J-Write
Component Library

J-Write Component Library User Manual

Issue 1.01

MWA Software
P.O. Box 37
Alresford
SO24 9ZF
England

Fax: (UK) 01962 735581
(Int.) +44 1962 735581

EMail: sales@mwassocs.demon.co.uk

WWW: http://www.demon.co.uk/mwa-soft

List of Contents

1Introduction

Welcome to the J-Write Component Library. The J-Write components are a powerful add-on to the Delphi Visual Component Library, and provide a comprehensive set of text viewing and editing classes. This User Manual describes:

- How to install or remove the J-Write Component Library
- The Architecture of the Component Library, and
    - The Example Application provided with the Component Library.

A reference manual for all publicly declared Properties, Events and Methods, is also provided. It is assumed that the reader is familiar with Borland Delphi and the use of a DOS/Windows computer.

The J-Write components have been designed to provide a powerful set of Delphi components to support the viewing, printing and editing of text files. Such files occur everywhere on most modern computer systems. They may contain network logs, configuration information, or technical papers, such as Internet RFCs.

Speed and effectively unlimited file sizes are major design goals of the J-Write Components. The J-Write components can access any file whose size may be expressed within a 32-bit signed number and. moreover, the time taken to move around a file and to edit it is independent of the size of the file. Fast search and replace functions complete the basic function set.

Word wrap is comprehensively supported, with four separate word wrap algorithms (on line breaks only, on window boundaries, on printer page width and after a set number of characters). Tab stops can also be set (in logical inches). The components can also recognise hard page breaks and calculate where soft page breaks occur. In addition, any ANSI font (including True Type) can be used.

There is full keyboard and Mouse support. All Edit Keys are configurable by using the Delphi Object Inspector, and the Mouse may be used to select text, and to drag and drop selected text to any other part of a file.

Text files are now commonly used for EMail, and the J-Write Components include support for the features necessary when processing EMail. There are uuEncode and Decode engines, and text merging facilities including support for prefixing every merged in line with a common text prefix - this may be used to quote from earlier EMails. There is even support for calling an external DOS compression utility in order to compress a file prior to it being uuencoded and merged into a file being edited.

If you are evaluating the J-Write Component Library then we hope that it proves useful to you. Remember that the source code of all the components is also available.

If you are a registered user of J-Write, then thank you for choosing J-Write. We hope that you will find it a useful set of components in your Delphi applications.

2Installing and Removing the Component Library

2.1Installation

The following instructions describe how to install the J-Write components and help file into the Delphi Desktop. Note that the procedures are slightly different depending on whether you are using the shareware version or the registered version that was installed by J-Write.

If you are installing the Shareware version of the J-Write Component Library:

The Shareware version of the J-Write component library is supplied in a single (.zip) archive. This must first be expanded into a directory created to hold the J-Write components. For example, you may have installed Delphi in the directory `C:\DELPHI` and you wish to keep the J-Write components in a subdirectory (JWRITE) of the Delphi directory. You should therefore start by creating the directory `C:\DELPHI\JWRITE` and then expand the J-Write archive into that directory. Note that you should preserve the archived directory structure when expanding the archive. For example, if you are using pkunzip to expand the archive, then you should using the “-d” switch when running pkunzip. Assuming that the J-Write archive is in the root directory of the C: drive, the following DOS script may be used to expand the J-Write components:

```
C:
cd \DELPHI
md JWRITE
cd JWRITE
pkunzip -d C:\JCMPT010.ZIP
```

You may now proceed to install the J-Write components under Delphi, as follows:

1. Backup your 'COMPLIB.DCL' file. You will find this file in the 'BIN' subdirectory, in your Delphi directory. For example, if you have installed Delphi in "`C:\DELPHI`", then you will need to take a backup of `'C:\DELPHI\BIN\COMPLIB.DCL'`. If for any reason, the installation fails, you will need to restore a working version COMPLIB.DCL from this backup.

2. Start Delphi and select "Install Components" from the "Options" menu. The "Install Components" Dialog box will now appear.

3. Append the path of the J-Write components directory to the "Search Path". You will need to separate this path name from the existing search path by a semi-colon. For example, if the J-Write components were installed in `C:\DELPHI\JWRITE`, then add ";`C:\DELPHI\JWRITE`" to the end of the existing search path.

4. Click on the "Add" button, and browse for the J-Write components directory. Select the "`FILEVIEW.DCU`" file.

5. Repeat the "Add" operation and select the "`MAILEDIT.DCU`" file.

6. Click on "OK". The component library should now be rebuilt. Once this is complete, then you will have three new components on the standard palette, and one additional component on the "Additional" palette. In the former case, these are the "FileViewer", the "FileEditor" and the "MailEditor". In the latter case, this is the "ProgressMeter" component. A property editor will also have been installed for changing the edit keys.

7. You may now merge the J-Write component library help keywords with the Delphi Help System. These procedures are described in 2.2.

8. Finally, you may wish also to add the J-Write Components Help File to the Delphi program group.

- Under Windows 3.1, you may do this by selecting File|New when the Delphi program group is active. You need to create a new Program Item, and then enter the name of the help file (e.g. `C:\DELPHI\JWRITE\JCMPT.HLP`) as the command line.

- Under Windows 95, you should again select the Delphi Program Group in Windows Exploring, and, making sure that no icons are selected, select the File|New menu item and create a new shortcut. Enter the command line as the path to the help file (e.g. `C:\DELPHI\JWRITE\JCMPT.HLP`).

If you are installing the register version of the J-Write Component Library:

The J-Write installation program will have copied the .dcu and other component files to the directory specified at installation time. However, these must then be separately installed on the Delphi desktop, as described below. We recommend that if you have previously installed the Shareware version then you first remove this version using the instructions given in 2.2, and then install the registered user version.

1. Backup your 'COMPLIB.DCL' file. You will find this file in the 'BIN' subdirectory, in your Delphi directory. For example, if you have installed Delphi in "`C:\DELPHI`", then you will need to take a backup of '`C:\DELPHI\BIN\COMPLIB.DCL`'.

2. Start Delphi and select "Install Components" from the "Options" menu. The "Install Components" Dialog box will now appear.

3. Append the path of the J-Write components directory to the "Search Path". You will need to separate this path name from the existing search path by a semi-colon. For example, if the J-Write components were installed in "`C:\JWRITE\COMPNTS`", then add "`;C:\JWRITE\COMPNTS`" to the end of the existing search path.

4. Click on the "Add" button, and browse for the J-Write components directory. Select the "`FILEVIEW.DCU`" file.

5. Repeat the "Add" operation and select the "`MAILEDIT.DCU`" file.

6. Click on "OK". The component library should now be rebuilt. Once this is complete, then you will have three new components on the standard palette, and one additional component on the "Additional" palette. In the former case, these are the "FileViewer", the "FileEditor" and the "MailEditor". In the latter case, this is the "ProgressMeter" component. A property editor will also have been installed for changing the edit keys.

7. You may now merge the J-Write component library help keywords with the Delphi Help System. These procedures are described in 2.2.

8. Finally, you may wish also to add the J-Write Components Help File to the Delphi program group.

- Under Windows 3.1, you may do this by selecting File|New when the Delphi program group is active. You need to create a new Program Item, and then enter the name of the help file (e.g. `C:\DELPHI\JWRITE\JCMPT.HLP`) as the command line.

- Under Windows 95, you should again select the Delphi Program Group in Windows Exploring, and, making sure that no icons are selected, select the File|New menu item and create a new shortcut. Enter the command line as the path to the help file (e.g. `C:\DELPHI\JWRITE\JCMPT.HLP`).

2.2Installing the J-Write Components Help File

1. First close Delphi and then take a backup of the 'DELPHI.HDX' file. This is located in the BIN subdirectory, in your Delphi directory. For example, if you have installed Delphi in "`C:\DELPHI`", then you will need to take a backup of '`C:\DELPHI\BIN\DELPHI.HDX`'. If the help index installation procedure fails, then you may restore the help index from its backup.

2. Now run the "HelpInst" program located in the Delphi program group. Use File|Open to open 'DELPHI.HDX.

3. Select Keywords|Add Keyword File, and select the 'JCOMPNTS.KWF' file from the directory in which you installed the J-Write component library.

4. Save the new index using File|Save.

Once the J-Write Keyword index has been installed you can search for J-Write classes, properties, etc., from the Delphi IDE, using Help|Topic Search. They are now part of the same list of keywords, as are Delphi's own VCL. Note that the first time you try to access a J-Write class, etc. using the Topic Search, you may find that you have to browse for the help file. This is contained in the directory in which you installed the J-Write Component Library. Thereafter, there should be no need to browse for this file again.

2.3Removing the J-Write Component Library

If you longer wish to use the J-Write Components, then you should perform the following before deleting the J-Write components from your system.

1. Backup your 'COMPLIB.DCL' file. You will find this file in the 'BIN' subdirectory, in your Delphi directory. For example, if you have installed Delphi in "`C:\DELPHI`", then you will need to take a backup of '`C:\DELPHI\BIN\COMPLIB.DCL`'.

2. Start Delphi and select "Install Components" from the "Options" menu. The "Install Components" Dialog box will now appear.

3. Select FileView from the list of installed units, and click on the Remove button. Do the same for MailEdit.

4. Remove the J-Write components directory from the Search Path.

5. Finally, click on OK and the component library will be rebuilt less the J-Write Components.

6. Exit Delphi and run the HelpInst utility from the Delphi Program Group to remove the J-Write help files by removing ‘JCOMPNTS.KWF’ from the list of ‘DELPHI.HDX’ keyword files, and delete the J-Write Components Help File icon from the Delphi program group, if present.

3The Architecture of Component Library

The J-Write components have been designed as a set of integrated object classes following a Document/Viewer model. That is, there is one set of object classes that is responsible for managing text data, and another set responsible for rendering text oriented data within a window. This separation of function allows for a clear division between the background functions that manipulate text and the foreground functions that display it on the screen.

The Document Object Classes follow a hierarchy that starts with an abstract model of a text storage class (TTextBuffer), where each character is individually indexed by a longint index. A pointer may be obtained for any such integer, and that pointer may itself be modified by pointer arithmetic to point to any other character in the text storage, within a limited range. This abstract model is then refined (TLinesBuffer) by adding parsing methods that permit the text store to be organised into lines of text, and finally refined into a usable object class (TTextStream), with the addition of methods that map the abstract text store onto a single stream. This object class is itself satisfactory for read only access to text files. However, to allow for editing of text files, this object class is then further refined (TEditStream) to permit the additional of "deltas" to the text, including undo and redo facilities. This final object class supports a text file editor.

The Viewer Object Classes descend from the Delphi VCL TCustomControl. This is first refined (TFlickerFreeControl) to permit flicker free updating of a window - very important for an edit control - and then refined again (TLinesViewer) to create an abstract class supporting the rendering of line oriented text onto a window using a Delphi Canvas. This abstract class is then refined in order to marry it within the document classes (TTextViewer). The refined class is a further abstract class that supports the rendering or line oriented text held in a text storage class, as discussed above. The first usable object class (TStreamViewer) is then derived from this class by a simple refinement whereby the companion text storage object is created as one that holds a single text stream; this allows a further simple refinement to create a text file viewer class (TFileViewer). A separate refinement of TTextViewer creates an abstract editor class (TBigEditor). This is then similarly refined to that above, whereby the companion text storage object is created as an editing text storage class (TStreamEditor), creating the full text editor (TFileEditor).

The J-Write Component Library adds to this basic text editing hierarchy with a "bolt-on" uuencode/decode engine, which operates on text storage classes (TUUEncoder and TUUDecoder). A further refinement of the text editor (TMailEditor) is created to use this engine and to include some other useful extensions for EMail.

The current component library is completed by a simple progress bar (TProgressBar), derived from the flicker free control class.

3.1Document Components

Figure 3-1 shows the Document object class hierarchy. The TTextBuffer descendants are available as components in their own right and therefore descend from Tcomponent. However, the remaining classes do not have properties that are accessible through the Delphi object inspector, and thus

TObject is the preferred ancestor of these classes. The exception is TUndoList, which has TList as its ancestor.

3.1.1TTextBuffer

TTextBuffer provides an abstract text storage class, and presents an idealised view of a text buffer for use by TLinesViewer descendants, including TBigEditor. To the using class, the text is presented as an array of characters "Text", indexed by a longint. A longint index is not possible for normal object Pascal arrays, as object Pascal does not support the Huge Memory model.

"Text" is a publicly available property and returns a PChar to the indexed character. Pointer arithmetic may be performed on the returned value, as long as the result is within the range given by the "Valid" property. Valid holds the PChar values of the first and last pointers in the valid range. Pointer arithmetic on a PChar returned from Text that results in a pointer greater than or less than the valid range, will give a pointer to an undefined memory area. Note: for example, valid describes the memory segment holding the character at Text[index].

TTextBuffer is an abstract object that does not define how the PChar returned by Text[ ] is determined. This is the subject of descendant classes (e.g. from memory, streams, etc). The required text may be made available by getting a pointer to the appropriate memory block, or it may be a pointer into a buffer after the required text is read from a stream.

3.1.2T

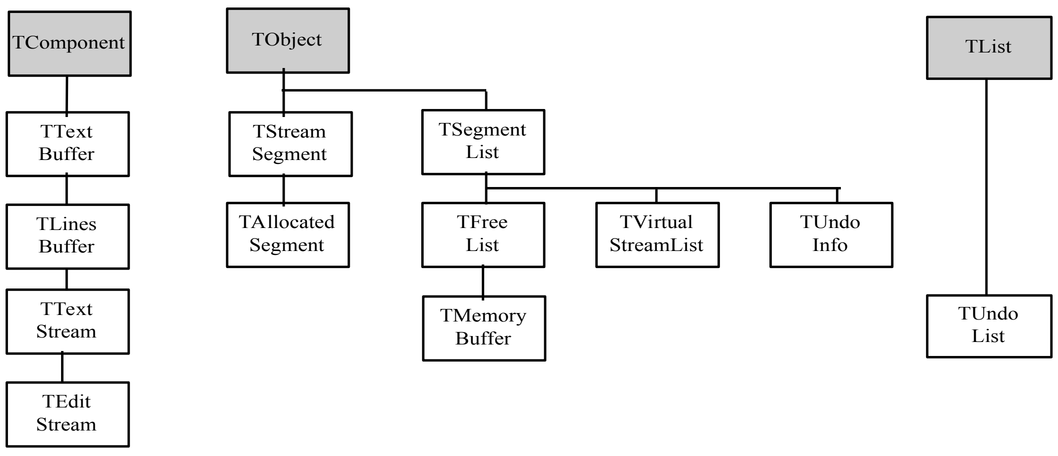

Figure 3-1 Document Components Hierarchy

LinesBuffer

TLinesBuffer is a descendant of TTextBuffer that includes additional functionality for parsing the text buffer into lines of text. As such, it simply adds additional methods to the TTextBuffer base class. The value of TLinesBuffer is that it separates out the functionality specific to parsing text into lines from the abstract text storage model. The user can then be clear as to what belongs where.

The parsing methods are critical for performance and are therefore written using Delphi's built-in assembler. There are hand optimised functions for:

- locating the previous hard line break (single LF, FF or CR/LF[1]) to a given point in the text;
- determining the end of a line, relative to a given start point; and
   - searching the text for a given string.

The greatest complexity is in determining the end of a line. This is because four different line wrap modes are supported:

On Hard line breaks only	There is no set maximum length to the line, and end of line is only declared at end of text or when the next LF, FF or CR/LF is found.
After *n* characters	End of line is declared after a specified number of characters, if a hard line break or end of text does not occur first. The line is truncated to end at the end of the last whole word.
After the display width of the line exceeds a limit	This limit is either the width of the window, or the printable width of paper in the current printer. The former is used to provide word wrap on window boundaries, and the latter for printing and for word wrap on printer page width (WYSIWYG). Again, the line is truncated to end at the end of the last whole word.

The above is further complicated by tab characters and the support of configurable tab settings at specific points, measured in logical inches from the start of the line.

To support the above, two conventional terms are introduced in this manual to describe the metric associated with a line. These are “width” and “length”, where the width of a line is the number of pixels it occupies on the display device, and the length of a line is the number of characters in it, including unexpanded tab characters.

All line parsing is performed by a single method “ScanForEol”. This method scans the text from a defined start point until either a hard line break is found, end of text, or the line width exceeds a given limit. A line width of zero conventionally means that there is no width limit on the line. This method calculates the display width of each line and expands tabs. To do this, it has to be provided with a font metric table, and a table of tab positions in pixels.

The need for font metric information implies a closer relationship between the viewer class and the text storage class than would have been preferred, but this is inevitable. In fact, in this set of components, the TLinesBuffer is given a pointer to the viewer’s Canvas object. It can therefore make direct enquiries on the font metric information necessary for creating the font metric table and tab position table. Given that access to the Canvas is anyway necessary, the TLinesBuffer also includes methods to draw text directly onto the Canvas at the direction of the viewer object. The benefit of including such methods in the TLinesBuffer, is then that the problem of dealing with a limited window on to the text storage, as provide by the TTextBuffer ancestor, is then localised within the TLinesBuffer, and the viewer class then only needs to deal in lines of text, with no worry about how they are stored.

Note that swapping the Canvas pointer to the current Printer Canvas, readily changes the rendering medium to the printer.

The low level parsing methods are then accessed though higher level methods. For example, *GetNextLine* is used to return the metrics of the line of text starting at a specified position.

[1] Following standard convetions, CR, LF and FF refer to the ASCII control characters Carriage Return, Line Feed and Form Feed, respectively.

GetStartofParagraph is used to locate the position of the preceding hard line break, and *GetDistanceBetween* and *GetCharAt* are used respectively to determine the number of pixels between two characters on the same line, and to find the character, so many pixels from another character on the same line.

Finally, it should be noted that the line scanning algorithm does not provide a means to limit a line by the number of characters in it. Even though an "after *n* characters" line wrap mode is required, word wrap after a limited number of characters is not a realistic option for the low level parsing routine. This is due to the fact that tab settings are expressed in inches and not characters, and hence tab expansion cannot be readily related to a set number of characters. Instead, word wrap after *n* characters must be simulated in the viewer class by multiplying *n* by the average character width in pixels. With a fixed pitch font this always gives the correct result. With a variable pitch font, there is a risk of the occasional early word wrap, but as the display width is rarely an exact number of characters, in any font, and that word wrap always truncates a line anyway, this effect is very unlikely to be visible to the user.

3.1.3TTextStream

The abstract object TLinesBuffer only added to TTextBuffer's functionality in respect of how text could be manipulated, but did nothing in respect of how the text is stored and accessed. TTextStream deals with this by providing a single stream as the text storage element. This can be any descendant of TStream, and the stream is viewed as read only. TTextStream is the text storage element for a viewer class that does not permit data to be edited, and includes a buffer to hold text read from the stream.

3.1.4TStreamSegment

The J-Write components are intended to support a fast editor of effectively unlimited size text files. Bearing this in mind, it is not realistic to simply modify a stream *in situ*. The overhead of doing so is too great. Instead, in order to support a text storage class that provides for an editable stream, it is necessary to create a means to store "deltas" to a stream, which can then be applied once the user wants to actually save the edited text, - and which could be discarded if the edited text was not wanted. The TStreamSegment Class provides the basis for such deltas.

A TStreamSegment object describes an area or segment of a stream. It includes a reference to the stream and the offset and length of the segment, and supports a method to read all or part of the stream segment into a buffer.

A TStreamSegment object is also designed to be a member of a doubly linked list. It includes methods for manipulating itself as a member of the list, such that:

a) one segment can be split into two segments, each describing two contiguous parts of the original stream segment, and as adjacent list members, and

b) two adjacent segments on the same list can be merged into a single segment, if they describe contiguous segments on the same stream.

It is thereby possible to construct a list of TStreamSegments that includes both elements of an original stream and "deltas" inserted between the two parts of a split segment. It is also possible to remove segments from the list to support deletions, or to restore removed segments, as in an undo operation.

Note that TStreamSegment also provides a virtual method "DeAllocate" in order to Free an object. This is provided to allow for changed semantics of the Free method in descendant objects, and is always used instead of Free for "disposing" of such an object.

3.1.5TSegmentList

A TSegmentList object is used to reference and manipulate a list of TStreamSegment objects. It has a "size", which is the total size of all its constituent TStreamSegments, and provides methods to manipulate the list. For example, to insert segments and to remove them.

3.1.6TVirtualStreamList

Although a TSegmentList has methods to manipulate a list, it provides no semantics as to a list's use. TVirtualStreamList imposes one possible set of semantics. That is, the list's segments together describe a "virtual stream". Like a real TStream descendant, such a stream has a size and may be viewed as comprising an ordered set of bytes. The user can "seek" to any position within the virtual stream, and then "read" a contiguous series of bytes from that point in the stream.

However, unlike a TStream descendant, this class provides no "write" method. Instead, it provides methods to insert a TStreamSegment or segments, and to cut out the TStreamSegments that describe part of the virtual stream - if necessary splitting existing segments in order to do this. A TVirtualStream therefore provides the functionality necessary to edit a stream by applying deltas rather than manipulating the stream itself.

In practice, a TVirtualStream object starts with a single segment list, that segment describing the whole of a single stream. It is then possible to remove (delete) part of that stream, by cutting out an area of the virtual stream. This is achieved by splitting the original segment into three, and removing the middle one. This cut out segment may be saved and later re-inserted by an undo operation. The insert would recognise that the re-inserted segment is contiguous with its neighbours and merge them together. None of this changes the original stream, only when the virtual stream is copied to a new stream in a later "save" operation are the changes in effect applied.

Similarly, text may be inserted by simply putting it in a new stream segment and inserting this into the virtual stream, at the appropriate position.

3.1.7TFreeList

New text inserted into a virtual stream has to be held on a proper TStream descendent. This cannot be the original stream, so another stream has to be created and managed for this purpose. The TFreeList class is defined for this purpose. This is another descendent of TSegmentList, but with different semantics to TVirtualStreamList.

The segments on a TFreeList describe the unallocated parts of a stream, A "FreePtr" is also provided to identify the point on the stream from which new allocations can take place.

A TFreeList object satisfies requests for stream segments of a given size by searching its segment list for a segment of the required size or greater. If one is found, then it is split to create a segment of the correct size, which is then allocated to the requester. If none is found, then the FreePtr is increased and a segment of the appropriate size created from the end of the stream.

A single "SaveBuf" method supports a single step allocation of a segment and the writing of data to that segment. It is therefore not possible to allocate a segment without also initialising it.

3.1.8TAllocatedSegment

Segments allocated from a TFreeList have a different semantic from normal segments when it comes to freeing the segment. In this case, the segment of the free list should be deallocated and the segment returned to the list.

TAllocatedSegment is a descendant class of TStreamSegment which additionally knows the TFreeList from which it was allocated and supports a refined “DeAllocate” method that returns it to the appropriate free list instead of disposing of the object. Such a segment is only disposed of when its TFreeList is disposed of.

3.1.9TMemoryBuffer

TMemoryBuffer simply refines TFreeList by making the stream from which allocations are made, a TMemoryStream. Inserted text is therefore held in a single memory buffer.

3.1.10TEditStream

The TEditStream class is a descendant class of TTextStream and adds to the TStream descendant of TTextStream with a TVirtualStream. The original single stream is then described by a single initial segment on the virtual stream. A TMemoryBuffer is also created to hold inserted text.

TEditStream supports Insert, Delete and Replace operations on text it manipulates. It is a text storage class with TTextBuffer as its ancestor. It provides methods for the corresponding Insert, Delete and Replace abstract methods of TTextBuffer.

Insert, Delete and Replace operations are typically carried out by modifying the text current held in the buffer created by TTextStream. Only when they affect text not held in the buffer, or when a different parts of the buffer affected are the changes flushed out to the virtual stream. For example, insertion of a single character simply updates the buffer. Only when the insertion point overflows the buffer or changes to a new part of the buffer, are the inserted characters copied to a TMemoryBuffer segment and inserted into the virtual file.

3.1.11TUndoInfo

Text cut out from a virtual stream is described by segments which can be saved for use in a later undo operation. The TUndoInfo class provides a means to save such segments, and is another TSegmentList descendent.

A TUndoInfo objects holds a list of segments describing a contiguous area of a virtual stream and which are the result of one or more cut operations. In addition to a TSegmentList, this class holds the offset in the virtual stream at which the cut took place, and the size of any text that replaced the cut out segments at the same point. It also includes state information from the corresponding Viewer object, in order that a properly synchronised undo operation may take place. It is also possible for a TUndoInfo object to have an empty segment list - for example, if it describes an insert operation with no corresponding deletion of text.

3.1.12TUndoList

It is generally necessary to keep several undo operations in the order that the operations occurred. The TUndoList class supports this, and a TUndoList object is created by each TEditStream in order to hold its undo transactions.

TUndoList is a TList descendant, with the list organised into a push down stack. TUndoInfo objects may be inserted at the top of the stack, and are read from the top of the stack, whenever an undo operation is requested. The number of objects on the list may be limited and if this limit is exceeded then the object at the bottom of the stack is discarded, this always keeping the most recent list of undo operations.

TUndoInfo objects also keep state information from the Viewer object, and to enable this information to be accessed when an undo operation is saved, and restored when the undo is performed; event handlers are provided to support the necessary linkage with the Viewer object.

In some cases, several separate undo operations have to be linked together for a single undo. For example, a “Replace All” operation results in many modifications in a text stream, and the expected semantic of undo is to undo all these changes in one go. TUndoInfo objects therefore have a “linked” flag that may be set to indicate that they are part of a linked chain of undo operations. Such linked operations are also treated as a single operation when calculating the number of transactions on the stack, thus avoiding the possibility of discarding part of a linked set of undo operations, when the number of transactions on the stack exceeds the limit.

There is also a clear symmetry between undo and redo. When an undo operation is performed, the cut out segments, together with the size of the restored segments, forms a new TUndoInfo object, which is then added to the redo list. This is another TUndoList object created by TEditStream. However, unlike the undo list, the redo list is always cleared once the text stream is modified again, thereby avoiding a corruption of the text.

3.2Viewer Components

Figure 3-2 shows the class hierarchy for the J-Write document viewer classes. Most classes are all derived from the Delphi VCL TCustomControl class and therefore share the common properties and methods of all Delphi controls. TWinScrollBar encapsulates functionality associated with a control’s scroll bars, and TKeyManager maintains the relationship between edit keys and response methods.

Published properties from all these classes, excepting TKeyManager, may be accessed through the Delphi Object Inspector at design time.

3.2.1TFlickerFreeControl

TFlickerFreeControl is, as its name suggests, a descendent of TCustomControl, modified to support flicker free drawing on its Canvas.

This trick is performed by intercepting the WM_EraseBkgnd message and deferring its effect until the WM_Paint message is received. Further, drawing is done on a memory DC, with its contents BitBlt'ed to the real DC only when update is complete. This results in a small amount of additional overhead, but a much crisper display.

3.2.2TLinesViewer

TLinesViewer is an abstract class that provides the core functionality for rendering line oriented text onto the Canvas of a custom control. TLinesViewer is itself a descendent of TFlickerFreeControl, and takes advantage of the crisp display that this gives.

TLinesViewer assumes a text model, whereby the text is structured into lines. Each line has a sequence number, starting from zero, and has a “length” in characters and a “width” in pixels. The largest possible line number is given by the LastLine property. All lines are displayed in the current Font, as defined by the control's Font property. TLinesViewer can both display these lines of text on the control' Canvas, or print them onto the current Printer.

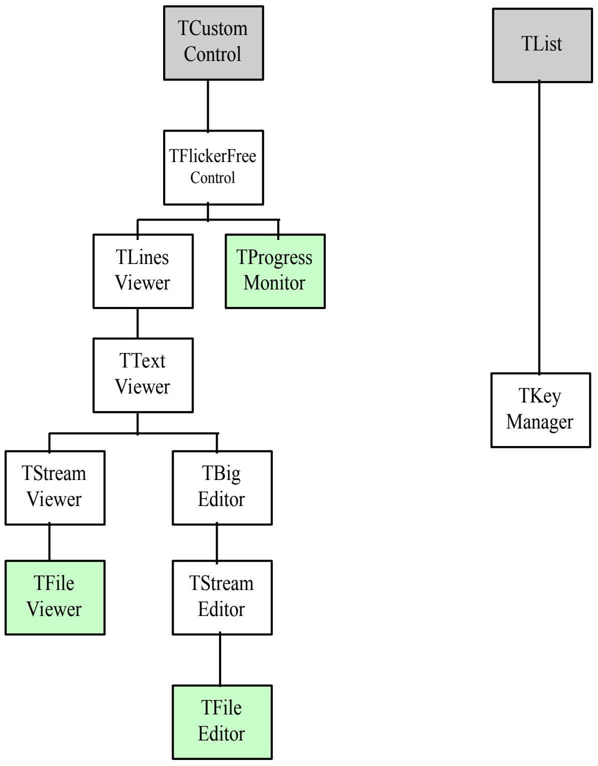

Figure 3-2 Document Viewer Class Hierarchy

Typically, there will be more lines of text to display than there are lines on the screen. TLinesViewer handles this by maintaining information on the current view presented to the user, in the form of two variables: DeltaY, which is the line number of the first line displayed by the control, and DeltaX, which is the pixel offset into each line of the leftmost pixel displayed. The CaretPos property also keeps track of where the Caret is displayed, as the relative pixel and line co-ordinates of the Caret.

TLinesViewer supports Caret movement using both the keyboard and the mouse. Text can also be selected by holding down the shift key while moving the Caret, or by "dragging" with the mouse. Selected text can then be copied to the clipboard.

TLinesViewer methods also support free text searching for the next occurrence of a string (of up to 256 characters) in either direction, the positioning of the Caret on any line number, and for the saving of the text to a file.

TLinesViewer also maintains a State property. This is a set of possible states and is used by a using application to update menu item enable/disable state and a status line. An OnStateChange event reports a change in the current state.

For reporting progress on lengthy operations, TLinesViewer has a structured set of methods for entering a busy mode (hour glass cursor displayed), exiting a busy mode, and reporting progress.

Together with the OnProgressEvent Handler and State information, this readily enables the support of progress bars and other means to report progress.

3.2.3TWinScrollBar

Delphi already provides two different encapsulations of the windows scroll bar. One provides for support of a scroll bar control, while the other supports a TScrollBox's scrollbar. However, neither is a satisfactory basis for the encapsulation of TLinesViewer's scroll bar.

TWinScrollBar supports a scaled scroll bar which provides for position and range expressed as longints. It also supports both dragging the scrollbar thumb and moving the thumb to a new position, in the sense of whether the display is updated while the thumb is dragged or at the end of the drag operation.

3.2.4TKeyManager

A design goal for TLinesViewer is to enable all cursor movement and other keys to be configurable at design time using the Delphi Object Inspector. This is in addition to shortcut keys that might be set through the menus. The keystrokes themselves are readily intercepted by the KeyDown method, but there remains the problem of relating keystrokes to methods, when those keystrokes themselves are defined by properties. This problem is solved by TKeyManager.

TKeyManager is a TList descendant that holds a list of records that relate defined keystrokes to the methods that implement the corresponding functions. The list is ordered by keycode and searched using a binary chop search method. An "OnKeyCode" property is provided in order to update the list. This is an indexed property with the index being a keycode and the value of the property being a method. Assigning a method to this property has the semantic that this method is called in response to the keycode used as the index. Assigning a nil value removes the keycode from the list.

Keycode values are the standard windows virtual key codes, modified by kbControl, kbShift and kbQuickKey modifiers (added to the keycode value). The semantics of these modifiers is that, for example, if kbControl is added to the virtual key code, then the control key must be pressed down when the keystroke is received in order for the method to be called. kbShift has a similar semantic with the Shift key. kbQuick is for two keystroke function keys, a Quick Key (determined by the QuickKey property) followed by another keystroke. If the kbQuickKey modifier is used then the semantic is that the method is called only if the keystroke was immediately preceded by the defined QuickKey.

In use, the TranslateKey method is called every time a wm_KeyDown message is received. If the keystroke is in the list then the corresponding method is called.

A property editor is provided to allow Keystroke values and appropriate modifiers to be readily maintained from the Delphi Object Inspector.

3.2.5TTextViewer

TLinesViewer has only an abstract model of the text it displays. TTextViewer refines TLinesViewer such that this text is stored in a TLinesBuffer container. In order to do this, TTextViewer has to match the TLinesViewer concept of a set of lines, each with a unique line number, to the TLinesBuffer model of text parsed into unnumbered lines. TTextViewer also manages the different word wrap modes.

When trying to find a given line, one possible mode of operation for TTextViewer would be to simply start at the first line, or the last known position, and just work its way through the lines in the

TLinesBuffer, until the required line was found. This is satisfactory for small files, but quickly becomes inefficient for large files, and so this approach is only done when the user wants to explicitly go to a given line number.

Instead, TTextViewer optimises the search for a given line number by the following algorithm:

a) If the required line is within two screens of the last known position, then it simply works its way through the TLinesBuffer to find the required line.

b) On the other hand, if the required line is further way then the offset of the start of this line is estimated, and the viewer synchronised with the line nearest this estimated position. The operation is further optimised by recognising jumps to the first and last line, and synchronising with the first and last line in the TLinesBuffer respectively.

This technique works very well as, in practice, the only time a jump is made to a line number which has to be estimated is when the scroll bar thumb is moved, and the user does not then accurately know which line should be displayed anyway. The same approach is also used to synchronise the viewer to a given offset in the file. For example, to view the result of a search operation.

The drawback of this approach is that there is an uncertainty over which line number is actually being displayed. This uncertainty is accepted temporarily for performance reasons. However, to determine the actual line number following this kind of jump, a background process is started to progress through the text from the last known line number until the current line is found. The correct line number is then accurately known.

This background process is performed during application idle time. The HandleOnIdle method performs this task and must be assigned to the TApplication OnIdle property, or called from a method that is assigned to this property. Whenever this method is called, it checks to see if the current line is estimated and then proceeds to locate the line accurately. In order so as not to interfere with foreground processing, this method works through no more than 100 lines at a time, and may be called many times before its work is done.

TTextViewer carries out its task by maintaining a “thumb”, locating a known line in the TLinesViewer. This “thumb” takes the form of the offset of this line (LinePos), the offset of the immediately preceding line (PrevLinePos) and the next line (NextLinePos). It also keeps the length of the current line and its width, and the estimated/known line number for this line. Most of the methods of TTextViewer are really concerned with relating this “thumb” to a line as seen by TLinesViewer.

Similary, TTextViewer also calculates the position of soft page breaks. Normally, this is done by simply counting the number of lines of each page. However, the algorithm is complicated by Hard Page Breaks that may be inserted anywhere in a text file. When scrolling down a file, a hard page break simply resets the line counter. However, when scrolling up a file, the preceding page break cannot be estimated. TTextViewer always keeps track of a preceding page break in order to optimise its work, but, when necessary, will re-compute the location of soft page breaks by re-running the background process that calculates line numbers accurately.

3.2.6TStreamViewer

TStreamViewer is a simple refinement of TTextViewer, where the TLinesBuffer the TTextViewer uses, is created as a TTextStream i.e. this class supports the viewing of text sourced from any TStream descendant.

3.2.7TFileViewer

TFileViewer goes one stage further and defines the stream to be a TFileStream. It provides a FileName property, and the semantics of assigning a filename to this property, is that the named file is opened, becomes the stream accessible though the TTextStream, and viewed through the TTextViewer. Any currently open file is closed.

3.2.8TBigEditor

TBigEditor provides an alternative refinement to TTextViewer that leads towards a file editor. TBigEditor adds methods that support the editing of text in the viewer. Methods provide for insertion of characters, line breaks, and text pasted from the Clipboard. They provide for the deletion of selected text and a single character, and "Drag and Drop" editing with the mouse is also supported. TBigEditor also includes the necessary event handlers to support the Undo/Redo mechanism provided by TEditStream, and provides methods to flow the text into paragraphs, capitalise words, and to support search/replace.

The TLinesViewer KeyManager is also used in order to support additional Edit Keys.

Note that the insertion/deletion of characters is optimised, both by directly updating the "thumb" maintained by TTextViewer, and by drawing directly on the Canvas rather than using the windows paint message. This is necessary to work at full typing speed on a low speed processor (e.g 386sx).

3.2.9TStreamEditor

TStreamEditor is a simple refinement of TBigEditor, which creates the TLinesBuffer referenced, as a TEditStream, thereby providing for the editing of any text stream.

3.2.10TFileEditor

TFileEditor's relationship with TStreamEditor, is like that between TFileViewer and TStreamViewer. TFileEditor refines TStreamEditor to create the stream as a TFileStream, thereby providing a file editor. Initially, this is an untitled file. The editor can also be cleared to this initial state, at any time, by calling the Clear method.

Additionally, TFileEditor provides a method to *save* the file currently being edited, replacing the current file, and renaming it with a .bak extension. To support file saving, TFileEditor also provides three Events:

1. **OnSaveModified**: this is called when an attempt is made to clear the current contents of the file editor (e.g. when the Clear method is called, or when the FileName property is assigned to a new file name), and when that contents has been modified. This gives an opportunity for the user to be prompted to save the file, lose the edits, or to cancel the request to clear the file editor.

2. **OnSaveNewFile:** this is called when the save method is called for an untitled file. It gives the user a chance to specify a file name and to save the file, or to cancel the save.

3. **OnNewFileName:** this is called whenever the file name changes, or is cleared. It gives an opportunity for (e.g.) a window caption to be changed in line with the name of the file currently being edited.

3.3EMail Extensions

The EMail extensions comprise the uuencode and decode engines and the TMailEditor class. The object classes for EMail Extensions are shown in Figure 3-3.

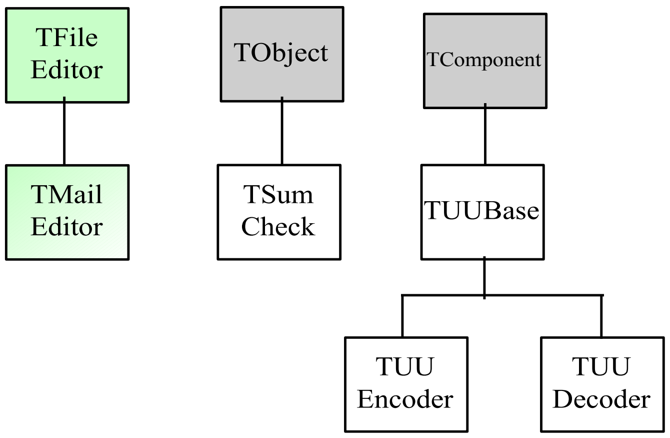

Figure 3-3 Mail Extensions Object Hierarchy

3.3.1TSumCheck

Often, uuencoded files are provided with an error detecting sum check on both the uuencoded text and the original binary data. This sum check is generated using the Unix sum utility. This mechanism is support by the J-Write EMail extensions through the TSumCheck class.

A TSumCheck object supports a single accumulator and byte count, that can be updated either by adding a line of text, or a block of data. Two methods are provided to support each of these approaches, respectively. A Clear method is also provided to reset the accumulator, and a GetSum method is used to obtain the formatted result of the checksum computation. The format is the same as generated by the Unix sum utility.

In use, a TSumCheck object is used either to generate a checksum on uuencoded text or on the binary data. Therefore, for a given object instance, only the Addline or the AddBlock methods are used.

3.3.2TUUBase

TUUBase provides the common aspects of both the uuencode and the uudecode engines. These are concerned with the creation/deletion of two TSumCheck objects to support error checksums on the uuencoded text and the original binary data respectively.

3.3.3TUUEncode

TUUEncode supports a single public method “UUEncodeFile”. A File name and TTextBuffer object are parameters to this method. The method encodes the contents of the file using the uuencode algorithm, and inserts the lines of encoded text into the TTextBuffer, starting at the indicated offset.

The algorithm is table driven and the actual encoding is performed using Delphi's built-in assembler. The OnBlockEncode event is provided to report progress in encoding the file.

3.3.4TUUDecode

TUUDecode supports the decoding of one or more uuencoded files held in a TLinesBuffer. It provides a single public method "UUDecode" which has a TLinesBuffer and offset into the TLinesBuffer as its parameters. Starting at the indicated offset, the method passes through the lines of text looking for the header of a uuencoded file. When one is found, the OnStartDecode event is called.

The OnStartDecode event gives the user a chance to change the filename into which the file will be saved. By default, this is the file name given in the uuencode header and it will be saved in the current directory. This event also allows the user to instruct the decoder to ignore a uuencoded file and to continue looking for another uuencode header.

The OnEndDecode event is called following completion of the decode process and reports the outcome.

The OnBlockDecode event is provided to report progress in both decoding files and in passing through the TLinesBuffer looking for uuencode headers.

3.3.5TMailEditor

TMailEditor adds to TFileEditor support for the EMail Extensions. These are:

1. File Merging
2. uuencode and decode
3. Paste or insert text with mail quotes
4. Interface to a DOS compression utility.

3.3.5.1File Merging

The TMailEditor InsertFile method can be used to insert text files at any offset into the file. On request, the InsertFile method will also call the Windows OemToAnsi conversion API, in order to convert DOS text files to windows text files. This will be necessary if the file uses the extended DOS character set.

The TMailEditor InsertQuoteFile method may be used to insert text files as above, but also to prefix each line by the text string held in the MailQuote Property. This is a general requirement for replying to EMails. This operation is performed by creating a temporary TTextViewer container to hold the input text file, and using this to read the text line by line, prefixing each line by the MailQuotes prior to inserting it into the file.

Binary files can also be inserted into a text file by uuencoding them, so that they are represented by lines of text. They can then be sent as text only EMails. The TMailEditor UUEncode method supports this, and creates a temporary TUUEncoder object to perform the uuEncode and insert operation.

3.3.5.2UUDecode

TMailEditor may also be used to decode text files that contain uuencoded binary files. The TMailEditor UUDecode method may be called to perform the decode operation and creates a temporary TUUDecoder object to perform this operation. The OnStartDecode event may also be used to redirect a decoded file to the appropriate directory, or even to change the file name or to ignore a uuencoded file. The OnEndDecode event may be used to report the conclusion, success or failure, of each decode.

The UUDecode method will decode multiple uuencoded files, encoded into the same text file, and can cope with files encoded into multiple sections.

3.3.5.3Paste Quote

The PasteQuoteFromClipboard method is very similar to the InsertQuoteFile method, except that the source of the text is the clipboard rather than a file. A temporary TEditStream is created to support the method and the contents of the clipboard inserted into it. The subsequent actions are then identical to InsertQuoteFile.

3.3.5.4File Compression

The uuencoding algorithm expands files by 30% and it can therefore be very costly to send a large file by EMail. There is every incentive to compress files prior to uuencoding them and this is supported by TMailEditor.

The InsertCompressedFile method is used to compress, uuencode and insert a file into the file being edited. It requires as parameters the name of the file, the normal file extension used by the compression algorithm that will be applied, and a DOS command line to run the DOS compression utility that will perform the actual compression. The command line should be as you would type it in, in response to DOS command line prompt, including any options switches that need to be set. The difference is that where the name of the (output) compressed file name would be entered, the two character symbol “&Z” must be used, and where the (input) file name would be entered, the two character symbol “&F” must be used. The InsertCompressedFile method replaces these symbols with the file names that it uses.

The method works by first compressing the input file into a temporary file and then uuencoding and inserting in to the text file being edited, the contents of the temporary file. The temporary file is then deleted. The DOS compression utility is invoked using the RunDOSProgram function provided by the JWRUN unit. This unit is responsible for invoking a DOS program, communicating with it and obtaining the programs completion code.

The JWRUN unit may also be compiled using Turbo Pascal to create the JWRUN.EXE program. This is used as a shell to execute the required DOS Program and communicates with the calling Windows program through a temporary file.

Note that the J-Write component library does not contain a compression utility. It has been designed to work with industry standard compression utilities, such as pkzip and lha, and these must be obtained separately.

3.4Progress Meter

TProgressMeter is a simple control that provides a progress bar and percentage done indicator. It is derived from TFlickerFreeControl in order to ensure a crisp display.

The Progress Meter comprises a rectangular window in which a percentage count is displayed centrally. The actual value displayed is taken from the PercentDone Property which must be a number between zero and one hundred. Every time the value of this property changes, the display is updated.

In addition to drawing the text representing the percentage done, a proportion of the pixels in the control's window are also inverted. Starting from the left hand side a rectangular area is calculated that is the same height as the control and has a width that is exactly the percentage of the width of the control as is given by the value of the PercentDone property. It is the pixels in this rectangle that are inverted. The result is a bar that appears to grow as the percentage done increases until it covers the whole window once 100% is reached.

3.5Units

The J-Write Component Library comprises the following units.

3.5.1TextBuff

This unit comprises the text container classes and the support classes used to maintain virtual streams. TextBuff defines:

TAllocatedSegment	TStreamSegment
TEditStream	TTextBuffer
TFreeList	TTextStream
TLinesViewer	TUndoInfo
TMemoryBuffer	TUndoList
TSegmentList	TVirtualStreamList

3.5.2FileView

This unit contains the document viewer classes and the progress meter class, comprising:

TBigEditor	TLinesViewer
TFileEditor	TStreamEditor
TFileViewer	TStreamViewer
TFlickerFreeControl	TTextViewer
TKeyManager	TWinScrollBar

3.5.3UUEncode

This unit contains the classes providing the "bolt-on" uuencode and decode engines, comprsing:

TSumCheck	TUUDecode
TUUBase	TUUEncode

3.5.4MailEdit

This unit contains the TFileEditor descendant TMailEditor, which provides support for theEMail Extensions.

3.5.5JWRun

This unit may be compiled with either Turbo Pascal to create a real mode DOS program, or with Dephi to create a Delphi unit. In the latter case, the unit provides the "RunDOSProgram" function, which may be called to run a DOS program in a separate DOS box, with the calling windows application suspended until it completes. The DOS completion code is return.

When compiled with Turbo Pascal, the JWRun program is created. This is a wrapper program that is invoked by the *RunDOSProgram* utility function to execute a DOS program and return the program's exit code.

3.5.6KeyEdit

This module provides a property editor for the TKeyDefinition type defined in the FileView unit.

4Using the J-Write Components

An example application (MDIEdit) is provided with the J-Write Component Library. Once the J-Write components have been installed, this application can be built and used to understand how the J-Write components work. This chapter describes how the example application was built.

4.1Creating The MDIEdit Application

MDIEdit is based on the Delphi MDI Application template and to re-create this application, you should start by using the Project Expert to create a basic MDI application. Using the Project Manager, the MDI Child form may then be displayed, and the J-Write FileEditor component dragged onto this form in the normal way. By default, the alignment of the FileEditor is *alClient* and hence it will immediately expand to fill the entire form. Note that you should typically set the Form's "ActiveControl" property to the name of the FileEditor, and this is essential if there are other components on the form such as status panels and speed button panels.

Work may then start on re-arranging the menus to work properly with the File Editor. In the example application, the Edit menu has been removed from the main form and the File menu reduced to the New, Open and Exit items only.

4.2File|New and File|Open

The response methods for these two menu items should be changed to reflect the way the FileEditor works. When a FileEditor component is created, the edit window is by default initialised to an un-named file, and hence there is no work in this case to be done. When opening a named file all that additionally needs to be done is to set the FileName property to the pathname of the file to be opened.

In the example application, the response method for the File|New menu item needs to do no more than to set the Caption for the newly created child form to a suitable NONAMEn title, while the response method for the File|Open menu item needs to have an assignment statement added to assign the file name returned from the OpenDialog to the FileEditor's FileName property.

4.3The Child Form's Menu

A menu for the child form may now be created by dragging the menu component from the palette, on to the form. This should have its "AutoMerge" property set to true so that his menu is merged with the main form's menu when the child form is active. The GroupIndex properties of the Main Form's "Windows" and "Help" menus should also be set to a high number (80 and 90 are used respectively in the example), so that the child form's menu can be merged in before these.

The File, Edit, Search and Options menus shown in the example may now be created. The File menu should be given a GroupIndex of zero, so that it replaces the Main Form's File menu, and the other menus given a GroupIndex intermediate between zero and that given to the MainForm's Window menu. The response methods for the child menu items may now be set up.

Most of these are straightforward. For example, the child menu's File|New and File|Open menu items are simply redirected to the corresponding methods of the Main Form. The Edit|Cut, Edit|Copy, etc. response methods simply call the corresponding method of the FileEditor.

The response methods for Search|Goto Line and Options|Max Line Length are slightly more complex as these methods use an InputQuery to get the required line number and maximum line length from the user. Options|Font and Options|Colour have a similar structure but use the standard Font and Colour Dialogs instead of InputQuery.

4.4File|Save and File|SaveAs

The response method for File|Save simply calls the FileEditor Save method, while File|SaveAs first uses the SaveDialog standard dialog to get the file name to save the file under and then calls the FileEditor's SaveAs method. However, for these methods to work effectively, some event handlers also need to be defined.

4.4.1The OnCloseQuery Event

Firstly, The MDI Child Form's OnCloseQuery event should be set up to ensure that the user is prompted to save a modified file before the editor is closed. However, the response method does not have to do this directly. All it needs to do is to call the FileEditor's Clear method, and a give a positive response to the query if the vsModified state bit in the FileEditor's State property is then clear.

4.4.2The OnSaveModified Event

The FileEditor's OnSaveModified event handler is where the user should be prompted to save a modified file. This is called by the Clear method if the text has been modified, and is similarly also called if the FileName property is changed (i.e. a new file is to be loaded into the edit window). This event provides a general method for prompting the user to save a file.

The OnSaveModified event handler may prompt the user using a simply MessageDlg and must return the result using the TModalResult codes (mrYes, mrNo or mrCancel). In the first case, the Save method will then be called to save the file. In the second case, the vsModified state is simply turned off, allowing the edit window to be cleared, while, in the last case, the method that called the event handler is abandoned.

Of course, calling the Save method either implicitly in the above case, or explicitly as with the File|Save response method, is not always sufficient to save a file. If the filename is already known then Save will replace the existing file on the disk, renaming the old version with a .bak extension. However, if the filename is unknown (i.e. the FileName property is empty), then it is necessary to prompt the user for a filename. This requirement is satisfied by the OnSaveNewFile event.

4.4.3The OnSaveNewFile Event

The response method for this event is as simple as possible. It is typically the same response method as for the File|SaveAs menu item. The event is called when the FileEditor is unable to complete a Save operation because the FileName is empty and File|SaveAs is essentially what is now required: the SaveDialog should be displayed to prompt the user for a filename, and then SaveAs method called to effect the save.

4.4.4The OnNewFileName Event

This OnNewFileName Event supports a change of FileName (e.g. as a result of the SaveAs method) and is called whenever the value of the FileName property changes. The example application makes a typical use of this event by changing the Form's Caption in response to the change of FileName.

4.4.5The OnSaveModePrompt Event

The last event supporting file saving is the OnSaveModePrompt Event. The J-Write FileEditor is capable of saving the text buffer either as it is, or with line breaks inserted where line wrap occurs. The save algorithm is determined by the SaveMode property. However, before the actual save takes place, the OnSaveModePrompt event is called. This gives a chance for the save mode to be modified to meet local requirements.

The example application provides a typical response method for this event. Firstly, if the WrapMode property is set to *opNone* i.e. no line wrap, then the response method returns the save mode as the *smParagraph* mode. This saves the text as paragraph text i.e. as it is. This is clearly sensible if there is no line wrap. If the current SaveMode is *smPrompt* then the user is prompted for the required save mode, otherwise the SaveMode is returned unchanged.

Other response methods may be defined depending on local requirements.

4.5Search and Replace

The FindDialog and ReplaceDialog standard dialogs are used to support search and replace in the example application, and their use follows a standard pattern.

The FindDialog and ReplaceDialog Execute methods are called from the response methods for the Search|Find and Search|Replace menu items respectively. Note that in each case, the initial find text is set up using the FileEditor's GetInitialFindStr method. This returns the word under the Caret, if any, and enables searches to be quickly set up for the next occurrence of a given word.

The FindDialog and ReplaceDialog objects themselves have event handlers which are called when the Find and Replace buttons are clicked. The OnFind response method is in common to both and simply calls the FileEditor's Find method. Note that this method returns false if the search fails and such an outcome is reported by a MessageDlg.

The ReplaceDialog's OnReplace event also requires a response method. This simply calls the FileEditor's Replace Method in order to carry out the replace operation. This method then searches for the next occurrence of the FindStr and like the Find method can return false if this search fails. Again this reported by a MessageDlg.

To support the ReplaceAll Operation (which is invoked through the corresponding search option bit), the FileEditor provides an OnReplaceAll event. This is called on completion of a ReplaceAll and reports how many items were replaced. In the example application, a response method is provided to report the replace count to the user.

4.6The Speed Buttons

Speed buttons typically call the same response methods as do the corresponding menu items. However, in an MDI application this is more complicated as some of the response methods are defined in the Child Form, whilst the speed buttons are defined by the Main Form.

The technique used here for such speed buttons makes an underlying assumption that the Child Form is always a TMDIChild (at least when the button is enable); cast the ActiveChild Property of the Main Form to TMDIChild, and then to call the response method.

4.6.1Word Wrap Speed Buttons

In the example application, the rightmost group of speed buttons work together to both indicate the current wrap mode and to provide a mechanism for quickly changing the word wrap mode. They are defined as forming a common group, by setting their GroupIndex Property to a common non-zero integer, so that one button is down and the others are in their up state. The button that is down indicates the word wrap mode, and clicking on another button changes to the corresponding mode.

To meet this requirement, it is necessary to provide a simple response method for each button, which sets the FileEditor’s WrapMode to the required value. However, there is a complication, as with all speed button’s, there is an assumption that the ActiveChild is a TMDIChild object.

To avoid problems that might occur if this assumption is invalid, event handlers are set up for the FileEditor’s OnEnter and OnExit events. These events are called, respectively, when the FileEditor gets and loses the focus. In the OnEnter response method, the group of speed buttons is enabled, in the OnExit response method, they are disabled. This ensures that the buttons can only be used when a TMDIChild is the active control.

4.7Enabling Menu Items and Speed Buttons

Generally, menu items and speed buttons depend not only on a TMDIChild having the focus, but also on the state of the editor. The FileEditor’s OnStateChange event is provided for this purpose of setting the proper enabled state of the relevant menu items and speed buttons.

The OnStateChange event is called whenever the FileEditor’s state flags change, or, very conveniently, when the FileEditor gets or loses the focus. In the example application, the response method for this event queries the current state and sets each menu items’ and speed buttons’ enabled state as appropriate. This response method is also responsible for setting the insert mode and modified status items on the Main Form’s status line, as these information fields are also dependent on the FileEditor’s state.

4.8Reporting Progress

Several operations performed by the FileEditor can take a long time. This includes searching large files and the uuencoding and decoding functions of the Mail Editor variant. A progress bar is available to provide the user with an indication of progress. This is included in the J-Write component library and may be found on the “Additional” component palette. In the example application, a Progress Bar is included on the Main Form’s status bar, in the rightmost panel.

The Progress Bar is normally invisible. It needs only to be made visible when a lengthy operation is in progress and, in the FileEditor, such an operation is synonymous with the vsBusy state bit being set. The visible status of the Progress Bar is therefore readily determined as part of the response method to the OnStateChange event.

The FileEditor’s OnProgressEvent is used to report incremental changes in the progress of a lengthy operation and, for example, reports how many characters have been searched out of the total to be searched. In the example application, the OnProgressEvent response method simply updates the progress bar in order to give the user an indication of progress.

During lengthy operations, the windows message queue is still serviced (the FileEditor is a good citizen!) and a check is made for the CancelKey (usually set to Esc). If this is found (i.e. the user has pressed Esc) then the operation is cancelled. However, the user is given a chance to confirm the cancel through the OnAbortRequest event. This event is called when the CancelKey is found and a TModalResult code (mrYes or mrNo) must be returned. If this is mrYes then the operation is cancelled, otherwise it continues. In the example application, a MessageDlg is used to prompt the user.

4.9Position Information

The other piece of status information reported by the example application is the current Caret position in line number, character number, co-ordinates. The FileEditor's OnCaretMoved event is called when the FileEditor receives the focus and whenever the Caret position changes. This event may therefore be used to generate status line information reporting the current Caret position of the active child form, and this is how it is used in the example application.

However, it should be noted that with large files, the FileEditor will estimate the current line number when a random jump is made into the file. This gives a very fast response, but requires a background "idle time" process to calculate the actual line number. In the example application, this situation is recognised by the response method for the OnCaretMoved event, and while the line number is estimated, a set of question marks is displayed. Once the line number has been properly calculated, OnCaretMoved is again called, enabling the proper line number to be indicated.

Of course to enable this to occur, the background process has to be set up. The FileEditor provides a HandleOnIdle method which needs to be assigned to the Application's OnIdle event and, once this is done, background calculation is automatic. In an SDI Application, this assignment can be done when the form is created. However, in an MDI Application, each MDI child has its own HandleOnIdle method and child forms may come and go during the lifetime of the task.

In the example application, this situation is handled by assigning HandleOnIdle to Application.OnIdle in the FileEditor's OnEnter event response method, and by assigning Application.OnIdle to nil in the FileEditor's OnExit method. This ensures that while a form has the focus, background calculation can take place and there is never a chance that a form can be destroyed while its HandleOnIdle method is still referenced by Application.OnIdle. Clearly, more complicated strategies are possible to enable background processing when the form does not have the focus, but this simple approach is satisfactory for most cases.

4.10Printer Setup

The example application also includes a Printer Setup Dialog. This is readily activated by calling its Execute method in the response method for the File|Print Setup dialog box. However, changing the current printer can affect a FileEditor object. This is because, if it is calculating word wrap on the page width of the *current* printer, then a change the current printer needs to be recognised by the FileEditor, so that the effective page width is changed.

TFileEditor provides a method (HandlePrinterChange) which may be called to alert the editor to a possible change in the current printer, and which will force it to re-compute the current page width. After the Printer Setup Dialog executes, the response method in the example application calls the HandlePrinterChange method for every active MDI child, thus ensuring that every instance of the FileEditor recognises the printer change. Note also that a check is made to ensure that an MDI Child Form is a TMDIChild (i.e. contains a FileEditor) before the method is invoked.

5Reference Manual

This chapter provides the definitions for each property, event and method made publicly available by the J-Write Components. In each the syntax is given, followed by a list of the classes to which the item applies. Notes on usage then follow.

5.1Public Properties

5.1.1ActualCharWidth

```
Property ActualCharWidth: integer
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This is the average character width in pixels on the current printing device.

5.1.2AltCopyKey

```
Property AltCopyKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the alternative copy to clipboard key.

5.1.3AltCutKey

```
Property AltCutKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the alternative key to cut out the current selection, if any, and copy it to the clipboard.

5.1.4AltPasteKey

```
Property AltPasteKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to paste the current contents of the clipboard in the cf_text format, if any, and insert it at the current caret position, replacing the current selection, if any. Note that the paste operation always inserts text and does not overwrite text other than the current selection.

5.1.5BackSpaceKey

```
Property BackSpaceKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to perform the backspace operation.

5.1.6BottomPrinterMargin

```
Property BottomPrinterMargin: TPrinterMargin
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The size of the bottom printer margin in inches.

5.1.7BusyCursor

```
Property BusyCursor: integer
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Cursor used when the viewer is in the busy state.

5.1.8CancelKey

```
Property CancelKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to remove (deselect) the current selection.

5.1.9Canvas

```
Property Canvas: TCanvas
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This is the current device canvas. Used to generate the font width tables, and tabstop tables and for text output.

5.1.10Caret

```
Property Caret: HBitmap
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The Bitmap or selector for the Caret.

5.1.11CaretPos

```
Property CaretPos: TPoint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the position on the screen of the Caret. In the X-direction, this is a pixel co-ordinate. In the Y-Direction, it is a line number.

5.1.12CharHeight

```
Property CharHeight: integer
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the character height allowed for under the current Font and is the height of each line in pixels.

5.1.13CharNumber

```
Property CharNumber: longint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the character on the current line immediately preceded by the Caret.

5.1.14ClickSize

```
Property ClickSize: integer
```

Applies to: TWinScrollBar

The clicksize is the amount by which the Position property is changed when the arrows at either end of the scroll bar are clicked.

5.1.15CopyKey

```
Property CopyKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the copy to clipboard key.

5.1.16CutKey

```
Property CutKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to cut out the current selection, if any, and copy it to the clipboard.

5.1.17DeleteKey

```
Property DeleteKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to delete either the current selection, if any, or the character immeidately after the caret.

5.1.18DeleteLineKey

```
Property DeleteLineKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to delete the line on which the caret is placed.

5.1.19DeleteWordKey

```
Property DeleteWordKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to delete the current word, if any, on which the caret is placed.

5.1.20DelLineEndKey

```
Property DelLineEndKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to delete all text from the current caret position, or start of selection, if text is selected, to the end of the current line.

5.1.21DelLineStartKey

```
Property DelLineStartKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to delete all text from the start of the current line to the current caret position, or end of selection if text is selected.

5.1.22DisplayCharWidth

```
Property DisplayCharWidth: integer
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This is the average character width in pixels on the current display device.

5.1.23DragCursor

```
Property DragCursor: integer
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the cursor used when a selected block of text is being dragged to a new location.

5.1.24Error

```
Property Error: TUUErrors
```

Applies to: TUUDecoder

This property contains the current error status code.

5.1.25EstimatedLineNumber

```
Property EstimatedLineNumber: boolean
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

When true this implies that the value of the LineNumber property inherited from TLinesViewer is an estimate and may change once it has been calculated exactly.

5.1.26FileName

```
Property FileName: String
```

Applies to: TFileViewer

This property contains the name of the file being viewed. To view a different file, simply assign a new value to this property.

5.1.27FileName

```
Property FileName: String
```

Applies to: TFileEditor, TMailEditor

This property contains the name of the file being edited. To edit a different file, simply assign a new value to this property. If the current file is modified then the OnSaveModifed event applies.

5.1.28HorzScrollBar

```
Property HorzScrollBar: TWinScrollBar
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The TWinScrollBar that encapsulates the TLinesViewer's horizontal scrollbar.

5.1.29InsertKey

```
Property InsertKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to toggle the insert mode.

5.1.30InsertOn

```
Property InsertOn: boolean
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This property holds the current insert mode. When true, new text is inserted at the current insertion point. When false, new text overwrites existing text.

5.1.31LastLine

```
Property LastLine: longint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The line number of the last line in the text.

5.1.32LeftArrowKey

```
Property LeftArrowKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret one character to the left.

5.1.33LeftPrinterMargin

```
Property LeftPrinterMargin: TPrinterMargin
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The size of the left printer margin in inches.

5.1.34Limit

```
Property Limit: longint
```

Applies to: TWinScrollBar

The limit is the maximum value the Position property can take. Zero is always the minimum value.

5.1.35LineDownKey

```
Property LineDownKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret one line down.

5.1.36LineEndKey

```
Property LineEndKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret to the rightmost position on the current line.

5.1.37LineLength

```
Property LineLength: longint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the length of the current line in characters.

5.1.38LineNumber

```
Property LineNumber: longint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the number of the "current line" i.e. the line on which the caret is currently placed.

5.1.39LineStartKey

```
Property LineStartKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret to the leftmost position on the current line.

5.1.40LineUpKey

```
Property LineUpKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret one line up.

5.1.41LineWidth

```
Property LineWidth: longint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the width in pixels of the current line.

5.1.42MailQuote

```
Property MailQuote: string
```

Applies to: TMailEditor

This is the Mail Quote string used for the Paste from the clipboard/insert text files operations, when it is required to precede each line of the inserted text with a "Mail Quotes" string.

5.1.43MaxLineLength

```
Property MaxLineLength: integer
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

When word wrap on a specified number of characters is selected, the value of this property is used as the maximum number of characters permitted to be in a line.

5.1.44MaxUndoLevels

```
Property MaxUndoLevels: integer
```

Applies to: TUndoList

This is the maximum number of undo transactions retained on the list, as a number between zero and *maxint*. If this is exceeded then the oldest transactions are deleted until the list is withn MaxUndoLevels.

5.1.45MaxUndoLevels

```
Property MaxUndoLevels: integer
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor, TEditStream

This is the maximum number of undo/redo transactions retained on the list. If this is exceeded then the oldest transactions are deleted until the list is withn MaxUndoLevels.

5.1.46MaxWidth

```
Property MaxWidth: longint
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This is the longest width (in pixels) a line is permitted to have before word wrap is applied. Zero implies no limit.

5.1.47Mode

```
Property Mode: TlbModes
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This is used for interpretation of MaxWidth.

5.1.48Modified

```
Property Modified: boolean
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

Set to true by a descendant class when the text buffer has been modified.

5.1.49NewLineKey

```
Property NewLineKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to insert a line break.

5.1.50NumberOfPages

```
Property NumberOfPages: integer
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the nufmber of pages of printable text contained in the current buffer.

5.1.51OutputDevice

```
Property OutputDevice: TOutputDevice
```

Applies to: TLinesBuffer, TTextStream, TEditStream

Set to odDisplay to use the display canvas as the output device, and set to odPrinter to use the current Printer.

5.1.52PageDownKey

```
Property PageDownKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret down by the number of lines on the screen.

5.1.53PageSize

```
Property PageSize: integer
```

Applies to: TWinScrollBar

The page size is the amount by which the Position property is changed when the scrollbar is clicked either side of the thumb.

5.1.54PageUpKey

```
Property PageUpKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret up by the number of lines on the screen.

5.1.55PasteKey

```
Property PasteKey: TKeyDefinition
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to paste the current contents of the clipboard in the cf_text format, if any, and insert it at the current caret position, replacing the current selection, if any. Note that the paste operation always inserts text and does not overwrite text other than the current selection.

5.1.56PercentDone

```
Property PercentDone: integer
```

Applies to: TProgressMeter

This is the percentage done. Setting this to a value between 0 and 100 updates the progress bar to show the percentage done in figures and inverts a corresponding proportion of the progress bar to give a graphical indication of progress.

5.1.57Position

```
Property Position: longint
```

Applies to: TWinScrollBar

The Position property reflects the current position of the Thumb. Assigning a value to Position within the valid range, explicitly changes the position of the scroll bar thumb and causes the OnScroll event to be called in order to change the view to correspond to the new position.

5.1.58PrinterCharWidth

```
Property PrinterCharWidth: integer
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This is the average character width in pixels on the current printer device.

5.1.59PrinterFooterMargin

```
Property PrinterFooterMargin: TPrinterMargin
```

Applies to:	TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The distance between the page footer and the bottom of the page in inches.

5.1.60PrinterHeaderMargin

```
Property PrinterHeaderMargin: TPrinterMargin
```

Applies to:	TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The distance between the page header and the top of the page in inches.

5.1.61QuickKey

```
Property QuickKey: TKeyDefinition
```

Applies to:	TKeyManager,TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the virtual key code plus modifiers for the Quick Key.

5.1.62RedoCount

```
Property RedoCount: integer
```

Applies to:	TEditStream

This is the current number of redo transactions.

5.1.63RightArrowKey

```
Property RightArrowKey: TKeyDefinition
```

Applies to:	TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret one character to the right.

5.1.64RightPrinterMargin

```
Property RightPrinterMargin: TPrinterMargin
```

Applies to:	TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The size of the right printer margin in inches.

5.1.65SaveMode

```
Property SaveMode: TSaveMode
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This property defines whether the text is saved as paragraph text (i.e. as it is in the text buffer); as Line oriented text (i.e. with line breaks inserted between each line displayed on the screen; or whether the user is prompted for the appropriate save mode (see the OnSaveModePrompt event).

5.1.66ScrollDownKey

```
Property ScrollDownKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to scroll down the screen by one line.

5.1.67ScrollUpKey

```
Property ScrollUpKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to scroll up the screen by one line.

5.1.68SelectionCursor

```
Property SelectionCursor: integer
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the cursor displayed while the cursor is over a selected block of text.

5.1.69ShowLineBreaks

```
Property ShowLineBreaks: boolean
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

When true, line breaks are shown using the '¶' character, and the end of text is shown using the '¤' character. Hard page breaks are shown as a solid line across the viewable area.

5.1.70ShowPageBreaks

```
Property ShowPageBreaks: TPageBreakDisplayModes
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The property determines when soft page breaks are shown. They may be never shown, always shown, or only shown when the word wrap mode is set to wrap on printer page width.

5.1.71Size

```
Property Size: longint
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This is a longint holding the current size of the text buffer. Note: Text[Size-1] is the last character in the Text buffer.

5.1.72Size

```
Property Size: longint
```

Applies to: TSegmentList, TFreeList, TVirtualStreamList, TUndoInfo

This is the total size of all of the segments in the list.

5.1.73SourceStream

```
Property SourceStream: TStream
```

Applies to: TStreamViewer, TFileViewer

This property has the current stream as its value. To view a different stream, simply assign a new stream object to the property. Note that the stream is freed when the viewer is closed or a new stream assigned to this property.

5.1.74SourceStream

```
Property SourceStream: TStream
```

Applies to: TStreamEditor, TFileEditor, TMailEditor

This property has the current stream as its value. To edit a different stream, simply assign a new stream object to the property. Note that the stream is freed when the viewer is closed or a new stream assigned to this property.

5.1.75SourceStream

```
Property SourceStream: TStream
```

Applies to: TTextStream, TEditStream

This is the stream which is presented by the TTextViewer as line oriented text.

5.1.76State

```
Property State: TViewerStates
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This property is the current set of TLinesViewer states and may be used when displaying status information and determining the state of menu items.

5.1.77Stream

```
Property Stream: TStream
```

Applies to: TFreeList, TVirtualStreamList, TUndoInfo

This read only property is the stream from which allocations are made. It may be a memory or a disk stream.

5.1.78TabCount

```
Property TabCount: integer
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor,TLinesBuffer, TTextStream, TEditStream

This property specifies the number of tab stops defined in the TapStop indexed property.

5.1.79TabStop[]

```
Property TabStop[n: integer]
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor, TLinesBuffer, TTextStream, TEditStream

This indexed property contains the defined tab stops in inches. The index must be in the range 0..TabCount-1, and the tab stops must be defined in order of increasing value.

5.1.80Text[]

```
Property Text[Pos: longint]: PChar
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This read only property is indexed by a longint and returns PChar to character in the idealised text buffer indexed by *Pos*.

5.1.81TextEndKey

```
Property TextEndKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret to the last line.

5.1.82TextSize

```
Property TextSize: longint
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The number of characters in the text including any line separator characters.

5.1.83TextStartKey

```
Property TextStartKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret to the first line.

5.1.84TopPrinterMargin

```
Property TopPrinterMargin: TPrinterMargin
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The size of the top printer margin in inches.

5.1.85Tracking

```
Property Tracking: boolean
```

Applies to: TWinScrollBar

If Tracking is true then the scroll bar reacts to the thumb being dragged by updating the Position property and calling the OnScroll event. If false, then any reaction is deferred until the drag is finished and the mouse button creases to be pressed down.

5.1.86UndoCount

```
Property UndoCount: integer
```

Applies to: TEditStream

This is the current number of undo transactions.

5.1.87Valid

```
Property Valid: TValidText
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This is a record holding two PChars: “First” and “Last” giving inclusive range of the current valid pointers to the idealised text buffer.

5.1.88VertScrollBar

```
Property VertScrollBar: TWinScrollBar
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The TWinScrollBar that encapsulates the TLinesViewer's vertical scrollbar.

5.1.89Visible

```
Property Visible: boolean
```

Applies to: TWinScrollBar

This property reflects the current Visible state of the scroll bar. This can be changed explicitly. However, the Visible state is always re-computed every time Limit is changed, such that Visible is set to true if and only if Limit is greater than the PageSize.

5.1.90WordLeftKey

```
Property WordLeftKey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret one word to the left.

5.1.91WordRightKey

```
Property WordRightkey: TKeyDefinition
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This is the Virtual Key Code plus modifiers for the key to move the caret one word to the right.

5.1.92WPMode

```
Property WPMode: boolean
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

When true the Caret wraps round from the last character on one line to the first character on the next line, and cannot be placed in the white space to the right of the end of a line. When false, the Caret can enter the white space and does not wrap around.

5.1.93WrapMode

```
Property WrapMode: TWrapModes
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This property determines how the text is organised into lines (i.e. wrapped to fit on the screen). Either no word wrap is displayed (i.e. lines may overflow the screen until a had line break is encountered); word wrap is on window boundaries; word wrap is after a specified number of characters (see the MaxLineLength property); or word wrap is on printer page width.

5.2Events

5.2.1OnAbortRequest

```
Property OnAbortRequest: TAbortEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called to enable the user to confirm a request to abort a busy state operation. The abort is requested by pressing the excape key which is regularly checked during this state. When the event returns, the result is checked to see if the user really does want to abort. A response of mrYes confirms the abort, mrNo requests continue. If this event handler is not defined then the result is assumed to be mrYes.

5.2.2OnBlockDecode

```
Property OnBlockDecode: TProgressEvent
```

Applies to: TUUDecoder

This event provides a notification of progress through a text buffer during a uudecode operation. It may be used, for example, to update a progress indicator.

5.2.3OnBlockEncode

```
Property OnBlockEncode: TProgressEvent
```

Applies to: TUUEncoder

This event provides a notification of progress through a text buffer during a uuencode operation. It may be used, for example, to update a progress indicator.

5.2.4OnCaretMoved

```
Property OnCaretMoved: TNotifyEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called to report a change in the position of the caret and may be used to provide status line information on the caret co-ordinates.

5.2.5OnChange

```
Property OnChange: TNotifyEvent
```

Applies to: TWinScrollBar

This event occurs whenever the scroll bar state changes, including the Visible state.

5.2.6OnChange

```
Property OnChange: TNotifyEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called whenever text changes in the underlying text container.

5.2.7OnChange

```
Property OnChange: TNotifyEvent
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This event is called to report that text in a text buffer has been changed.

5.2.8OnEndDecode

```
Property OnEndDecode: TUUEndDecodeEvent
```

Applies to: TUUDecoder, TMailEditor

This event provides a notification of the completion of a uuDecode operation and reports the result of the operation. The user may thus be informed as to whether the decode operation was successful.

5.2.9OnEnter

```
Property OnEnter: TNotifyEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called whenever the TLinesViewer gets the focus.

5.2.10OnExit

```
Property OnExit: TNotifyEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called whenever the TlinesViewer loses the focus.

5.2.11OnKeyCode

```
Property OnkeyCode[const KeyDef: TKeyDefinition]:
      TKeyEventHandler
```

Applies to: TKeyManager

This is a write only indexed property. The index is a virtual key code plus any modifiers required, and the assigned value is the handler for that key code. Assiging a value of nil removes the key code from the list maintained by TKeyManger.

5.2.12OnModified

```
Property OnModified: TNotifyEvent
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

Event handler for change to modified status.

5.2.13OnNewFileName

```
Property OnNewFileName: TNotifyEvent
```

Applies to: TFileEditor

This event provides a notification of a change in the filename property. It may, for example, be used to set the window caption.

5.2.14OnPagePrint

```
Property OnPagePrint: TPrintEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called every time a page is printed and may be used to report the number of pages printed.

5.2.15OnProgressEvent

```
Property OnProgressEvent: TProgressEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called to report progress during a lengthy operation and while in the busy state. It may be used to update a progress bar.

5.2.16OnReplaceAll

```
Property OnReplaceAll: TReplaceAllEvent
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called to report the completion of a Replace All action. It may be used to report the number of items replaced.

5.2.17OnSaveModePrompt

```
OnSaveModePrompt: TSavePromptEvent
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event is called when the file is saved and gives a chance for the user to override the current save mode when appropriate. This event is most useful when SaveMode is set to prompt the user for the appropriate save mode, and should then return the save mode as paragraph or line oriented text. In this case, this event provides a mechanism for an appropriate prompt to be displayed to the user and for them to indicate the preferred save mode. If no event handler is defined and the SaveMode is set to prompt the user, then save as line oriented text is assumed.

5.2.18OnSaveModified

```
Property OnSaveModified: TSaveModifiedEvent
```

Applies to: TFileEditor, TMailEditor

This event occurs when the clear method is called, or a new filename assigned, and the file currently being edited has been modified. It should return an indication of the action to take as either mrYes (save the file by calling the Save method), mrNo (don't save the file), or mrCancel (abandon the Clear or change in file. This event permits a simple dialog box to be displayed, prompting the user to save the file, with Yes/No/Cancel as the options.

5.2.19OnSaveNewFile

```
Property OnSaveNewFile: TNotifyEvent
```

Applies to: TFileEditor, TMailEditor

This event provides a notification that the Save method has been called when the FileName property is empty, or when the SaveAs method has been called with an empty filename parameter. The event handler should prompt the user for the filename under which to save the file, and then call SaveAs again in order to perform the save. If no event handler is defined then the save operation fails. This event should always be handled as it is the mechanism by which an untitled file is saved and the user prompted for a filename.

5.2.20OnSaveUndo

```
Property OnSaveUndo: TUndoSaveUndoEvent
```

Applies to: TUndoList

This event handler is called when a new undo operation is created. It is expected to return the current viewer state information to be saved with the undo operation.

5.2.21OnSaveUndo

```
Property OnSaveUndo: TSaveUndoEvent
```

Applies to: TEditStream

This event handler is used to report that Undo information is about to be saved. Parameters may be used to return the current selection and modification status for saving with the Undo information.

5.2.22OnScroll

```
Property OnScroll: TNotifyEvent
```

Applies to: TWinScrollBar

This event is occurs whenever the Position of the thumb is changed.

5.2.23OnSearchStatus

```
Property OnSearchStatus: TSearchStatus
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This event reports progress on a long search

5.2.24OnStartDecode

```
Property OnStartDecode: TUUDecodeEvent
```

Applies to: TUUDecoder, TMailEditor

This event provides a notification of when a uuDecode header has been found in the text (when the uuDecode method has been called) and gives the user the opportunity to change the filename/location for the uudecoded file, or to skip the uudecode.

5.2.25OnStateChange

```
Property OnStateChange: TNotifyEvent
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

The TLinesViewer maintains a State variable (a Set) which includes state information for editors as well as Viewers, and may be set by descendant classes, as well as TLinesViewer. Whenever the State variable is updated, this event handler is called. It may, for example, be used to update menu items and/or a status line.

5.2.26OnUndo

```
Property OnUndo: TUndoUndoEvent
```

Applies to: TUndoList

This event handler is called when an undo operation is performed and notifies the handler of the viewer state that is to be restored.

5.2.27OnUndo

```
Property OnUndo: TUndoEvent
```

Applies to: TEditStream

This event handler is used to report the Undo operation. Parameters of call report the selection status and modification status at the time that the undo information was saved, and the number of undo records remaining

5.2.28OnUndoChange

```
Property OnUndoChange: TNotifyEvent
```

Applies to: TUndoList

This event handler is called when the undo transaction count changes. It may be used to change status information, e.g. the state of menu items.

5.2.29OnUndoChange

```
Property OnUndoChange: TNotifyEvent
```

Applies to: TEditStream

This event handler is called when the undo transaction count changes. It may be used to change status information, e.g. the state of menu items.

5.2.30OnWrapModeChange

```
Property OnWrapModeChange: TNotifyEvent
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This event provides a notification of a change in the current wrap mode and may be used to (e.g.) update a status bar.

5.3Public Procedures

5.3.1AddBuffer

```
procedure AddBuffer(var Buffer; BufLen: Word);
```

Applies to: TSumCheck

This method adds the binary values of each byte in the buffer to the sum check.

5.3.2AddLine

```
procedure AddLine(const S: String);
```

Applies to: TSumCheck

This method adds the ASCII values of the characters in S to the sumcheck, adding an assumed LF separator to the end of the string.

5.3.3Append

```
procedure Append(P: TStreamSegment);
```

Applies to: TSegmentList, TFreeList, TVirtualStreamList, TUndoInfo

This method appends P to the end of the TSegmentList.

5.3.4AtPageBreak

```
function AtPageBreak(Pos: longint): boolean;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method returns true if the character at Pos is a Form Feed.

5.3.5BackSpace

```
procedure BackSpace;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method moves the caret back one character and then deletes the next character. The exception is if the caret is at the start of the line when it wraps back to the end of the preceding line. Note that a check is made to see if preceding line was wrapped or ended in a real CR/LF. In the former case, the caret is stepped back passed the last character on the line, in the latter case, the caret is placed at the end of the line to ensure that the line break is deleted.

5.3.6Capitalise

```
procedure Capitalise;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is used to capitalise either the current word (i.e. the word on which the caret is located) or all words in the selection. If the first word is already capitalised then all words are converted to all upper case. If the second letter is already upper case, then all are converted to lower case. Note that the Ansilower and upper functions are used to ensure international character set support.

5.3.7Clear

```
procedure Clear; virtual;
```

Applies to: TSegmentList, TFreeList, TVirtualStreamList, TUndoInfo

This method deallocates all segments on the TSegmentList and re-initialises the list to the initial state.

5.3.8ClearLastUndoLink

```
procedure ClearLastUndoLink; virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method forces the undo operation at the top of the undo stack to be a separate undo transaction from any undo operations saved afterwards. Note that the method is defined as an abstract method by TTextBuffer and overridden by TEditStream.

5.3.9ClearTransactionBoundary

```
procedure ClearTransactionBoundary;
```

Applies to: TUndoList

The "Linked" field in each member of an undo list is used to indicate the boundary between undo transactions. When true, it implies that this undo operation is linked to the one that precedes it in the list (i.e. nearer top of stack) and should be processed in the same undo operation. This method removes a transaction boundary and effectively joins the current undo operation to whatever undo operation is later added to the top of the stack.

5.3.10Close

```
procedure Close;
```

Applies to: TLinesViewer

This method closes the window by sending it a wm_close message

5.3.11Copy

```
constructor copy(P: TStreamSegment);
```

Applies to: TStreamSegment

The Copy constructor creates a new object which is a copy of P.

5.3.12Copy

```
constructor copy(P: TAllocatedSegment);
```

Applies to: TAllocatedSegment

The Copy constructor creates a new object which is a copy of P.

5.3.13CopyOf

```
function copyOf: TStreamSegment; virtual;
```

Applies to: TStreamSegment, TAllocatedSegment

This is a virtual method and creates a copy of the the object using its copy constructor. Note that every descendant object that defines a copy constructor overrides this method to call its own copy constructor.

5.3.14CopyText

```
procedure CopyText(Dest: pointer; Pos, Length: longint);
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method copies 'Length' characters starting at Pos from the text buffer to a memory block pointed to by 'Dest'. This memory block may be a huge memory block i.e. exceeding 64KB. Note that the method is defined as an abstract method by TTextBuffer and overridden by TTextStream and TEditStream.

5.3.15CopyToClipBoard

```
procedure CopyToClipBoard; virtual;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

Copies the current selection to a memory area and passes it to the clipboard

5.3.16CopyToMem

```
procedure CopyToMem(Mem: THandle; Pos, Length: longint);
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method copies Length characters of text at position Pos to the memory segment Mem.

5.3.17CopyToStream

```
procedure CopyToStream(S: TStream; Pos, Length: longint);
virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method copies 'Length' characters from the Text Buffer, starting at Pos to the stream S. Note that the method is defined as an abstract method by TTextBuffer and overridden by TTextStream.

5.3.18CopyToStream

```
procedure CopyToStream(S: TStream; Count: Longint);
```

Applies to: TVirtualStreamList

This method copies 'Length' characters from the virtual stream, starting at Pos to the stream S.

5.3.19Cut

```
function Cut(Length: Longint): TSegmentList;
```

Applies to: TVirtualStreamList

This method cuts out a region of 'Length' characters from the virtual stream, starting at the current position, and returns the cut out stream segments as a TSegmentlist, reflecting their order in the stream.

5.3.20CutToClipBoard

```
procedure CutToClipBoard;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method copies the current selection to the clipboard and then deletes it. Note that an exception will occur (and hence the deletion skipped if the copy fails).

5.3.21DeAllocate

```
procedure DeAllocate(List: TSegmentList); virtual;
```

Applies to: TStreamSegment, TAllocatedSegment

This method is called when a segment is deallocated from a list. By default the segment is freed, However, the method is overridden by TAllocatedSegment to return the segment to its freelist.

5.3.22Delete

```
procedure Delete(Pos, Count: longint); virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method is called to delete "count" characters from the Text Buffer, starting at Pos. Note that the method is defined as an abstract method by TTextBuffer and overridden by TEditStream.

5.3.23DeleteChar

```
procedure DeleteChar;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method deletes the character at the current cursor position or the current selection, if text is selected.

Deletion of a single character may take place at

1) a hard line break, when the number of lines in the file will decrease by one unless the hard line break is replaced by a soft line break.

2) a soft line break, when line wrap may occur.

3) or in the middle or start of a line, when again, line wrap may occur. In this case, deletion of a word break character may result in word wrap in the preceding line.

5.3.24DeleteLine

```
procedure DeleteLine;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method deletes all characters on the current line (as given by the *LineNumber* property), including the line break, if any.

5.3.25DeleteLineEnd

```
procedure DeleteLineEnd;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method deletes all characters from the caret position to the end of the line, including selected text, if any.

5.3.26DeleteLineStart

```
procedure DeleteLineStart;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method deletes all characters from the caret position to the start of the line, including selected text, if any.

5.3.27DeleteSelection

```
procedure DeleteSelection;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method deletes all text in the current selection, if any.

5.3.28DeleteWord

```
procedure DeleteWord;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method deletes the word at the current caret position, if any.

5.3.29Find

```
function Find(Sender: TFindDialog; const AFindStr: String;
                        SearchOptions: TFindOptions): boolean;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

Find is used in conjunction with TFindDialog to search the text for a string of characters. The string to search for is given by *AFindStr*, and the search options selected by the user, in *SearchOptions*. *Sender* should either be set to nil, if a TFindDialog is not used as the dialog box, to should be set to the TFindDialog object.

Find returns true if successful, and false otherwise.

Example:

```
procedure TMDIChild.FindDialog1Find(Sender: Tobject);
begin
     With Sender As TFindDialog Do
     If not FileEditor1.Find(TFindDialog(Sender),FindText,Options)
           Then MessageDlg('Text not Found',mtError,[mbOK],0)
end;
```

5.3.30FindNext

```
function FindNext(Sender:TFindDialog): boolean; virtual;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

FindNext is called to repeat the previous Find operation, if any. It is typically called as the result of a menu selection, in which case the *Sender* is usually set to nil.

5.3.31FlowText

```
procedure FlowText;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method removes all single Line Breaks from the current selection, or from the Caret position to end of text, if no selection. Double line breaks are left intact. The result is text organised into paragraphs rather than lines.

5.3.32GetCharAt

```
function GetCharAt(var Pos: longint; Distance: longint; var
ActualOffset: longint): boolean;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method is used to find the character at a given pixel offset from another character on the same line. The pixel offset is always that of the display device, not the printer.

On input, Pos is the index of the starting character, and Distance is the number of pixels away from that point of the character to be located.

Returns: true, then ActualOffset holds the actual number of pixels between the two characters and Pos is adjusted to index the identified character.

false, if Pos is not a valid index.

5.3.33GetDistanceBetween

```
function GetDistanceBetween(StartPos, EndPos: longint): longint;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method works out the distance between StartPos and EndPos on the current canvas, and returns the result, in pixels. If StartPos >= EndPos then the distance is always zero. Result is always bounded by MaxWidth (in pixels). Distance includes the character indicated by startpos and all characters up to but not including EndPos. The pixel offset is always that of the display device, not the printer.

5.3.34GetInitialFindStr

```
function GetInitialFindStr: string; virtual;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

GetInitialFindStr returns the word under the caret, if any. This method is typically used prior to executing a TFindDialog box, in order to initialise the find string to the word under the caret.

Example:

```
procedure TMDIChild.Find1Click(Sender: Tobject);
begin
     FindDialog1.FindText := FileEditor1.GetInitialFindStr;
     FindDialog1.Execute
end;
```

5.3.35GetNextLine

```
function GetNextLine(var Pos: longint; var LineWidth: longint;
                        var LineLength: longint): boolean;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method scans the buffer starting at “Pos” looking for the next end of line. The function returns true if found, or false if already at the end of the text buffer. If successful, the value of “Pos” on return is the position of the next line in the Text array. One return, LineLength is number of characters in line TextWidth is display width of line - interpretation depends on Mode.

On Entry: Pos is index of start position

On Return: If true, Pos is index of start of next Line

LineLength is number of characters in line

LineWidth is display width of line

If false, already at end of text buffer

5.3.36GetStartOfParagraph

```
Procedure GetStartOfParagraph(var Pos: longint);
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method scans back from Pos for the start of the previous line i.e. the character following the preceding LF character. If Pos is immediately preceded by a CR/LF or a LF, then this is skipped. On return, Pos is set to the character index of the first character of the line. Note: if Pos is zero on entry, then this method is a no-op.

On Entry Pos = start index

On Return: Pos is index of next character after preceding LF, or start of text buffer (i.e. 0).

5.3.37GetString

```
function GetString(Pos: longint; Length: integer): string;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method returns the string of length “Length” at “Pos”. Note: the returned string may be shorter if end of text buffer is encountered.

5.3.38GetSum

```
function GetSum: String;
```

Applies to: TSumCheck

This method returns the current sum check values as a formatted string. This will be identical to the result obtained from the Unix 'sum' utility.

5.3.39GetUndoList

```
function GetUndoList: TUndoInfo;
```

Applies to: TUndoList

This method returns the TUndoIndo object at the top of the undo stack. Note that it also calls the OnUndo event, as the semantic of this call is to initiate an undo operation.

5.3.40GotoLineNumber

```
procedure GotoLineNumber(LineNo: Longint);virtual;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is intended to be called from a user dialog box to position the cursor explicitly on the line number given by *LineNo*. Note that this method uses line numbers relative to zero i.e. line 0 is the first line.

5.3.41HandleOnIdle

```
procedure HandleOnIdle(Sender: TObject; var Done: Boolean);
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is intended to be called during a TApplication Idle procedure and is used to support background calculation of the number of lines in the text, and the current line number when it is estimated.

5.3.42HandlePrinterChange

```
procedure HandlePrinterChange(Sender: TObject);
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is called by a user to notify the TTextViewer than the Printer has changed. The various metrics that depend upon the current printer are then updated. Note that this is defined as a TNotifyEvent so that it may be used as the handler for a notification event.

5.3.43Insert

```
procedure Insert(const buf; Pos: longint; Count: longint);
```

```
virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

Inserts count characters from buf into the text stream at Pos. Note that the method is defined as an abstract method by TTextBuffer and overridden by TEditStream.

5.3.44Insert

```
Procedure Insert(P: TStreamSegment);
```

Applies to: TVirtualStreamList

This method inserts a single TStreamSegment into the list at the current position.

5.3.45InsertAfter

```
procedure InsertAfter(P, NewRec: TStreamSegment);
```

Applies to: TSegmentList, TFreeList, TVirtualStreamList, TUndoInfo

This method inserts a File segment (NewRec) into the list after P. If P = nil then NewRec is inserted at head of list.

5.3.46InsertChar

```
procedure InsertChar(C: Char);
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method inserts the character ‘C’ at the current insertion point, or replaces the current selection, or the current character if in overwrite mode. Note that this method is optimised to provide a short path for character insertion. It therefore does its own checking for the need to update the object’s state variables, and which parts of the screen need to be redrawn.

5.3.47InsertCompressedFile

```
function InsertCompressedFile(CommandLine: String;
             const Extension, FileName: string): TEncodeResult;
```

Applies to: TMailEditor

This method is called to insert the contents of a named file at the current insertion point. Before being inserted, the file is compressed using a specified compression utility and the result uuencoded. The pathname of the file is given in 'FileName', and the DOS command line to invoke the compression utility is given in 'CommandLine'. The extension to be used for the compressed file (replaces the current extension of 'FileName', is given by 'Extension'. Note that 'CommandLine' must include the strings &Z and &F. These are replaced by the name of the temporary file to hold the compressed data, and the name of the input file, respectively.

e.g. if pkzip is the compression utility, then a typical command line will be:

```
pkzip &Z &F
```

5.3.48InsertDateTime

```
procedure InsertDateTime(DateTime: TDateTime);
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method inserts the specified date and time at the current insertion point, in the format determined by international settings. The method is provided to ensure the J-Write editor is a complete replacement for Notepad.

5.3.49InsertEOL

```
procedure InsertEOL;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method inserts a line break at the current caret position.

5.3.50InsertFile

```
procedure InsertFile(const FileName: String;
                                ConvertToANSI: boolean);
```

Applies to: TMailEditor

This method is called to insert the contents of the text file 'FileName' at the current insertion point. If 'ConvertToANSI' is true then the text is assumed to be in the OEM (DOS) character set and converted to the ANSI character set when it is inserted.

5.3.51InsertListAfter

```
procedure InsertListAfter(P: TStreamSegment; List: TSegmentList);
```

Applies to: TSegmentList, TFreeList, TVirtualStreamList, TUndoInfo

This method inserts the members of ‘List’ after P in the current list. If P = nil then the List is inserted at start. Note that List is always freed when this method completes.

5.3.52InsertMem

```
function InsertMem(M: THandle; Pos: longint): longint; virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method is defined as an abstract method by TTextBuffer and overridden by TEditStream. The semantic is that the contents of the memory segment 'M' are inserted into the Text Buffer at 'Pos'. Note that 'M' is NOT disposed of by this method, but may be disposed of by the caller if it is no longer needed.

5.3.53InsertQuoteFile

```
procedure InsertQuoteFile(const FileName: String);
```

Applies to: TMailEditor

This methods inserts the content of the text file given by 'FileName', and inserts the current MailQuote string (as given by the property of the same name), at the beginning of each line.

5.3.54InsertSegmentList

```
procedure InsertSegmentList(P: TSegmentList);
```

Applies to: TVirtualStreamList

This method is like Insert, except that a whole segment list is inserted into the virtual stream.

5.3.55InsertSpaces

```
procedure InsertSpaces(Pos,Count: longint);
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method inserts 'Count' spaces at text buffer position 'Pos'.

5.3.56InsertString

```
procedure InsertString(const S: string; Pos: longint);
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This methods inserts the string 'S' at text buffer position 'Pos'.

5.3.57IsEditable

```
function IsEditable: boolean; virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

By default, this method returns false, implying that the object does not support the editing methods (e.g. Insert, Delete, etc.). This is overridden by descendants that do support these methods and enables a user to dynamically determine if these methods are supported.

5.3.58IsLinked

```
function IsLinked: boolean;
```

Applies to: TUndoList

Returns true if the undo operation at the top of the stack has the "linked" flag set. This implies that it is linked to the previous Undo transaction. When performing an Undo Transaction, the user should iteratively call GetUndoList to initiate and perform undo operations until IsLinked returns false.

5.3.59LineBreakAt

```
function LineBreakAt(Pos: longint): boolean;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method is used to determine if a hard line break is present at position ‘Pos’, and returns true if Pos indexes a CR/LF, LF or FF..

5.3.60Merge

```
function Merge(P: TStreamSegment): boolean;
```

Applies to: TStreamSegment, TAllocatedSegment

This method attempts to merge P with this file segment. Merging is successful if they are on the same stream and describe adjacent segements in the stream. If successful. true is return, the file segment is updated, and P is disposed of.

5.3.61PasteFromClipBoard

```
procedure PasteFromClipBoard;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

If the current clipboard selection is in text format, then it is inserted into the file at the current caret position.

5.3.62PasteQuoteFromClipboard

```
procedure PasteQuoteFromClipboard;
```

Applies to: TMailEditor

This method inserts the content of the clipboard (if in cf_text format) at the current insertion point, and inserts the current MailQuote string (as given by the property of the same name), at the beginning of each line.

5.3.63PopUndoStack

```
function PopUndoStack: TUndoInfo;
```

Applies to: TUndoList

This method removes the topmost UndoInfo object, and returns the object as the function value.

5.3.64Prepend

```
procedure Prepend(P: TStreamSegment);
```

Applies to: TSegmentList, TFreeList, TVirtualStreamList, TUndoInfo

This method inserts P at head of the TSegmentlist.

5.3.65Print

```
procedure Print(PageTitle, Caption: string; PageNumbers: boolean;
                 WrapToPage: boolean; PrintRange: TPrintRange;
                 FromPage, ToPage: integer);
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method prints all lines of text on the current printer. If *title* is not empty, then this is printed at the top of each page. If *PageNumbers* is true, then page numbers are printed at the bottom of each page. If *WrapToPage* is true then word wrap is set to Wrap on Printer Page Width prior to printing. Otherwise the current word wrap mode is used, which may result in text overflowing the page. *Caption* is used as the name of the print job.

PrintRange determines whether all the text is printed, the current selection, or the range of pages given by *FromPage* to *ToPage*.

5.3.66PushUndoStack

```
procedure PushUndoStack(AUndoPos: Longint);
```

Applies to: TUndoList

This method creates a new Undo Info object and puts in on the "top" of the Undo List. If the list size exceeds maxUndoLevels, then the oldest entry is disposed of.

The input parameter AUndoPos is taken as defining the position in the text buffer at which the undo transaction is located. Other editor state information is determined by calling the OnSaveUndo event.

5.3.67QueryUndo

```
function  QueryUndo(var AUndoPos, AUndoSize,
                        AInsertSize: longint): boolean;
```

Applies to: TUndoList

This method is called to determine the information held by the undoinfo object at the top of the stack, without initiating an undo operation. Returns false if stack empty.

5.3.68Read

```
function Read(var Buf; From, Count: longint): longint;
```

Applies to: TStreamSegment, TAllocatedSegment

This method is used to read an area of the file segment into Buf.

5.3.69Read

```
function Read(var Buf; Count: longint): longint;
```

Applies to: TVirtualStreamList

This method reads Count characters into 'buf' starting from the current position in the virtual stream.

5.3.70Redo

```
procedure Redo;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is called to perform the most recent redo operation. This reverses a preceding Undo operation.

5.3.71Redo

```
function Redo:longint; virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method is called to carry out the most recent Redo Operation, and returns the position at which the redo took place.

5.3.72Refresh

```
procedure Refresh;
```

Applies to: TWinScrollBar

Refresh updates the “Visible” state of the scrollbar. The Scroll Bar is made visible, if and only if the Limit property > PageSize - 1

5.3.73Replace

```
function Replace(Sender: TFindDialog; const FindStr,
                  ReplaceStr: string;
                  SearchOptions: TFindOptions): boolean;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is called to perform a replace operation, and is normally used in conjunction with the TReplaceDialog standard dialog. The function will replace the next occurrence of *FindStr* with the contents of *ReplaceStr*, according to the *SearchOptions*.

Example:

```
procedure TMDIChild.ReplaceDialog1Find(Sender: Tobject);
begin
     With Sender As TReplaceDialog Do
           If not FileEditor1.Replace(TFindDialog(Sender),
                                  FindText, ReplaceText,Options) Then
                 MessageDlg('Text not found',mtError,[mbOK],0)
end;
```

5.3.74ReplaceChar

```
procedure ReplaceChar(Pos: longint; c: char); virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method replaces the character in the text buffer at position 'Pos' with the character 'c'. The replacement is recorded as an undo transaction. Note that the method is defined as an abstract method by TTextBuffer and overridden by TEditStream.

5.3.75Reset

```
procedure Reset; virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

Reset is called to clear down the buffer and restore it to its initial state.

5.3.76RunDosProgram

```
function RunDosProgram(const CommandLine: string): boolean;
```

Applies to: JWRUN Module

This function is called to run a DOS program. The program is run synchronously (i.e. the function waits for the program to complete) in a separate virtual machine. The function returns true if program complete without errors.

5.3.77Save

```
procedure Save;
```

Applies to: TFileEditor, TMailEditor

This method saves the current contents of the editor. If a named file is being edited, then the file is renamed with the ".bak" extension, and replaced with the saved text. Otherwise, a new file is created.

5.3.78SaveAs

```
procedure SaveAs(const AFileName: String); virtual;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method saves the text to the file given by *AFileName*.

5.3.79SaveBuf

```
function SaveBuf(const Buf; Count: longint): TStreamSegment;
```

Applies to: TFreeList

This method is used to save Count bytes in Buf to a TStreamSegment allocated from the freelist. The function returns the allocated TStreamSegment if successful, and otherwise returns nil.

5.3.80SaveInsertInfo

```
procedure SaveInsertInfo(Pos, InsertCount: longint);
```

Applies to: TUndoList

This method saves insert information into the current Undo operation. The semantic is that the method records the 'InsertCount' characters have been inserted at position 'Pos' in the text buffer. If

'Pos' is not the current undo position, then a new Undo operation is created at the top of the stack, recording 'Pos' as the undo position.

5.3.81SaveUndo

```
procedure SaveUndo(List: TSegmentList; Pos: longint);
```

Applies to: TUndoList

This method saves characters deleted from the text buffer into the current Undo operation. The semantic is that the TSegmentList 'List' has been 'Cut' out of the text buffer at position 'Pos'. If 'Pos' is the current Undo position and the insert count for this undo operation is zero, then 'List' is appended to the current list of deleted text segments. Otherwise a new Undo operation is created at the top of the stack, recording 'Pos' as the undo position.

5.3.82ScrollMessage

```
procedure ScrollMessage(Msg: TWMScroll);
```

Applies to: TWinScrollBar

This method must be called from owning control in response to a wm_vscroll or a wm_hscroll message directed to a window’s scrollbar, in order to process scroll bar events. The method parameter is set to the current message.

Example:

```
procedure TLinesViewer.WMHScroll(var Msg: TWMScroll);
Begin
    If Msg.ScrollBar = 0 Then HorzScrollBar.ScrollMessage(Msg)
    Else inherited
End;
```

5.3.83Search

```
function Search(var Pos: longint; const FindStr: String;
                    SearchOptions: TFindOptions):boolean;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method is used to scan the text buffer starting at “Pos”, for the string “FindStr”. Depending on the SearchOptions, the search may be forwards, or backwards (if search backwards is true), for whole words only or any occurrence of the string, and case sensitive or case blind (case blind true). If successful, function returns true, and “Pos” contains the index of the found string. If unsuccessful, then function returns false.

5.3.84Seek

```
Procedure Seek(At: longint);
```

Applies to: TVirtualStreamList

This method is used to set the current position to 'At'. The internal pointer 'Current' is set to the TStreamSegment in which At is located, and Offset is set to the difference between the offset of the current segment and 'At'.

5.3.85SelectAll

```
procedure SelectAll;
```

Applies to: TLinesViewer, TTextViewer, TStreamViewer, TFileViewer, TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method selects all text currently being viewed/edited.

5.3.86SetTransactionBoundary

```
procedure SetTransactionBoundary;
```

Applies to: TUndoList

This method is the reverse of ClearTransactionBoundary, and is used to ensure that the undo operation at the top of the stack starts a new undo transaction.

For example, when the editor performs a multiple set of operations (e.g. replace all) it is not known until after the last operation was performed, that it was the operation. The technique is therefore to set each corresponding undo operation as linked to the next, and when it is determined that the operation is complete, to call this method to unlink the operation on the top of the stack and thereby create an undo transaction boundary.

5.3.87SkipBackOverLineBreak

```
function SkipBackOverLineBreak(Pos: longint): longint;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

This method skipd back passed a line break that immediately precedes Pos, if any. On return, the function value is the position of the last character in the preceding line. If 'Pos' is not preceded by a line break then the function returns 'Pos'.

5.3.88Split

```
function Split(At: longint): TStreamSegment;
```

Applies to: TStreamSegment, TAllocatedSegment

This method is used to split a TStreamSegment in two, at offset "At". The function returns the file segment at "At".

5.3.89TabbedTextOut

```
procedure TabbedTextOut(X,Y: integer; Pos, Length: longint;
                              TabOrigin: integer);
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method is called to draw the text string starting at Pos, and of the specified 'Length', on the current Canvas at position (X,Y), expanding tabs as required. The TabOrigin defines the offset from which the tab stops are computed.

5.3.90TextOut

```
procedure TextOut(X,Y: integer; Pos, Length: longint);
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method is called to draw the text string starting at Pos, and of the specified 'Length', on the current Canvas at position (X,Y).

5.3.91ToggleInsertMode

```
procedure ToggleInsertMode;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method toggles the insert mode from insert to overwrite mode, and back again.

5.3.92ToggleWrapMode

```
procedure ToggleWrapMode;
```

Applies to: TTextViewer, TStreamViewer, TFileViewer, TBigEditor,
TStreamEditor, TFileEditor, TMailEditor

This method changes the value of the WrapMode property to the next in the enumerated type, wrapping round to the first, if the current value is the last.

5.3.93TranslateKey

```
function TranslateKey(KeyCode: word; Shift: TShiftState):
boolean;
```

Applies to: TKeyManager

This method is called by a control in response to a wm_keydown message. The method searches the list of entries for a corresponding key handler. If one is found then it is called and the function returns true, otherwise it returns false.

5.3.94Undo

```
function Undo: longint; virtual;
```

Applies to: TTextBuffer, TLinesBuffer, TTextStream, TEditStream

Undoes the last edit, or chain of edits. The function returns the position of the earliest undo operation. Note that the method is defined as an abstract method by TTextBuffer and overridden by TEditStream.

5.3.95Undo

```
procedure Undo;
```

Applies to: TBigEditor, TStreamEditor, TFileEditor, TMailEditor

This method is called to perform the most recent undo operation. The preceding deletion/insertion is thereby reversed.

5.3.96UUDecode

```
procedure UUDecode;
```

Applies to: TMailEditor

This method scans the current file, starting at the beginning, for uuencoded files, decoding their contents if required. To intercept the decision to decode a file, and control where it is stored, use the 'OnStartDecode' event. To report the completion of each decode, use the 'OnEndDecode' event.

5.3.97UUDecode

```
function UUDecode(Buffer: TLinesBuffer; Pos: longint): boolean;
```

Applies to: TUUDecoder

This method is called to run the decoder on the TLinesbuffer from Pos onwards. UUdecoded file(s) are stored in the files according to their name, unless overridden by the response to the OnStartDecodeEvent.

5.3.98UUEncode

```
function UUEncode(const FileName, Name: string): TEncodeResult;
```

Applies to: TMailEditor

This method is called to uuencode the contents of the file given by 'FileName', and inserted the encoded result at the current insertion point. Note that the uuencoded file header includes the name of the encoded file. This does not necessarily have to be the same as the name of the file being uuencoded and is anyway given by 'Name'.

5.3.99UUEncodeFile

```
function UUEncodeFile(const FileName,Name : String; Buffer:
            TTextBuffer; Pos: longint; var Cancelled: boolean):
            longint;
```

Applies to: TUUEncoder

This method encodes the file given by FileName, into Buffer, starting at Pos. On Return, the function returns the number of characters encoded into the Buffer.

5.3.100WidthOf

```
function WidthOf(c: char):integer;
```

Applies to: TLinesBuffer, TTextStream, TEditStream

This method returns the width of the ‘c’ in pixels using the display device and the current font.

5.4Type Definitions

5.4.1TAbortEvent

```
TAbortEvent = Procedure(Sender: TLinesViewer; var Response: Word) of
object;
```

The *Response* should be either `mrYes,` or `mrNo`, as defined by the TModalResult type.

5.4.2TKeyDefinition

```
TKeyDefinition =  0..$0500;
```

TKeyDefinition is the type used for all edit keys and holds the Virtual Key Code for the edit key, which may be modified by adding to the Virtual Key Code one of the following kbXXXX codes:

```
kbControl  = $0100;  {Offset for control key in Key Definition}
kbShift    = $0200;  {Offset for Shift key in Key Definition}
kbQuickKey = $0400;  {Offset for Quick Key in Key Definition}
```

The semantics of a modified virtual key code is that TKeyManager requires the Virtual key to be in combination with the associated shift key and/or preceded by the Quick Key (two key stroke function keys) in order to invoke the associated key code handler.

For example, to set up the KeyManager to use CNTL+Y to delete a line, use the following code in a TLinesViewer method:

```
FKeyManager.OnKeyCode[ord('Y') or kbControl] := DeleteLine
```

Note that in practice a more sophisticated method than the above is used in order to make it easy to define edit key codes as properties - see the FileView.Pas source file for further details

5.4.3TKeyEventHandler

```
TKeyEventHandler = Procedure of object;
```

5.4.4TlbModes

```
 TlbModes =  (lbDisplay,    {Use display device for calculating
                            line widths}

             lbPrinter);    {User printer for calculating line
                            widths}
```

The TlbModes type is used to determine the way a TLinesBuffer determines the width of a line in pixels. This may be either display device relative or printer relative.

5.4.5TOutputDevice

```
TOutputDevice =  (odDisplay,  {Output to display device}
                  odPrinter); {Output to printer device}
```

Used by the OutputDevice property to determine whether the printer or the display canvas is to be used.

5.4.6TPageBreakDisplayModes

```
 TPageBreakDisplayModes =
    (pbNever,     {Soft page breaks not shown}
     pbAlways,    {Soft page breaks always shown}
     pbWYSIWYG); {Soft Page breaks shown when wrap on printer page
                  width}
```

The TPageBreakDisplayModes type is used by the ShowPageBreaks property to determine when soft page breaks are shown by a TTextViewer

5.4.7TPrinterMargin

```
TPrinterMargin =  Real;
```

The TPrinterMargin type is used for a printer margin specified in inches.

5.4.8TPrintEvent

```
TPrintEvent =
     Procedure(Sender: TLinesViewer; {the object}
     PageNum,                        {now printing page number}
     TotalPages: longint)            {total pages in document}
     of object;
```

5.4.9TProgressEvent

```
TProgressEvent =
     Procedure(Sender: TLinesViewer;      {the object}
     Done,                                {Chars processed so far}
     Total:Longint;                       {out of chars to process}
     var Cancel: boolean)                 {set to true on return in
                                           order to abort process}
     of object;
```

5.4.10TReplaceAllEvent

```
TReplaceAllEvent = Procedure(Sender: TObject; Count: integer) of Object;
```

5.4.11TSaveMode

```
 TSaveMode =   (smParagraph,   {save text as paragraphs}
               smLineBreaks,   {save text with hard line breaks}
               smPrompt);      {prompt for save mode}
```

The TSaveMode type is used by a TLinesViewer to determine how the text is to be saved to a file.

5.4.12TSaveModifiedEvent

```
TSaveModifiedEvent = procedure(Sender: TObject; var SaveRequest:
TModalResult) of object;
```

The *SaveRequest* response should be either `mrYes, mrNo` or `mrCancel`, as defined by the TModalResult type.

5.4.13TSavePromptEvent

```
TSavePromptEvent = procedure(Sender: TObject; var Mode: TSaveMode) of
object;
```

5.4.14TSaveUndoEvent

```
TSaveUndoEvent = Procedure(var CaretPos, SelStart, SelEnd: longint; var
     SelMode: TSelModes;
     var Linked: boolean) of object;
```

5.4.15TSearchStatus

```
TSearchStatus = Procedure (Done: longint;
      var Cancelled: boolean) of object;
```

5.4.16TSelModes

```
TSelModes = (                {Selection Modes}
     smNone,                 {no selection}
     smDouble,               {Selection in double click mode}
     smExtend);              {Selection using shift key}
```

5.4.17TUdStates

```
 TUdStates =
     (udInit,                {Initial State}
      udDecoding,            {Decoding a file}
      udSectionBreak,        {Unexpected Section break found}
      udSectionChecked,      {Section text checksum found}
      udEnd,                 {End of encoded text found}
      udEndSectionChecked,   {Last section text checksum found}
      uuCancelled,           {Cancelled by user}
      udFileChecked);        {File binary checksum found}
```

5.4.18TUndoSaveUndoEvent

```
TUndoSaveUndoEvent =  Procedure(var Modified: boolean;
      var CaretPos, SelStart, SelEnd: longint;
      var SelMode: TSelModes; var Linked: boolean) of object;
```

5.4.19TUndoEvent

```
TUndoEvent =
     Procedure(CaretPos, SelStart, SelEnd: longint;
     SelMode: TSelModes) of object;
```

5.4.20TUndoUndoEvent

```
TUndoUndoEvent = Procedure(Modified: boolean;
     CaretPos, SelStart, SelEnd: longint;
     SelMode: TSelModes) of object;
```

5.4.21TUUDecodeEvent

```
TUUDecodeEvent = procedure(Sender: TUUDecoder;
     var FileName: string; var Cancel: boolean) of object;
```

5.4.22 TUUEndDecodeEvent

```
TUUEndDecodeEvent =
      procedure(Sender: TUUDecoder;
      State: TUdStates;                  {Current State of Decoder}
      Error: TUUErrors;                  {Error Status}
      const FileName: string)            {Name of decoded file}
      of object;
```

5.4.23 TUUErrors

```
TUUErrors =
   (uuNoError,              {Decode completes with no error}
   uuUserError,             {Decode interrupted by user}
   uuNoSpace,               {Line longer than specifed maximum}
   uuLineError,             {Line shorter than indicated length}
   uuFileSumCheck,          {Decode completes with a binary sum
                             check error}
   uuSectionSumCheck,       {Section decode completes with a text
                             sum check error}
   uuBadCharacter);         {Illegal character in encoded text}
```

The TUUError type is used to indicate the result of a uudecode operation (see OnEndDecode event).

5.4.24 TWrapMode

```
TWrapModes = (opNone,          {Line wrap on Cr/LF only}
     opWrapScreenWidth,        {wraps lines at right edge of window}
     opAbsoluteWrap,           {wraps lines at absolute character count}
     opPrinter);               {wraps lines according to printer width i.e. WYSIWYG}
```

5.4.25 TValidText

```
TValidText = Record
     First,      {Pointer to first valid address in buffer}
     Last: PChar {Pointer to last valid address in buffer}
End;
```

5.4.26 TViewerState

```
TViewerState =
   (vsEmpty,                {Set when there is no text to view/edit}
   vsSelection,             {Set when text is selected}
   vsSearchString,          {Set when a search string is available
                             for FindNext}
   vsInsertOn,              {Set when an editor is in insert mode}
   vsMouseDown,             {Set when the Left Mouse button is held
                             down during selection}
   vsDragging,              {Set when the Left Mouse button is held
                             down during dragging}
   vsBusy,                  {Set when the Hour Glass Cursor is
                             visible}
   vsDropping,              {Set while dragged text is being
                             inserted into the text}
   vsModified,              {Set after an editor has changed the
                             text and reset when saved}
   vsUndoAvailable,         {Set when an Undo operation can be
```

```
                               performed}
      vsRedoAvailable,        {Set when a Redo operation can be
                               performed}
      vsPasteAvailable,       {Set when CF_Text is in the Windows
                               ClipBoard}
      vsCopying,              {Set during a copy to clipboard
                               operation}
      vsPrinting,             {set during a print operation}
      vsGotoLine,             {set during a search for a specified
                               line number}
      vsSearching);           {set during a search operation}
```

5.4.27TViewerStates

TViewerStates = Set of TViewerState;