TransEdit

A TransSkel Edit Window Module

Release 3.05
26 February 1994

Introduction

This document describes TransEdit, a plug-in module that runs on top of the TransSkel Macintosh application skeleton, and that may be added to any TransSkel project to provide an arbitrary number of text editing windows. The editing operations provided are quite basic, but the module is sufficiently open-ended that the host application can add another layer of functionality on top of it. If you want to present display-only windows, TransEdit is overkill. A related module, TransDisplay, may be used instead.

The host application may exert quite a bit of control over edit windows if it wishes, but the minimum investment required to use them is small. With a single call, TransEdit provides a standard document window that does the following with no intervention on the part of the host:

- Keyboard input handling (including display of the line containing the caret when the Enter key is typed).
- Caret positioning and text selection with the mouse (including autoscrolling).
- Text scrolling with the scroll bar.
- Window resizing and updating.
- Window activation and deactivation (including highlighting and unhighlighting of the scroll bar, size box and selected text or the caret).
- Keeping track of whether the text in a window has been modified.
- Asking the user whether to save to a file when the close box of a window containing changed text is clicked.
- Asking for confirmation of revert operations on windows containing changed text.
- Changing the cursor to a watch during file I/O, and to an I-beam otherwise when the cursor is in the window's text area.

In addition, facilities are provided to do the following with little effort:

- Pass Edit menu selections to TransEdit.
- Change font, point size, word wrap type and justification of a window.

- Allow external modification of the text and update forcing.
- Bind a window to a file at window creation time.
- Perform save operations to the current file, or save under a different name (optionally changing the file that the window is bound to).
- Revert to version of file on disk.
- Edit windows may be told to report key, activate and close events to the host. This is useful for applications that enable or disable menu items according to which window is frontmost and whether the window is dirty.

TransEdit windows use standard TextEdit operations, and so are bound by the usual TextEdit limitations. In particular, TransEdit windows are limited to 32K characters of text.

Distribution Information

TransEdit is public domain, so distribution is unrestricted. I am interested in hearing about any additions or corrections, for possible inclusion in future releases. I may be reached via electronic mail at *dubois@primate.wisc.edu* or via U.S. mail at:

Paul DuBois
Wisconsin Regional Primate Research Center
1220 Capitol Court
Madison, WI 53715-1299
USA

The version of TransEdit described in this document is written for THINK C 6.0. THINK C is a trademark of:

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014 USA

This distribution of TransEdit consists of:

Documentation
- Release Notes
- Manual

TransEdit library Csource code
- TransEdit.c —TransDisplay source
- FakeAlert.c — special "resourceless alert" code used by TransEdit.c

Demonstration application source code
- TinyEdit — minimal TransEdit application
- DumbEdit — multiple window editor

Interface files
- TransEdit.h — C header file
- TransEdit.intf — Pascal interface file
- TransEdit — binary TransEdit library document

The remainder of this document describes the demonstration programs included in the distribution and provides a detailed specification of the TransEdit interface. TransEdit 3.0

requires TransSkel 3.0. Familiarity with TransSkel is assumed.

Demonstration Applications

The demonstrations are an introduction to the ways in which TransEdit can be used. TinyEdit shows how to use an edit window with a minimal amount of work. DumbEdit demonstrates how edit windows may be fully integrated into an application, including appropriate menu item enabling/disabling.

TinyEdit

This demonstration puts up an Apple menu with desk acessories in it, a File menu with a New, Open and Quit items. A single edit window is provided. To begin a new file, select New. To open

an existing file, select Open. To stop editing the window, click its close box or select Close. Terminate the application by selecting Quit from the File menu, or by typing Command-Q.

DumbEdit

This demonstration shows how multiple edit windows may be used. It also demonstrates how to use the full set of TransEdit calls.

The TransEdit Interface — General Information

TransEdit.c contains the source of the TransEdit module. *FakeAlert.c* contains some auxiliary code that is used by *TransEdit.c*. These files can be made into a project or library document for inclusion in the your application project document, or the source can be included directly in your project.

The interface to the host application is procedural, but you should `#include` the header file *TransEdit.h* in source files containing TransEdit calls.

The available routines are:

`NewEWindow()`	Create edit window
`GetNewEWindow()`	Create edit window from resource template
`SetEWindowProcs()`	Install event notification procedures
`SetEWindowStyle()`	Set window text display characteristics
`GetEWindowTE()`	Return handle to window's text
`GetEWindowFile()`	Get name of file bound to window
`EWindowOverhaul()`	Redo display
`IsEWindow()`	Test whether a window is an edit window
`IsEWindowDirty()`	Test whether edit window is dirty
`EWindowClose()`	Close edit window
`ClobberEWindows()`	Shut down all existing edit windows
`EWindowSave()`	Save window's text to file
`EWindowSaveAs()`	Save window's text to another file, change window name
`EWindowSaveCopy()`	Save window's text to another file without changing name
`EWindowRevert()`	Revert to version of file stored on disk
`EWindowEditOp()`	Perform Edit menu operation on window
`SetEWindowCreator()`	Set creator of files created by TransEdit

All other variables and procedures are declared `static`, to preclude name conflicts with your application.

The general logic of applications using TransEdit is:

- Initialize TransSkel, plus whatever other initialization is desired.
- For each edit window to be used, call `NewEWindow()` or `GetNewEWindow()` to create it.
- Call `SetEWindowProcs()` if desired to install event notifiers for the window.
- Pass selections from the Edit menu to the edit window when it is frontmost.
- If the host wishes to get rid of the edit window, call `EWindowClose()`. To destroy all the edit windows at once, call `ClobberEWindows()`.

TransEdit operates fairly autonomously. The only thing it needs from the host is to receive item numbers when a selection is made from the Edit menu whenever an edit window is active. The host, on the other hand, may wish to know what TransEdit is or has been doing, for purposes of integrating it more seamlessly into the Macintosh-style interface.

For example, most programs that provide editing capabilities have Open, Close and Save items under the File menu. Many also have Save As and Revert. Typically items like Close and Save would only be enabled when an edit window is active. Additionally, Save might be disabled if the active edit window is not bound to (associated with) a file, or if it is bound to a file but not dirty. Revert might be enabled if the active edit window is bound to a file and is dirty.

For such reasons, the host may wish to be provided with information about:

- window activation/deactivation/closing
- the state of the window contents (clean/dirty)
- the type of the window contents (whether bound to file or not)

Besides needing to be able to solicit information about edit windows, the host also needs to be able to tell an edit window to do certain things, e.g., save itself, save itself under a different name, close itself. Typically, file operations give the user the option of canceling the operation. This is provided as well.

If an edit window is bound to a file when it is created, its initial contents are read in from the file.

An unbound window may become bound to a file by being saved to that name. A window that is bound to a file may be bound to a different file by being saved to the new name. It is also possible to save a window to a file without binding it to the file.

An edit window is initially clean. Keyboard input or edit operations selected from the Edit menu (or the keyboard equivalents) make the window dirty. A dirty window becomes clean again when it is saved to a file that it is bound to, or when it is reverted.

The TransEdit Interface — Procedural Specification

Each of the TransEdit interface routines is described in detail below.

Except where noted, routines expecting a `WindowPtr` to an edit window do nothing if the pointer is not pointing to an edit window.

```
pascal WindowPtr
NewEWindow (Rect *bounds, StringPtr title, Boolean visible,
                          WindowPtr behind, Boolean goAway,
```

```
                                long refNum, Boolean bindToFile);
```

`NewEWindow()` creates a new edit window. The window is created as a standard document window, with a size box and a scroll bar along the right edge. The current port is preserved. A pointer to the new window is the return value. If the window could not be created the return value is `nil`. The window is subject to whatever the current TransSkel window sizing defaults are. They may be changed with `SkelSetGrowBounds()` in the usual manner.

Most of the other parameters have meanings similar to the corresponding parameters of the Toolbox routine `NewWindow()` (see Inside Macintosh), with the following exceptions.

The `bindToFile` parameter determines whether the window is to be bound to a file when it is created. If not, the window is created as an empty window not bound to any file. If so, a Standard File dialog is presented so the user may select a file. (If the user cancels the file dialog or there is an error reading the file, no window is created and `NewEWindow()` returns `nil`.)

The meaning of the `title` parameter interacts with `bindToFile`. If `bindToFile` is `true`, the window title is always the file name. Otherwise, if `title` is not `nil` or the empty string, it is used for the window title. If `title` is `nil` or empty, the window is untitled. The first such title is "untitled", subsequent untitled windows are named "untitled 1", "untitled 2", etc.

The default text display characteristics for edit windows are monaco 9-point font, word wrap on, and left justification. By default there are no event notification procedures. These values may be changed with `SetEWindowStyle()` and `SetEWindowProcs()`.

To destroy an edit window and its data structures, pass the window pointer to `EWindowClose()`. Pay attention to the return value of `EWindowClose()`, though. If it returns `false`, the window was not closed. (See description of `EWndowClose()` below to understand why.)

```
pascal WindowPtr
GetNewEWindow (short resourceNum, WindowPtr behind,
                                  Boolean bindToFile);
```

`GetNewEWindow()` is like `NewEWindow()` except that it creates the window from the 'WIND' resource with the given ID number.

Watch out for the following when using resources to create new edit windows:

- The `procId` value from the template is ignored.
- Set the title to the empty string if you want the window to come up untitled when it's not bound to a file. Otherwise the title in the resource will be used.
- If your application that creates multiple edit windows, you may want to specify that the window is invisible in the resource. Then you can position the window before calling `ShowWindow()` to make it visible.

```
pascal void
SetEWindowProcs (WindowPtr w,
                 TEditKeyProcPtr pKey,
                 TEditActivateProcPtr pActivate,
```

```
                    TEditCloseProcPtr pClose);
```

`SetEWindowProcs()` associates procedures with `w`, to be called when `w` receives an activate or deactivate event, mouse or key click, or the close box is clicked. Any of the procedure parameters may be `nil`, to disable notification for the corresponding event type.

If `w` is `nil`, `pKey`, `pActivate`, and `pClose` becomes the default notification procedures associated with new edit windows created with subsequent calls to `NewEWindow()` or `GetNewEWindow()`.

The key and close box click procedures, if they are not `nil`, should be declared to take no parameters:

```
pascal void
MyKey (void)
```

```
{
}

pascal void
MyClose (void)
{
}
```

The activate event procedure should be declared to take a single boolean parameter, which will be `true` if the window is coming active, `false` if it is going inactive:

```
pascal void
MyActivate (Boolean active)
{
}
```

When any notification procedure is called, the edit window to which the event applies is the current port, so it may be obtained with the QuickDraw `GetPort()` procedure. This is useful when you associate a notification procedure with more than one window. TransEdit handles activating the window properly (e.g., highlighting the scroll bar and text and drawing the size box appropriately), before calling the notification procedure.

Notification is useful for applications that change enabling of menu items according to which window is frontmost. The reason for notification of key clicks is that a window becomes dirty as soon as keyboard entry occurs in it. The host may wish immediate notification of this.

If no close procedure is installed, the window is closed with `EWindowClose()`; see below.

```
pascal TEHandle
GetEWindowTE (WindowPtr w);
```

`GetEWindowTE()` returns a handle to the TextEdit record associated with `w`, or `nil` if it's not an edit window. This call allows the host to perform arbitrary text operations not supported by the standard TransEdit calls.

```
pascal Boolean
GetEWindowFile (WindowPtr w, SFReply *fileInfo);
```

`GetEWindowFile()` returns `true` if `w` is bound to a file and `false` if it is not, or if `w` is not an edit window. Additionally, if `w` is bound to a file, a copy of the file info is placed in the `SFReply` pointed to by the second parameter. The `fName` and `vRefNum` fields of this record indicate the name and location of the file.

If the host only wants to know whether `w` is bound to a file or not, `nil` may be passed for the second parameter.

```
pascal void
SetEWindowStyle (WindowPtr w,
                 short font,
                 short size,
                 short wrap,
                 short just);
```

`SetEWindowStyle()` sets the text display characteristics for `w`. The value of `wordWrap` should be non-negative to specify wrapping on, negative to specify wrapping off. The

justification values are the usual TextEdit justification constants `teJustLeft`, `teJustCenter`, and `teJustRight` to specify left, center or right justification, respectively.

If `w` is `nil`, the style parameters become the defaults for new edit windows created with subsequent calls to `NewEWindow()` or `GetNewEWindow()`.

`SetEWindowStyle()` takes care of text display recalculation and redrawing.

```
pascal void
EWindowOverhaul (WindowPtr w, Boolean showCaret,
                                        Boolean recalc, Boolean dirty);
```

`EWindowOverhaul()` redoes the display of the text area of `w`. It is used when the host does some non-TransEdit operation, such as loading the text into the text record itself. If `showCaret` is `true`, the text is scrolled to show the caret. If `recalc` is `true`, the `lineStarts` of the text record are recalculated. The window is set dirty or not according to the value of `dirty`.

```
pascal Boolean
IsEWindow (WindowPtr w);
```

`IsEWindow()` returns `true` if `w` is an edit window, `false` otherwise.

```
pascal Boolean
IsEWindowDirty (WindowPtr w);
```

`IsEWindowDirty()` returns `true` if the text in `w` is dirty, `false` if the text is not dirty or if `w` is not an edit window.

A window is dirty if the text has been changed since the last save (or since the window was created, if the window is not bound to a file). Key clicks and the Cut, Paste, and Clear operations from the Edit menu (as well as the associated command-key equivalents) are considered to change the text.

A window is initially clean. A dirty window is made clean again by saving it to disk (unless it is not bound to the file it's saved to), or reverting.

```
pascal Boolean
EWindowClose (WindowPtr w);
```

Close `w`. If it's dirty and is either bound to a file or (if not bound) has some text in it, the user is asked about saving it first, and given the options of saving changes, tossing them, or canceling altogether.

`EWindowClose()` returns `true` if the file was saved (or the changes were discarded) and the window was closed, `false` if the user canceled or there was an error or `w` isn't an edit window. The return value indicates whether the window was closed, not whether its contents were saved: if the user discards changes to a dirty window, the window is closed without saving and `EWindowClose()` returns `true`.

Note

Normally `EWindowClose()` should be called to shut down edit windows, rather than `SkelRmveWind()`; if the window is dirty, `SkelRmveWind()` won't ask about saving the text. The host should also always inspect the return value, since it's not safe to assume the user didn't cancel the close.

```
pascal Boolean
ClobberEWindows (void);
```

`ClobberEWindows()` is typically called when the user selects Quit from the File menu. It finds each existing edit window and tries to shut it down by calling `EWindowClose()`. If the user saves or discards the changes for each dirty window, so that all windows are shut down properly, `ClobberEWindows()` returns `true` and the host may quit. If the user cancels a save request for any dirty window, or if there is some error, `ClobberEWindows()` returns `false`; the host should not quit.

```
pascal Boolean
EWindowSave (WindowPtr w);
```

Save the contents of `w` under its current name.

If the window is not bound to a file, a name is requested (using a Standard File dialog). The window contents are saved to that file, the window becomes bound to it and `true` is returned. If the user cancels the request or there is an error writing the file, the window remains unbound, and `EWindowSave()` returns `false`.

The window becomes clean if the file is successfully saved.

```
pascal Boolean
EWindowSaveAs (WindowPtr w);
```

Save the contents of `w` under a new name (obtained with the standard `SFPPutFile()` dialog) and bind it to that name. Returns `false` if the user cancels `SFPPutFile()` or an error occurs while writing the file (in either case the window is not bound to the new name.)

The window becomes clean if the file is successfully saved.

```
pascal Boolean
EWindowSaveCopy (WindowPtr w);
```

Save the contents of `w` under a new name (obtained with the standard `SFPPutFile()` dialog) without binding it to that name. Returns `false` if the user cancels `SFPPutFile()` or an error occurs while writing the file.

`EWindowSaveCopy()` does not change whether the window is dirty or not.

```
pascal Boolean
EWindowRevert (WindowPtr w);
```

Revert to saved version of file on disk. `w` must be an edit window, and must be bound to a file. Returns `false` if one of these conditions is not met, or if they are met but there was an error reading the file.

If the window is dirty, the user is asked whether to really revert. If the user cancels the request, `EWindowRevert()` returns `false`. A successfully reverted window becomes clean again.

```
pascal void
EWindowEditOp (short item);
```

When the user selects an item from the Edit menu and an edit window is active, pass the item number to `EWindowEditOp()` for processing. The Edit menu is assumed to have the standard items Undo, gray line, Cut, Copy, Paste, and (optionally) Clear, in that order. (It may have other items following these, but the host should handle them itself.)

Undo is not supported. This means that when an edit window is frontmost, your application should dim the Undo item, unless it provides some way for Undo to be performed.

The Cut, Paste, and Clear edit operations make an edit window dirty. Copy does not, and Undo is not supported.

```
pascal void
SetEWindowCreator (OSType creat);
```

`SetEWindowCreator()` sets the file creator for any files created by TransEdit.

Using Notification Procedures

The notification procedures for an edit window, if any are installed, are called whenever an appropriate event occurs for the window, allowing the host to take appropriate action (such as menu enabling or disabling). The key click notification procedure is called whenever keyboard input occurs in an edit window. The activate notification procedure is called whenever the window is activated or deactivated. The close procedure is called when the user clicks in the close box. If no close procedure is installed, TransEdit simply does an `EWindowClose()` on the window, otherwise it executes the installed procedure, and the host can take appropriate action — perhaps simply hiding the window.

Generally, Macintosh applications allow the user the option of closing windows via a Close item in the File menu. If the host provides such an option, you can cause whatever close procedure is associated with an edit window to be executed like this:

```
SkelClose (FrontWindow ());
```

All notification procedures may assume that the current port is set to the edit window to which the event applies, and may obtain the port by calling `GetPort()`.