManager* component to reflect the size of the attachment and the fact that the attachment is stored.

The *Update* method is used to change any information regarding an attachment. The prototype for the *Update* method is

```
procedure Update(Number: LongInt;AttachRec: TMIMEAttachment);
```

where *Number* is the attachment number which was returned by the *NewAttachment* method and *AttachRec* is a record of type *TMIMEAttachment*. The following example shows how to handle the *OnAttachmentStored* event of the *GetMail* component.

```
var
    AttNumber: LongInt;

procedure Form1.GetMail1AttachmentGetLocation(Sender:TObject;
Attachment: TMIMEAttachment);
begin
    AttNumber:=AttachmentManager1.NewAttachment(Attachment);
end;

procedure Form1.GetMail1AttachmentStored(Sender:TObject;
Attachment: TMIMEAttachment);
begin
    Attachment.Stored:=True;
    AttachmentManager1.Update(AttNumber,Attachment);
end;
```

Keeping Track of Attachments

Not only must we store of attachments, we must also associate attachments with a message. The *Mailbox* component stores two properties for each message, *Msg_AttachStart* and *Msg_AttachEnd*. The following code illustrates how to store this information in the mailbox.

```
var
    AttNumber, StartAtt, EndAtt: LongInt;

procedure Form1.GetMail1MailInfo(Sender: TObject; info:
    GetMailInfo; addinfo: string);
begin
    case info of
        gmmessageSize: {signals start of new message}
            begin
                StartAtt:=0;
                EndAtt:=0;
            end;
        ...
    end;
end;

procedure Form1.GetMail1AttachmentGetLocation(Sender:TObject;
    Attachment: TTIMEAttachment);
begin
    AttNumber :=AttachmentManager1.NewAttachment(Attachment);
    if StartAtt=0 then StartAtt:=AttNumber;
    EndAtt:=AttNumber;
end;

procedure Form1.GetMail1AttachmentStored(Sender:TObject;
    Attachment: TTIMEAttachment);
begin
    Attachment.Stored:=True;
    AttachmentManager1.Update(AttNumber, Attachment);
end;

procedure Form1.GetMail1MessageLoaded(Sender: TObject);
begin
    NMailbox1.Msg_AttachStart:=StartAtt;
    NMailbox1.Msg_End:=EndAtt;
    NMailbox1.CurrentMailboxNumber:=IN_BOX;
    NMailbox1.Msg_From:=GetMail1.Msg_From;
    NMailbox1.Msg_To:=GetMail1.Msg_To;
    NMailbox1.Msg_Date:=GetMail1.Msg_Date;
    NMailbox1.Msg_Subject:=GetMail1.Msg_Subject;
    NMailbox1.Flag:='U';
    NMailbox1.Msg_Text.Assign(GetMail1.Mail_Text);
    NMailbox1.AddMail;
end;
```

With this code, the mailbox now keeps track of attachments for each message. If the *Msg_AttachStart* property is 0 the message does not have any attachments. If it has any other value, then attachments number *Msg_AttachStart* to *Msg_AttachEnd* are part of the message.

Retrieving Information about an Attachment

Now that we can store attachments, we need a way to retrieve information about the attachments. The *Retrieve* method of the *TAttachmentManager* property can be used to get information about an attachment. The *Retrieve* method is prototyped as follows:

```
function Retrieve(Number: LongInt): TMimeAttachment;  
where Number is the attachment number returned by the NewAttachment  
method. The function returns a record of type TMimeAttachment.
```

Copying an Attachment

The copy an attachment under the control of the *TAttachmentManager* component to another location, you will use the *Copy* method. The *Copy* method is prototyped as follows:

```
procedure Copy(AttNumber: LongInt; Location: String);  
where AttNumber is the attachment number returned by the NewAttachment  
method and Location is the path where to store the attachment.
```

Example:

```
AttachmentManager1.Copy(23, 'c:\test.txt');
```

Deleting an Attachment

It is sometimes desirable to delete an attachment, especially if the attachment has been copied to another location or if it is no longer needed. The *Delete* method can be used to delete an attachment. The *Delete* method is prototyped as follows:

```
procedure Delete(Number: LongInt; NewLocation: string);  
where Number is the attachment number returned by the NewAttachment  
method. If a copy of the attachment is stored at a different location, the  
NewLocation parameter can be used to specify that location. Otherwise, the  
NewLocation parameter should be set to an empty string.
```

Example:

```
AttachmentManager1.Copy(23, 'c:\test.txt');  
AttachmentManager1.Delete(23, 'c:\test.txt');
```

Viewing or Executing Attachments

Before viewing or executing an attachment, it should be stored in a temporary file. The *TAttachmentManager* component can create any needed temporary files, and will also delete any temporary file when the component is destroyed. The *GetTempFile* method is used to create temporary files. This method is prototyped as follows:

```
function GetTempFile(AttNumber: LongInt; Extension: string):  
string;
```













where *Number* is the attachment number returned by the *NewAttachment* method and *Extension* is the desired extension for the temporary file. We will see later how to specify an attachment based on an attachment's mime type, when we discuss the *TMimeManager* object. The function returns the path of the

temporary file containing the attachment. The following example shows how one might view a text attachment:

```
s:=AttachmentManager1.GetTempFile(23,'txt');
Memo1.Lines.LoadFromFile(s);
```

The NMailboxViewer Component

The NMailboxViewer is a graphical component used to present a listing of a mailbox to the user.

Status	From	Date	Size	Subject
	Tony BenBrahim <startect	Thu, 18 Jan 1996 10:31:3	1	Re: test
	Tony BenBrahim <startect	Thu, 18 Jan 1996 10:57:0	2	Re: test
	Tony BenBrahim <startect	Thu, 18 Jan 1996 11:05:0	2	Re: test
	Christian AUBLE <frog@y	Thu, 18 Jan 1996 10:45:0	22	Revue de presse RFI
	Rick Colman <rickcol@un	Thu, 18 Jan 1996 08:39:3	2	Need Developer's Prii
	maillist-delphi@shockwav	Thu, 18 Jan 1996 13:56:4	1	JoinList
	Antonello Carlomagno <st	Thu, 18 Jan 1996 20:44:0	2	INFORMATIONS
	maillist-delphi@shockwav	Thu, 18 Jan 1996 15:31:3	1	JoinList
	Tony BenBrahim <startect	Thu, 18 Jan 1996 16:06:3	2	test
	Don Schueler <dons@axi	Sat, 20 Jan 1996 08:28:3	2	Re: Open Socket Err
	Russell Rice <russell@ph	Sat, 20 Jan 96 11:07:22 -	2	Re: Has the mail arriv
	fatman <fatman@cscs.c	Sun, 21 Jan 1996 07:50:0	3	Thanks

The *NMailboxViewer* component can perform two functions:

- The *NmailboxViewer* displays the content of a mailbox, automatically staying in sync with the mailbox as messages are added, deleted, transferred to another mailbox or when a message's status changes.
- Optionally, the *NMailboxViewer* component can create and manage windows used to display the messages, and can provide functionality to show the next or previous message in a window, as well as delete or transfer selected messages.

The *NMailboxViewer* component also has the following features:

- customizable column headers text.
- resizeable columns at run time
- 4 bitmaps for message status and 1 bitmap for attachments
- sortable columns by clicking on header (left click ascending, right click descending)
- drag and drop of messages between *NMailboxViewer* components.

Initializing the NMailboxViewer component

Before displaying the *NMailboxViewer* component, a few properties must be set to let the component know which mailbox to use and to set the physical appearance of the control.

Selecting a Mailbox

The *Mailbox* property is used to specify which *TNMailbox* component the control will read its information from. Once you have set the *Mailbox* property, you will need to specify the mailbox number to display in the control, using the *MailboxNumber* property.

Example:

```
const
    IN_BOX=1;
...
NMailboxViewer1.Mailbox:=MainForm.Mailbox1;
NMailboxViewer1.MailboxNumber:=IN_BOX;
```

Once you set the *MailboxNumber* property, there might be a slight delay depending on the mailbox size while the component reads in the message headers.

Changing the appearance of the Control

You can change the text of the headers column, the font used for the control, the width of the various columns and the bitmaps displayed for a message's status by setting various property.

The SectionNames Property

To set the text of various header columns, you can use the *SectionNames* property. If you do not set the *SectionNames* property, the column headers will be named 'Status', 'From', 'Date', 'Size' and 'Subject'. The *SectionNames* is of type *Tstrings* and can be set as the following example demonstrates, which replaces the English headers with Spanish headers:

```
NMailboxViewer.SectionNames.Clear;
NMailboxViewer.SectionNames.Add('Estado');
NMailboxViewer.SectionNames.Add('Desde');
NMailboxViewer.SectionNames.Add('Fecha');
NMailboxViewer.SectionNames.Add('Tamano');
NMailboxViewer.SectionNames.Add('Sujeto');
```

The *SectionNames* property can also be edited at design time from the Object Inspector.

The SetHeaderSize Method

The *SetHeaderSize* method can be used to set the starting width of one or more columns. The prototype of the *SetHeaderSize* method is:

```
procedure SetHeaderSize(x,s: integer);
```

where *x* is the column number and *s* is the new size in pixels. Columns are numbered starting with 1. For example, to change the width of the first column to 200 pixels, you would use the following code:

```
NMailboxViewer.SetHeaderSize(1,200);
```


Setting Status Bitmaps

You must set the *AttachmentBitmap*, *UnreadBitmap*, *QueuedBitmap*, *RepliedBitmap* and *ReadBitmap* properties if you want bitmaps to be displayed in the first column.

The best way to specify the desired bitmaps is to store them in a resource file and load them at run time. This involves the following steps.

- Create a resource file using the image ImageEdit program that ships with Borland Delphi. Note: You must name this file a different name than your project. For example, if your program project file is 'mymail.dpr', do not name the resource file 'mymail.res'. In this example, we will save our resource file as 'extra.res'.
- In the resource file, create five bitmaps, 16 pixels wide by 16 pixels high, 16 colors. We now 5 bitmaps, named 'BITMAP_1' to 'BITMAP_5'. You can rename these bitmaps to give them a more descriptive name. In this example, we renamed the bitmaps MAIL_READ,MAIL_UNREAD, etc..
- In the code for the project's main form, add the following line in the implementation section:

```
{ $R xtra.res }
```

- The following code can be used to load the bitmaps into a Tbitmap object:

```
interface
```

```
var
```

```
    ReadBM,QueuedBM,RepliedBM,UnreadBM,AttachmentBM;
```

```
...
```

```
implementation
```

```
($R extra.res)
```

```
procedure InitializeProgram;
```

```
begin
```

```
    ReadBM:=TBitmap.Create;
```

```
    ReadBM.Handle:=LoadBitmap(HInstance,'MAIL_READ');
```

```
    UnreadBM:=TBitmap.Create;
```

```
        UnreadBM.Handle:=LoadBitmap(HInstance,'MAIL_UNREAD');
```

```
    RepliedBM:=TBitmap.Create;
```

```
    RepliedBM.Handle:=LoadBitmap(HInstance,'MAIL_REPLIED');
```

```
    AttachmentBM:=TBitmap.Create;
```

```
    AttachmentBM.Handle :=LoadBitmap(HInstance,'MAIL_ATTACHMENT');
```

```
;
```

```
    QueuedBM:=TBitmap.Create;
```

```
    QueuedBM.Handle:=LoadBitmap(HInstance,'MAIL_QUEUED');
```

```
...
```

```
end;
```

- You can now assign these bitmaps to the bitmap properties of the *NMailboxViewer* control whenever needed.

```
NMailboxViewer1.AttachmentBitmap.Assign(AttachmentBM);  
NMailboxViewer1.UnreadBitmap.Assign(UnreadBM);  
NMailboxViewer1.QueuedBitmap.Assign(QueuedBM);  
NMailboxViewer1.RepliedBitmap.Assign(RepliedBM);  
NMailboxViewer1.ReadBitmap.Assign(ReadBM);
```

Which bitmap is displayed in the first column will depend on the value of the *Msg_Status* property of the *TNMailbox* component:

Msg_Status	Bitmap Displayed
'U'	UnreadBitmap
'V'	ReadBitmap
'Q'	QueuedBitmap
'R'	RepliedBitmap

The Font property

The *Font* property (type *Tfont*) controls the font used by the control's text. Refer to the *Delphi Language Reference* for more information about working with the *Font* property.

The Align Property

The *Align* property controls the alignment of the control within the parent control. Refer to the *Delphi Language Reference* for more information about working with the *Align* property.

Working with the NMailboxViewer component

The *NMailboxViewer* automatically keeps track of any changes in its associated mailbox and displays any changes as they happen. For example, to change the status of a message in the component, you would use the *UpdateFlag* method of the associated mailbox and the change will be automatically reflected in the *NMailboxViewer* component. To remove a message from the component, you must delete or transfer the message to another mailbox.

We will first look at the automatic actions of the *NMailboxViewer*. We will then look at the two modes of operations for the *NMailboxViewer* component, *mail viewer manager* mode with the *NMailboxViewer* component managing the windows used to view messages, and manual mode.

Automatic Operations of the NMailboxViewer Component

Moving Messages

The *NMailboxViewer* component supports drag and drop of messages from one *NMailboxViewer* to another. There are two properties which control the operation

of the drag and drop feature, the *DragEnabled* and the *DropEnabled* property. If the *DragEnabled* property is set to *True*, the user will be able to drag items from the component. If the *DropEnabled* property is set to *True*, the component will accept items from other components.

When an item is dragged from a *NMailboxViewer* component and dropped to another *NMailboxViewer* component, the components automatically takes care of the details of moving the associated from the origin mailbox to the destination mailbox, as long as both *NMailboxViewer* components use the same *TNMailbox* component.

Message Count

To find out how many messages are being displayed by the *NMailboxViewer* component, you should use the *NumMessages* property. Additionally, you can use the *OnChange* event to be notified when the number of messages changes. The following example shows how one might set the caption of a form containing a *NMailboxViewer* component to reflect the number of messages .

```
procedure Form1.NMailboxViewer1Change(Sender: Tobject);
begin
    Caption:=IntToStr(NMailboxViewer1.NumMessages)+' messages';
end;
```

Sorting messages

The component automatically sorts messages displayed in the *NmailboxViewer* component when the user clicks on one the component's headers. Left clicking on a header causes the messages to be sorted in ascending order according to the values in the header's column. For example, left clicking on the *Date* column header will cause all messages to be sorted in ascending order based on the order they were received. When a column header is right clicked, the messages are sorted in descending order.

The *NewMessageTop* property can be used to select the initial sort order of the messages displayed by the component. Setting the *NewMessageTop* property to *True* causes the newest messages to be displayed on top when the component is first displayed. If *NewMessageTop* is set to *False*, the messages are displayed in the order they were received, with the oldest message on top.

Finding selected messages

To find out which messages are selected, you can use the *Selected* property in conjunction with the *IndexToMessageNumber* method. The *Selected* property is prototyped as follows:

```
property Selected[X: Integer]: Boolean;
```

where the X parameter is the position of the item in the component, with the first item having an X value of 0. The value of the *Selected* property is *True* if the item is selected and *False* if the item is not selected.

The *IndexToMessageNumber* property translates the index of a message (its position in the component) to a message number. This is necessary because the first message displayed in the component is not always the first message in the

mailbox, depending on the value of the *NewMessageTop* property and depending on the sort order selected by the user by clicking on a column header.

The following example shows how to iterate through all the entries in a *NMailboxViewer* component in response to a button click and doing something with the selected entries:

```
procedure Form1.Button1Click(Sender: TObject);
var
    i,msg: integer;
begin
    for i:=0 to NMailboxViewer1.NumMessages-1 do
    begin
        if Selected[i] then
        begin
            msg:=IndexToMessageNumber(i);
            {do something with selected message}
        end;
    end;
end;
```

Note, however that if you plan to delete or transfer messages to another mailbox, a slightly different procedure should be used, as the number of messages (*NumMessages*) will decrease with each message deletion or transfer.

```
procedure Form1.Button1Click(Sender: TObject);
var
    i,msg: integer;
begin
    i:=0;
    while i<NMailboxViewer1.NumMessages do
    begin
        if Selected[i] then
        begin
            msg:=IndexToMessageNumber(i);
            {delete or transfer message}
        end
        else Inc(i);
    end;
end;
```

Methods Available in Message Viewer Manager or Manual mode

There are two methods available whether the component is in the *Message Viewer Manager* mode or in manual mode.

The *DeleteSelectedMessages* method is used to delete all selected messages from the mailbox associated with the *NMailboxViewer* component. (The associated entries in the component will also automatically be removed by the automatic synchronization with the mailbox). The *DeleteSelectedMessages* method has no parameters.

The *TransferSelectedMessages* method is used to transfer all selected messages to another mailbox. The *TransferSelectedMessages* method is prototyped as follows:

```
procedure TransferSelectedMessages(TargetMailboxNumber: integer);
```

where *TargetMailboxNumber* is the mailbox number to which messages should be transferred.

The following example shows how one might implement a trash button for the *NMailboxViewer* component:

```
interface

const
    IN_MAILBOX      = 1;
    OUT_MAILBOX     = 2;
    TRASH_MAILBOX   = 3;
    ...

implementation

procedure Form1.TrashButtonClick(Sender: TObject);
begin
    NMailboxViewer1.TransferSelectedMessages(TRASH_MAILBOX);
end;
```

Manual mode

Double Clicking on a Message

When a user double clicks on a message, the component triggers an *OnMessageDoubleClick* event. The *OnMessageDoubleClick* event is prototyped as follows:

```
procedure(Sender: TObject;MessageNumber: integer);
```

where *Sender* is the control which triggered the event and *MessageNumber* is the message that was clicked on. The following example loads a message into a memo box when the control is double clicked:

```
procedure Form1.NMailboxViewerMessageDoubleClick(Sender: TObject;
    MessageNumber: integer);
begin
    NMailbox1.CurrentMailboxNumber:=(Sender as
    NMailboxViewer).Mailbox;
    NMailbox1.Msg_Number:=MessageNumber;
    NMailbox1.LoadMail;
    Memo1.Lines.Assign(NMailbox1.Msg_Text);
end;
```

Note that the *OnMessageDoubleClick* event is not available when the *NMailboxViewer* component is in the *Message Viewer Manager* mode.

Message Viewer Manager mode

When the *NMailboxViewer* component is placed in *Message Viewer Manager* mode, the component manages all aspects of displaying messages in any form provided by the user. This mode provides methods to display the next or previous message in a window, automatically creates a new form when a message is double clicked and closes forms whose messages have been

deleted or transferred to a another mailbox. *Message Viewer Manager* mode is intended to be used with MDI applications.

To convince you of the usefulness of *Message Viewer Manager* mode, let's look at what would be involved with some common operations if you handled these operations in manual mode:

- Displaying a message when a user double clicks on an entry: Handling the *OnMessageDoubleClicked* event gives you access to the message number. But before creating a new window to display the message, you would need to check if a window containing the same message is already displayed. If that is the case, you would need to bring that window to the front, rather than creating a new window. For each window you create, you also need to keep track of the associated mailbox number and message number.
- Handling message deletion: When you delete a message, you would need to update the message number information for all windows displaying succeeding messages. For example, if a window displaying message 5 was open and you deleted message 3, you would need to update the information for the open window, which is now displaying message 4. Failure to do that would cause a function which deleted a windows message to delete the wrong message or a function to display the next message in a window to display the wrong message. Additionally, when you delete a message, you should check to see if a window displaying that message is open, and if that is the case, close that window.
- Showing the next or previous message: If you have correctly kept track of message deletion, you should have the correct message number for the message to be operated on. However, showing the next or previous message is not simply a matter of incrementing or decrementing the message number. You also have to consider that messages might be sorted in the *NMailboxViewer* component. To correctly display the next or previous message, you would have to find the index of the message in the component, increment or decrement the index and convert the index back to a message number. You would then have to display the message, avoiding duplicate windows for the same message.

The *NMailboxViewer* component relieves you of all these details in *Message Viewer Manager* mode. To place the component in this mode, you will use the *EnableMessageViewerManager* method, which is prototyped as follows:

```
procedure EnableMessageViewerManager(MessageViewerFormClass: TFormClass);
```

where the *MessageViewerFormClass* is the class of the form you want the *NMailboxViewer* component to create to display a message. Any class derived from *TForm* can be used to display messages. This means that any form that you create with *Delphi* can be used to display messages. Note that *Delphi* assigns a class to a form based on the name you give the form, by prepending a 'T' to the form name. For example, if a form is named *Form1*, its class is *TForm1*.

There are only two restrictions that are placed on the form used to display messages:

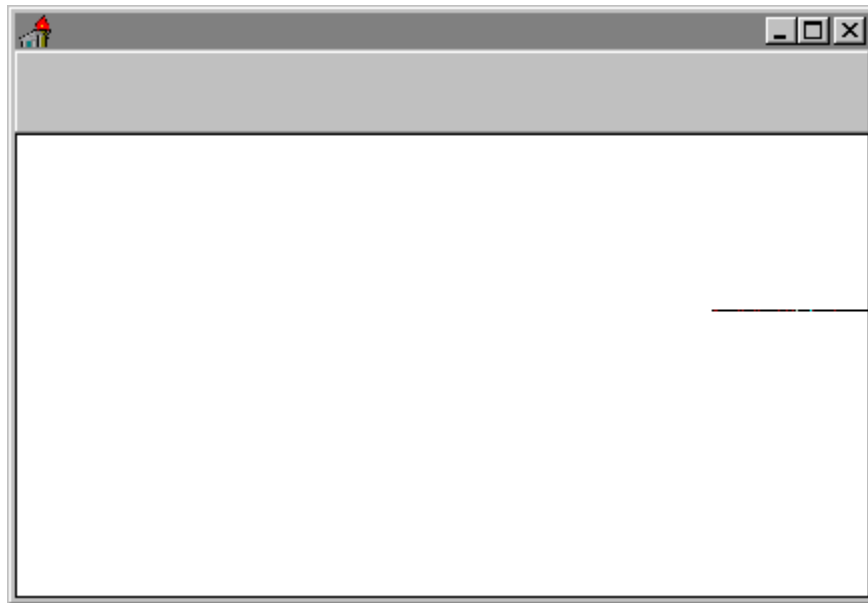
- It must have a form style *fsMDIChild*, as the *Message Viewer Manager* mode is only intended to manage forms in an MDI application.

- It must support four custom windows messages, *MBV_PARENT*, *MBV_SETMESSAGE*, *MBV_UPDATEINFO* and *MBV_CLOSE*. We will see later how to handle how to write message handlers for these messages.

Designing the message viewer form

The first step is to design a form to display the messages, just as you would design any other form.

In the following example, we design a simple form with a memo to display a message and a panel which will use later to add buttons to the form.



At this point, the source code for the form might look as follows:

```
unit Unit2;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls,
  Forms, Dialogs;

type
  TViewMail = class(TForm)
    Panel1: TPanel;
    Memo1: TMemo;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  ViewMail: TViewMail;
```

```
implementation
{ $R *.DFM }
end.
```

To enable a *NMailboxViewer* component to use this form to display messages, you would need to add the following line when the component's parent form is displayed:

```
procedure Form1.Show(Sender:TObject);
begin
    NMailboxViewer1.EnableMessageViewerManager(TViewMail);
end;
```

We will then have to modify our *TViewMail* form to handle the *MBV_PARENT*, *MBV_SETMESSAGE*, *MBV_UPDATEINFO* and *MBV_CLOSE* windows messages.

Handling Windows Messages

To modify a form to handle a specific message, you need to write a procedure which is prototyped as follows:

```
procedure SomeProcedure(var Message: TMessage);message
WM_SOMEMESSAGE;
```

where *WM_SOMEMESSAGE* is the message to be handled and the *Message* parameter is a record of type *TMessage* prototyped as follows:

```
type TMessage=record
    wParam: word;
    lParam: LongInt;
    Result: LongInt;
end;
```

In using the *NMailboxViewer* component, we are only interested in the *wParam* and *lParam* parameters, which will be used to pass information between the *NMailboxViewer* and the message viewer form it manages.

The MBV_PARENT message

The *MBV_PARENT* message is sent by a *NMailboxViewer* component to a message viewer form right after it creates it. The *lParam* parameter contains the address of the *NMailboxViewer* which created the form receiving the message. The following example shows the code, highlighted by a bold typeface, that you would add to handle the *MBV_PARENT* message.

```
interface
uses ...,nmbxvwr;

type TViewMail=class(TForm)
...
private
    ParentViewer: NMailboxViewer;
    procedure MBVParent(var Message: TMessage);message
MBV_PARENT;
```



```

...
end;

implementation

procedure TViewMail.MBVParent(var Message: TMessage);
begin
    ParentViewer:=NMailboxViewer(Message.lParam);
end;

```

The MBV_SETMESSAGE message

The *MBV_SETMESSAGE* message is sent whenever the *NMailboxViewer* component wants the message viewer form to display a different message. The *wParam* parameter contains the mailbox number and the *lParam* parameter contains the message number to be displayed. The following example shows the code, highlighted by a bold typeface, that you would add to handle the *MBV_SETMESSAGE* message.

```

unit Unit2;

interface

uses... ,Nmbxvwr;

type TViewMail = class(TForm)
...
private
    ParentViewer: NMailboxViewer;
    MailboxNumber,MessageNumber: integer;
    procedure MBVParent(var Message: TMessage);message
MBV_PARENT;
    procedure MBVSetMessage(var Message: Tmessage); message
MBV_SETMESSAGE;
...
end;

implementation

procedure TViewMail.MBVSetMessage(var Message: TMessage);
begin
    with ParentViewer.Mailbox do
    begin
        MailboxNumber:= Message.wParam;
        MessageNumber:= Message.lParam;
        CurrentMailboxNumber:=Message.wParam;
        Msg_Number:=Message.lParam;
        LoadMail;
        Memol.Assign(Msg_Text);
    end;
end;

```

The MBV_UPDATEINFO message

The *MBV_UPDATEINFO* message is used to notify the message viewer form that the message it is displaying now has a different message number. For example, if a form is displaying message 5 in mailbox 1, when message 3 is deleted in the same mailbox, the *NMailboxViewer* component will send an

MBV_UPDATEINFO message to inform the form that it is now displaying message 4 in mailbox 1. The following example shows the code, highlighted by a bold typeface, that you would add to handle the *MBV_UPDATEINFO* message.

```
unit Unit2;

interface

uses... ,Nmbxvwr;

type TViewMail = class(TForm)
...
private
    ParentViewer: NMailboxViewer;
    MailboxNumber,MessageNumber: integer;
    procedure MBVParent(var Message: TMessage);message
MBV_PARENT;
    procedure MBVSetMessage(var Message: TMessage); message
MBV_SETMESSAGE;
    procedure MBVUpdateInfo(var Message: TMessage); message
MBV_UPDATEINFO;
    ...
end;

implementation

procedure TViewMail.MBVUpdateInfo(var Msg: TMessage);
begin
    MailboxNumber:=Msg.wParam;
    MessageNumber:=Msg.lParam;
end;
```

The **MBV_CLOSE** message

The *MBV_CLOSE* message is sent whenever the *NMailboxViewer* component wants the message viewer form to close, for example when the message a form displays has been deleted or transferred to another mailbox. The following example shows the code, highlighted by a bold typeface, that you would add to handle the *MBV_CLOSE* message.

```
unit Unit2;

interface

uses... ,Nmbxvwr;

type TViewMail = class(TForm)
...
private
    ParentViewer: NMailboxViewer;
    MailboxNumber,MessageNumber: integer;
```

```

        procedure MBVParent(var Message: TMessage);message
MBV_PARENT;
        procedure MBVSetMessage(var Message: TMessage); message
MBV_SETMESSAGE;
        procedure MBVUpdateInfo(var Message: TMessage); message
MBV_UPDATEINFO;
        procedure MBVClose(var Message: TMessage); message
MBV_CLOSE;
        ...
    end;

implementation

procedure TViewMail.MBVClose(var Message: TMessage);
begin
    Close;
end;

```

Adding Functionality to the Message Viewer Form

Once the message viewer form you have designed can handle the four messages described above, you can add additional functionality by manipulating the message directly. For example, to transfer the message displayed in a message viewer form to the Trash mailbox, you would use the message and mailbox number supplied by the *MBV_SETMESSAGE* and *MBV_UPDATEINFO* messages to move the message, as the following example shows:

```

procedure TViewMail.DeleteButtonClick(Sender: Tobject);
begin
    with ParentViewer.Mailbox do
    begin
        CurrentMailboxNumber:=MailboxNumber;
        Msg_Number:=MessageNumber;
        TransferTo(MailboxFile(TRASH_MAILBOX));
    end;
end;

```

Note that the *NmailboxViewer* component will automatically take care of removing the message from the component and of closing the form displaying the message.

The NextMessage and PreviousMessage methods

The *NMailboxViewer* component supplies two methods for displaying the next or previous message relative to the message currently displayed in a message viewer form, keeping within the sequence in which they are displayed in the component. The *NextMessage* and *PreviousMessage* methods are prototyped as follows:

```

procedure NextMessage(f: TForm);
procedure PreviousMessage(f: TForm);

```

where *f* is the mail viewer form (created by the *NMailboxViewer* component) for which to display the next or previous message. Note that the message will be displayed in the same form, unless the next or previous message is already displayed in another form, in which case focus will shift to that form.

The following example shows how trivial it is to add a next or previous button to a mail viewer form:

```
procedure TViewMail.NextButtonClick(Sender: TObject);
begin
    ParentViewer.NextMessage(self);
end;

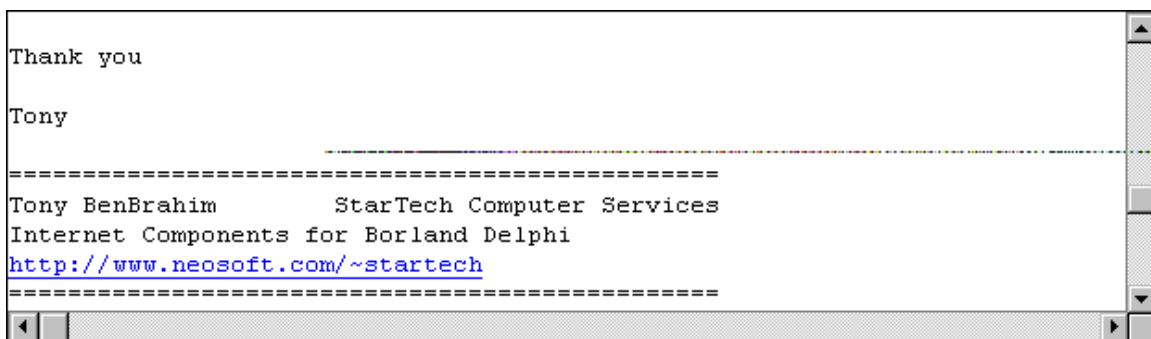
procedure TViewMail.PreviousButtonClick(Sender: TObject);
begin
    ParentViewer.PreviousMessage(self);
end;
```

The ViewSelectedMessages Method

The *ViewSelectedMessages* method displays all selected messages in the *NMailboxViewer* component. Although the *NMailboxViewer* component checks for available resources before creating each window, it is possible that a user might select a large number of messages to be displayed, leaving the system in a state with very few resources left. This could cause problems in poorly designed programs.

The TStringsViewer Component

The *TMemo* component included with *Borland Delphi* has limitations on the size of the text it can display, which can vary anywhere from 20K to 32K. Unfortunately, many Internet messages will exceed this size, making the *TMemo* component inappropriate to display messages. The *TStringsViewer* component was designed to overcome this limitation, and can be used to display text which can exceed 1.5 Megabytes in size. Additionally, the *TStringsViewer* provides a feature which is becoming more popular, highlighting of http URL's, so that a specific action can be taken when the link is clicked, such as launching a browser to display the URL.



TStringsViewer properties

The *TstringsViewer* component was designed to be as functionally similar to the *Tmemo* component as possible. You will find that most of the properties and methods of the *TStringsViewer* component are similar to those of the *Tmemo* component.

Modifying the Control's Appearance

The following properties can be used to modify the control's appearance:

- Align
- Color
- Font

All of these properties operate in the same fashion as they do on the *TMemo* component. The only exception is the *Font* property, which should specify a monospaced font, such as '*Courier New*', in order for marking to work properly.

Modifying the Control's Operation

The following properties can be used to modify the control's operation:

- HelpContext
- Hint
- PopupMenu
- ShowHint
- TabOrder
- TabStop

These properties are identical to the similarly named properties of the *TMemo* component.

Modifying the Control's Contents

The *Lines* property can be used to change the text of the *TStringsViewer* component. It is of type *TSVStrings*, which is descended from *Tstrings*, so all properties and methods of the *Tstrings* objects, such as *Add*, *Delete* and *Insert*, can be used with the *Lines* property. The *Modified* property is set to *True* and the *OnChange* event is triggered whenever the strings referenced by the *Lines* property are modified. The *Clear* method can be used to clear the contents of the *TStringsViewer* component.

TStringsViewer Events

The *TStringsViewer* component provides the following events, which are analogous to the similarly named events of the *TMemo* component:

- OnClick
- OnDbClick
- OnKeyDown
- OnKeyUp
- OnKeyPress
- OnMouseDown
- OnMouseUp
- OnMouseMove
- OnEnter
- OnExit
- OnDragDrop
- OnEndDrag
- OnDragOver

Additionally, the *TStringsViewer* component provides an *OnLinkClicked* event, which is triggered when an http link is clicked. The *OnLinkClicked* event is prototyped as follows:

```
procedure(Sender: TObject; Link: string);
```

where *Sender* is the control which triggered the event and *Link* is the link which was clicked.

Clipboard Operations

The *TStringsViewer* component supports marking of text with the mouse, and copying of text to the clipboard (up to 30K) using the Ctrl+C or Ctrl+X key combinations, or using the *CopytoClipboard* or *CuttoClipboard* methods. Text can also be selected using the *SelectAll* method. Note that since the component's text is read only, pasting of text to the component is not supported.

THE FTP PACKAGE

The FTP package contains three components that greatly facilitate file transfers using the File Transfer Protocol (FTP). The TFtp component is a basic implementation of the FTP protocol. The FtpUrlDialog component simplifies file transfer operations by combining file transfer and status display functionality into one component. The TFtpPlus component is an enhanced FTP component, which allows for transferring several files or directories recursively with just one command.

The TFTP Component



The TFtp component operates in two different modes:

- In *interactive mode*, you issue commands through the component and wait for an event to signal that the transaction has completed.
- In *URL mode*, you issue one command which initiates a file transfer transaction, including logging in, file transfer and logging off.

We will first look at the operation of the TFtp component in *interactive mode*.

The TFtp Component in interactive mode

There are three steps to using the TFtp component in interactive mode:

- Provide the information required for the component to log on to a server, such as the server alias or IP address, the user name and the user password.
- Issue commands such as *Login*, *GetFile*, and *ChangeDirectory* to navigate the directory structure of the FTP site, transfer files, delete or rename files, etc...
- Respond to events which indicate when a particular command has completed. Each command has a corresponding event associated with it. For

example, when the *Login* command completes, you will receive an *OnLoggedIn* event.

Initializing the FTP component

Before the TFTP component is able to connect into an FTP server, you must provide some information so that the component is able to connect and log in. The following properties are used to initialize the component:

The FTPServer property

You will use the *FTPServer* property to specify which server to connect to. In the *FTPServer* property, you will specify the name (or alias) of the FTP server connect to, or its IP address (numerical address)

Example:

```
FTP1.FTPServer:='ftp.microsoft.com';    {using alias}
```

or

```
FTP1.FTPServer:='266.45.34.1';    {using IP address}
```

Should you use the server alias or its IP address? If you have physical control of the mail server (i.e. you will know if its address changes), using the server address in the *FTPServer* property will save you the overhead of name resolution every time you connect. Most of the time, however, you will want to use the server name, as there is no guarantee that the server's address will remain the same over time.

The FTPPort property

You will use the *FTPPort* property to specify which server to connect to. The port for an FTP server is almost always 21.

Example:

```
FTP1.FTPPort:=21;
```

Logging in: the UserName, UserPassword and UserAccount properties

There are two ways in which you can access an FTP server:

anonymous access: Anonymous access is general access to an FTP server which will allow you to download files and sometimes upload files in a specified directory (such as the *incoming* directory).

Private access: Private access allows you to manage a portion of the FTP server's directories, allowing you to download and upload files, delete or rename files and create directories within your assigned home directory.

In both types of access, you will need to specify a user ID and password, and you optionally need to specify an account for private access. You will use the *UserName*, *UserPassword* and *UserAccount* properties to provide that

information. The following examples demonstrate the information one might provide to login anonymously or privately.

Example:

```
FTP1.UserName:='anonymous';    {anonymous login}
FTP1.UserPassword:='jdoe@acme.com;

or

FTP1.UserName:='jsmith';    {private login}
FTP1.UserPassword:='mysecret';
FTP1.UserAccount:='12345';    {this line would only be needed
if logging in to an ftp server which required an account to login}
```

To log in anonymously into an FTP server, you will specify a user name of 'anonymous' or 'guest' and a password consisting of an e-mail address. Although it is preferable to send your e-mail address as a password, note that any string consisting of a '@' character with preceeding and succeeding alphanumeric characters will suffice as a password.

To log in privately into an FTP server, you will specify the user name, user password and optionally the user account provided by the FTP server administrator.

Logging in to the FTP Server

Once you have provided the *FTPServer*, *FTPPort*, *UserName*, *UserPassword* and optionally the *UserAccount* properties, you are ready to log in. The *Login* method is used to initiate a connection to an FTP server and negotiate the login procedure.

Example:

```
Ftp1.Server:='ftp.microsoft.com';
Ftp1.Port:=21;
FTP1.UserName:='anonymous';
FTP1.UserPassword:='me@here';
FTP1.Login;
```

Once the login transaction has completed, you will receive an *OnLoggedIn* event. Error handling will be covered in a succeeding section. When you receive an *OnLoggedIn* event, the *InitialDirectory* property will contain the current path on the FTP server.

After Logging In: doing something useful

Once you have successfully logged in to an FTP server, there are several useful commands you can issue to navigate through the directory structure, modify the directory structure by deleting, creating or renaming files, list the files in a directory, or upload, download, delete or rename files. The following table shows the various tasks that can be accomplished, the method that is used to accomplish that task and the event that indicates completion of that task.

Task	Method	Event
Log in	Login	OnLoggedIn
List Files and Directories	FTPRefresh	OnListingDone
Delete a File	DeleteFile	OnFileDeleted
Create a Directory	CreateDirectory	OnDirectoryCreated
Delete a Directory	DeleteDirectory	OnDirectoryDeleted
Rename a File or Directory	RenameFile	OnFileRenamed
Upload a File	PutFile	OnFileStored
Download a File	GetFile	OnFileReceived
Change Directory	ChangeDirectory	OnDirectoryChange
End the session	Quit	OnFTPQuit

Listing Files and Directories

You will use the *FTPRefresh* method to list files and directories. Once the command has completed, you will receive an *OnListingDone* event.

The *Listing* property (type *Tstrings*) will contain the file listing as obtained from the FTP server. The following shows an example of what the *Listing* property might contain after obtaining a listing from an FTP server:

```
total 42
-rw-rw-r-- 1 root    wheel    627 Sep  7 1995 .Links
-rw-rw-r-- 1 root    wheel      80 Sep  5 1995 .about.html
drwxr-xr-x 4 root    wheel    512 Feb  8 18:24 archive2
drwxr-xr-x 2 root    wheel    512 Jun 19 1995 archive3
drwxrwxr-x 12 root    ftpmaint 512 Feb  5 22:03 archive4
d--x--x--x 2 root    wheel    512 Dec  1 05:50 bin
dr-xr-xr-x 2 root    wheel    512 May 22 1994 dev
dr-xr-xr-x 3 root    wheel    512 Nov 23 1994 etc
d--x--x--x 2 root    wheel    512 Dec 20 18:42 hidden
drwxrwxr-x 2 archiver ftpmaint 512 Dec  7 1994 info
dr-xrwxr-x 5 root    wheel    512 Sep  9 1995 neopolis
drwxrwxr-x 5 archiver ftpmaint 512 Oct 27 00:57 neosoft
drwxrwxr-x 2 archiver ftpmaint 512 Sep 26 1995 organizations
drwxrwxr-x 18 archiver ftpmaint 512 Feb  9 21:53 pub
drwxrwx--x 5 archiver ftpmaint 512 Mar 27 1995 usr
drwxrwxr-x 3 archiver ftpmaint 512 Dec 12 19:48 vendor
```

The TFTP component also parses certain listings into two properties of type *TStrings*, the *Files* and *Directories* properties. After a listing, the *Directories* properties will contain the name of all the subdirectories and links found in the listing, and the *Files* property will contain the name of all the files found in the listing. The TFTP component is currently capable of parsing the following type of listings:

- Unix and variants (Ulrix, Win NT, BSD, etc..)
- Vax/VMS listings (VMS-Multinet, VMS-UCx, etc...)
- IBM-VM listings
- Novell File Server listings

After a listing command, the *InitialDirectory* property is also updated to reflect the current directory on the server.

Deleting a File

To delete a file on an FTP server, you will use the *DeleteFile* method. The *DeleteFile* method takes one argument, the name of the file or the directory. Once the command has completed, you will receive an *OnFileDeleted* event.

Example:

```
Ftp1.DeleteFile('/pub/jsmith/test.txt');
```

Creating a Directory

To create a directory on an FTP server, you will use the *CreateDirectory* method. The *CreateDirectory* method takes one argument, the name of the directory to be created. Once the command has completed, you will receive an *OnDirectoryCreated* event.

Example:

```
Ftp1.CreateDirectory('/pub/jsmith/newdir');
```

Deleting a Directory

To delete a directory on an FTP server, you will use the *DeleteDirectory* method. The *DeleteDirectory* method takes one argument, the name of the directory to be deleted. Once the command has completed, you will receive an *OnDirectoryDeleted* event.

Example:

```
Ftp1.DeleteDirectory('/pub/jsmith/newdir');
```

Renaming a File or Directory

To rename a file or directory, you will use the *RenameFile* method. The *RenameFile* method is prototyped as follows:

```
procedure RenameFile(oldname,newname: string);
```

where the *oldname* parameter contains the original file name and the *newname* parameter contains the new file name. When the command has completed, you will receive an *OnFileRenamed* event.

Example:

```
Ftp1.RenameFile('/pub/jsmith/files','/pub/jsmith/docs');
```

Uploading a File

To upload a file to an FTP server, you will set the *LocalFile* and *RemoteFile* properties and call the *PutFile* method. The *LocalFile* property contains the name of the file to be uploaded. The *RemoteFile* property contains the name of the destination file on the FTP server. Once this command has completed, you will receive an *OnFileStored* event.

Example:

```
FTP1.LocalFile:='c:\mirror\default.htm';  
FTP1.RemoteFile:='/pub/users/jsmith/default.htm';  
FTP1.Put File;
```

Downloading a File

To download a file from an FTP server, you will set the *RemoteFile* and *LocalFile* properties and call the *GetFile* method. The *RemoteFile* property contains the name of the file to be downloaded from the FTP server. The *LocalFile* property contains the destination file name on the local file system. Once this command has completed, you will receive an *OnFileReceived* event.

Example:

```
Ftp1.RemoteFile:='/pub/users/jsmith/default.htm';  
Ftp1.LocalFile:='c:\mirror\default.htm';  
Ftp1.GetFile;
```

Changing Directory

To change directory on an FTP server, you will use the *ChangeDirectory* method. The *ChangeDirectory* method takes one parameter, the name of the new directory to change to. Once this command has completed, you will receive an *OnDirectoryChanged* event.

Example:

```
Ftp1.ChangeDirectory('/pub/users');
```

Ending a Session

To end an FTP session, you will use the *Quit* method. The *Quit* method takes no parameters. Once the control has disconnected and is ready for another connection, you will receive an *OnFTPQuit* event.

Example:

```
FTP1.Quit;
```

Handling Errors

There are two types of error that can be encountered by the FTP component, fatal errors and non-fatal errors. Fatal errors are errors that hinder the further normal operation of the component. Examples of fatal errors are:

- Inability to log in after calling the *Login* method, either because the maximum user limit has been reached, the user id/password/account information is invalid or the FTP server is down.
- Inability to establish a data connection after calling the *GetFile* or *PutFile* method.
- Fatal errors generated by the FTP server, resulting in a premature disconnection.

After a fatal error, the FTP component will generate an *OnFtpError* and set the *FTPError* property to indicate the type of error that occurred, will set the *Success*

property to *False* and will disconnect from the server, which will result in an *OnFtpQuit* event to be triggered.

Non-fatal errors are errors which result in attempts to perform an invalid action by the user, but do not hinder the further operation of the component. Examples of non-fatal errors are:

- Attempting to create a directory with invalid characters in the directory name or in a directory where the user does not have sufficient privileges.
- Attempting to delete a directory which does not exist or in a directory where the user does not have sufficient privileges.
- Attempting to delete a file which does not exist or in a directory where the user does not have sufficient privileges.
- Attempting to upload a file with invalid characters in the directory name or in a directory where the user does not have sufficient privileges.

After a non-fatal error, the FTP component will generate an *OnFtpError* and set the *FTPErr* property to indicate the type of error that occurred, and will set the *Success* property to *False*.

The easiest to handle fatal and non-fatal errors is in the *OnFTPErr* event, which is prototyped as follows:

```
procedure (Sender : TObject; info: FtpInfo; addinfo: string);
```

where *Sender* is the *TFtp* component which triggered the event, *info* contains an error code. Note that the *addinfo* parameter is not used at this time.

The following table lists possible values of the *info* parameter and their meaning.

Value	Meaning	Fatal?
ftpWinsockNotInitialized	The winsock DLL could not be loaded.	Y
ftpConnAborted	The connection was aborted due to timeout or other error	Y
ftpConnReset	The connection was reset by the remote server	Y
ftpConnectTimeOut	No connection occurred before the specified timeout	Y
ftpOutOfSockets	The maximum number of sockets are already in use	Y
ftpNetworkUnreachable	The winsock interface has detected that the network is unreachable	Y
ftpAddressNotAvailable	The specified remote address is not available from this host	Y
ftpConnectionRefused	The remote server forcefully rejected the connection	Y
ftpProtocolError	A basic FTP command was not recognized by the FTP server	Y
ftpCanceled	The transfer was canceled with a <i>StopTransfer</i> call.	N*
ftpUnknown	An unknown error happened (you should not receive this error)	Y
ftpGeneralWinsockError	The winsock interface has failed	Y
ftpNetworkDown	The network is down and a connection could not be established	Y
ftpInvalidAddress	The address specified in <i>FTPServer</i> could not be resolved	Y
ftpInternalError	An internal error occurred. (you should not receive this error)	Y
ftpAddressResolutionError	An error occurred while trying to resolve the server address	Y
ftpPrematureDisconnect	The component disconnected before the transaction completed	Y
ftpHostUnreachable	The host is not reachable from this machine at this time	Y
ftpNoServer	No server specified in <i>FtpServer</i> .	Y
ftpAlreadyBusy	A command was issued before the previous command completed.	N
ftpAccessDenied	Access to the server was denied at login	Y
ftpFileOpen	The file specified in <i>LocalFile</i> could not be opened.	N
ftpFileWrite	An error occurred while writing to <i>LocalFile</i> (such as disk full)	N
ftpFileRead	An error occurred while reading from <i>LocalFile</i>	N
ftpFileNotFound	The file specified as a parameter or in <i>RemoteFile</i> was not found	N
ftpServerDown	The FTP server is down and not accepting connections	Y
ftpDataError	An error occurred while transferring data	Y
ftpBadURL	An improperly format URL has been specified	Y

*Although an *ftpCancelled* error is not fatal as far as the *TFtp* component is concerned, most servers will respond by closing the connection if the data connection is closed prematurely.

Non fatal errors can also be handled in the associated completion event for the command that was issued. For example, to check if a call to the *CreateDirectory* method succeeded, you could check the value of the *Success* property in the handler for the *OnDirectoryCreated* event. In the following example, the *CreateDirectory* method is called. In the event handler for the *OnDirectoryCreated* event, the *Success* property is checked. If it is true, the *ChangeDirectory* method is invoked to change to the new directory. If it is false, the *Quit* method is invoked to end the FTP session.

Example:

```

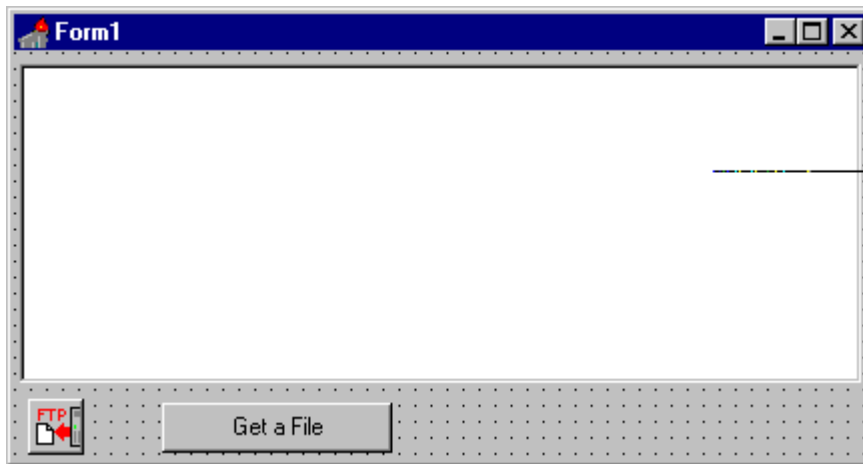
procedure TForm.SomeProcedure;
begin
    Ftp1.CreateDirectory(newdir);
end;
...
procedure TForm1.Ftp1DirectoryCreated(Sender: TObject);
begin
    if Success=True then Ftp1.ChangeDirectory(newdir)
    else Ftp1.Quit;
end;

```

Putting it together: a simple file transfer

When you put several commands together, you can accomplish useful tasks. For example, the steps involved in a simple file transfer are as follows:

- Log in to the server (using the *Login* method)
- Start the transfer (using the *GetFile* or *PutFile* method)
- End the session (using the *Quit* method)



The first example we will design is a simple program to retrieve a file from an FTP server when a button is pressed. In this example, we will write a program to retrieve the file `ftpmenu.htm` from the `/pub` directory on the Borland FTP server. We place a

TMemo, a TButton and a TFTP component on a form. When the button is pressed, we want to initialize the properties of the FTP component (such as what server to log into and what user ID and password to use) and call the Login method. The code for the button OnClick handler looks like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    {specify the password}
    Ftp1.FTPServer:='ftp.borland.com';
    Ftp1.FTPPort:=21;
    {specify login information}
    Ftp1.UserName:='anonymous';
    Ftp1.UserPassword:='startech@neosoft.com';
    Ftp1.UserAccount:='';
    {disable the button while the transaction is in progress}
    Button1.Enabled:=False;
    {now log in}
    Ftp1.Login;
end;
```

Once the component is logged into the FTP server, the *OnLoggedIn* event of the FTP component is triggered. When we receive this event, we need to start the transfer of the desired file. The code for the FTP component's *OnLoggedIn* event looks like this:

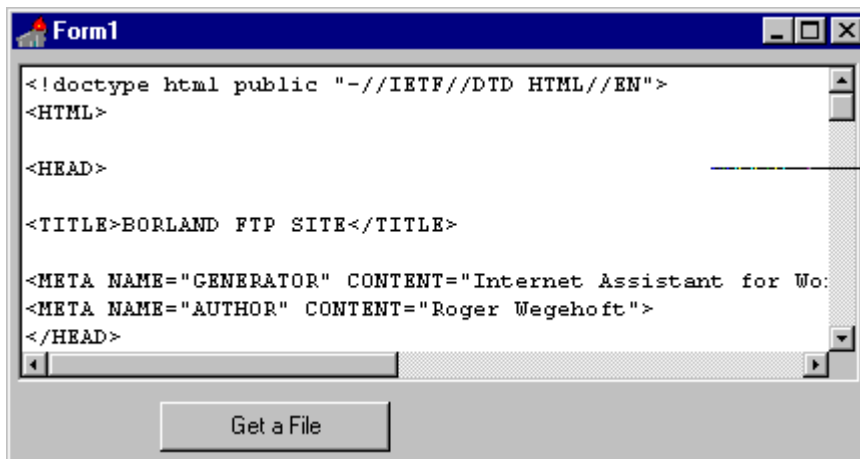
```
procedure TForm1.Ftp1LoggedIn(Sender: TObject);
begin
    {Location of file on ftp server}
    Ftp1.RemoteFile:='/pub/ftpmenu.htm';
    {specify where we want to store the file on the local
system}
    Ftp1.LocalFile:='c:\test.txt';
    {Finally start the transfer}
    Ftp1.GetFile;
end;
```

Once the file has been transferred, we will receive an *OnFileReceived* event. At this point, we want to display the file in the TMemo component and disconnect from the FTP server. The code for the FTP component's *OnFileReceived* event looks like this:

```
procedure TForm1.Ftp1FileReceived(Sender: TObject);
begin
    {load the file into the memo component}
    Memol.Lines.LoadFromFile('c:\test.txt');
    {disconnect}
    Ftp1.Quit;
end;
```

Once we have sent the request to be disconnected, we should wait for the component to indicate that it has been disconnected before enabling the button for another transaction. The *OnFtpQuit* event indicates when the component has been disconnected and is ready for another transaction. The code for the FTP component's *OnFtpQuit* event handler looks like this:

```
procedure TForm1.Ftp1FtpQuit(Sender: TObject);
begin
    Button1.Enabled:=True;    {reenable the button}
end;
```



It can be a little disconcerting to run this program, since there are no indications that the program is doing anything until the TMemo component is loaded. The next section looks at status notification.

Monitoring Transaction Progress

In an interactive FTP program, it is useful to display the current status of the transaction, especially over slow connections, to show if “anything is happening”. The *OnFtpInfo* event is used to report the progress of an FTP transaction. The *OnFTPInfo* event is prototyped as follows:

```
procedure (Sender : TObject; info: FtpInfo; addinfo: string);
```

where *Sender* is the *TFtp* component which triggered the event, *info* is the type of status notification which is being returned and *addinfo* contains additional textual information. The following table lists the possible values for the *info* parameter.

Value	Meaning
ftpServerConnected	A connection to a server has been established.

ftpServerDisconnected	The server has been disconnected from the server.
ftpTraceIn	Diagnostic trace of input received from server.
ftpTraceOut	Diagnostic trace of output sent to server.
ftpFileSize	Notification of file size before transfer
ftpDataTrace	A block of data has been sent or received.
ftpLoggedIn	The component has successfully logged in.
ftpListing	An individual line of the listing has been received
ftpStartListing	The file listing is about to start.
ftpPermissionDenied	The requested command is not permitted
ftpResolvingAddress	The server name is being resolved
ftpAddressResolved	The server name has been resolved into an address.
ftpTransferDone	Obsolete. The file transfer has completed
ftpReady	Obsolete. The last command has completed
ftpDirectoryRefresh	Obsolete. The file listing is ready

Notes:

- When you receive an *OnFTPInfo* event with an *info* value of *ftpServerConnected*, the *addinfo* property will contain the IP address of the server.
- The *ftpTraceIn* and *ftpTraceOut* notifications are very useful for diagnostic output. The *addinfo* parameter contains everything received (*ftpTraceIn*) or sent (*ftpTraceOut*) by the component. This could, for example, be directed to a *TMemo* component for display.

Example:

```

procedure TForm1.Ftp1FtpInfo(Sender: TObject; info: FTPInfo;
addinfo: string);
begin
    case info of
        ftpTraceIn, ftpTraceOut:
            try
                Memo1.Lines.Add(addinfo);
            except
                Memo1.Clear;
                Memo1.Lines.Add(addinfo);
            end;
    end;
end;

```

- The *FTPFileSize* and *FTPDataTrace* notifications are used to monitor the progress of a transaction. The *FTPFileSize* notification is sent to indicate the number of bytes to be transferred, after a call to the *GetFile* or *PutFile* method. The *addinfo* paramter will contain the number of bytes to be transferred. The *FtpDataTrace* notification is sent to indicate that a block of data has been transferred. The *BytesTransferred* property can be used to check how many bytes have been transferred so far. The *FtpDataTrace* notification is sent during a transfer initiated by a call to the *FtpRefresh*, *GetFile* or *PutFile* method. Note that during an *FtpRefresh*, or listing, the number of bytes to be transferred is not available. The *DoingListing* property, if True, indicates that a listing is in progress. If False, a file transfer is in progress. The *TransferTime* property contains the elapsed time for the file transfer, in milliseconds.
- The *ftpStartListing* and the *ftpListing* notifications are used to indicate that a new listing is starting and that an individual line of a listing has been

received. The *addinfo* parameter will contain the individual line of the listing, in the case of an *ftpListing* notification. These two notifications can be used if you want to process the file listing as it is being received, rather than after receiving an *OnListingDone* event at the completion of the listing.

Adding Status Notifications to the Example

We will now modify the previous example to display status information. We first add a TLabel component to the form. We can then use the label's caption to display status information to the user. The first notification we will add is when the user presses the button. The code for the OnClick handler of the button will now look like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Ftp1.FTPServer:='ftp.borland.com';
    Ftp1.FTPPort:=21;
    Ftp1.UserName:='anonymous';
    Ftp1.UserPassword:='startech@neosoft.com';
    Ftp1.UserAccount:='';
    Button1.Enabled:=False;
    {display the current status}
    Label1.Text:='Connecting to '+Ftp1.FtpServer;
    {now log in}
    Ftp1.Login;
end;
```

We will then use the OnFTPInfo event handler to add additional status notifications. The code for the OnFTPInfo event of the FTP component now looks like this:

```
procedure TForm1.Ftp1FtpInfo(Sender: TObject; info: FtpInfo;
    addinfo: string);
begin
    case info of
        ftpServerConnected: Label1.Caption:='Connected to '+addinfo;
        ftpLoggedIn: Label1.Caption:='Logged in';
        ftpFileSize: Label1.Caption:='File Size: '+addinfo;
        ftpDataTrace: Label1.Caption:='Transferring file:' +
            IntToStr(Ftp1.BytesTransferred)+' bytes';
        ftpTransferDone: Label1.Caption := IntToStr(
            Ftp1.BytesTransferred )+ ' transferred in ' + IntToStr(
            Ftp1.TransferTime ) + ' ms';
        ftpServerDisconnected: Label1.Caption:='Disconnected';
    end;
end;
```

Advanced usage

When the FTP protocol was first designed, it was envisioned that client programs would operate on a command line, with the user typing in commands and viewing the output on the console. Therefore, no one saw a need to define standards for the directory listing that an FTP server might return. Instead, listings often match the type of listing generated by the operating system. For example, compare the three listings presented below, generated from a Unix, VMS and VM system respectively.

```
total 42
-rw-rw-r-- 1 root    wheel    627 Sep  7 1995 .Links
-rw-rw-r-- 1 root    wheel    80 Sep  5 1995 .about.html
drwxr-xr-x 4 root    wheel    512 Feb  8 18:24 archive2
drwxr-xr-x 2 root    wheel    512 Jun 19 1995 archive3
drwxrwxr-x 12 root    ftpmaint  512 Feb  5 22:03 archive4
d--x--x--x 2 root    wheel    512 Dec  1 05:50 bin
dr-xr-xr-x 2 root    wheel    512 May 22 1994 dev
dr-xr-xr-x 3 root    wheel    512 Nov 23 1994 etc
d--x--x--x 2 root    wheel    512 Dec 20 18:42 hidden
drwxrwxr-x 2 archiver ftpmaint  512 Dec  7 1994 info
```

A Unix style listing

```
DISK$SHARE:[ANONYMOUS.PUB.WIN3.WINVN]

OLD.DIR;1          2 10-JUN-1993 23:42 [NASA,RUSHING] (RWE,RWED,RWE,RE)
README.TXT;1      25 23-SEP-1994 17:57 [NASA,RUSHING] (RWED,RWED,RE,RE)
UNZ51X.EXE;1     172 23-MAR-1994 17:47 [NASA,RUSHING] (RWED,RWED,RE,RE)
WINVNFAQ.TXT;2    70 8-OCT-1994 21:55 [NASA,DUMOULIN] (RWED,RWED,RE,RE)
WINVN_HAS_MOVED.TXT;1
                    1 7-APR-1995 00:47 [ANONYMOUS] (RWED,RWED,RE,RE)

Total of 270 blocks in 5 files.
```

A VMS-Multinet listing

\$READ-ME	FIRST	V	72	25	1	9/30/93	0:14:04	PUBLIC
RMAC	HELPCMS	V	79	133	2	11/21/88	19:31:03	PUBLIC
RMAC	MODULE	V	18000	3	5	11/21/88	19:19:15	PUBLIC
RMAC	SCRIPT	V	76	172	2	11/21/88	19:29:09	PUBLIC
WMAC	ASSEMBLE	F	80	2480	49	10/06/91	0:31:14	PUBLIC
WMAC	HELPCMS	V	79	189	3	11/21/88	19:52:59	PUBLIC
WMAC	MODULE	V	21976	3	6	5/25/89	0:29:48	PUBLIC
WMAC	SCRIPT	V	76	235	3	11/21/88	19:52:24	PUBLIC
XFER	READ-ME	V	67	39	1	9/25/91	20:48:43	PUBLIC
XMDMGEN	C	V	70	62	1	10/06/91	0:35:29	PUBLIC
XMDMTAB	ASSEMBLE	F	80	46	1	7/14/88	22:44:33	PUBLIC

An IBM-VM listing

As you can see, there are no similarities between the three listings. Most servers, however, support a command (the *SYST* command) which allows you to determine which type of server you are connected to. This command is issued when you login and the result is stored in the *ServerType* property. If the command is not supported, the component will assume that the server is a default FTP server and will treat it as a Unix type server, which represent over 90% of FTP servers in use today. An unfortunate situation still exists with a few servers which do not support the *SYST* command and are not Unix type servers.

The values that the *FTPSTServer* property can return are:

- *ftpstDefault*: the *SYST* command is not supported and *ftpstUnix* is assumed.
- *ftpstUnix*: a Unix style server
- *ftpstClix*: a Clix server (Unix variant)
- *ftpstUltrix*: an Ultrix server (Unix variant)
- *ftpstMVS*: a MVS server
- *ftpstQVT*: a QVT server
- *ftpstNCSA*: an NCSA server
- *ftpstChameleon*: a Chameleon server
- *ftpstVMSMultinet*: a VMS-Multinet server
- *ftpstVMSUCx*: a VMS-UCx server
- *ftpstVM*: an IBM-VM server
- *ftpstVMVPS*: a variation of a VM server

There are other possible server types which will be supported in the future version of the FTP package, but the types supported here represent over 99% of known servers.

The information provided by the *ServerType* property is invaluable in parsing a directory listing. A utility function has also been added to the FTP unit to help you in parsing directory listings. The *ParseListingLine* function is used to parse a line from a directory listing into name, size and date fields and to determine whether the entry represents a file or a directory. Note that the *ParseListingLine* function is a stand-alone function, not a method of the *TFTP* component. The *ParseListingLine* function is prototyped as follows:

```
function ParseListingLine(ServerType: TFTPSType; line:
string; var name, size, date: string; var IsDir: Boolean): Boolean;
```

where the *ServerType* parameter is the type of the server which generated the listing line (such as is returned by the *TFTP*'s component *ServerType* property) and the *line* parameter is a line from the listing. Upon return, the return value if True indicates that information about the name, size and date of a file or directory has been stored in the *name*, *size* and *date* parameters. The *IsDir* parameter also indicates if the *name* parameter represents a directory or a file. If the function returns false, the line passed to the *ParseListingLine* function did not contain any file information. The following example shows how one might handle the *TFTP* component's *OnListingDone* event to display files and directories in separate list boxes.

Example:

```
procedure Form1.FTP1ListingDone(Sender: TObject);
var
    i: integer;
    name, date, size: string;
    IsDir: Boolean;
begin
    FilesListBox.Clear;
    DirsListBox.Clear;
    for i:=0 to FTP1.Listing.Count-1 do
        begin
            if ParseListingLine( Ftp1.ServerType, FTP1.Listing[i],
                name, size, date, IsDir) then
                begin
                    if IsDir then DirsListBox.Items.Add(name)
                    else FilesListBox.Items.Add(name);
                end;
            end;
        end;
end;
```

Note that in this version of the FTP package, the *ParseListingLine* function supports the following servers:

- Unix and variants (Ulrix, Win NT, BSD, etc..)
- Vax/VMS listings (VMS-Multinet, VMS-UCx, etc...)
- IBM-VM listings
- Novell File Server listings

Techniques for Transferring Multiple Files.

There are several ways to transfer multiple files. Only one example is presented here. The basic technique involves keeping two string lists, one containing the fully qualified path and names of the local files, the other containing the same number of entries consisting of the fully qualified path and names of the file of the files on the remote server. The following step by step example how this is accomplished.

The following example shows how one might transfer all the files in a local directory to a directory on an FTP server:

Step 1: Building the list of files

Build a list of all the files to transfer and put it in a *TStringList*. Then build a list of the target files on the FTP server and put it in another *TStringList*.

Your two string lists might look like this:

Item	Local List	Server List
0	c:\html\files\default.htm	/pub/users/jdoe/default.htm
1	c:\html\files\banner.gif	/pub/users/jdoe/banner.gif
2	c:\html\files\me.gif	/pub/users/jdoe/me.gif
3	c:\html\files\page2.htm	/pub/users/jdoe/page2.htm
4	c:\html\files\page2girl.jpg	/pub/users/jdoe/page2girl.jpg

When you generate the list of files on the FTP server and the local file systems, keep in mind the differences of file naming conventions between the two systems. For example, spaces in file names are acceptable in Windows 95 or Windows NT, but are not acceptable in Unix. You will therefore want to change the spaces in a file name to underscore characters or other valid characters when storing the file on the FTP server.

Step 2: Transferring the first file

Upload the first file. Note that after we set up the FTP component, we delete the first file from the local and remote list. For example:

```
Ftp1.LocalFile:=LocalList[0];  
Ftp1.RemoteFile:=RemoteList[0];  
LocalList.Delete(0);  
RemoteList.Delete(0);  
Ftp1.PutFile;
```

Step 3: Transferring the next file

Handle the *OnFileStored* event and download the next file, if there are more files to download. For example:

```
procedure Form1.Ftp1FileStored(Sender: TObject);  
begin  
    if Ftp1.Success then  
    begin  
        if LocalList.Count>0 then  
        begin  
            Ftp1.LocalFile:=LocalList[0];  
            Ftp1.RemoteFile:=RemoteList[0];
```

```
        LocalList.Delete(0);
        RemoteList.Delete(0);
        Ftpl.PutFile;
        Exit;
    end;
end;
Quit; {if we get here, we are done, quit}
end;
```

Techniques for Transferring Multiple Directories

The technique for transferring multiple directories is similar to the technique for transferring multiple directories. You will need two additional string lists, for storing the directory names of the local and remote system. You would start by processing the directory listing in the following fashion:

- Build a list of files to transfer, as was done when transferring multiple files
- Build a list of subdirectories to process later.

You would then transfer the first file and handle the *OnFileReceived* or *OnFileStored* event as described above. The difference is that when you run out of files, you will process the first directory in the following fashion:

- Build a list of files to transfer, as was done when transferring multiple files
- Build a list of subdirectories to process later.
- Create the target directory on the FTP server or the local file system.
- Download the first file.

You will repeat this procedure until you run out files and directories to process. This algorithm is used to download directories recursively in the *StarFTP* example program which is included with source code in the FTP package.

The TFTP Component in URL Mode

If you want to avoid the burden of handling events, the *TFTP* component can be used in URL mode to download or upload single files, by specifying the URL of a file to transfer. A Uniform Resource Location, or a URL, is a string which uniquely identifies a resource on the Internet. The following shows examples of URLs:

http://www.borland.com ftp://www.neosoft.com/pub/users/s/startech/dl510.zip mailto: startech@neosoft.com
--

An FTP URL specifier consists of the string 'ftp://', followed by the server name, followed by the full path of the file to be transferred. For example, the URL 'ftp://www.neosoft.com/pub/users/s/startech/dl510.zip' specifies that to access this resource, you would use FTP to connect to www.neosoft.com and transfer file 'pub/users/s/startech/dl510.zip'.

Initializing the FTP component for URL mode.

Prior to starting a transfer with the FTP component in URL mode, you must specify the name of the local file you will be working with. This file will either be uploaded to the location specified by the URL, or will be used to store the file

referenced by the URL. You will specify the name of the local file using the *LocalFile* property. You must specify the URL that you will be working with, using the *Ftp_URL* property.

Example:

```
Ftp1.LocalFile:='c:\myfile.htm';  
Ftp1.Ftp_URL:='ftp://www.neosoft.com/pub/users/s/startech/de  
fault.htm';
```

Starting the File Transfer

To start a file transfer with the FTP component in URL mode, you will use the *GetURL* method to download a file or the *PutURL* method to upload a file. You can also use the *ListURL* method to store a listing in a file, provided that the URL in the *Ftp_URL* property refers to a directory and not to a file.

Example:

```
Ftp1.GetURL;  
or  
Ftp1.PutURL;  
or  
Ftp1.ListURL;
```

Once the file transfer is initiated, the following will happen:

- The component will connect to the server.
- The component will login anonymously.
- The component will transfer the file.
- Finally, the component will log off the server and disconnect.

Monitoring Transaction Progress and Completion

The *TFTP* component will generate an *OnFTPQuit* event once a transaction in URL mode has completed. You can also monitor the progress of the file transfer by handling the *OnFTPInfo* event, as was done in interactive mode.

Handling Errors

Unlike transactions in interactive mode where some errors are non-fatal, all errors in URL mode are fatal. To determine if an error occurred during the transaction, simply check the *FTPError* property when you handle the *OnFTPQuit* event. A value of *ftpNone* indicates that an error did not occur. Other possible values are listed below with their meaning:

Value	Meaning
ftpServerDown	The server is down and is not accepting logins.
ftpInvalidServer	The server alias specified in FTPServer could not be resolved.
ftpAlreadyBusy	A command was issued before the previous command completed.
ftpNoWinsock	The winsock DLL could not be loaded.
ftpNoServer	No server specified in FtpServer.
ftpSocketError	The FTP server refused the connection
ftpReadError	A fatal error occurred while reading the connection
ftpWriteError	A fatal error occurred while writing to the connection
ftpProtocolError	A basic FTP command was not recognized by the FTP server
ftpAccessDenied	Access to the server was denied at login
ftpCancelled	The transfer was cancelled with a <i>StopTransfer</i> call.
ftpFileOpen	The file specified in <i>LocalFile</i> could not be opened.
ftpFileWrite	An error occurred while writing to <i>LocalFile</i> (such as disk full)
ftpFileRead	An error occurred while reading from <i>LocalFile</i>
ftpFileNotFound	The file specified as a parameter or in <i>RemoteFile</i> was not found
ftpPermissionDenied	The requested command is not permitted

Other methods, properties and events of the FTP component

The FTP command supports other less frequently used methods and properties, which are described in the following sections:

The IssueCommand method

The *IssueCommand* method can be used in interactive mode to issue commands not supported by the component. The *IssueCommand* method is prototyped as follows:

```
procedure IssueCommand(command: string);
```

where the *command* parameter is the command to send to the FTP server.

Example:

```
FTP1.IssueCommand('SITE');
```

Note that it is the responsibility of the programmer to handle the response from the server, which can be done by using the *OnFtpInfo* event with an *info* parameter value of *ftpTraceIn*.

The StopTransfer method

The *StopTransfer* method can be used in interactive or URL mode to stop a file transfer in progress. In URL mode, the component will disconnect and the *FTPError* property will contain the value *ftpCancelled*. In interactive mode, the component will disconnect the data connection to interrupt the file transfer and will generate a non fatal *ftpCancelled* error. Note that although the FTP component treats this error as non fatal, note that most FTP servers will react negatively to the data channel closing prematurely while downloading a file and will close the command connection, causing the component to disconnect.

Example:

```
FTP1.StopTransfer;
```


The TransferMode property

The TransferMode property can be used to specify which transfer mode to use when transferring files, either *image mode* (more commonly known as binary mode) or *text mode*. In *image mode*, the file transferred is identical byte for byte with the original. In *text mode*, the line ends are adjusted to account for the different conventions of Unix and other systems. You should set the TransferMode property prior to starting a transfer with a *GetFile*, *GetURL*, *PutFile* or *PutURL* method. Listings are automatically downloaded in *text mode*.

Example:

```
FTP1.TransferMode:=BinaryTransfer;    {image mode}
```

or

```
FTP1.TransferMode:=AsciiTransfer;    {text mode}
```

The WinsockStarted property

The WinsockStarted property can be used to detect the presence of a Winsock Dynamic Link Library on the system, which is required for the component to function. While it can be assumed that a 32 bit Windows OS supports Winsock, the same cannot be said about a 16 bit Windows OS. To check for the presence of a Winsock DLL, first set the WinsockStarted property to *True*. Then check that the WinsockStarted property is still *True*. This allows you to take corrective action should your program be running on a platform with no Winsock DLL. Such corrective actions may include disabling the FTP features if they are not essential to the program or informing the user of the problem and exiting gracefully.

Example:

```
Ftp1.WinsockStarted:=True;
if not Ftp1.WinsockStarted then
begin
    {take corrective actions}
    ShowMessage('Winsock is not installed on this system.'
        +' The Upload Web Site feature will be disabled.');
```

```
    UploadMenu1.Enabled:=False;
end;
```

The Connected property

The *Connected* property is *True* when the component is connected to an FTP server and *False* otherwise.

The OnFTPNeedInfo event

It is possible to start an interactive session with the FTP component without first specifying the *FTPServer*, *UserName*, *UserPassword* and *UserAccount* properties. In this case, the *OnFTPNeedInfo* event will be triggered when the information is needed. The *OnFTPNeedInfo* event is prototyped as follows:

```
procedure (Sender:TObject;need:TftpInfoNeeded;var value:string);
type TftpInfoNeeded=(Host,User>Password,Account);
```

where the *Sender* parameter is the FTP component which triggered the event and the *need* parameter is the type of information needed. The *value* parameter should be set to the needed info prior to exiting the event handler. The *need* parameter is one of:

- Host: specify the FTP server name or address in the *value* parameter.
- User: specify the user name in the *value* parameter.
- Password: specify the user password in the *value* parameter.
- Account: specify the user account in the *value* parameter.

The following example shows how one might handle the *OnFTPNeedInfo* event, by displaying an input box and passing the value entered in the *value* parameter.

Example:

```
procedure Form1.Ftp1FTPNeedInfo(Sender: TObject; need:
TFTPInfoNeeded; var value: string);
var
    s: string;
begin
    case need of
        Host: s:='server alias or address';
        User: s:='user ID';
        Password: s:='user password';
        Account: s:='user account';
    end;
    value:=InputBox('Need Login Info','Please enter the '+s,'');
end;
```

The FTPURLDialog Component

The FTPURLDialog component allows you to retrieve a file on an FTP server, or store a file to an FTP server, with a minimum of user interaction.

Initializing the FTPURLDialog component

Before starting the transfer, there are several properties which must be initialized, which tell the component which file to download or upload for example.

The LocalFile and URL properties

Prior to starting a transfer with the FTPURLDialog component, you must specify the name of the local file you will be working with. This file will either be uploaded to the location specified by the URL, or will be used to store the file referenced by the URL. You will specify the name of the local file using the *LocalFile* property. You must specify the URL that you will be working with, using the *URL* property.

Example:

```
FtpURLDialog1.LocalFile:='c:\myfile.htm';
FtpURLDialog1.URL:='ftp://www.neosoft.com/pub/users/s/startech/default.htm';
```

Specifying the direction and type of the transfer

You will specify the direction of the transfer with the *Action* property. Set *Action* to *fd_GetURL* to transfer the file specified by the URL property to the file specified by the LocalFile property. Set *Action* to *fd_PutURL* to transfer the file specified by the LocalFile property to the file specified by the URL property.

Example:

```
FtpURLDialog1.Action:=fd_GetURL; {download a file}
```

or

```
FtpURLDialog1.Action:=fd_PutURL; {upload a file}
```

You will specify the type of transfer with the *TransferType* property. Set *TransferType* to *BinaryTransfer* to transfer binary files, or to *AsciiTransfer* to transfer text files.

Specifying Login information

If you want to upload a file to a private directory or access a private FTP server, you must specify the user ID and user password using the *UserID* and *UserPass* properties. If the *UserID* and *UserPass* properties are not set, you will log in anonymously.

Example:

```
FTPURLDialog1.UserID:='jsmith';  
FTPURLDialog1.UserPass:='mypassword';
```

Starting the File Transfer



Once you have set the *LocalFile*, *URL*, *Action* and optionally the *TransferType*, *UserPass* and *UserID* properties, you can initiate the file transfer by calling the *Execute* method. A dialog box will

popup, with a status bar, showing the progress of the transfer. If *Execute* returns *False*, then the transfer did not succeed and you can check the *Error* property to determine the cause of the failure. At the end of the transfer, the *Execute* method returns *true* if the file transfer succeeded, *false* if the file transfer failed, in which case you can determine the exact cause of the failure by reading the *Error* property.

Transaction Completion

When the transaction completes, the *Execute* method returns *true* if the file transfer succeeded or *false* if the file transfer failed. The *Error* property yields additional information about what type of error occurred. The following table lists the possible values of the *Error* property and their meaning.

Value	Meaning
ftpServerDown	The server is down and is not accepting logins.
ftpInvalidServer	The server alias specified in FTPServer could not be resolved.
ftpAlreadyBusy	A command was issued before the previous command completed.
ftpNoWinsock	The winsock DLL could not be loaded.
ftpNoServer	No server specified in FtpServer.
ftpSocketError	The FTP server refused the connection
ftpReadError	A fatal error occurred while reading the connection
ftpWriteError	A fatal error occurred while writing to the connection
ftpProtocolError	A basic FTP command was not recognized by the FTP server
ftpAccessDenied	Access to the server was denied at login
ftpCancelled	The transfer was cancelled with a <i>StopTransfer</i> call.
ftpFileOpen	The file specified in <i>LocalFile</i> could not be opened.
ftpFileWrite	An error occurred while writing to <i>LocalFile</i> (such as disk full)
ftpFileRead	An error occurred while reading from <i>LocalFile</i>
ftpFileNotFound	The file specified as a parameter or in <i>RemoteFile</i> was not found
ftpPermissionDenied	The requested command is not permitted

A sample FTPURLDialog program

The following code snippet is the code for a button's *OnClick* event handler, which performs a file transfer:

```
procedure Form1.Button1Click(Sender: TObject);
begin
    FtpURLDialog1.LocalFile:='c:\myfile.htm';
    FtpURLDialog1.URL:='ftp://www.acme.com/users/jsmith/default.
htm';
    FTPURLDialog1.Action:=fd_PutURL;
    FTPURLDialog1.TransferType:=BinaryTransfer;
    FTPURLDialog1.UserID:='jsmith';
    FTPURLDialog1.Password:='mysecret';
    if not FTPURLDialog1.Execute then
        ShowMessage('Transfer Failed!');
end;
```

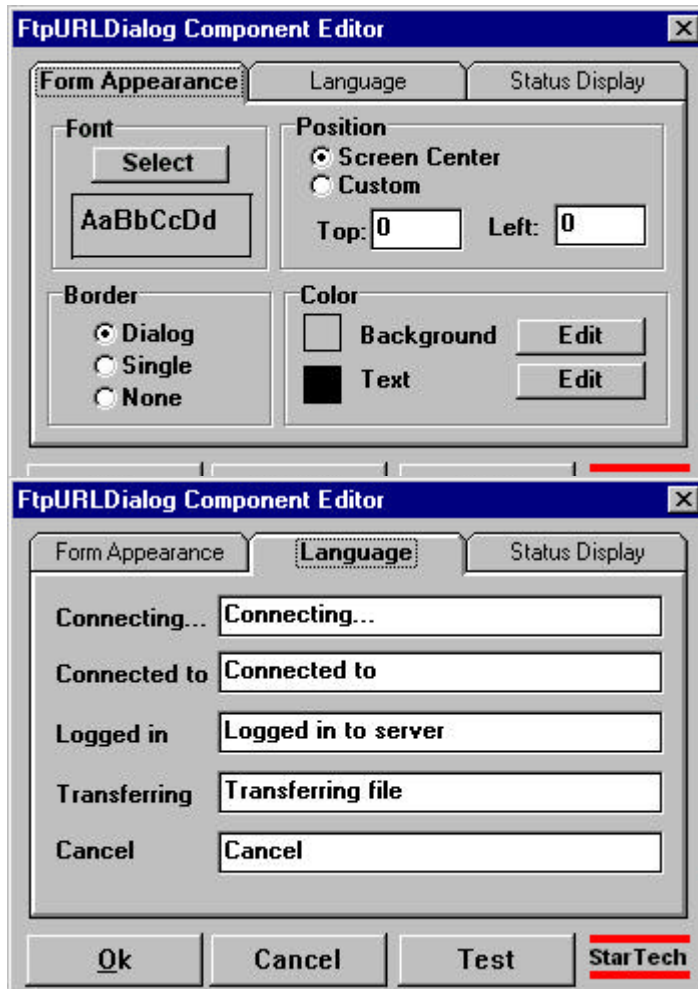
As you can see, the *FTPURLDialog* component greatly simplifies the task of FTP file transfers and minimizes the amount of code needed.

Customizing the appearance of the FTPURLDialog Component

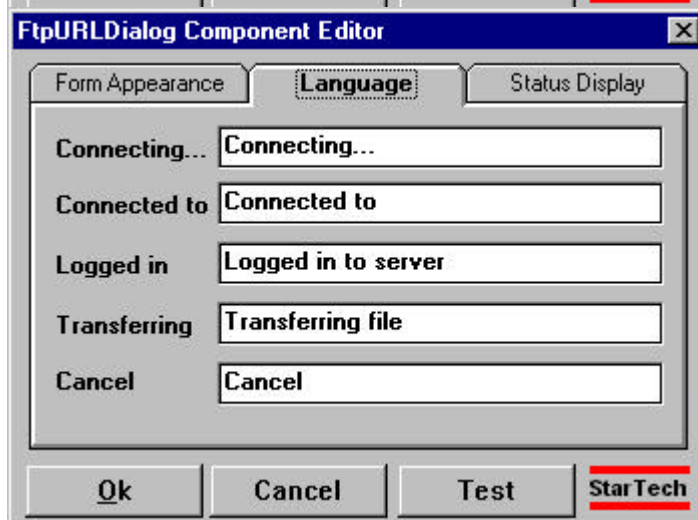
You can customize the appearance, placement and strings that are used by the *FTPURLDialog* component. This can be done either with the component editor or programmatically.

Using the component editor to customize the FTPURLDialog component

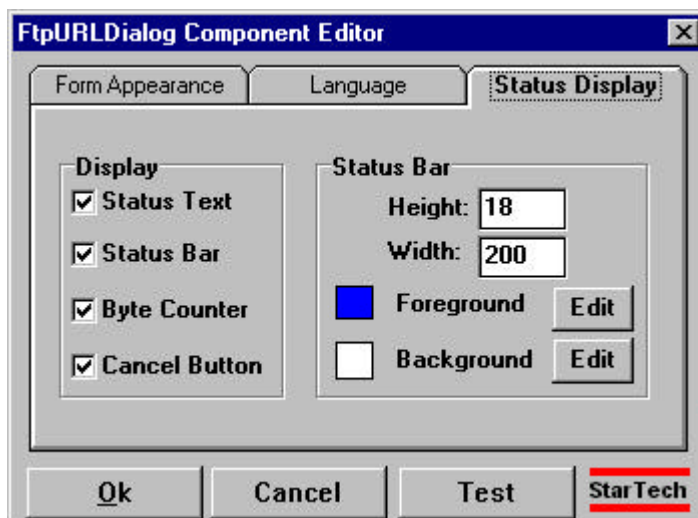
You can access the component editor in design mode by double clicking the component or by highlighting the component, right clicking and choosing *Edit* from the pop up menu.



The first page of the component editor lets you specify the font to be used for the text messages, the color of the background and the text, the position of the window and the border used by the window.

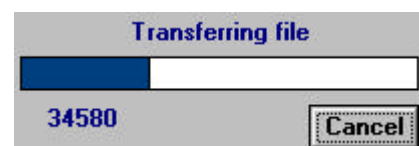


The second page of the component editor lets you specify the strings that will be used to display completion status to the user. This allows you to customize the component for foreign language applications.



The third page of the component editor lets you specify which elements of the status display to use, the size of the status bar and the colors used for the status bar.

Once you have modified the desired properties, you can press the Test button to preview the appearance of the control. The



FTPURLDialog component displayed on the right is an example of a component with no borders, a different font and all four elements of the status display in use. Note that displaying the FTPURLDialog with no border is useful if you want to give the impression that the status display is part of another form rather than a dialog box.

Modifying the FTPURLDialog Programmatically

If you want to modify the appearance of the control at run time, you can use the following properties:

- The *Border* property can be set to one of the following values:
 - *bsNone* for no border
 - *bsDialog* for a dialog box border
 - *bsSingle* for a single line border
- The *Color* property can be used to set the background color of the dialog box.
- The *Font* property can be used to set the font used by the component for status display.
- The *StatusBarColor* and *StatusBarBackground* properties can be used to set the foreground and background color of the status bar.
- The *StatusBarHeight* and *StatusBarWidth* properties can be used to set the size of the status bar.
- The *Position*, *WindowLeft* and *WindowTop* properties can be used to set the position of the dialog box.
- You can select which elements of the status display will appear in the dialog box by adding one or more of the following values to the *Options* set property:
 - *StatusText*: status text is displayed at the top of the window.
 - *StatusBar*: a progress indication bar is displayed in the middle of the window.
 - *ByteCounter*: a line/byte counter is displayed at the bottom of the status window.
 - *CancelButton*: a cancel button is displayed at the bottom of the status window.
- You can specify the text of the status display strings by using the *LanguageStrings* property, of type *TStrings*. The *LanguageStrings* property for the *FTPURLDialog* component consists of five strings, which must be in the exact order specified below:
 - Connecting... string
 - Connected to... string
 - Logged in string
 - Transferring file string
 - Cancel button caption

Additional properties of the FTPURLDialog Component

The *Caption* property is used to specify the caption of the status display window. Note that if the window has no border, it will not have a title bar and it is therefore not necessary to provide a caption.

The *TimeOut* property is used to specify a timeout value for the FTP transaction. The timeout period is specified in milliseconds. For example, to specify a timeout value of 30 seconds, you will set the *TimeOut* property to 30000. If nothing is received within the time specified in the *TimeOut* property, the file transfer will be aborted and the *Execute* method will return *False*. In this case, the *Error* property should contain the value *ftpTimeOut*.

THE HTTP PACKAGE

The THttp Component

This chapter has yet to be written, the following list of public methods, properties and events is included as minimal help.

```
const
    HTTP_SIZEUNKNOWN=-1;

type THttpInfo=(httpConnecting, httpConnected, httpSocketError,
httpInvalidServer,httpTraceIn, httpTraceOut, httpReadError,
httpWriteError, httpDisconnected, httpHeadersLoaded, httpCancelled
);
type THttpMethod=(httpGet,httpPost);
type THttpVersion=(HTTP_1_0,HTTP_0_9);

type THttpInfoEvt=procedure(Sender: TObject;info:
THttpInfo;addinfo: string) of object;
type THttpOutputEvt=procedure(Sender: TObject;data: string) of
object;

type THttp = class(TStarSock)
public
    pURL: PChar;
    ResultDate,ResultContentType: string;
    ResultStatus,ResultSize: LongInt;
    procedure Get;
published
    property Proxy: string read FProxy write FProxy;
    property ProxyPort: integer read FProxyPort write
FProxyPort;
    property UseProxy: Boolean read FUseProxy write FUseProxy;
    property URLEdit: TEdit read FUrLEdit write FURLEdit;
    property OnHttpInfo: THttpInfoEvt read FHttpInfoEvent
write FHttpInfoEvent;
    property OnHttpOutput: THttpOutputEvt read
FHttpOutputEvent write FHttpOutputEvent;
    property WinsockStarted;
    property Version;
    property HttpVersion: THttpVersion read FHttpVersion write
FHttpVersion;
    property ResultHeaders: TStrings read FHeaders write
SetHeaders;
    property URL: string read FURL write FURL;
end;

type EInvalidHttpURL=class(Exception);
```

The TStarImage Component

This section has yet to be written, the following list of public methods, properties and events is included as minimal help.

```
type TStarImage=class(TCustomControl)
published
    property Filename: TFilename read FFileName write
SetFilename;
```

end;

THE IRC PACKAGE

This chapter has yet to be written, the following list of public methods, properties and events is included as minimal help.

```

type TIRCChannel=record
    Name: string[16];
    Topic: string[100];
    Members: TStrings;
    Limit: string[10];
    Mode: string[20];
    InUse: Boolean;
end;

type TIRCCChannelList=record
    Name: PChar;
    Topic: PChar;
    NumUsers: integer;
end;

type
    TIRC = class(TStarSock)
    private
    public
        property Channels[ChannelNum: Integer]: TIRCCChannel read
            GetChannel;
        property ChannelList[Num: Integer]: TIRCCChannelList read
            FGetChannelList;
        function Connect: Boolean;
        procedure Quit(QuitMessage: string);
        function JoinChannel(channel,password: string): integer;
        procedure WriteChannelMessage(ChannelNum: integer;Msg:
            string);
        procedure WritePrivateMessage(Nickname, Msg: string);
        procedure QueryMode(ChannelNum: integer);
        procedure QueryNames(ChannelNum: integer);
        procedure LeaveChannel(ChannelNum: integer;msg: string);
        procedure
            SetChannelMode(ChannelNum:integer;mode,parameter: string);
        procedure SetChannelTopic(ChannelNum: integer;Topic:
            string);
        procedure InviteToChannel(ChannelNum: integer;Nickname:
            string);
        procedure KickFromChannel(ChannelNum:
            integer;Nickname,Comment: string);
        procedure FindWho(Mask: string);
        procedure Write(line: string);
        procedure GetChannelList;
        procedure GetCTCPInfo(Nickname: string;InfoType:
            TCTCPInfo);
    published
        property IRCServer:string read FServer write SetServer;
        property IRCPort:Word read FPort write FPort;
        property UserNickname: string read FNick write SetNick;
        property UserRealName: string read FRealName write
            FRealName;
        property OnIRCInfo: TIRCInfoEvent read FOnInfo write
            FOnInfo;
    end;
end;

```

```

        property OnServerMessage: TServerMessageEvent read
FServerMessage write FServerMessage;
        property OnPrivateMessage: TPrivateMessageEvent read
FPrivateMessage write FPrivateMessage;
        property OnChannelMessage: TChannelMessageEvent read
FChannelMessage write FChannelMessage;
        property OnChannelEnter: TChannelEnterEvent read
FChannelEnter write FChannelEnter;
        property OnChannelLeave: TChannelEvent read FChannelLeave
write FChannelLeave;
        property OnChannelCreate: TChannelEvent read
FChannelCreate write FChannelCreate;
        property OnChannelDestroy: TChannelInfoEvent read
FChannelDestroy write FChannelDestroy;
        property OnChannelTopic: TChannelMessageEvent read
FChannelTopic write FChannelTopic;
        property OnChannelMembersLoaded: TChannelInfoEvent read
FChannelMembers write FChannelMembers;
        property OnChannelModeChange: TChannelMessageEvent read
FCModeSet write FCModeSet;
        property OnChannelsListDone: TNotifyEvent read
FChannelListDone write FChannelListDone;
        property OnChannelsListUpdating: TNotifyEvent read
FChannelsUpdating write FChannelsUpdating;
        property OnFindResult: TFindResultEvent read FResultEvent
write FResultEvent;
        property Connected: Boolean read FConnected write
DummyBoolean;
        property NumChannels: integer read FNumChannels write
FNumChannels;
        property OnCTCPInfoRequest: TCTCPEvent read FCTCPRequest
write FCTCPRequest;
        property OnCTCPInfoResult: TCTCPEvent read FCTCPResult
write FCTCPResult;
    end;

```

TTCPCLIENT AND TTCPSERVER

This chapter has yet to be written, the following list of public methods, properties and events is included as minimal help.

```
TTCPClient=class(TStarSock)
protected
    constructor Create(AOwner: TComponent);override;
published
    property Handle;
    property Server: string read FServer write SetServer;
    property Host: string read FHost;
    property Address: string read FAddress;
    property Port: u_short read FPort write FPort;
    property OnConnect: TNotifyEvent read FOnConnect write
FOnConnect;
    property OnDisconnect: TNotifyEvent read FOnDisconnect
write FOnDisconnect;
    property OnRead: TNotifyEvent read FOnRead write FOnRead;
    property OnWrite: TNotifyEvent read FOnWrite write
FOnWrite;
    property WinsockStarted;
end;

TTCPServer=class(TServSock)
protected
    constructor Create(AOwner: TComponent);override;
published
    property Handle;
    property Server:string read FServer write SetServer;
    property Host: string read FHost;
    property Address: string read FAddress;
    property Port: u_short read FPort write FPort;
    property OnAccept: TServerEvent read FOnAccept write
FOnAccept;
    property OnDisconnect: TServerEvent read FOnDisconnect
write FOnDisconnect;
    property OnRead: TServerEvent read FOnRead write FOnRead;
    property OnWrite: TServerEvent read FOnWrite write
FOnWrite;
    property WinsockStarted;
end;
```

****TTCPClient inherits from TStarSock:

```
TStarSock = class(TComponent)
public
    procedure Close;
    procedure Open;
    property WinsockStarted: Boolean read ISStarted write
TestStarted;
    function GetBytesSent : integer;
    function RecvText : string;
    procedure SendText(const s : string);
    function SendBuffer(buf: PChar; cnt : integer) : integer;
    procedure SetServer(server: string);
    property Conn : TStSocket read FConn;
    function GetLocalHost: string;
```

```

        function GetLocalAddress: string;
        function GetLocalPort: integer;
        constructor Create(AOwner : TComponent);override;
        destructor Destroy;override;
end;

```

****TTCPServer inherits from TServSock:

```

TServSock = class(TComponent)
public
    procedure Close;
    procedure Listen;
    function GetLocalHost: string;
    function GetLocalAddress: string;
    function GetLocalPort: integer;
    procedure SetServer(server: string);
    function DoAccept: integer;
    function GetBytesSent : integer;
    function RecvText : string;
    procedure SendText(const s : string);
    function GetClient(cid : integer) : TStSocket;
    function ClientGetBytesSent(cid: integer) : integer;
    function ClientRecvText(cid: integer) : string;
    function ClientRecvBuffer(cid: integer;buf: PChar; cnt :
integer) : integer;
    function ClientSendBuffer(cid: integer;buf: PChar;cnt:
integer): integer;
    procedure ClientSendText(cid: integer;const s : string);
    procedure ClientClose(cid: integer);
    constructor Create(AOwner : TComponent);override;
    destructor Destroy;override;
end;

```

APPENDIX A - INTERNET RFC

This section contains information about the Internet RFCs (standards documents) that apply to the various protocols employed by the StarTech components.

The SMTP Protocol

The SMTP protocol is used by the components in the *SendMail* package to send mail. It is described in RFC 821, which can be obtained from:

<http://www.cis.ohio-state.edu/htbin/rfc/rfc821.html>

The POP3 Protocol

The POP3 protocol is used by the components in the *GetMail* package to retrieve mail from POP3 servers. It is described in RFC 1725, which can be obtained from:

<http://www.cis.ohio-state.edu/htbin/rfc/rfc1725.html>

THE MIME Protocol

The MIME protocol is used to format multimedia content for transfer over the Internet. It is described in RFC 1521, which can be obtained from:

<http://www.cis.ohio-state.edu/htbin/rfc/rfc1521.html>

The FTP Protocol

The FTP protocol is used by the components of the FTP package to transfer files. It is described in RFC 959, which can be obtained from:

<http://www.cis.ohio-state.edu/htbin/rfc/rfc959.html>

Must Read if you are Designing a Mail Client

RFC 1824, "Multimedia E-mail (MIME) User Agent checklist", provides a mechanism for testing your MIME compatible mail client. It includes the procedure for contacting an auto responder which will send you various types of MIME formatted messages.

APPENDIX B - ADDITIONAL MATERIAL

Registered MIME Types

The following document describes registered MIME types, which are discussed in the *GetMail* section. For the most updated listing and detailed information about each type, see:

<ftp://venera.isi.edu/in-notes/iana/assignments/media-types/>

Type	Subtype	Description	Reference
----	-----	-----	-----
text	plain		[RFC1521,Borenstein]
	richtext		[RFC1521,Borenstein]
	enriched		[RFC1896]
	tab-separated-values		[Paul Lindner]
	html		[RFC1866]
	sgml		[RFC1874]
multipart	mixed		[RFC1521,Borenstein]
	alternative		[RFC1521,Borenstein]
	digest		[RFC1521,Borenstein]
	parallel		[RFC1521,Borenstein]
	appledouble		[MacMime,Patrik Faltstrom]
	header-set		[Dave Crocker]
	form-data		[RFC1867]
	related		[RFC1872]
	report		[RFC1892]
	voice-message		[RFC1911]
message	rfc822		[RFC1521,Borenstein]
	partial		[RFC1521,Borenstein]
	external-body		[RFC1521,Borenstein]
	news		[RFC 1036, Henry Spencer]
application	octet-stream		[RFC1521,Borenstein]
	postscript		[RFC1521,Borenstein]
	oda		[RFC1521,Borenstein]
	atomicmail		[atomicmail,Borenstein]
	andrew-inset		[andrew-inset,Borenstein]
	slate		[slate,terry crowley]
	wita	[Wang Info Transfer,Larry Campbell]	
	dec-dx	[Digital Doc Trans, Larry Campbell]	
	dca-rft	[IBM Doc Content Arch, Larry Campbell]	
	activemessage		[Ehud Shapiro]
	rtf		[Paul Lindner]
	applefile		[MacMime,Patrik Faltstrom]
	mac-binhex40		[MacMime,Patrik Faltstrom]
	news-message-id		[RFC1036, Henry Spencer]
	news-transmission		[RFC1036, Henry Spencer]
	wordperfect5.1		[Paul Lindner]
	pdf		[Paul Lindner]
	zip		[Paul Lindner]
	macwriteii		[Paul Lindner]
	msword		[Paul Lindner]
	remote-printing		[RFC1486,Rose]
	mathematica		[Van Nostern]
	cybercash		[Eastlake]
	commonground		[Glazer]
	iges		[Parks]
	riscos		[Smith]
	eshop		[Katz]
	x400-bp		[RFC1494]
	sgml		[RFC1874]
	cals-1840		[RFC1895]
	vnd.framemaker		[Wexler]
	vnd.mif		[Wexler]
	vnd.ms-excel		[Gill]
	vnd.ms-powerpoint		[Gill]

	vnd.ms-project	[Gill]
	vnd.ms-works	[Gill]
	vnd.ms-tnef	[Gill]
	vnd.svd	[Becker]
image	jpeg	[RFC1521,Borenstein]
	gif	[RFC1521,Borenstein]
	ief	Image Exchange Format [RFC1314]
	g3fax	[RFC1494]
	tiff	Tag Image File Format [Rose]
	cgm	Computer Graphics Metafile [Francis]
	naplps	[Ferber]
	vnd.dwg	[Moline]
	vnd.svf	[Moline]
	vnd.dxf	[Moline]
audio	basic	[RFC1521,Borenstein]
	32kadpcm	[RFC1911]
video	mpeg	[RFC1521,Borenstein]
	quicktime	[Paul Lindner]
	vnd.vivo	[Wolfe]