# The TextView DLL

**Version 1.00**

# INTRODUCTION TO TEXTVIEW

Thank you for using the **TextView** system. The author hopes that it will prove to be a powerful and useful tool to Windows applications writers, providing a facility that is both useful in the final application, and helpful in the development phase.

This manual covers all aspects of the system. It assumes that you are an experienced Windows application developer, and are conversant with the use of the Windows API.

The author is always interested to hear any reports about how you find TextView, what further facilities you would like it to include, and any problems you may find when using it. Contact addresses are given later in this manual.

## What is TextView?

**TextView** is a system that provides a Windows 3 application with the ability to write lines of text to a window with the minimum of effort. **TextView** itself handles all the many operations needed to manage the window displaying the text; you need only call the function that writes the text, in exactly the same way as you would call **`printf`** in a DOS application.

You can create as many **TextView** windows as you require, and all will operate independently. **TextView** windows can be resized by the user, minimized, maximized and scrolled horizontally and vertically with no work needed by the application. They can also have *File* menus that notify the application when the user requests that the window contents be written to disk.

**TextView** windows can, of course, be used for an infinite variety of purposes. One possibility is to use **TextView** to provide a way of outputting tracing and debugging information when developing an application. Included with the system are the sources for a demonstration application that contain a flexible tracing system that may be incorporated into your own code.

**TextView** is supplied as a *Dynamic Link Library* or *DLL*. In this form it is not added to the application's code at link time, but instead is bound to it dynamically by Windows whenever an application needing it is run. All applications using **TextView** will share the same code segments, making it very efficient in memory utilisation.

## What you need to use TextView

In order to use **TextView** you will need to have a suitable compiler that can generate Windows 3 code, such as Microsoft C 6.0. The **TextView** functions follow the same conventions as used in the Windows API, and their descriptions later in this guide use the same layout as in the Microsoft Windows Programmer's Reference.

## Distribution and Use of TextView

distribution. It may be stored on Bulletin Board systems and other archives so long as all the files comprising the original distribution are included. It may be repackaged to suit the storage conventions in use for the system concerned. It may not be distributed as part of commercial disk libraries without the prior agreement of the author.

**TextView** may be freely used in any non-commercial Windows application. Authors of such applications may include the **TextView** DLL with the products either with or without the other files comprising the full distribution set. However, such application should include in their documentation (or should display in their *About* dialog box or in their online help) that they are using **TextView**, and should list the author's copyright.

The demonstration sources supplied with **TextView** may be freely adapted and included in other applications as required.

Authors of ShareWare or commercial products may not use **TextView** without the author's written permission.

## Disclaimer

**TextView** is distributed on an *as is* basis. No guarantee is offered, and none should be inferred, of its correct functionality, nor of its suitability for any task whatsoever. The author accepts no liability for any loss or damage whatsoever caused as a result of using **TextView** with any application, whether written by the user or a third party.

**TextView** is written as a private activity by the author, unconnected in any way with his employment at the University of Lancaster.

## Contacting the Author

The author may be contacted *c/o* **The Computer Centre, Lancaster University, Lancaster LA1 4YW, United Kingdom**. Electronic mail may be addressed to one of the following addresses:

```
alan@uk.ac.lancaster      JANET network
alan@lancaster.ac.uk      Internet or BITNET
alan%uk.ac.lancaster@ukc UUCP network
```

# INSTALLING TEXTVIEW

There are three components to the **TextView** system. The DLL itself, **`textview.dll`**, should be copied to your Windows directory. The model-independent import library **`textview.lib`** should be copied to one of the directories named in your **`LIB`** environment variable - the directory containing your Windows SDK libraries or your C compiler libraries would normally be suitable. The header file **`textview.h`** should be copied to one of the directories named in your **`INCLUDE`** environment variable - the directory containing your **`windows.h`** header file would be suitable.

# USING TEXTVIEW

The **TextView** DLL extends the Windows API with a number of specific functions, which you call from your application. All the functions have names beginning with **TV**, to avoid clashes with Windows functions, those in your application or in other DLLs.

All the functions use the Pascal calling convention, and must be declared **FAR**. Full function prototypes are contained in the include file **`textview.h`**; including this in your source will automatically select the right calling mode, and will also perform any necessary casts to far pointers.

You must take two additional steps when building your application:

- Include the **TextView** import library **`textview.lib`** in the list of libraries that you name for the linker to scan.

- Make sure you declare sufficient stack size in your application's **`.DEF`** file. As a rough guide, try increasing the size by 2Kbytes over what your application itself uses.

You must not use the Windows **LoadLibrary** and **GetProcAddress** functions to link TextView routines at run-time. This is because some of the functions are actually within the import library and *must* be statically-linked.

The sections below describe how to use **TextView** in your application. The text assumes that you're familiar with the Windows API and with how to program Windows applications: there is general information on this in the documentation supplied with the Microsoft Windows Software Developer's Kit, and in several other published books.

Besides this manual, there is a comprehensively-commented set of C source routines supplied in the distribution set. These show you how you can use **TextView** to add a dynamic tracing utility to your application, which provides you with a very powerful development aid that enables you to keep a log of your application's activity that can be scrolled back and consulted when necessary.

## A Summary of TextView Usage

Although it offers your application powerful facilities, **TextView**'s API is very straightforward, and has been written to parallel closely the way you use the Windows API. The basic concepts of how the system works are these:

### The TextView Window

A window created by **TextView** is an ordinary *overlapped* window that belongs to your application. Within certain limits you have full control over the appearance of the window: you can allow the user to resize it or not, supply it with maximize and minimize boxes, and so on.

Creation of a **TextView** window is done with the **TVCreateWindow** function, which looks similar to the Windows **CreateWindow** function. The major difference is that for a **TextView** window the message loop is handled within the DLL and not by your

application, so that you do not need to concern yourself with managing the window. **TextView** will look after aspects such as text scrolling on your behalf.

Although your application does not supply the message loop, it can still interact fully with the window. A set of functions allow you to test what state the window in and to change that state, and to destroy the window when you've finished with it.

If you choose, you can specify that a **TextView** window displays a *File* menu with *Save*, *Save As* and *Print* options. You do not see clicks on these menu items directly, since you do not supply the message loop, but you can arrange for **TextView** to call back into a routine in your application to notify you when menu items are selected.

## How Text is Stored

**TextView** maintains the lines written to each window in a cyclic buffer held in an area of memory private to that window. Each window operates independently of the others, and the only practical limit to the total amount of data stored will be how much memory your system has available.

When you create a window by calling **TVCreateWindow** you specify the maximum number of lines that each window can store, up to a limit of 4096 lines. Specifying a larger number requires **TextView** to allocate more control memory for the window, so you should not request a larger capacity that you need.

Whenever a line is written, **TextView** stores the contents in dynamically-allocated memory. When the window contains the specified maximum number of lines, the next to be written will replace the oldest line, and so on. A line can be up to 512 characters long.

## Scrolling

When you create a **TextView** window it will be in *automatic scrolling* state. In this state **TextView** will automatically move existing lines of text up to make room as new ones are written, without your application needing to be aware of what is happening. Old lines, of course, will disappear off the top of the window, but they will remain stored in memory until the window reaches its set capacity and they are overwritten with new lines of text.

If the user wishes to look at older messages that are no longer visible in the window, he can put the window into *manual scrolling* state. **TextView** will draw horizontal and vertical scroll bars on the window (you may select either or both) and the user may use them to scroll around the stored text. You can specify when you create the window that it is to have a *Scrolling* menu to enable the user to select manual scrolling mode, or you may choose to control the window from your own application using the **TVSetScrollMode** function.

When the window is in manual scroll mode it is unable to display any text written to it. However, it will count the number of messages lost should this occur, and will add a line itself to the window to warn the user when the window returns to automatic scroll mode.

## Registering a TextView Window Class

As with the Windows **CreateWindow** function, you must have registered a suitable window class before you can create a **TextView** window using **TVCreateWindow**. Since most of the details of the window class have to be supplied by **TextView** itself, you must call a **TextView** function **TVRegisterClass** to do this, and must not use the ordinary Windows technique.

When you call **TVRegisterClass** you specify four arguments: for example

```
TVRegisterClass(hInstance,"TV_WINCLASS",
                LoadIcon(hInstance,"TV_WINICON"),
                GetStockObject(WHITE_BRUSH));
```

will register a class using an icon defined in your application's resource area, and will use a white background for the window.

The function will return **FALSE** if it fails to register the window class, or if you pass incorrect arguments.

## Creating a TextView Window

Creating a Window with **TextView** is closely analogous to how you use Windows' own **CreateWindow** function. The routine you call is **TVCreateWindow**, and many of the arguments you need to pass are have exact **CreateWindow** equivalents.

The *lpClassName*, *lpWindowTitle*, *X*, *Y*, *nWidth*, *nHeight* and *hInstance* arguments are used in the same way as the **CreateWindow** arguments of the same name. The one restriction is that the window class you specify with *lpClassName* must have been registered with the **TVRegisterClass** function.

The remaining arguments are specific to TextView and have no **CreateWindow** equivalents.

*hFont* specifies a handle to the font that you want text written in the window to appear in. If you specify the argument as **NULL**, **TextView** will use the system font; otherwise, you can use any font created with the **CreateFont** function.

The *dWflags* argument specifies a series of bit settings that describe the appearance you want the window to take, and what facilities it provides. These are described in detail below.

The *nTabSize* value specifies how you want tabs to be expanded in lines written to the window. You give the value as a number of characters; TextView multiplies this by the width of the average character in the selected font to calculate the actual spacing. If you give a value of zero, tabs will be expanded to a width of 8 characters.

The *nMaxLines* argument tells TextView how many lines of text should be stored with the window (note that this is not the size of the actual *window*, but the size of its data storage area). You can set this value to be from 128 to 4096. If you write more lines than this to the window, the oldest lines will be progressively overwritten.

The *lpMenuHandler* argument is the procedure instance address of a function within your application that is to receive notification of things happening in the window. This is discussed in more detail below in the section on *Receiving Menu Notifications*. The value you pass as this argument must have been obtained by using the **MakeProcInstance** function. Depending on the values you have specifies in the *dwFlags* argument, you may be allowed to give a **`NULL`** value.

The return value from **TVCreateWindow** is a normal window handle, which you use in other **TextView** functions. You can also pass this handle to Windows functions to manipulate the window; however you should avoid calling **DestroyWindow** to close it. Instead, use the **TextView** equivalent **TVDestroyWindow**, which is guaranteed to work in future releases.

### The dwFlags Argument

This argument is a collection of bit settings that tells **TextView** details of how you want the window to appear, and what facilities it should support.

The **TVS_MAXIMIZE**, **TVS_MINIMIZE** and **TVS_SYSMENU** settings correspond directly to *dwStyle* settings in **CreateWindow**, and control whether the window has a maximize box, a minimize box, and a system menu.

The **TVS_NOCLOSE** setting can be used to disable the *close* option in the system menu; if you select this, your application must destroy the window, as the user will have no way to do so. If you specify a system menu, and do *not* disable the close option, you *must* supply a procedure address with the *lpMenuHandler* argument so that **TextView** can notify your application when the user closes the window.

The **TVS_NORESIZE** setting allows you to create the window without a thick "resizing" frame. The user will not be able to alter the size of the window other than by minimizing or maximizing it.

**TVS_HSCROLL** and **TVS_VSCROLL** specify whether the window is to show horizontal and vertical scroll bars if the user or your application puts it into *manual scroll* mode. If you don't include either of the settings, manual scroll mode cannot be selected, and the user will be unable to scroll back to look at text no longer in the window.

**TVS_TIMESTAMP** controls whether TextView is to timestamp text written to the window. If you specify it, all lines will be prefixed with the current Windows time (that is, the number of milliseconds since Windows started, obtained from the **GetCurrentTime** function). Timestamping messages can be useful if you are using TextView to trace the path taken by your application in response to external events.

The other settings all control what menu items should appear in the window's menu bar. **TVS_SCROLLMENU** selects a *Scrolling* menu item, which will let the user switch between manual and automatic scroll modes. You can use this setting only if you've also given either or both of **TVS_HSCROLL** and **TVS_VSCROLL**. If you specify a procedure address with the *lpMenuHandler* argument, your application will be notified when the user clicks these menu items.

**TVS_FILESAVE**, **TVS_FILESAVEAS** and **TVS_FILEPRINT** specify that the window is to have a *File* menu, which should include *Save*, *Save As* and *Print* options respectively.

You can use the three settings independently. If you use any of these settings you *must* specify a procedure address in the *lpMenuHandler* argument to receive notification when the items are clicked.

## Writing Text to a TextView Window

Once a **TextView** window is created, writing text to it is simply done with the **TVOutputText** function. This takes three arguments:

| | |
|---|---|
| *hWnd* | This is the handle to the **TextView** window. |
| *lpBuffer* | This is a long pointer to a buffer containing the line you wish to write. |
| *nSize* | This is the number of characters in the buffer. If you set this value to zero, **TVOutputText** will assume the text is a zero-terminated string and will write it out completely. |

You can write up to 512 bytes in a line, less the length of the timestamp details if you used the **TVS_TIMESTAMP** option in the *dwFlags* argument to **TVCreateWindow**. If you supply a longer line than that, **TextView** will truncate it.

What happens to the text depends on the current state of the **TextView** window you are writing it to. If the window is in *automatic scroll mode*, it will be displayed in the window, which will be scrolled up by one line if necessary to make room for it. The text will be written in the colour last specified with the **TVSetTextColor** function (or in black, if you haven't set another colour).

If the window is in *manual scroll mode*, the text will be discarded. **TextView** will record the fact that a line has been lost, and when the user (or your application) returns the window to automatic scroll mode will add a line itself noting the number of lines that have ben lost. If the window has been suspended with a call to **TVSuspendWindow**, or you are currently calling the **TVReturnData** function to read back the data stored in the window, the text will also be discarded. Here, though, **TextView** will *not* record the fact.

If you choose, you can determine the status of the window by calling the **TVGetWindowStatus** function, and so avoid making calls to **TVOutputText** at inappropriate moments.

## Receiving Menu Notifications

A **TextView** window is an independent entity, all of whose functions are controlled within the **TextView** DLL itself. Your application does not provide the message loop for the window, and is normally unaware of what the user is doing to the window: **TextView** handles window resizing, scrolling, iconizing and so on.

However, you may wish to make your application aware of what the window is doing for one of two reasons. Firstly, you may wish to allow the user to control the window from either its own menu or from the *application's* menu - for example, you might include in your application's menu the ability for the user to switch the **TextView** window between manual and automatic scrolling. In order to keep the application's menu state in line with the window, it will be necessary for the application to be informed whenever the state is changed using the window's

own menu. The application will probably also wish to be informed when the user closes the window with the *Close* option in its *System* menu, so that it can keep track of which windows are still active.

The second reason is that some services *must* be provided by the application. TextView allows you to specify that the window should possess a *File* menu with options such as *Save* and *Print*, but it does not itself handle these functions. Instead, it notifies the application when one of these menu items is clicked, so that it may then perform the required operations.

All these examples of menu notifications are done via the *menu handler* routine specified when you create the window with **TVCreateWindow**. Whenever the user clicks a menu item (including the *Close* option in the *System* menu) the **TextView** DLL will call this routine, passing it the handle of the window concerned, and a code indicating the menu item.

Your code can then take what action it wishes. For example, if the user clicked on *File Save As*, you might run a dialog to ask the user for a file name, open it, and then copy the data currently in the window to the file using the **TVSaveWindowToFile** function.

The example sources contained in the distribution set contain examples of how you use menu notification. The code keeps the main window's menu in line with the current state of the window, graying some items when the window is destroyed and checking the relevant scroll state selections.

## Saving Data in a Window to File

**TextView** allows you to save the contents of a window to a file on disk at any time. There are two techniques for doing this, depending on how much processing of the data you wish your application to do before it is written.

If you application created the window and specified a *File* menu with *File Save* or *File Save As* options, it will be notified when they are clicked, as described above, but you can, of course, initiate a file save at any time and not solely when this occurs.

The first technique for saving the data is the simplest. **TextView** provides a function **TVSaveWindowToFile** which will write a range of lines to a file exactly as they as stored, and if you don't need to process the data yourself this will be the easiest option. Your application will need to determine the name of the file to be written, probably by giving the user a dialog box to choose it, and open the file for writing; then it can call **TVSaveWindowToFile**.

For example, if the name of the file is stored in **file_name**, you could use code like this to save all the data stored for the window:

```
int     hFile;          /* handle to file */

...

hFile = _lcreat(file_name,0);
TVSaveWindowToFile(hWnd,hFile,0,-1,0L);
```

Specifying the start line as 0 and the number of lines to write as -1 causes all the stored data to be saved. The demonstration program sources in the distribution set show you this technique in use.

The second technique for saving data allows you to process the lines before they are written. You can obtain successive lines from the window by calling the **TVReturnData** function, described in a following section, perform the actions required, and write to the file yourself.

If you wish to save only the data that is actually visible in the window, you can use the **TVGetWindowStatus** function to obtain the required line numbers, as shown here:

```
TVWSTATUS   status;          /* window status details */
int         hFile;           /* handle to file */

...

/* Get details of the window */

TVGetWindowStatus(hWnd,&status);

/* Open the file and write the data that is visible now */

hFile   = _lcreat(file_name,0);
TVSaveWindowToFile(hWnd,hFile,status.nTopLine,
                                status.nRows,0L);
```

Giving the number of lines to be written as the number of rows in the window ensures that all the visible data is written. If the window is not actually full, the function will adjust the number requested itself.

## Printing Data in a Window

**TextView** does not itself provide facilities to print the contents of a window to file. If you wish your application to do this, you will need to use the **TVReturnData** function described below to read back the window's contents, and handle the printer yourself.

If you create the window to have a *File* menu with a *Print* option, **TextView** will notify your application when the user requests a print action by clicking it.

## Reading Back the Contents of a TextView Window

Your application can call the **TextView** DLL at any time to read back the data stored with any **TextView** window, using the **TVReturnData** function.

The initial call to **TVReturnData** nominates a buffer that you wish to use to receive the contents of the window, one line at a time. You also specify the address of a *callback* function; **TextView** will copy the text of one line into your buffer and call this function to allow you to process it, repeatedly until either every line has been processed, or you terminate the sequence.

For example, the callback function might be declared like this:

```
int FAR PASCAL  data_handler(HWND hWnd, LPSTR lpBuffer,
                             int nCount, BOOL nTruncated)

{
    /* Make sure we didn't lose any bit of the line */

    if ( nTruncated )
    {
        /* Buffer was too short, so a line has been cut
        *  short. Warn the user and abort the process
        */

        MessageBox(NULL,"Buffer too short",NULL,
                    MB_ICONSTOP | MB_TASKMODAL);
        return(0);
    }

    /* Got all the line, so process it */

    process_line(lpBuffer);

    /* And return non-zero to get the next line */

    return(1);

}
```

This routine will be called once for every line of text in the window. It checks that no data was lost due to the buffer being too small, and if it was it terminates the read of data by returning a value of zero to the DLL. If not, it calls some routine called **`process_file`** to do something to the data, and returns a non-zero value requesting the DLL to pass the next line to it.

While your application is engaged in reading back lines of text from a window, any calls to **TVOutputText** to add more text to it, and most of the functions controlling the window, will be disabled.

The example sources supplied in the distribution give you a full example of how to use **TVReturnData**. In this case, they read back the data from one window and make a copy of it in another.

## Destroying a TextView Window

A **TextView** window may be destroyed in one of two ways. Your application may simply call the **TVDestroyWindow** function, which will remove the window from the screen and release all its memory resources. This may be done at any time.

Alternatively, if you specified in the **TVCreateWindow** call that the window was to have a System Menu, and did not inhibit the *Close* option, the user may close the window directly by clicking that option. In this case, **TextView** will remove the window from the screen and release the memory resources used by the window. It will then notify your application that the window has been destroyed by calling the *menu handler* function, passing it a code of **TVMI_CLOSE**.

You should not use the normal Windows **DestroyWindow** function to destroy a TextView window as this may be incompatible with future releases.

## Control Functions

**TextView** provides your application with a range of control functions that control the operation of its windows, as described below:

### Setting Text and Background Colour

By default, text will be displayed in a **TextView** window in black. The **TVSetTextColor** function allows you to specify any RGB value to be used for subsequent output. The **TVSetBKColor** function allows you to set the text background colour.

### Setting the Scroll State

If you specify the **TVS_SCROLLMENU** setting in the *dwFlags* argument to **TVCreateWindow**, the window will have a *Scrolling* menu that will permit the user to switch between manual and automatic scroll modes at will.

If you wish, your application can force a particular scroll state itself with the **TVSetScrollState** function. If you do not specify a *Scrolling* menu, this is the *only* way to change scroll modes.

### Suspending Text Output

At some point in your application you may wish to suppress output to a **TextView** window. One way of doing so would, of course, be to set a global flag in your application's data segment that is checked by every routine that calls **TVOutputText**, and this technique is used in the demonstration application to activate or disable tracing.

An alternative that does not involve a global flag is to use the **TVSuspendWindow** function. When a window is marked as *suspended*, **TVOutputText** functions as normal, but the text is discarded.

### Temporarily Inhibiting Window Updates

Writing text to a window, particularly if the existing contents must be scrolled to make room, can be an expensive operation in terms of processor power. If your application has to write a large number of lines to a **TextView** window at one time, it will be considerably slowed while the data is scrolled up through the display.

The **TVSetRedraw** function allows you to configure a window so that text lines are stored as normal, but the window is not updated as they are received, letting you write the data with **TVOutputText** very rapidly. Then, when you have written all the lines, you can tell

**TextView** to update the window: it will display only the final windowfull of lines, which will involve no scrolling.

The demonstration program supplied with **TextView** lets you see the effect of suppressing window updates: the *Write Batch* menu option writes 200 lines to the window, scrolling for each one; the *Write Batch Fast* option disables window updating, writes 200 lines and then updates the window.

### Clearing a Window

The **TVResetWindow** will destroy all data stored for a window, and it will be redrawn empty. The window will be set into automatic scroll mode, and the text colour used will be set to be black.

## Information Functions

You can find out the exact status of a **TextView** window at any time by calling the **TVGetWindowStatus** routine. This fills in a **TVWSTATUS** struct that you supply; the details of the format are shown in the *Data Types and Structures* section.

# FUNCTIONS DIRECTORY

This chapter contains an alphabetical list of functions comprising the **TextView** system. The specifications are laid out in the same way as those in the Microsoft Windows Programmer's reference Manual.

All **TextView** functions use the Pascal calling convention, and must be declared **FAR**. Including the header file **`textview.h`** in sources using these interfaces will declare their prototypes automatically, and will ensure that they are being correctly used.

Applications that make use **TextView** routines must be linked with the **TextView** import library **`textview.lib`**.

# TVCreateWindow

**Syntax** **BOOL TVCreateWindow**(*lpClassName, lpWindowName, X, Y, nWidth, nHeight, hInstance, hFont, dwFlags, dwUnused, nTabSize, nMaxLines, lpMenuHandler*)

This function creates a **TextView** window. Many of the arguments are similar to those used with the CreateWindow function, but there are some extra values appropriate to **TextView**, and some inappropriate arguments may not be specified.

**TVCreateWindow** will always create an overlapped window. Option flags allow the selection of minimize and maximize boxes, and whether the user will be able to resize it by dragging its borders.

**TextView** allows the application to interact with the window in a controlled way. All message handling for the window is performed by **TextView**, but the application can specify that it wished to be notified when events such as menu item clicks occur. For example, this allows the window to appear to the user as a normal application window complete with a *File* menu, but for handling of the corresponding functions to be done by the application.

| **Parameter** | **Type/Description** |
|---|---|
| *lpClassName* | **LPSTR** Points to a null-terminated string that names the window class. This class must have been previously registered with a call to **TVRegisterClass.** |
| *lpWindowName* | **LPSTR** Points to a null-terminated character string that represents the window name. |
| *X* | **int** Specifies the initial *x*-position of the window. The value is the initial *x*-coordinate of the upper left corner, in screen coordinates. If the value is CW_USEDEFAULT, Windows selects the default position for the window's upper-left corner. |
| *Y* | **int** Specifies the initial y-position of the window. The value is the initial y-coordinate of the upper left corner, in screen coordinates. |
| *nWidth* | **int** Specifies the width of the window in device units. If the value is CW_USEDEFAULT, Windows selects a default width and height for the window, and the *nHeight* argument is ignored. |
| *nHeight* | **int** Specifies the height of the window in device units. The argument is ignored if *nWidth* is CW_USEDEFAULT. |

*hInstance* **HANDLE** Identifies the instance of the module to be associated with the window.

*hFont* **HFONT** Specifies a handle to the font to be used when text is written to the window. If the value is NULL, the system font is used.

*dwFlags* **DWORD** Specifies various facilities required in the window. It can be any combination of the values given in the list below.

*dwUnused* **DWORD** Must be zero.

*nTabSize* **int** Specifies the width of a tab stop to be used when writing text to the window.

*nMaxLines* **int** Specifies how many lines of text are to be stored in the window's buffers. The value must be between 128 and 4096; values outside this range will be silently adjusted.

*lpMenuHandler* **FARPROC** A procedure instance of a routine to be called when the user clicks on an item in the window's menu. See the "Comments" section for details.

**Return Value** The return value is a handle to the new window. It is NULL if the window could not be created.

**Comments** Where the *X* argument is CW_USEDEFAULT, the *Y* argument can be one of the show-style parameters described with the **ShowWindow** function.

The address passed as the *lpMenuHandler* argument must be created by using the **MakeProcInstance** function. The callback function must use the Pascal calling convention and be declared **FAR**.

The *dwFlags* argument should contain values from the list below:

TVS_FILESAVE Specifies that the window is to have a *File* menu that will incorporates a *Save* option. If this option is specified the *lpMenuHandler* argument must not be NULL.

TVS_FILESAVEAS Specifies that the window is to have a *File* menu that will incorporates a *Save As* option. If this option is specified the *lpMenuHandler* argument must not be NULL.

TVS_FILEPRINT Specifies that the window is to have a *File* menu that includes a *Print* option. If this option is specified the *lpMenuHandler* argument must not be NULL.

TVS_HSCROLL Specifies that the window is to permit horizontal scrolling by the user.

TVS_MAXIMIZE Specifies that the window is to have a maximize box.

TVS_MINIMIZE Specifies that the window is to have a minimize box.

TVS_NOCLOSE Specifies that the *Close* option on the window's system menu is to be inhibited. If this option is used the window can only be destroyed by a call to **TVDestroyWindow**. This option is ignored if TVS_SYSMENU is not specified also.

TVS_NORESIZE Specifies that the window is not to have a thick frame allowing the user to resize it.

TVS_SCROLLMENU
Specifies that the window is to have a Scroll menu, allowing the user to switch between manual and automatic scrolling. This option requires one or both of TVS_HSCROLL and TVS_VSCROLL to also be defined.

If this option is omitted, the application must switch scrolling modes if required with a call to **TVSetScrollState**.

TVS_SYSMENU Specifies that the window is to have a system menu. If this option is used and the TVS_NOCLOSE option is not used, the *lpMenuHandler* argument must not be NULL.

TVS_TIMESTAMP Specifies that lines written to the window are to automatically be timestamped. **TextView** will prepend the current Windows time (the number of milliseconds since Windows was started) to the text supplied.

TVS_VSCROLL Specifies that the window is to permit vertical scrolling by the user.

**Callback** **void FAR PASCAL** *MenuHandler(hWnd,nMenuItem)*
**HWND** *hWnd***;**
**WORD** *nMenuItem***;**

*MenuHandler* is a place-holder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module definition file.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Identifies the window whose menu item has been selected |

*nMenuItem* **WORD** Identifies the menu item concerned. The value will be one of those in the table below.

The menu handler callback function will be informed of which menu option has been clicked in the *nMenuItem* argument. The value will be one of those in the list below; if the *dwFlags* argument to **TVCreateWindow** did not enable a menu item the corresponding notification value will not occur.

TVMI_AUTOSCROLL
: Specifies that the user has clicked the *Automatic* option in the window's *Scrolling* menu. TextView will have already set the window into automatic scrolling state and hidden any scroll bars.

TVMI_CLOSE
: Specifies that the user has clicked the *Close* option on the window's system menu. The window will already have been destroyed when this event is notified.

TVMI_FILESAVE
: Specifies that the user has clicked the *Save* option in the window's *File* menu.

TVMI_FILESAVEAS
: Specifies that the user has clicked the *Save As* option in the window's *File* menu.

TVMI_FILEPRINT
: Specifies that the user has clicked the *Print* option in the window's *File* menu.

TVMI_MANUALSCROLL
: Specifies that the user has clicked the *Manual* option in the window's *Scrolling* menu. TextView will already have set the window into manual scrolling mode and drawn the required scroll bars.

# TVDestroyWindow

**Syntax** **BOOL** **TVDestroyWindow**(*hWnd*)

Destroys a window created with the **TVCreateWindow** function.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window to be destroyed**.** |

**Return Value** The function returns **TRUE** if the window has been destroyed**.**

# TVGetWindowStatus

**Syntax** **BOOL** **TVGetWindowStatus**(*hWnd,lpStatusBlock*)

This function returns status information for a **TextView** window.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *lpStatusBlock* | **LPTVWSTATUS** A FAR pointer to a **TVWSTATUS** block whose values will be filled in by this call. |

**Return Value** The return value will be TRUE if the call succeeded and the status information was returned.

# TVOutputText

**Syntax** **BOOL** **TVOutputText**(*hWnd,lpString,nCount*)

Writes one line of text to a **TextView** window. The text is stored in the window's buffer; if this contains the maximum number of lines specified when the window was created, the oldest stored line will be replaced.

If the **TVS_TIMESTAMP** option was specified when the window was created, **TextView** will prefix the supplied text with the current Windows time before displaying it.

Tab characters will be expanded to the width specified when the window was created.

The text will be written using the colour defined by the last call to **TVSetTextColor**, or in black by default.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *lpString* | **LPSTR** Points to the string to be written. |
| *nCount* | **int** Specifies the number of characters in the string. If the value is zero the string will be assumed to be null-terminated and will be written in its entirety. |

**Return Value** The return value will be **TRUE** if the text was successfully displayed.

**Comments** There are three conditions in which the text will not be displayed in the window as requested:

1. If the window has been put into *suspended* state with a call to **TVSuspendWindow**, the call to **TVOutputText** will be silently ignored.

2. If the application has called **TVReturnData** to read back the lines stored in the window, and has not yet ended the call-back loop, the call to **TVOutputText** will be silently ignored.

3. If the window is in manual scroll mode, the call will also be silently ignored. However, **TextView** records the number of lines lost when this occurs, and will add a line to the window itself to notify the user when the window returns to automatic scroll mode.

In all these cases, **TVOutputText** returns a value of **TRUE** to the caller. A **FALSE** value is used solely to indicate a system problem such as insufficient memory.

Text longer than 512 bytes will be truncated.

# TVRegisterClass

**Syntax** **BOOL** **TVRegisterClass**(*hInstance,lpClassName,hIcon,hbrBackground*)

This function registers a window class that can subsequently be passed to **TVCreateWindow** to create a **TextView** window. The use of the function is similar to that of the **RegisterClass** function, with the exception that **TextView** itself will supply most of the details required.

| **Parameter** | **Type/Description** |
|---|---|
| *hInstance* | **HANDLE** Specifies the application's instance handle. |
| *lpClassName* | **LPSTR** Points to a null-terminated string containing the name to be given to the window class. |
| *hIcon* | **HICON** Specifies a handle to the icon to be used when the **TextView** window is minimized. It must not be NULL. |
| *hbrBackground* | **HBRUSH** Specifies the class background brush to be applied to the window. The meaning is as defined for the *hbrBackground* item of the **WNDCLASS** structure, with the exception that the value must not be NULL. |

**Return Value** The function returns **TRUE** if the class was registered successfully.

**Comments** An error return value of **FALSE** will result both from the class name already having been registered, and from invalid arguments being supplied.

# TVResetWindow

**Syntax** **BOOL** **TVResetWindow**(*hWnd*)

This functions resets a **TextView** window to an 'empty' state. All stored text is discarded, and the window is redrawn empty. The window will be put into automatic scroll mode, and will be marked as not suspended. The text colour will be set to black.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle of the window concerned. |

**Return Value** The function will return **TRUE** if the operation succeeded.

# TVReturnData

**Syntax** **int** **TVReturnData**(*hWnd,lpBuffer,nSize,lpNotifyFunc*)

This function requests TextView to return the data stored in a TextView window into an application-supplied buffer. **TextView** will copy successive lines into the buffer, and will call back into a notification function to inform the application that each line is ready. The process continues until either all the lines have been returned or the callback function returns zero.

| Parameter | Type/Description |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *lpBuffer* | **LPSTR** Points to the application's buffer into which **TextView** will copy successive lines. |
| *nSize* | **int** Specifies the size of the application's buffer. |
| *lpNotifyFunc* | **FARPROC** The procedure-instance address of the callback function used to notify the application when each line has been copied into the buffer. |

**Return Value** The return value specifies the last value returned by the callback function. Its meaning is user-defined.

**Comments** The address passed as the *lpNotifyFunc* argument must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and be declared **FAR**.

**Callback** **int FAR PASCAL** *NotifyFunc(hWnd,lpBuffer,nCount,nTruncated)*
**HWND** *hWnd;*
**LPSTR** *lpBuffer;*
**int** *nCount;*
**BOOL** *nTruncated;*

*NotifyFunc* is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the application's module definition file.

| Parameter | Type/Description |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |

| | |
|---|---|
| *lpBuffer* | **LPSTR** Points to the application-supplied buffer that now contains the text of a line of text from the window. |
| *nCount* | **int** Specifies the line number within the window corresponding to this call. The first line is numbered zero. |
| *nTruncated* | **BOOL** This value will be TRUE if the text had to be truncated to fit into the supplied buffer. |

# TVSaveWindowToFile

**Syntax** **BOOL** **TVSaveWindowToFile**(*hWnd,hFile,nStart,nLines,dwFlags*)

This function writes the data stored for a **TextView** window into a file. It can be called at any time, and not only in response to the user clicking an item in the window's *File* menu.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *hFile* | **int** Specifies the MSDOS handle to the file to which the data is to be written. The file must have been opened for writing. |
| *nStart* | **int** Specifies the number of the first stored line to be written to the file. The first line is numbered zero. |
| *nLines* | **int** Specifies the number of lines to be written. A value of -1 will write all the lines stored from the first specified to the end of the buffer. |
| *dwFlags* | **DWORD** Must be set to 0L. |

**Return Value** The function returns **TRUE** if the operation succeeded. A value of **FALSE** indicates either that the range of lines requested was invalid, or that an error occurred writing the file. In the latter case, **TextView** will display a message to alert the user.

# TVSetBkColor

**Syntax** **DWORD** **TVSetBkColor**(*hWnd,crColor*)

This function sets the background colour used when text is written to a **TextView** window.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *crColor* | **COLORREF** Specifies the new background colour. |

**Return Value** The function will return the previously set colour, or -1 if an error occurs.

**Comments** **TextView** does not record background colours with individual messages. As the screen is scrolled messages that are written will use the text colour defined when they were stored, but the *current* background colour value.

# TVSetRedraw

**Syntax** **WORD** **TVSetRedraw**(*hWnd,nNewState*)

This function sets the redraw of a **TextView** window. If the state is set to **FALSE**, lines written to the window with **TVOutputText** will be stored, but the screen will not be updated until the state is changed to **TRUE**.

| **Parameter** | **Type/Description** | |
|---|---|---|
| *hWnd* | **HWND** | Specifies the handle to the window concerned. |
| *nNewState* | **BOOL** | Specifies the new state to be set. |

**Return Value** The function will return the previous redraw state of the window.

# TVSetScrollState

**Syntax** **WORD** **TVSetScrollState**(*hWnd,nNewState*)

This function sets the scrolling state of a **TextView** window.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *nNewState* | **WORD** Specifies the new state to be set. The value should be **TV_SCR_AUTO** to set automatic scrolling, or **TV_SCR_MANUAL** to set manual scrolling. |

**Return Value** The function will return the previous scrolling state of the window as wither **TV_SCR_AUTO** or **TV_SCR_MANUAL**. It will return zero if an error occurs, or if the call to **TVCreateWindow** did not specify one or both of the **TVS_HSCROLL** and **TVS_VSCROLL** options.

# TVSetTextColor

**Syntax** **DWORD** **TVSetTextColor**(*hWnd,crColor*)

This function sets the colour in which text is written to a **TextView** window.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *crColor* | **COLORREF** Specifies the colour in which text is to be written. |

**Return Value** The function will return the previously set colour, or -1 if an error occurs.

**Comments** Using this function does not affect the colour of text that has already been written.

# TVSuspendWindow

**Syntax** **BOOL** **TVSuspendWindow**(*hWnd,nNewState*)

Sets the *suspend* state of a **TextView** window.

| **Parameter** | **Type/Description** |
|---|---|
| *hWnd* | **HWND** Specifies the handle to the window concerned. |
| *nNewState* | **BOOL** Specifies the state to be set. It should be **TRUE** to suspend the window, and **FALSE** to desuspend it. |

**Return Value** The return value is the previous suspend state of the window.

# TVVersion

**Syntax** **void** **TVVersion**(*lpMark,lpVersion,lpCycle*)

Returns the version identification of the **TextView** DLL. This consists of a *mark number*, a *version number* within that mark, and a *compilation cycle* number within that version.

| **Parameter** | **Type/Description** | |
|---|---|---|
| *lpMark* | **LPINT** | Pointer to an **int** to receive the mark number. |
| *lpVersion* | **LPINT** | Pointer to an **int** to receive the version number. |
| *lpCycle* | **LPINT** | Pointer to an **int** to receive the cycle number. |

**Return Value** The function has no return value.

# DATA TYPES AND STRUCTURES

This section describes the data types and structures that are defined by **TextView**. Definitions of all the items described here are contain in the header file **`textview.h`**.

## Data types

The data types in the following list are key words that define the size and meaning of arguments and return values associated with **TextView** functions.

| Type | Definition |
|---|---|
| **LPTVWSTATUS** | Long pointer to a **TVWSTATUS** data structure. |

## Data structures

This section lists data structures that are used by **TextView**.

## TVWSTATUS

## Status Information for a TextView Window

The **TVWSTATUS** structure contains information giving the current status of a **TextView** window.

```
typedef struct sTVWSTATUS
{
     DWORD      dwFlags;
     int        nMaxLines;
     int        nLinesStored;
     BOOL       nSuspended;
     BOOL       nReturningData;
     BOOL       nRedraw;
     WORD       nScrollState;
     int        nTabSize;
     int        nMessagesLost;
     int        nRows;
     int        nColumns;
     int        nTopLine;
     int        nNextRow;
     COLORREF   crColor;
     COLORREF   crBkColor;
} TVWSTATUS;
```

The TVWSTATUS structure has the following fields:

| Parameter | Description |
|---|---|
| **dwFlags** | Contains a copy of the *dwFlags* argument passed to the **TVCreateWindow** call that created the window. |
| **nMaxLines** | Specifies the maximum number of lines that may be stored with the window. |
| **nLinesStored** | Specifies the number of lines currently stored with the window. |
| **nSuspended** | Specifies whether the window is currently in *suspended* state. |
| **nReturningData** | Specifies whether **TextView** is currently servicing a **TVReturnData** call for this window. |
| **nRedraw** | Specifies whether lines written to the window are being displayed immediately, or are being held until a window redraw is permitted. |
| **nScrollState** | Specifies the current scroll state of the window. The value will be either **TV_SCR_AUTO** or **TV_SCR_MANUAL**. |
| **nTabSize** | Specifies the width of a tab stop, in characters. |
| **nMessagesLost** | Specifies the number of messages that have been lost by being written to the window while it is in manual scroll state. The value is cleared whenever the scroll state is set to automatic. |
| **nRows** | Specifies the current depth of the window, in text rows. |
| **nColumns** | Specifies the current width of the window, in average characters. |
| **nTopLine** | Specifies the line number of the line currently displayed at the top of the window. |
| **nNextRow** | Specifies the window row to which the next line of text will be written (the first row is numbered zero). If the value is negative, the window is full and will need to be scrolled to accept the next line. |
| **crColor** | The current text colour. |
| **crBkColor** | The current background colour. |

# APPENDIX : THE DEMONSTRATION PROGRAM

Supplied with the **TextView** DLL in the full distribution set are full sources for a demonstration program that both show you what the system can do, and supplement the information in this guide on how to program with it.



## A real-time program trace facility

The purpose of the demonstration program is to show you how you can use **TextView** to add a real-time tracing facility to your application.

Of course, there are many ways to debug a program while you are developing it. You can use Microsoft's CodeView system to monitor it; but this is relatively clumsy and, until CodeView 3.05, required you to have two monitors attached to your system. Even with single monitor capability, CodeView debugging can be a time-consuming business.

If you know the area where a bug is occurring, you might plant **MessageBox** calls at suitable points to stop the program and display details. This, though, is very laborious, requiring you to click an *OK* box every time; and if your bug is deep inside a loop might well be totally impractical.

**TextView** lets you add a real-time trace facility to your program with ease. In essence, you can regard it as giving you the ability you have in DOS programs of performing **printf** calls to the screen: the messages appear as you write them, and scroll off the top of the screen to make room for new ones. However, with **TextView** you have a permanent record of as many of the lines as you choose, and can at any time scroll back through them.

Typically, you would add simple text output calls to your program as you develop it, writing out progress records and useful values as you go. The **trace** routine in the supplied sources is a basic example of such a routine. Its action is controlled by a simple *on or off* flag that is settable from a menu, so you can activate or de-activate tracing whenever you want. Of course, you could add more sophisticated criteria, to allow you to trace, for example, only certain categories of events or at certain debug levels.

## Running the demonstration

To run the demonstration you must first have installed the **TextView** DLL as described earlier. Then, you can start it from the Program Manager, the File Manager or any other program launcher you may have.

## Trace menu items

All the actions of the demonstration program are controlled by options in the trace menu. Some of the options control the trace system's operation; others simulate an application writing messages as it runs.

### Start tracing

This option activates tracing. If a trace window does not already exist, it causes a call to **TVCreateWindow** to create one. The global *tracing* flag is set to *on*, so that the trace routine becomes active.

### Stop tracing

Sets the global *tracing* flag to *off*, so that the trace routine does nothing. The trace window is not affected. These two options show on way of controlling the system from your application.

### Write one message

Causes one call to the trace routine to write a message. Depending on the setting of the global *tracing* flag, text either will or will not appear in the trace window.

### Write batch

Writes a batch of 200 messages in a loop, to demonstrate how the **TextView** window scrolls.

### Write batch fast

Writes a batch of 200 messages in a loop. Unlike the previous option, the **TVSetRedraw** function is called to inhibit updating of the **TextView** window while this is being done. The effect is that the messages are written substantially faster.

### Reset trace window

Calls the **TVResetWindow** function to clear all data stored in the **TextView** window.

### Show trace window

This option uses normal Windows API functions to make the **TextView** window visible, and bring it to the top of the screen. It shows you one case in which you can manipulate the window with normal Windows functions as well as with **TextView** functions.

### Show trace status

Calls the **TVGetWindowStatus** function to display some details of what the **TextView** window is doing.

### Trace window scrolling

This menu option shows you a further popup menu item containing two items: *Automatic* and *Manual*. These let you change the scrolling state of the **TextView** window from the *application's* menu.

You can also change the scrolling state from the *Scrolling* item in the **TextView** window's *own* menu. Note how the application uses the menu notification facility to keep the check marks on its menu item in line with the window state when this is done.

### Copy trace window

This menu item creates a second **TextView** window, and uses the **TVReturnData** function to copy all the data from the first one into it.

### Kill trace window

This destroys the **TextView** window and deletes all the stored data.

## Source files

The demonstration program source comprises a number of modules, which are described briefly below. For full details you should consult the actual sources, which contain a large amount of explanatory comment.

### main.c

This is the entry point module for the demonstration program. It contains the message processing routine that services messages sent to the main window. Note, of course, that the application contains *no* code at all to service messages sent to the **TextView** window - the DLL handles all of this itself.

### trace.c

This module contains the routines that service most of the *Trace* menu items, and the *trace* routine itself. This routine is written to accept a format string, such as you pass to *printf*, and a variable number of arguments of any type.

Also in this module is the *menu handler* function. This routine is called back from the Windows DLL to notify the application when the user clicks an item in the **TextView** window's menu.

### copy.c

This module handles the *Copy Trace Window* menu item. It sets up a second **TextView** window and calls **TVReturnData** to read the contents of the first.

### utils.c

This contains a few small utility routines.

## Building the program

The demonstration sources are written to be compiled with Microsoft C 6.00A. You may need to make adjustments if you have a different compiler.

The makefile is written for a UNIX-compatible make program such as *ndmake*. If you use the Microsoft *nmake* utility in the Programmer's Work Bench you will need to amend it.