

23

DRAG AND DROP

Demonstration Program: Drag

Overview

The Drag Manager

Using the Drag Manager, you can provide your application with the capability to drag objects from your application's windows, to your application's windows, and within your application's windows.

The Three Phases of a Drag

All drag and drop operations comprise the following three distinct phases:

- Starting the drag.
- Tracking the drag.
- Finishing the drag.

Different applications may be involved in each of these three phases. For example, when the user drags an item from one application to another, the source application starts the drag, other applications through whose windows the user drags the item may track the drag, and the application owning the target window finishes the drag by accepting the drop (see Fig 1).

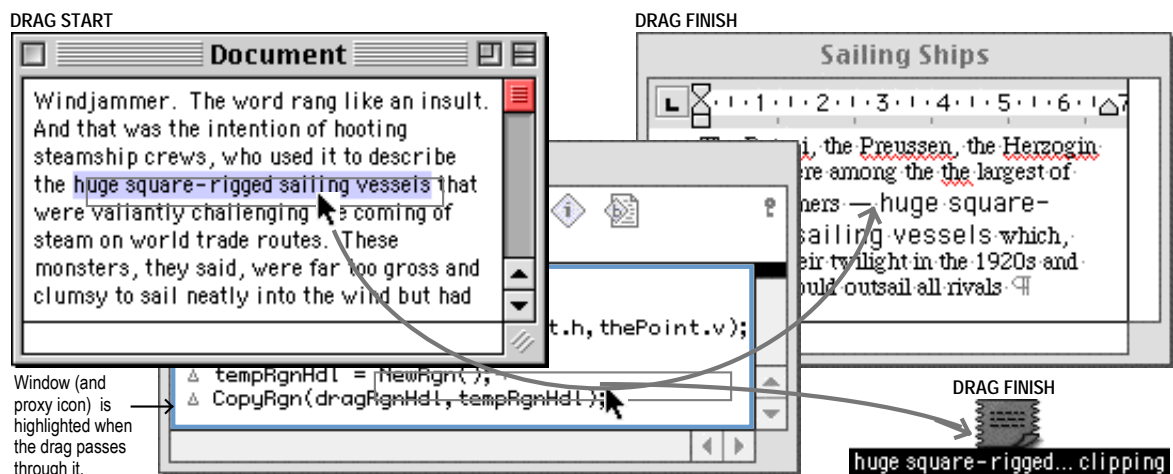


FIG 1 - STARTING, TRACKING, AND FINISHING A DRAG IN DIFFERENT APPLICATIONS

Of course, when the drag and drop is wholly within one of an application's document windows, that single application starts the drag, tracks it through the window, and accepts the drop.

Starting the Drag

A drag is initiated when the user clicks on a selected object, keeps the mouse button down, and begins to move the mouse (see the left hand window at Fig 1). Using the function `WaitMouseMoved`, you can determine if the mouse has moved far enough to initiate a drag. If `WaitMouseMoved` returns `true`, your application should call `NewDrag` to create a new **drag reference**. A drag reference is an opaque data type that you use to refer to a specific drag process.

Having created a drag reference, and for each item in the drag, you must then add at least one **drag item flavour** to that reference using the function `AddDragItemFlavor`. Drag item flavours represent the various data formats in which each individual item may be produced. The concept is similar to that applying to Scrap flavours as described at Chapter 20.

With the drag item flavour (or flavours) added, your application can then initiate tracking by calling the function `TrackDrag`.

Tracking the Drag — Drag Tracking Handlers

The Drag Manager tracks the drag through each window the cursor moves over as the user drags the selection across the screen. During this process, the user is presented with the following feedback:

- Destination highlighting will occur when the cursor has left the source location. (See the middle window at Fig 1, whose owner application supports drag and drop and can accept the drop).
- If a container under the cursor (such as a folder in the Finder) can accept the drop, that container will be highlighted.

As the cursor moves through, for example, the middle window at Fig 1, the Drag Manager calls a function, called a **drag tracking handler**, defined by that window's owner application. Drag tracking handlers inspect the description of the item, or items, being dragged and highlight the window if this inspection reveals that the window can accept the drop. The application's drag tracking handler uses the functions `ShowDragHilite` and `HideDragHilite` to create and remove the highlighting. (The acceptable drop region, and thus the region highlighted, does not necessarily have to be the whole of the window's content region.)

Finishing the Drag — Drag Receive Handlers

When the user releases the mouse button in, for example, the right hand window at Fig 1, the Drag Manager calls a function, called a **drag receive handler**, defined by that window's owner application. Drag receive handlers accept the drop and insert the selection at its final destination.

More on Drag Items, Drag Item Flavours, and Handlers

Drag Items and Item Reference Numbers

A **drag item** is a single distinct object, which could be, for example, a selection from a painting in a painting program, a selection of objects in a drawing program, or a continuous range of text in a text editing program. A discontinuous selection of text (resulting from using the Command key in some programs) would result in multiple drag items.

A new drag item is created if the **item reference number** passed in the `theItemRef` parameter of a call to `AddDragItemFlavor` is different from any other item reference number.¹

¹ In many cases it is easiest to use index numbers as item reference numbers (e.g., 1, 2, 3...). Item reference numbers are only used as unique "key" numbers for each item. Item reference numbers do not need to be given in order, nor must they be sequential. Depending on your application, it might be easier to use your own internal memory addresses as item reference numbers (as long as each item being dragged has a unique item reference number).

Drag Item Flavours

Any item that can be dragged can generally be represented using several different flavours (that is, data formats). At the start of a drag, your application must inform the Drag Manager, using the function `AddDragItemFlavor`, of the flavours that you can provide to the receiver of the drag. For example, if the drag item is a selection of text, you might elect to provide the data in both the standard 'TEXT' format and in the Rich Text Format (RTF). Alternatively, instead of supplying the data in RTF, you might supplement the 'TEXT' data with 'styl' data. You might also provide the data in your application's own internal data format to cater for a drag and drop to one of your application's documents, including the source document.

Your application adds additional flavours to an item by passing the same item reference number in the `AddDragItemFlavor` call as was used in the `AddDragItemFlavor` call which created the item and added the first flavour.

Different destinations may prefer different data formats, and there is no certainty as to whether a drag will contain the preferred, or an acceptable, flavour. Thus, as previously stated, the destination application needs to inspect the description of the items being dragged to determine whether an acceptable flavour is available and only highlight the relevant window if it can accept a drop. The function `GetFlavorFlags` is used to determine whether the drag contains a specific flavour type.

Flavour Flags

The following flavour flags, which are used by the Drag Manager and its clients to provide additional information about a specific drag item flavour, may be passed in the `theFlags` parameter of the function `AddDragItemFlavor` and retrieved by the function `GetFlavorFlags`:

<i>Flag</i>	<i>Description</i>
<code>flavorSenderOnly</code>	Set by the sender if the flavour should only be available to the sender.
<code>flavorSenderTranslated</code>	Set if the flavour data is translated by the sender. Useful if the receiver needs to ascertain whether the sender is performing its own translation to generate this data type.
<code>flavorNotSaved</code>	Set by the sender if the flavour data should not be stored by the receiver. Useful for marking flavour data that will become stale after the drag is completed.
<code>flavorSystemTranslated</code>	Set if the flavour data is translated by the Translation Manager. If this flavour is requested, the Drag Manager will acquire the required data type from the sender and then use the Translation Manager to provide the data requested by the receiver.

Note that the Finder does not save flavour types marked with the `flavorSenderTranslated`, `flavorNotSaved`, and `flavorSystemTranslated` flags into clippings files.

Drag Handlers

Drag tracking handlers and drag receive handlers are callback functions that your application registers with the Drag Manager using the functions `InstallTrackingHandler` and `InstallReceiveHandler`. The window reference of the window on which the handlers are to be installed is passed in the `theWindow` parameter of the two handler installer functions. You can install more than one tracking handler and more than one receive handler on the same window, in which case the Drag Manager calls each of the handlers in the order they were installed.

If you pass `NULL` in the `theWindow` parameter of the handler installer functions, the Drag Manager will register the handler in a special area that is used when the drag occurs in any window of your application. Handlers installed in this special area are called **default handlers**.

Drag Tracking Handlers

As the user moves the mouse through your application's windows during the drag, the Drag Manager sends status messages to your tracking handler. These messages are as follows:

Message	Description
Enter handler	Received when the focus of the drag enters a window handled by your tracking handler from a window not handled by your tracking handler.
Enter window	Received when the focus of a drag enters any window handled by your tracking handler.
In window	Received as the user drags within a window handled by your tracking handler.
Leave window	Received when the focus of a drag leaves a window that is handled by your tracking handler.
Leave handler	Received when the focus of a drag enters a window that is not handled by your tracking handler.

Fig 2 illustrates the receipt of tracking messages involving multiple applications and windows.

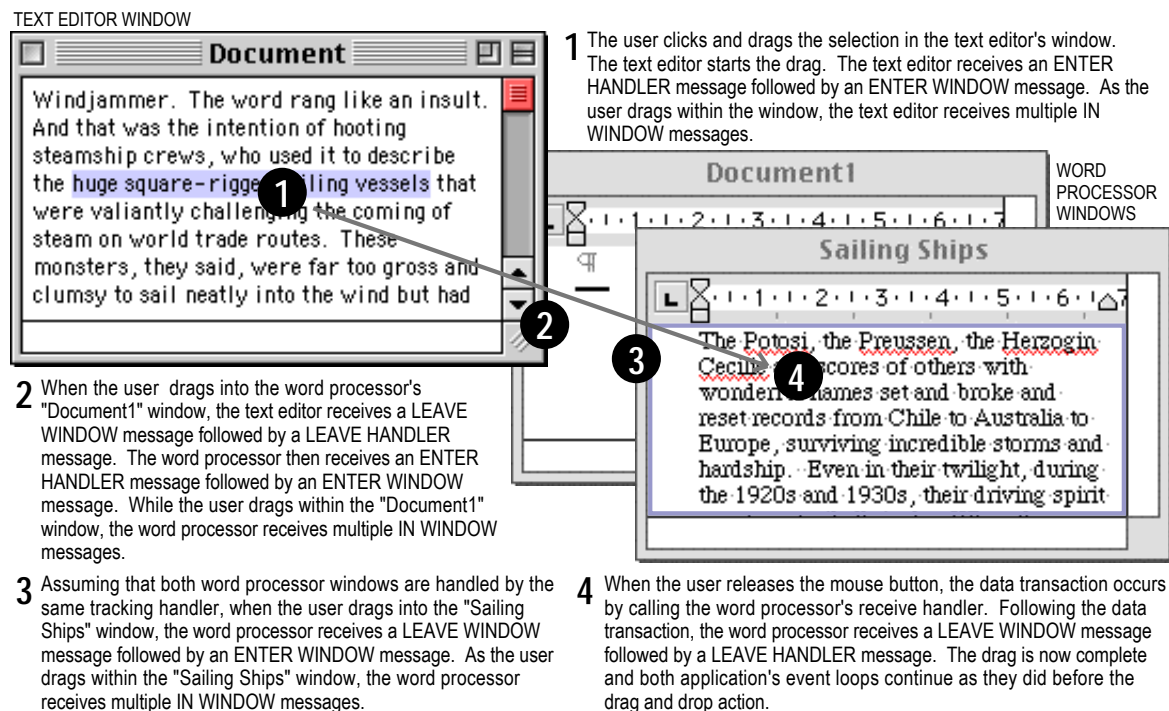


FIG 2 - DRAG TRACKING THROUGH MULTIPLE APPLICATIONS AND WINDOWS

Drag Receive Handlers

When the user drops an item, or a collection of items, in one of your application's windows, the Drag Manager calls any receive handlers installed on the destination window. Your receive handler can then request the drag item flavour, or flavours, that you wish to accept and insert the data at its final destination.

When the drop occurs within the window in which the drag originated, and the user did not press the option key either before the mouse button went down or before it was released, receive handlers must delete the original selection before inserting the dropped data. If the option button was not pressed, receive handlers must not delete the selection.

Creating the Drag Region and Setting the Drag Image

Creating the Drag Region

Recall that your application initiates tracking by calling the function `TrackDrag`. This function takes a region, specifically, the **drag region**, in its `theRgn` parameter. This is the region, drawn by the Drag Manager in the dithered gray pattern (Mac OS 8/9) or gray colour (Mac OS X), that moves with the mouse during the drag (see Fig 1).

As an example, to create a drag region when a single item is selected, your application should:

- Copy the drag item's region to a temporary region and inset the temporary region by 1 pixel.
- Use `DiffRgn` to subtract the temporary region from a copy of the item's region.

The resulting copy of the item's region has the same outline as the item's original region but is only one pixel thick (see Fig 3).

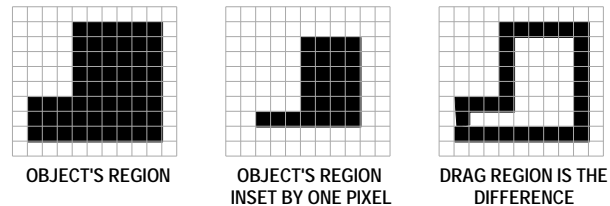


FIG 3 - CREATING THE DRAG REGION

Setting the Drag Image — Translucent Drags

The drag region may also be visually represented to the user by a translucent image of the original selection (see Fig 4).

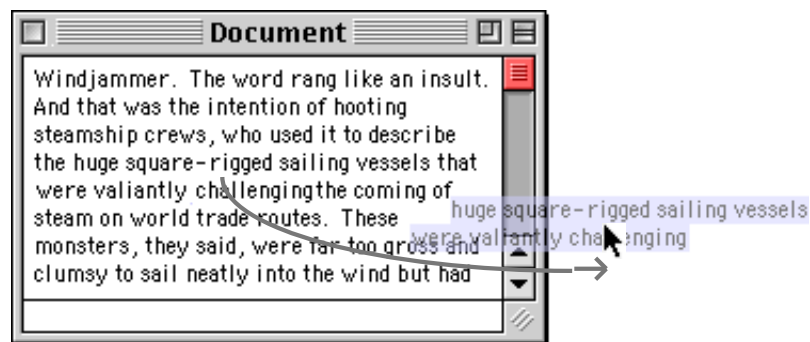


FIG 4 - A TRANSLUCENT DRAG

To support translucent dragging of a text selection on Mac OS 8/9, you should:

- Create an offscreen graphics world the size of the bounding rectangle of the highlight region. The offscreen graphics world should be eight bits deep.
- Copy the area of the window enclosed by the highlight region's bounding rectangle to the offscreen graphics world.
- Create a region for a mask which is exactly the same shape as the highlight region. (Note that, if you do not supply a mask, the entire rectangular area of the offscreen pixel map, including white space, will be dragged.)
- Call `SetDragImage` to associate the image with the drag reference, passing in the handle to the offscreen pixel map, the handle to the mask region, and a constant specifying the required level of translucency. `SetDragImage` installs a custom drawing function (see below) to perform the translucent drawing.

The level of translucency may be specified using one of the following **drag image flags**:

<i>Constant</i>	<i>Value</i>	<i>Description</i>
<code>kDragStandardTranslucency</code>	0L	65% image translucency. This is the recommended level.
<code>kDragDarkTranslucency</code>	1L	50% image translucency.
<code>kDragDarkerTranslucency</code>	2L	25% image translucency.
<code>kDragOpaqueTranslucency</code>	3L	0% image translucency (opaque).

Drag Functions — Overriding Drag Manager Default Behaviour

If you wish to override the Drag Manager's default behaviour, you can supply it with several different kinds of **drag function**. Only the sender can specify drag functions.

Drawing Function

When a drawing function is installed, the Drag Manager sends that function a sequence of messages that allow the application to assume responsibility for drawing the drag region (or similar feedback) on the screen. If translucent dragging is used, you must not install a custom drawing function.

Send Data Function

If you install a **send data function**, the Drag Manager calls that function when the receiver application requests a drag item flavour that the Drag Manager does not currently have the data cached for.

Ordinarily, the Drag Manager caches the flavour data for all flavours that were added to the drag with the `AddDragItemFlavor` function. If a receiver calls `GetFlavorData` to get a flavour's data, the Drag Manager simply returns the cached data to the receiver.

However, if your application passes `NULL` as the pointer to the flavour data in the `AddDragItemFlavor` call, the Drag Manager does not cache the data. In effect, your application has made a "promise" to later supply the data in the event that a receiver requests it. Thus, if a receiver calls `GetFlavorData` for that particular flavour, the Drag Manager will call your send data function to get the data from the sender. Your send data function should then create that flavour's data and call `SetDragItemFlavorData` to send the data to the Drag Manager.

This mechanism of "promising" data that may, in the event, not be called for by a receiver is useful where the sender must perform expensive computations to produce the data or if the resulting data requires a large amount of memory to store.

Drag Input Function

If you install a **drag input function**, the Drag Manager calls that function when sampling the mouse position and keyboard state to allow the application to override the current state of the input devices. The Drag Manager passes the current mouse location, mouse button state, and keyboard modifier status to your drag input function, which can then modify those parameters.

Main Drag Manager Constants, Data Types, and Functions

Constants

Flavour Flags

flavorSenderOnly = (1 << 0)
flavorSenderTranslated = (1 << 1)
flavorNotSaved = (1 << 2)
flavorSystemTranslated = (1 << 8)

Drag Attributes

kDragHasLeftSenderWindow = (1L << 0)
kDragInsideSenderApplication = (1L)
kDragInsideSenderWindow = (1L << 2)

Drag Tracking Handler Messages

kDragTrackingEnterHandler = 1
kDragTrackingEnterWindow = 2
kDragTrackingInWindow = 3
kDragTrackingLeaveWindow = 4
kDragTrackingLeaveHandler = 5

Drag Image Flags

kDragStandardTranslucency = 0L
kDragDarkTranslucency = 1L
kDragDarkerTranslucency = 2L
kDragOpaqueTranslucency = 3L

Drag Drawing Handler Messages

kDragRegionBegin = 1
kDragRegionDraw = 2
kDragRegionHide = 3
kDragRegionIdle = 4
kDragRegionEnd = 5

Data Types

```
typedef struct OpaqueDragRef *DragRef;  
typedef UInt32 DragItemRef;  
typedef OSType FlavorType;  
typedef UInt32 DragAttributes;  
typedef UInt32 FlavorFlags;  
typedef SInt16 DragTrackingMessage;  
typedef SInt16 DragRegionMessage;  
typedef UInt32 DragImageFlags;
```

Functions

Installing and Removing Drag Handlers

```
OSErr InstallTrackingHandler(DragTrackingHandlerUPP trackingHandler, WindowRef theWindow,  
void *handlerRefCon);  
OSErr InstallReceiveHandler(DragReceiveHandlerUPP receiveHandler, WindowRef theWindow,  
void *handlerRefCon);  
OSErr RemoveTrackingHandler(DragTrackingHandlerUPP trackingHandler, WindowRef theWindow);  
OSErr RemoveReceiveHandler(DragReceiveHandlerUPP receiveHandler, WindowRef theWindow);
```

Creating and Disposing of Drag References

```
OSErr NewDrag(DragRef *theDrag);  
OSErr DisposeDrag(DragRef theDrag);
```

Adding Drag Item Flavors

```
OSErr AddDragItemFlavor(DragRef theDrag, DragItemRef theItemRef, FlavorType theType,  
const void *dataPtr, Size dataSize, FlavorFlags theFlags);
```

```
OSErr SetDragItemFlavorData(DragRef theDrag, DragItemRef theItemRef, FlavorType theType,
    const void *dataPtr, Sized ataSize, UInt32 dataOffset);
```

Performing a Drag

```
OSErr TrackDrag(DragRef theDrag, const EventRecord *theEvent, RgnHandle theRegion);
```

Getting Drag Item Information

```
OSErr CountDragItems(DragRef theDrag, UInt16 *numItems);
OSErr GetDragItemReferenceNumber(DragRef theDrag, UInt16 index, DragItemRef *theItemRef);
OSErr CountDragItemFlavors(DragRef theDrag, DragItemRef theItemRef, UInt16 *numFlavors);
OSErr GetFlavorType(DragRef theDrag, DragItemRef theItemRef, UInt16 index, FlavorType *theType);
OSErr GetFlavorFlags(DragRef theDrag, DragItemRef theItemRef, FlavorType theType,
    FlavorFlags *theFlags);
OSErr GetFlavorDataSize(DragRef theDrag, DragItemRef theItemRef, FlavorType theType,
    Size *dataSize);
OSErr GetFlavorData(DragRef theDrag, DragItemRef theItemRef, FlavorType theType, void * dataPtr,
    Size *dataSize, UInt32 dataOffset);
```

Getting and Setting Drag Status Information

```
OSErr GetDragAttributes(DragRef theDrag, DragAttributes *flags);
OSErr GetDragMouse(DragRef theDrag, Point *mouse, Point *globalPinnedMouse);
OSErr SetDragMouse(DragRef theDrag, Point globalPinnedMouse);
OSErr GetDragOrigin(DragRef theDrag, Point *globalInitialMouse);
OSErr GetDragModifiers(DragRef theDrag, SInt16 *modifiers, SInt16 *mouseDownModifiers,
    SInt16 *mouseUpModifiers);
```

Window Highlighting Utilities

```
OSErr ShowDragHilite(DragRef theDrag, RgnHandle hiliteFrame, Boolean inside);
OSErr HideDragHilite(DragRef theDrag);
OSErr UpdateDragHilite(DragRef theDrag, RgnHandle updateRgn);
```

Drag Manager Utilities

```
Boolean WaitMouseMoved(Point initialMouse);
```

Setting Drag Functions

```
OSErr SetDragSendProc(DragRef theDrag, DragSendDataUPP sendProc, void *dragSendRefCon);
OSErr SetDragInputProc(DragRef theDrag, DragInputUPP inputProc, void *dragInputRefCon);
OSErr SetDragDrawingProc(DragRef theDrag, DragDrawingUPP drawingProc, void *dragDrawingRefCon);
```

Setting the Drag Image — Translucent Dragging

```
OSErr SetDragImage (DragRef theDrag, PixMapHandle imagePixMap, RgnHandle imageRgn,
    Point imageOffsetPt, DragImageFlags theImageFlags);
```

Creating and Disposing of Universal Procedure Pointers

```
DragTrackingHandlerUPP NewDragTrackingHandlerUPP(DragTrackingHandlerProcPtr userRoutine);
DragReceiveHandlerUPP NewDragReceiveHandlerUPP(DragReceiveHandlerProcPtr userRoutine);
DragSendDataUPP NewDragSendDataUPP(DragSendDataProcPtr userRoutine);
DragInputUPP NewDragInputUPP(DragInputProcPtr userRoutine);
DragDrawingUPP NewDragDrawingUPP(DragDrawingProcPtr userRoutine);
void DisposeDragTrackingHandlerUPP(DragTrackingHandlerUPP userUPP);
void DisposeDragReceiveHandlerUPP(DragReceiveHandlerUPP userUPP);
void DisposeDragSendDataUPP(DragSendDataUPP userUPP);
void DisposeDragInputUPP(DragInputUPP userUPP);
void DisposeDragDrawingUPP(DragDrawingUPP userUPP);
```

Application Defined (Callback) Functions

```
OSErr myDragTrackingHandler(DragTrackingMessage message, WindowRef theWindow,
    void *handlerRefCon, DragRef theDrag);
OSErr myDragReceiveHandler(WindowRef theWindow, void *handlerRefCon, DragRef theDrag);
OSErr muDragSendDataFunction(FlavorType theType, void *dragSendRefCon, DragItemRef theItemRef,
    DragRef theDrag);
OSErr myDragInputFunction(Point *mouse, SInt16 *modifiers, void *dragInputRefCon,
    DragRef theDrag);
OSErr myDragDrawingFunction(DragRegionMessage message, RgnHandle showRegion,
    Point showOrigin, RgnHandle hideRegion, Point hideOrigin, void *dragDrawingRefCon,
    DragRef theDrag);
```


Relevant TextEdit Function

OSErr TEGetHiliteRgn(RgnHandle region,TEHandle hTE);

Demonstration Program Drag Listing

```
// *****
// Drag.h CARBON EVENT MODEL
// *****
//
// This program demonstrates drag and drop utilising the Drag Manager. Support for Undo and
// Redo of drag and drop within the source window is included.
//
// The bulk of the code in the source code file Drag.c is identical to the code in the file
// Text1.c in the demonstration program MonoTextEdit (Chapter 21).
//
// The program utilises the following resources:
//
// • A 'plst' resource.
//
// • An 'MBAR' resource, and 'MENU' resources for Apple, File, and Edit menus.
//
// • A 'WIND' resource (purgeable) (initially visible).
//
// • 'CNTL' resources (purgeable) for the vertical scroll bars in the text editor window and
// Help dialog, and for the pop-up menu in the Help Dialog.
//
// • A 'STR#' resource (purgeable) containing error text strings.
//
// • A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
// doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// *****

// ..... includes

#include <Carbon.h>

// ..... defines

#define rMenuBar 128
#define mAppleApplication 128
#define iAbout 1
#define mFile 129
#define iNew 1
#define iOpen 2
#define iClose 4
#define iSaveAs 6
#define iQuit 12
#define mEdit 130
#define iUndo 1
#define iCut 3
#define iCopy 4
#define iPaste 5
#define iClear 6
#define iSelectAll 7
#define rWindow 128
#define rVScrollbar 128
#define rErrorStrings 128
#define eMenuBar 1
#define eWindow 2
#define eDocStructure 3
#define eTextEdit 4
#define eExceedChara 5
#define eNoSpaceCut 6
#define eNoSpacePaste 7
#define eDragHandler 8
#define eDrag 9
#define eDragUndo 10
#define kMaxTELength 32767
#define kTab 0x09
#define kBackSpace 0x08
```

```

#define kForwardDelete    0x7F
#define kReturn           0x0D
#define kEscape           0x1B
#define kReturn           0x0D
#define kFileCreator      'KJbb'
#define topLeft(r)        (((Point *) &(r))[0])
#define botRight(r)       (((Point *) &(r))[1])

// ..... typedefs

typedef struct
{
    TEHandle    textEditStrucHdl;
    ControlRef  vScrollbarRef;
    WindowRef   windowRef;
    Boolean     windowTouched;
    Handle      preDragText;
    SInt16      preDragSelStart;
    SInt16      preDragSelEnd;
    SInt16      postDropSelStart;
    SInt16      postDropSelEnd;
} docStructure, *docStructurePointer;

// ..... function prototypes

void          main                (void);
void          doPreliminaries     (void);
OSStatus      appEventHandler     (EventHandlerCallRef,EventRef,void *);
OSStatus      windowEventHandler  (EventHandlerCallRef,EventRef,void *);
void          doIdle              (void);
void          doKeyEvent          (SInt8);
void          scrollActionFunction (ControlRef,SInt16);
void          doInContent         (Point,EventRecord *,Boolean);
void          doDrawContent       (WindowRef);
void          doActivateDeactivate (WindowRef,Boolean);
void          doOSEvent           (EventRecord *);
WindowRef     doNewDocWindow      (void);
EventHandlerUPP doGetHandlerUPP   (void);
Boolean       customClickLoop     (void);
void          doSetScrollBarValue (ControlRef,SInt16 *);
void          doAdjustMenus       (void);
void          doMenuChoice        (MenuID,MenuItemIndex);
void          doFileMenu          (MenuItemIndex);
void          doEditMenu          (MenuItemIndex);
SInt16        doGetSelectLength   (TEHandle);
void          doAdjustScrollbar   (WindowRef);
void          doAdjustCursor      (WindowRef);
void          doCloseWindow       (WindowRef);
void          doSaveAsFile        (TEHandle);
void          doOpenCommand       (void);
void          doOpenFile          (FSSpec);
void          doErrorAlert        (SInt16);
void          navEventFunction    (NavEventCallbackMessage,NavCRecPtr,
                                  NavCallBackUserData);

OSErr         doStartDrag         (EventRecord *,RgnHandle,TEHandle);
OSErr         dragTrackingHandler  (DragTrackingMessage,WindowRef,void *,DragRef);
SInt16        doGetOffset         (Point,TEHandle);
SInt16        doIsOffsetAtLineStart (SInt16,TEHandle);
void          doDrawCaret         (SInt16,TEHandle);
SInt16        doGetLine           (SInt16,TEHandle);

OSErr         dragReceiveHandler   (WindowRef,void *,DragRef);
Boolean       doIsWhiteSpaceAtOffset (SInt16,TEHandle);
Boolean       doIsWhiteSpace      (char);
char          doGetCharAtOffset   (SInt16,TEHandle);
SInt16        doInsertTextAtOffset (SInt16,Ptr,SInt32,TEHandle);
Boolean       doSavePreInsertionText (docStructurePointer);
void          doUndoRedoDrag      (WindowRef);

```

```

// *****
// Drag.c
// *****

// ..... includes

#include "Drag.h"

// ..... global variables

ControlActionUPP      gScrollActionFunctionUPP;
TEClickLoopUPP       gCustomClickLoopUPP;
DragTrackingHandlerUPP gDragTrackingHandlerUPP;
DragReceiveHandlerUPP gDragReceiveHandlerUPP;
Boolean               gRunningOnX           = false;
SInt16                gNumberOfWindows     = 0;
SInt16                gOldControlValue;
Boolean               gEnableDragUndoRedoItem = false;
Boolean               gUndoFlag;

// ***** main

void main(void)
{
    MenuBarHandle menubarHdl;
    SInt32         response;
    MenuRef        menuRef;
    EventTypeSpec applicationEvents[] = { { kEventClassApplication, kEventAppActivated  },
                                          { kEventClassApplication, kEventAppDeactivated },
                                          { kEventClassCommand,      kEventProcessCommand  },
                                          { kEventClassMenu,        kEventMenuEnableItems },
                                          { kEventClassMouse,       kEventMouseMove      } };

    // ..... do preliminaries

    doPreliminaries();

    // ..... create universal procedure pointers

    gScrollActionFunctionUPP = NewControlActionUPP((ControlActionProcPtr) scrollActionFunction);
    gCustomClickLoopUPP      = NewTEClickLoopUPP((TEClickLoopProcPtr) customClickLoop);

    gDragTrackingHandlerUPP = NewDragTrackingHandlerUPP((DragTrackingHandlerProcPtr)
                                                       dragTrackingHandler);
    gDragReceiveHandlerUPP  = NewDragReceiveHandlerUPP((DragReceiveHandlerProcPtr)
                                                       dragReceiveHandler);

    // ..... set up menu bar and menus

    menubarHdl = GetNewMBar(rMenuBar);
    if(menubarHdl == NULL)
        doErrorAlert(eMenuBar);
    SetMenuBar(menubarHdl);
    DrawMenuBar();

    Gestalt(gestaltMenuMgrAttr,&response);
    if(response & gestaltMenuMgrAquaLayoutMask)
    {
        menuRef = GetMenuRef(mFile);
        if(menuRef != NULL)
        {
            DeleteMenuItem(menuRef,iQuit);
            DeleteMenuItem(menuRef,iQuit - 1);
        }

        gRunningOnX = true;
    }
    else

```

```

{
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
        SetMenuItemCommandID(menuRef,iQuit,kHICommandQuit);
}

// ..... install application event handler

InstallApplicationEventHandler(NewEventHandlerUPP((EventHandlerProcPtr) appEventHandler),
                             GetEventTypeCount(applicationEvents),applicationEvents,
                             0,NULL);

// ..... install a timer

InstallEventLoopTimer(GetCurrentEventLoop(),0,TicksToEventTime(GetCaretTime()),
                      NewEventLoopTimerUPP((EventLoopTimerProcPtr) doIdle),NULL,
                      NULL);

// ..... open window

doNewDocWindow();

// ..... run application event loop

RunApplicationEventLoop();
}

// ***** doPreliminaries

void doPreliminaries(void)
{
    MoreMasterPointers(192);
    InitCursor();
}

// ***** appEventHandler

OSStatus appEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                        void * userData)
{
    OSStatus    result = eventNotHandledErr;
    UInt32      eventClass;
    UInt32      eventKind;
    HICommand   hiCommand;
    MenuID      menuID;
    MenuItemIndex menuItem;
    WindowClass windowClass;

    eventClass = GetEventClass(eventRef);
    eventKind  = GetEventKind(eventRef);

    switch(eventClass)
    {
    case kEventClassApplication:
        if(eventKind == kEventAppActivated)
            SetThemeCursor(kThemeArrowCursor);
        break;

    case kEventClassCommand:
        if(eventKind == kEventProcessCommand)
        {
            GetEventParameter(eventRef,kEventParamDirectObject,typeHICommand,NULL,
                             sizeof(HICommand),NULL,&hiCommand);
            menuID = GetMenuID(hiCommand.menu.menuRef);
            menuItem = hiCommand.menu.menuItemIndex;
            if((hiCommand.commandID != kHICommandQuit) &&
                (menuID >= mAppleApplication && menuID <= mEdit))
            {
                doMenuChoice(menuID,menuItem);
                result = noErr;
            }
        }
    }
}

```

```

    }
}
break;

case kEventClassMenu:
    if(eventKind == kEventMenuEnableItems)
    {
        GetWindowClass(FrontWindow(),&windowClass);
        if(windowClass == kDocumentWindowClass)
            doAdjustMenus();
        result = noErr;
    }
    break;

case kEventClassMouse:
    if(eventKind == kEventMouseMove)
    {
        GetWindowClass(FrontWindow(),&windowClass);
        if(windowClass == kDocumentWindowClass)
            doAdjustCursor(FrontWindow());
        result = noErr;
    }
    break;
}

return result;
}

// ***** windowEventHandler

OSStatus windowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                            void* userData)
{
    OSStatus      result = eventNotHandledErr;
    UInt32        eventClass;
    UInt32        eventKind;
    WindowRef     windowRef;
    UInt32        modifiers;
    Point         mouseLocation;
    Boolean        shiftKeyDown = false;
    EventRecord    eventRecord;
    ControlRef     controlRef;
    ControlPartCode controlPartCode;
    SInt8         charCode;

    eventClass = GetEventClass(eventRef);
    eventKind = GetEventKind(eventRef);

    switch(eventClass)
    {
    case kEventClassWindow: // event class window
        GetEventParameter(eventRef,kEventParamDirectObject,typeWindowRef,NULL,sizeof(windowRef),
                          NULL,&windowRef);
        switch(eventKind)
        {
        case kEventWindowDrawContent:
            doDrawContent(windowRef);
            result = noErr;
            break;

        case kEventWindowActivated:
            doActivateDeactivate(windowRef,true);
            result = noErr;
            break;

        case kEventWindowDeactivated:
            doActivateDeactivate(windowRef,false);
            result = noErr;
            break;
        }
    }
}

```

```

    case kEventWindowClickContentRgn:
        SetPortWindowPort(FrontWindow());
        GetMouse(&mouseLocation);
        GetEventParameter(eventRef, kEventParamKeyModifiers, typeUInt32, NULL,
            sizeof(modifiers), NULL, &modifiers);
        if(modifiers & shiftKey)
            shiftKeyDown = true;
        ConvertEventRefToEventRecord(eventRef, &eventRecord);
        doInContent(mouseLocation, &eventRecord, shiftKeyDown);
        result = noErr;
        break;

    case kEventWindowClose:
        doCloseWindow(windowRef);
        result = noErr;
        break;
}
break;

case kEventClassMouse: // event class mouse
    switch(eventKind)
    {
    case kEventMouseDown:
        GetEventParameter(eventRef, kEventParamMouseLocation, typeQDPoint, NULL,
            sizeof(mouseLocation), NULL, &mouseLocation);
        SetPortWindowPort(FrontWindow());
        GlobalToLocal(&mouseLocation);
        controlRef = FindControlUnderMouse(mouseLocation, FrontWindow(), &controlPartCode);
        if(controlRef)
        {
            gOldControlValue = GetControlValue(controlRef);
            TrackControl(controlRef, mouseLocation, gScrollActionFunctionUPP);
            result = noErr;
        }
        break;
    }
    break;

case kEventClassKeyboard: // event class keyboard
    switch(eventKind)
    {
    case kEventRawKeyDown:
    case kEventRawKeyRepeat:
        GetEventParameter(eventRef, kEventParamKeyMacCharCodes, typeChar, NULL,
            sizeof(charCode), NULL, &charCode);
        GetEventParameter(eventRef, kEventParamKeyModifiers, typeUInt32, NULL,
            sizeof(modifiers), NULL, &modifiers);
        if((modifiers & cmdKey) == 0)
            doKeyEvent(charCode);
        result = noErr;
        break;
    }
    break;
}

return result;
}

// ***** doIdle

void doIdle(void)
{
    WindowRef          windowRef;
    docStructurePointer docStrucPtr;

    windowRef = FrontWindow();
    if(GetWindowKind(windowRef) == kApplicationWindowKind)
    {

```

```

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    if(docStrucPtr != NULL)
        TEIdle(docStrucPtr->textEditStrucHdl);
}
}

// ***** doKeyEvent

void doKeyEvent(SInt8 charCode)
{
    WindowRef          windowRef;
    docStructurePointer docStrucPtr;
    TEHandle           textEditStrucHdl;
    SInt16              selectionLength;

    if(charCode <= kEscape && charCode != kBackSpace && charCode != kReturn)
        return;

    windowRef = FrontWindow();
    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    gEnableDragUndoRedoItem = false;

    if(charCode == kTab)
    {
        // Do tab key handling here if required.
    }
    else if(charCode == kForwardDelete)
    {
        selectionLength = doGetSelectLength(textEditStrucHdl);
        if(selectionLength == 0)
            (*textEditStrucHdl)->selEnd += 1;
        TDelete(textEditStrucHdl);
        doAdjustScrollbar(windowRef);
    }
    else
    {
        selectionLength = doGetSelectLength(textEditStrucHdl);
        if(((textEditStrucHdl)->teLength - selectionLength + 1) < kMaxTELength)
        {
            TEKey(charCode, textEditStrucHdl);
            doAdjustScrollbar(windowRef);
        }
        else
            doErrorAlert(eExceedChara);
    }
}

// ***** scrollActionFunction

void scrollActionFunction(ControlRef controlRef, SInt16 partCode)
{
    WindowRef          windowRef;
    docStructurePointer docStrucPtr;
    TEHandle           textEditStrucHdl;
    SInt16              linesToScroll;
    SInt16              controlValue, controlMax;

    windowRef = GetControlOwner(controlRef);
    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    controlValue = GetControlValue(controlRef);
    controlMax = GetControlMaximum(controlRef);

    if(partCode)
    {
        if(partCode != kControlIndicatorPart)

```



```

{
    switch(partCode)
    {
        case kControlUpButtonPart:
        case kControlDownButtonPart:
            linesToScroll = 1;
            break;

        case kControlPageUpPart:
        case kControlPageDownPart:
            linesToScroll = (((*textEditStrucHdl)->viewRect.bottom -
                (*textEditStrucHdl)->viewRect.top) /
                (*textEditStrucHdl)->lineHeight) - 1;
            break;
    }

    if((partCode == kControlDownButtonPart) || (partCode == kControlPageDownPart))
        linesToScroll = -linesToScroll;

    linesToScroll = controlValue - linesToScroll;
    if(linesToScroll < 0)
        linesToScroll = 0;
    else if(linesToScroll > controlMax)
        linesToScroll = controlMax;

    SetControlValue(controlRef,linesToScroll);

    linesToScroll = controlValue - linesToScroll;
}
else
{
    linesToScroll = gOldControlValue - controlValue;
    gOldControlValue = controlValue;
}

if(linesToScroll != 0)
    TEScroll(0,linesToScroll * (*textEditStrucHdl)->lineHeight,textEditStrucHdl);
}
}

// ***** doInContent

void doInContent(Point mouseLocation,EventRecord * eventStrucPtr,Boolean shiftKeyDown)
{
    WindowRef          windowRef;
    docStructurePointer docStrucPtr;
    TEHandle           textEditStrucHdl;
    RgnHandle          hiliteRgn;
    OSErr              osError;

    windowRef = FrontWindow();
    docStrucPtr = (docStructurePointer) GetWRefCon(windowRef);
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    if(PtInRect(mouseLocation,&(*textEditStrucHdl)->viewRect))
    {
        hiliteRgn = NewRgn();

        TEGetHiliteRgn(hiliteRgn,textEditStrucHdl);

        if(!EmptyRgn(hiliteRgn) && PtInRgn(mouseLocation,hiliteRgn))
        {
            if(WaitMouseMoved(mouseLocation))
            {
                osError = doStartDrag(eventStrucPtr,hiliteRgn,textEditStrucHdl);
                if(osError != noErr)
                    doErrorAlert(eDrag);
            }
        }
    }
}

```

```

else
{
    TEClick(mouseLocation,shiftKeyDown,textEditStrucHdl);
    gEnableDragUndoRedoItem = false;
}

DisposeRgn(hiliteRgn);
}
}

// ***** doDrawContent

void doDrawContent(WindowRef windowRef)
{
    docStructurePointer docStrucPtr;
    TEHandle            textEditStrucHdl;
    GrafPtr            oldPort;
    RgnHandle          visibleRegionHdl = NewRgn();
    Rect               portRect;

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    GetPort(&oldPort);
    SetPortWindowPort(windowRef);

    GetPortVisibleRegion(GetWindowPort(windowRef),visibleRegionHdl);
    EraseRgn(visibleRegionHdl);

    UpdateControls(windowRef,visibleRegionHdl);

    GetWindowPortBounds(windowRef,&portRect);
    TEUpdate(&(*textEditStrucHdl)->viewRect,textEditStrucHdl);

    DisposeRgn(visibleRegionHdl);
    SetPort(oldPort);
}

// ***** doActivateDocWindow

void doActivateDeactivate(WindowRef windowRef,Boolean becomingActive)
{
    docStructurePointer docStrucPtr;
    TEHandle            textEditStrucHdl;

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    if(becomingActive)
    {
        SetPortWindowPort(windowRef);

        (*textEditStrucHdl)->viewRect.bottom = (((*textEditStrucHdl)->viewRect.bottom -
                                                (*textEditStrucHdl)->viewRect.top) /
                                                (*textEditStrucHdl)->lineHeight) *
                                                (*textEditStrucHdl)->lineHeight) +
                                                (*textEditStrucHdl)->viewRect.top;
        (*textEditStrucHdl)->destRect.bottom = (*textEditStrucHdl)->viewRect.bottom;

        TActivate(textEditStrucHdl);
        ActivateControl(docStrucPtr->vScrollbarRef);
        doAdjustScrollbar(windowRef);
        doAdjustCursor(windowRef);
    }
    else
    {
        TDeactivate(textEditStrucHdl);
        DeactivateControl(docStrucPtr->vScrollbarRef);
    }
}

```

```

}

// ***** doNewDocWindow

WindowRef doNewDocWindow(void)
{
    WindowRef      windowRef;
    OSStatus       osError;
    Rect           contentRect = { 100,100,400,595 };
    WindowAttributes attributes = kWindowStandardHandlerAttribute |
                                kWindowStandardDocumentAttributes;

    docStructurePointer docStrucPtr;
    Rect               portRect, destAndViewRect;
    EventTypeSpec      windowEvents[] = { { kEventClassWindow, kEventWindowDrawContent },
                                           { kEventClassWindow, kEventWindowActivated },
                                           { kEventClassWindow, kEventWindowDeactivated },
                                           { kEventClassWindow, kEventWindowClickContentRgn },
                                           { kEventClassWindow, kEventWindowClose },
                                           { kEventClassMouse, kEventMouseDown },
                                           { kEventClassKeyboard, kEventRawKeyDown },
                                           { kEventClassKeyboard, kEventRawKeyRepeat } };

    osError = CreateNewWindow(kDocumentWindowClass,attributes,&contentRect,&windowRef);
    if(osError != noErr)
    {
        doErrorAlert(eWindow);
        return NULL;
    }

    ChangeWindowAttributes(windowRef,0,kWindowResizableAttribute);
    RepositionWindow(windowRef,NULL,kWindowCascadeOnMainScreen);
    SetWTitle(windowRef,"\puntitled");
    SetPortWindowPort(windowRef);
    TextSize(10);
    if(!(docStrucPtr = (docStructurePointer) NewPtr(sizeof(docStructure))))
    {
        doErrorAlert(eDocStructure);
        return NULL;
    }

    SetWRefCon(windowRef,(SInt32) docStrucPtr);
    SetWindowProxyCreatorAndType(windowRef,0,'TEXT',kUserDomain);

    InstallWindowEventHandler(windowRef,doGetHandlerUPP(),GetEventTypeCount(windowEvents),
                             windowEvents,0,NULL);

    gNumberOfWindows ++;

    docStrucPtr->windowRef      = windowRef;
    docStrucPtr->windowTouched = false;
    docStrucPtr->preDragText   = NULL;
    docStrucPtr->vScrollbarRef = GetNewControl(rVScrollbar,windowRef);

    GetWindowPortBounds(windowRef,&portRect);
    destAndViewRect = portRect;
    destAndViewRect.right -= 15;
    InsetRect(&destAndViewRect,2,2);

    if(!(docStrucPtr->textEditStrucHdl = TNew(&destAndViewRect,&destAndViewRect)))
    {
        DisposeWindow(windowRef);
        gNumberOfWindows --;
        DisposePtr((Ptr) docStrucPtr);
        doErrorAlert(eTextEdit);
        return NULL;
    }

    TSetClickLoop(gCustomClickLoopUPP,docStrucPtr->textEditStrucHdl);
    TAutoView(true,docStrucPtr->textEditStrucHdl);
}

```

```

TEFeatureFlag(teFOutlineHilite,teBitSet,docStrucPtr->textEditStrucHdl);

if(osError = InstallTrackingHandler(gDragTrackingHandlerUPP,windowRef,docStrucPtr))
{
    DisposeWindow(windowRef);
    gNumberOfWindows --;
    DisposePtr((Ptr) docStrucPtr);
    doErrorAlert(eDragHandler);
    return NULL;
}

if(osError = InstallReceiveHandler(gDragReceiveHandlerUPP,windowRef,docStrucPtr))
{
    RemoveTrackingHandler(gDragTrackingHandlerUPP,windowRef);
    DisposeWindow(windowRef);
    gNumberOfWindows --;
    DisposePtr((Ptr) docStrucPtr);
    doErrorAlert(eDragHandler);
    return NULL;
}

ShowWindow(windowRef);

return windowRef;
}

// ***** doGetHandlerUPP

EventHandlerUPP doGetHandlerUPP(void)
{
    static EventHandlerUPP windowEventHandlerUPP;

    if(windowEventHandlerUPP == NULL)
        windowEventHandlerUPP = NewEventHandlerUPP((EventHandlerProcPtr) windowEventHandler);

    return windowEventHandlerUPP;
}

// ***** customClickLoop

Boolean customClickLoop(void)
{
    WindowRef          windowRef;
    docStructurePointer docStrucPtr;
    TEHandle           textEditStrucHdl;
    GrafPtr            oldPort;
    RgnHandle          oldClip;
    Rect               tempRect, portRect;
    Point              mouseXY;
    SInt16             linesToScroll = 0;

    windowRef = FrontWindow();
    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    GetPort(&oldPort);
    SetPortWindowPort(windowRef);
    oldClip = NewRgn();
    GetClip(oldClip);
    SetRect(&tempRect,-32767,-32767,32767,32767);
    ClipRect(&tempRect);

    GetMouse(&mouseXY);
    GetWindowPortBounds(windowRef,&portRect);

    if(mouseXY.v < portRect.top)
    {
        linesToScroll = 1;
    }
}

```

```

doSetScrollBarValue(docStrucPtr->vScrollbarRef,&linesToScroll);
if(linesToScroll != 0)
    TEScroll(0,linesToScroll * ((*textEditStrucHdl)->lineHeight),textEditStrucHdl);
}
else if(mouseXY.v > portRect.bottom)
{
    linesToScroll = -1;
doSetScrollBarValue(docStrucPtr->vScrollbarRef,&linesToScroll);
if(linesToScroll != 0)
    TEScroll(0,linesToScroll * ((*textEditStrucHdl)->lineHeight),textEditStrucHdl);
}

SetClip(oldClip);
DisposeRgn(oldClip);
SetPort(oldPort);

return true;
}

// ***** doSetScrollBarValue

void doSetScrollBarValue(ControlRef controlRef,SInt16 *linesToScroll)
{
    SInt16 controlValue, controlMax;

    controlValue = GetControlValue(controlRef);
    controlMax = GetControlMaximum(controlRef);

    *linesToScroll = controlValue - *linesToScroll;
    if(*linesToScroll < 0)
        *linesToScroll = 0;
    else if(*linesToScroll > controlMax)
        *linesToScroll = controlMax;

    SetControlValue(controlRef,*linesToScroll);
    *linesToScroll = controlValue - *linesToScroll;
}

// ***** doAdjustMenus

void doAdjustMenus(void)
{
    MenuRef          fileMenuRef, editMenuRef;
    WindowRef        windowRef;
    docStructurePointer docStrucPtr;
    TEHandle          textEditStrucHdl;
    ScrapRef          scrapRef;
    OSStatus          osError;
    ScrapFlavorFlags scrapFlavorFlags;

    fileMenuRef = GetMenuRef(mFile);
    editMenuRef = GetMenuRef(mEdit);

    if(gNumberOfWindows > 0)
    {
        windowRef = FrontWindow();
        docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
        textEditStrucHdl = docStrucPtr->textEditStrucHdl;

        EnableMenuItem(fileMenuRef,iClose);

        if(gEnableDragUndoRedoItem)
        {
            EnableMenuItem(editMenuRef,iUndo);
            if(gUndoFlag)
                SetMenuItemText(editMenuRef,iUndo,"\pUndo Drag & Drop");
            else
                SetMenuItemText(editMenuRef,iUndo,"\pRedo Drag & Drop");
        }
    }
}

```

```

else
{
    DisableMenuItem(editMenuRef,iUndo);
    SetMenuItemText(editMenuRef,iUndo,"\pRedo Drag & Drop");
}

if((*textEditStrucHdl)->selStart < (*textEditStrucHdl)->selEnd)
{
    EnableMenuItem(editMenuRef,iCut);
    EnableMenuItem(editMenuRef,iCopy);
    EnableMenuItem(editMenuRef,iClear);
}
else
{
    DisableMenuItem(editMenuRef,iCut);
    DisableMenuItem(editMenuRef,iCopy);
    DisableMenuItem(editMenuRef,iClear);
}

GetCurrentScrap(&scrapRef);

osError = GetScrapFlavorFlags(scrapRef,kScrapFlavorTypeText,&scrapFlavorFlags);
if(osError == noErr)
    EnableMenuItem(editMenuRef,iPaste);
else
    DisableMenuItem(editMenuRef,iPaste);

if((*textEditStrucHdl)->teLength > 0)
{
    EnableMenuItem(fileMenuRef,iSaveAs);
    EnableMenuItem(editMenuRef,iSelectAll);
}
else
{
    DisableMenuItem(fileMenuRef,iSaveAs);
    DisableMenuItem(editMenuRef,iSelectAll);
}
}
else
{
    DisableMenuItem(fileMenuRef,iClose);
    DisableMenuItem(fileMenuRef,iSaveAs);
    DisableMenuItem(editMenuRef,iClear);
    DisableMenuItem(editMenuRef,iSelectAll);
}

DrawMenuBar();
}

// ***** doMenuChoice

void doMenuChoice(MenuID menuID,MenuItemIndex menuItem)
{
    if(menuID == 0)
        return;

    switch(menuID)
    {
        case mAppleApplication:
            if(menuItem == iAbout)
                SysBeep(10);
            break;

        case mFile:
            doFileMenu(menuItem);
            break;

        case mEdit:
            doEditMenu(menuItem);
    }
}

```

```

        break;
    }
}

// ***** doFileMenu

void doFileMenu(MenuItemIndex menuItem)
{
    docStructurePointer docStrucPtr;
    TEHandle             textEditStrucHdl;

    switch(menuItem)
    {
        case iNew:
            doNewDocWindow();
            break;

        case iOpen:
            doOpenCommand();
            doAdjustScrollbar(FrontWindow());
            break;

        case iClose:
            doCloseWindow(FrontWindow());
            break;

        case iSaveAs:
            docStrucPtr = (docStructurePointer) (GetWRefCon(FrontWindow()));
            textEditStrucHdl = docStrucPtr->textEditStrucHdl;
            doSaveAsFile(textEditStrucHdl);
            break;
    }
}

// ***** doEditMenu

void doEditMenu(MenuItemIndex menuItem)
{
    WindowRef          windowRef;
    docStructurePointer docStrucPtr;
    TEHandle           textEditStrucHdl;
    SInt32             totalSize, contigSize, newSize;
    SInt16             selectionLength;
    ScrapRef           scrapRef;
    Size               sizeOfTextData;

    windowRef = FrontWindow();
    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    switch(menuItem)
    {
        case iUndo:
            doUndoRedoDrag(windowRef);
            break;

        case iCut:
            if(ClearCurrentScrap() == noErr)
            {
                PurgeSpace(&totalSize,&contigSize);
                selectionLength = doGetSelectLength(textEditStrucHdl);
                if(selectionLength > contigSize)
                    doErrorAlert(eNoSpaceCut);
            }
            else
            {
                TECut(textEditStrucHdl);
                doAdjustScrollbar(windowRef);
                if(TEToScrap() != noErr)
                    ClearCurrentScrap();
            }
    }
}

```

```

    }
}
break;

case iCopy:
    if(ClearCurrentScrap() == noErr)
    {
        TECopy(textEditStrucHdl);
        if(TEToScrap() != noErr)
            ClearCurrentScrap();
    }
    break;

case iPaste:
    GetCurrentScrap(&scrapRef);;
    GetScrapFlavorSize(scrapRef, kScrapFlavorTypeText, &sizeofTextData);
    newSize = (*textEditStrucHdl)->teLength + sizeofTextData;
    if(newSize > kMaxTELength)
        doErrorAlert(eNoSpacePaste);
    else
    {
        if(TEFromScrap() == noErr)
        {
            TEPaste(textEditStrucHdl);
            doAdjustScrollbar(windowRef);
        }
    }
    break;

case iClear:
    TDelete(textEditStrucHdl);
    doAdjustScrollbar(windowRef);
    break;

case iSelectAll:
    Teselect(0, (*textEditStrucHdl)->teLength, textEditStrucHdl);
    break;
}
}

// ***** doGetSelectLength

SInt16 doGetSelectLength(TEHandle textEditStrucHdl)
{
    SInt16 selectionLength;

    selectionLength = (*textEditStrucHdl)->selEnd - (*textEditStrucHdl)->selStart;
    return selectionLength;
}

// ***** doAdjustScrollbar

void doAdjustScrollbar(WindowRef windowRef)
{
    docStructurePointer docStrucPtr;
    TEHandle textEditStrucHdl;
    SInt16 numberOfLines, controlMax, controlValue;

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));;
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    numberOfLines = (*textEditStrucHdl)->nLines;
    if((*textEditStrucHdl)->hText + (*textEditStrucHdl)->teLength - 1) == kReturn)
        numberOfLines += 1;

    controlMax = numberOfLines - (((*textEditStrucHdl)->viewRect.bottom -
        (*textEditStrucHdl)->viewRect.top) /
        (*textEditStrucHdl)->lineHeight);
    if(controlMax < 0)

```



```

    controlMax = 0;
    SetControlMaximum(docStrucPtr->vScrollbarRef, controlMax);

    controlValue = ((*textEditStrucHdl)->viewRect.top - (*textEditStrucHdl)->destRect.top) /
        (*textEditStrucHdl)->lineHeight;
    if(controlValue < 0)
        controlValue = 0;
    else if(controlValue > controlMax)
        controlValue = controlMax;

    SetControlValue(docStrucPtr->vScrollbarRef, controlValue);

    SetControlViewSize(docStrucPtr->vScrollbarRef, (*textEditStrucHdl)->viewRect.bottom -
        (*textEditStrucHdl)->viewRect.top);

    TEScroll(0, ((*textEditStrucHdl)->viewRect.top - (*textEditStrucHdl)->destRect.top) -
        (GetControlValue(docStrucPtr->vScrollbarRef) *
        (*textEditStrucHdl)->lineHeight), textEditStrucHdl);
}

// ***** doAdjustCursor

void doAdjustCursor(WindowRef windowRef)
{
    GrafPtr      oldPort;
    RgnHandle     arrowRegion, iBeamRegion, hiliteRgn;
    Rect         portRect, cursorRect;
    docStructurePointer docStrucPtr;
    Point        offset, mouseXY;

    GetPort(&oldPort);
    SetPortWindowPort(windowRef);

    arrowRegion = NewRgn();
    iBeamRegion = NewRgn();
    hiliteRgn   = NewRgn();
    SetRectRgn(arrowRegion, -32768, -32768, 32766, 32766);

    GetWindowPortBounds(windowRef, &portRect);
    cursorRect = portRect;
    cursorRect.right -= 15;
    LocalToGlobal(&topLeft(cursorRect));
    LocalToGlobal(&botRight(cursorRect));

    RectRgn(iBeamRegion, &cursorRect);
    DiffRgn(arrowRegion, iBeamRegion, arrowRegion);

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    TEGetHiliteRgn(hiliteRgn, docStrucPtr->textEditStrucHdl);
    LocalToGlobal(&topLeft(portRect));
    offset = topLeft(portRect);
    OffsetRgn(hiliteRgn, offset.h, offset.v);
    DiffRgn(iBeamRegion, hiliteRgn, iBeamRegion);

    GetGlobalMouse(&mouseXY);

    if(PtInRgn(mouseXY, iBeamRegion))
        SetThemeCursor(kThemeIBeamCursor);
    else if(PtInRgn(mouseXY, hiliteRgn))
        SetThemeCursor(kThemeArrowCursor);
    else
        SetThemeCursor(kThemeArrowCursor);

    DisposeRgn(arrowRegion);
    DisposeRgn(iBeamRegion);
    DisposeRgn(hiliteRgn);

    SetPort(oldPort);
}

```

```

// ***** doCloseWindow

void doCloseWindow(WindowRef windowRef)
{
    docStructurePointer docStrucPtr;

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));

    DisposeControl(docStrucPtr->vScrollbarRef);
    TEDispose(docStrucPtr->textEditStrucHdl);
    DisposePtr((Ptr) docStrucPtr);

    if(docStrucPtr->preDragText == NULL)
        DisposeHandle(docStrucPtr->preDragText);

    RemoveTrackingHandler(gDragTrackingHandlerUPP,windowRef);
    RemoveReceiveHandler(gDragReceiveHandlerUPP,windowRef);

    DisposeWindow(windowRef);

    gNumberOfWindows --;
}

// ***** doSaveAsFile

void doSaveAsFile(TEHandle textEditStrucHdl)
{
    OSErr          osError = noErr;
    NavDialogOptions dialogOptions;
    NavEventUPP    navEventFunctionUPP;
    WindowRef      windowRef;
    OSType         fileType;
    NavReplyRecord navReplyStruc;
    AEKeyword      theKeyword;
    DescType       actualType;
    FSSpec         fileSpec;
    SInt16         fileRefNum;
    Size           actualSize;
    SInt32         dataLength;
    Handle         editTextHdl;

    osError = NavGetDefaultDialogOptions(&dialogOptions);

    if(osError == noErr)
    {
        windowRef = FrontWindow();

        fileType = 'TEXT';

        navEventFunctionUPP = NewNavEventUPP((NavEventProcPtr) navEventFunction);
        osError = NavPutFile(NULL,&navReplyStruc,&dialogOptions,navEventFunctionUPP,fileType,
                            kFileCreator,NULL);
        DisposeNavEventUPP(navEventFunctionUPP);

        if(navReplyStruc.validRecord && osError == noErr)
        {
            if((osError = AEGetNthPtr(&(navReplyStruc.selection),1,typeFSS,&theKeyword,
                                     &actualType,&fileSpec,sizeof(fileSpec),&actualSize)) == noErr)
            {
                if(!navReplyStruc.replacing)
                {
                    osError = FSpCreate(&fileSpec,kFileCreator,fileType,navReplyStruc.keyScript);
                    if(osError != noErr)
                    {
                        NavDisposeReply(&navReplyStruc);
                    }
                }
            }
        }
    }
}

```

```

        if(osError == noErr)
            osError = FSpOpenDF(&fileSpec,fsRdWrPerm,&fileRefNum);

        if(osError == noErr)
        {
            SetWTitle(windowRef,fileSpec.name);
            dataLength = (*textEditStrucHdl)->teLength;
            editTextHdl = (*textEditStrucHdl)->hText;
            FSpWrite(fileRefNum,&dataLength,*editTextHdl);
        }

        NavCompleteSave(&navReplyStruc,kNavTranslateInPlace);
    }

    NavDisposeReply(&navReplyStruc);
}
}
}

// ***** doOpenCommand

void doOpenCommand(void)
{
    OSErr          osError = noErr;
    NavDialogOptions dialogOptions;
    NavEventUPP    navEventFunctionUPP;
    NavReplyRecord navReplyStruc;
    SInt32         index, count;
    AEKeyword      theKeyword;
    DescType       actualType;
    FSSpec         fileSpec;
    Size           actualSize;
    FInfo          fileInfo;

    osError = NavGetDefaultDialogOptions(&dialogOptions);

    if(osError == noErr)
    {
        navEventFunctionUPP = NewNavEventUPP((NavEventProcPtr) navEventFunction);
        osError = NavGetFile(NULL,&navReplyStruc,&dialogOptions,navEventFunctionUPP,NULL,NULL,
            NULL,NULL);
        DisposeNavEventUPP(navEventFunctionUPP);

        if(osError == noErr && navReplyStruc.validRecord)
        {
            if(osError == noErr)
            {
                if(osError == noErr)
                {
                    osError = AECountItems(&(navReplyStruc.selection),&count);

                    for(index=1;index<=count;index++)
                    {
                        osError = AEGetNthPtr(&(navReplyStruc.selection),index,typeFSS,&theKeyword,
                            &actualType,&fileSpec,sizeof(fileSpec),&actualSize);

                        {
                            if((osError = FSpGetFInfo(&fileSpec,&fileInfo)) == noErr)
                                doOpenFile(fileSpec);
                        }
                    }
                }
            }
        }

        NavDisposeReply(&navReplyStruc);
    }
}
}
}

```

```

// ***** doOpenFile

void doOpenFile(FSSpec fileSpec)
{
    WindowRef      windowRef;
    docStructurePointer docStrucPtr;
    TEHandle        textEditStrucHdl;
    SInt16          fileRefNum;
    SInt32          textLength;
    Handle          textBuffer;

    if((windowRef = doNewDocWindow()) == NULL)
        return;

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));
    textEditStrucHdl = docStrucPtr->textEditStrucHdl;

    SetWTitle(windowRef, fileSpec.name);

    FSpOpenDF(&fileSpec, fsCurPerm, &fileRefNum);

    SetFPos(fileRefNum, fsFromStart, 0);
    GetEOF(fileRefNum, &textLength);

    if(textLength > 32767)
        textLength = 32767;

    textBuffer = NewHandle((Size) textLength);

    FSRead(fileRefNum, &textLength, *textBuffer);

    MoveHHi(textBuffer);
    HLock(textBuffer);

    TETSetText(*textBuffer, textLength, textEditStrucHdl);

    HUnlock(textBuffer);
    DisposeHandle(textBuffer);

    FSClose(fileRefNum);

    (*textEditStrucHdl)->selStart = 0;
    (*textEditStrucHdl)->selEnd = 0;

    doDrawContent(windowRef);
}

// ***** doErrorAlert

void doErrorAlert(SInt16 errorCode)
{
    Str255 errorString;
    SInt16 itemHit;

    GetIndString(errorString, rErrorStrings, errorCode);

    if(errorCode < eWindow)
    {
        StandardAlert(kAlertStopAlert, errorString, NULL, NULL, &itemHit);
        ExitToShell();
    }
    else
    {
        StandardAlert(kAlertCautionAlert, errorString, NULL, NULL, &itemHit);
    }
}

// ***** navEventFunction

```

```

void navEventFunction(NavEventCallbackMessage callBackSelector,NavCBRecPtr callBackParms,
                    NavCallBackUserData callBackUD)
{
}

// *****
// StartAndTrackDrag.c
// *****

// ..... includes

#include "Drag.h"

// ..... global variables

Boolean gCursorInContent, gCanAcceptItems, gCaretShowFlag;
SInt16  gInsertPosition, gLastOffset, gCaretOffset;
UInt32  gSystemCaretTime, gCaretStartTime;

extern Boolean gRunningOnX;

// ***** doStartDrag
OSErr doStartDrag(EventRecord *eventStrucPtr,RgnHandle hiliteRgnHdl,TEHandle textEditStrucHdl)
{
    OSErr      osError;
    DragReference dragRef;
    Rect       originalHiliteRect, zeroedHiliteRect;
    RgnHandle  maskRgnHdl;
    Point      offsetPoint;
    QDErr      qdError;
    CGrafPtr   savedPortPtr;
    GDHandle    saveDeviceHdl;
    GWorldPtr  dragGWorldPtr = NULL;
    PixMapHandle dragPixMapHdl, windPixMapHdl;
    RgnHandle  dragRgnHdl, tempRgnHdl;

    // ..... create new drag

    if(osError = NewDrag(&dragRef))
        return osError;

    // ..... add 'TEXT' flavour

    osError = AddDragItemFlavor(dragRef,1,'TEXT',
                               (*textEditStrucHdl->hText) + (*textEditStrucHdl->selStart,
                               (*textEditStrucHdl->selEnd - (*textEditStrucHdl->selStart,0);

    // ..... get and set drag image for translucent drag and drop

    if(!gRunningOnX)
    {
        maskRgnHdl = dragRgnHdl = tempRgnHdl = NULL;

        GetRegionBounds(hiliteRgnHdl,&originalHiliteRect);
        zeroedHiliteRect = originalHiliteRect;
        OffsetRect(&zeroedHiliteRect,-originalHiliteRect.left,-originalHiliteRect.top);

        GetGWorld(&savedPortPtr,&saveDeviceHdl);

        qdError = NewGWorld(&dragGWorldPtr,8,&zeroedHiliteRect,NULL,NULL,0);
        if(dragGWorldPtr != NULL && qdError == noErr)
        {
            SetGWorld(dragGWorldPtr,NULL);
            EraseRect(&zeroedHiliteRect);

            dragPixMapHdl = GetGWorldPixMap(dragGWorldPtr);
            LockPixels(dragPixMapHdl);
        }
    }
}

```

```

windPixmapHdl = GetGWorldPixmap(savedPortPtr);

CopyBits((Bitmap *) *windPixmapHdl,(Bitmap *) *dragPixmapHdl,
        &originalHiliteRect,&zeroedHiliteRect,srcCopy,NULL);

UnlockPixels(dragPixmapHdl);
SetGWorld(savedPortPtr,saveDeviceHdl);

maskRgnHdl = NewRgn();
if(maskRgnHdl != NULL)
{
    CopyRgn(hiliteRgnHdl,maskRgnHdl);
    OffsetRgn(maskRgnHdl,-originalHiliteRect.left,-originalHiliteRect.top);

    SetPt(&offsetPoint,originalHiliteRect.left,originalHiliteRect.top);
    LocalToGlobal(&offsetPoint);

    SetDragImage(dragRef,dragPixmapHdl,maskRgnHdl,offsetPoint,kDragStandardTranslucency);
}
}
}

// ..... get drag region

dragRgnHdl = NewRgn();
if(dragRgnHdl == NULL)
    return MemError();

CopyRgn(hiliteRgnHdl,dragRgnHdl);
SetPt(&offsetPoint,0,0);
LocalToGlobal(&offsetPoint);
OffsetRgn(dragRgnHdl,offsetPoint.h,offsetPoint.v);

tempRgnHdl = NewRgn();
if(tempRgnHdl == NULL)
    return MemError();

CopyRgn(dragRgnHdl,tempRgnHdl);
InsetRgn(tempRgnHdl,1,1);
DiffRgn(dragRgnHdl,tempRgnHdl,dragRgnHdl);
DisposeRgn(tempRgnHdl);

// ..... perform the drag

osError = TrackDrag(dragRef,eventStrucPtr,dragRgnHdl);
if(osError != noErr && osError != userCanceledErr)
    return osError;

if(dragRef)    DisposeDrag(dragRef);
if(maskRgnHdl) DisposeRgn(maskRgnHdl);
if(dragGWorldPtr) DisposeGWorld(dragGWorldPtr);
if(dragRgnHdl)  DisposeRgn(dragRgnHdl);
if(tempRgnHdl)  DisposeRgn(tempRgnHdl);

return noErr;
}

// ***** dragTrackingHandler

OSErr dragTrackingHandler(DragTrackingMessage trackingMessage,WindowRef windowRef,
        void *handlerRefCon,DragRef dragRef)
{
    docStructurePointer docStrucPtr;
    DragAttributes      dragAttributes;
    UInt32              theTime;
    UInt16              numberOfDragItems, index;
    ItemReference        itemRef;
    OSErr               result;
    FlavorFlags         flavorFlags;

```

```

Point          mousePt, localMousePt;
RgnHandle     windowHiliteRgn;
Rect          correctedViewRect;
SInt16       theOffset;

if((trackingMessage != kDragTrackingEnterHandler) && !gCanAcceptItems)
    return noErr;

docStrucPtr = (docStructurePointer) handlerRefCon;

GetDragAttributes(dragRef,&dragAttributes);
gSystemCaretTime = GetCaretTime();
theTime = TickCount();

switch(trackingMessage)
{
// ..... enter handler

case kDragTrackingEnterHandler:
    gCanAcceptItems = true;

    CountDragItems(dragRef,&numberOfDragItems);

    for(index=1;index <= numberOfDragItems;index++)
    {
        GetDragItemReferenceNumber(dragRef,index,&itemRef);
        result = GetFlavorFlags(dragRef,itemRef,'TEXT',&flavorFlags);
        if(result != noErr)
        {
            gCanAcceptItems = false;
            break;
        }
    }
    break;

// ..... enter window

case kDragTrackingEnterWindow:
    gCaretStartTime = theTime;
    gCaretOffset = gLastOffset = -1;
    gCaretShowFlag = true;
    gCursorInContent = false;
    break;

// ..... in window

case kDragTrackingInWindow:

    GetDragMouse(dragRef,&mousePt,NULL);
    localMousePt = mousePt;
    GlobalToLocal(&localMousePt);

    if(dragAttributes & kDragHasLeftSenderWindow)
    {
        if(PtInRect(localMousePt,&(**(docStrucPtr->textEditStrucHdl)).viewRect))
        {
            if(!gCursorInContent)
            {
                windowHiliteRgn = NewRgn();
                correctedViewRect = (**(docStrucPtr->textEditStrucHdl)).viewRect;
                InsetRect(&correctedViewRect,-2,-2);
                RectRgn(windowHiliteRgn,&correctedViewRect);
                ShowDragHilite(dragRef,windowHiliteRgn,true);
                DisposeRgn(windowHiliteRgn);
            }
            gCursorInContent = true;
        }
        else
        {

```

```

        if(gCursorInContent)
            HideDragHilite(dragRef);
        gCursorInContent = false;
    }
}

// ... start caret drawing stuff, first get the offset into the text
theOffset = doGetOffset(mousePt,docStrucPtr->textEditStrucHdl);

// ... if in sender window, defeat caret drawing in selection
if(dragAttributes & kDragInsideSenderWindow)
{
    if((theOffset >= (*(docStrucPtr->textEditStrucHdl))->selStart) &&
        (theOffset <= (*(docStrucPtr->textEditStrucHdl))->selEnd))
    {
        theOffset = -1;
    }
}

// ... save the offset to a global for use by dragReceiveHandler
gInsertPosition = theOffset;

// ... if offset has changed, reset the caret flashing timer
if(theOffset != gLastOffset)
{
    gCaretStartTime = theTime;
    gCaretShowFlag = true;
}

gLastOffset = theOffset;

// ... if caret-flashing interval has elapsed, toggle caret "show" flag, reset timer
if(theTime - gCaretStartTime > gSystemCaretTime)
{
    gCaretShowFlag = !gCaretShowFlag;
    gCaretStartTime = theTime;
}

// ... if caret "show" flag is off, set variable to defeat caret drawing
if(!gCaretShowFlag)
    theOffset = -1;

// ... if offset has changed, erase previous caret, draw new caret at current offset
if(theOffset != gCaretOffset)
{
    // ... if first pass this window, don't erase, otherwise erase at old offset
    if(gCaretOffset != -1)
        doDrawCaret(gCaretOffset,docStrucPtr->textEditStrucHdl);

    // ... if "show" flag says show, draw caret at current offset
    if(theOffset != -1)
        doDrawCaret(theOffset,docStrucPtr->textEditStrucHdl);
}

gCaretOffset = theOffset;

break;

// ..... Leave window

```



```

case kDragTrackingLeaveWindow:

    if(gCaretOffset != -1)
    {
        doDrawCaret(gCaretOffset,docStrucPtr->textEditStrucHdl);
        gCaretOffset = -1;
    }

    if(gCursorInContent && dragAttributes & kDragHasLeftSenderWindow)
        HideDragHilite(dragRef);

    break;

// ..... Leave handler

case kDragTrackingLeaveHandler:
    break;
}

return noErr;
}

// ***** doGetOffset

SInt16 doGetOffset(Point mousePt,TEHandle textEditStrucHdl)
{
    WindowRef windowRef;
    SInt16 theOffset;
    Point thePoint;

    theOffset = -1;

    if(FindWindow(mousePt,&windowRef) == inContent)
    {
        SetPortWindowPort(windowRef);
        GlobalToLocal(&mousePt);

        if(PtInRect(mousePt,&((*textEditStrucHdl)->viewRect)))
        {
            theOffset = TEGetOffset(mousePt,textEditStrucHdl);
            thePoint = TEGetPoint(theOffset - 1,textEditStrucHdl);

            if((theOffset) &&
                (doIsOffsetAtLineStart(theOffset,textEditStrucHdl)) &&
                ((*textEditStrucHdl)->hText)[theOffset - 1] != 0x0D) &&
                (thePoint.h < mousePt.h))
            {
                theOffset--;
            }
        }
    }

    return theOffset;
}

// ***** doIsOffsetAtLineStart

SInt16 doIsOffsetAtLineStart(SInt16 offset,TEHandle textEditStrucHdl)
{
    SInt16 line = 0;

    if((*textEditStrucHdl)->teLength == 0)
        return(true);

    if(offset >= (*textEditStrucHdl)->teLength)
        return((*textEditStrucHdl)->hText)[(*textEditStrucHdl)->teLength - 1] == 0x0D);

    while((*textEditStrucHdl)->lineStarts[line] < offset)
        line++;
}

```

```

    return ((*textEditStrucHdl)->lineStarts[line] == offset);
}

// ***** doDrawCaret

void doDrawCaret(SInt16 theOffset,TEHandle textEditStrucHdl)
{
    Point thePoint;
    SInt16 theLine, lineHeight;

    thePoint = TEGetPoint(theOffset,textEditStrucHdl);
    theLine = doGetLine(theOffset,textEditStrucHdl);

    if((theOffset == (*textEditStrucHdl)->teLength) &&
        ((*textEditStrucHdl)->hText)[(*textEditStrucHdl)->teLength - 1] == 0x0D)
    {
        thePoint.v += TEGetHeight(theLine,theLine,textEditStrucHdl);
    }

    PenMode(patXor);
    lineHeight = TEGetHeight(theLine,theLine,textEditStrucHdl);
    MoveTo(thePoint.h - 1,thePoint.v - 1);
    Line(0,1 - lineHeight);

    PenNormal();
}

// ***** doGetLine

SInt16 doGetLine(SInt16 theOffset,TEHandle textEditStrucHdl)
{
    SInt16 theLine = 0;

    if(theOffset > (*textEditStrucHdl)->teLength)
        return ((*textEditStrucHdl)->nLines);

    while((*textEditStrucHdl)->lineStarts[theLine] < theOffset)
        theLine++;

    return theLine;
}

// *****
// ReceiveAndUndoDrag.c
// *****

// ..... includes

#include "Drag.h"

// ..... global variables

extern Boolean gEnableDragUndoRedoItem;
extern Boolean gUndoFlag;
extern Boolean gCanAcceptItems;
extern SInt16 gInsertPosition, gCaretOffset;

// ***** dragReceiveHandler

OSErr dragReceiveHandler(WindowRef windowRef,void *handlerRefCon,DragRef dragRef)
{
    docStructurePointer docStrucPtr;
    TEHandle textEditStrucHdl;
    SInt32 totalTextStart;
    Size totalTextSize;
    Boolean wasActive, moveText, gotUndoMemory = false;
    DragAttributes dragAttributes;
    SInt16 mouseDownModifiers, mouseUpModifiers, selStart, selEnd;

```

```

UInt16          numberOfDragItems, index;
ItemReference   itemReference;
OSErr          osError;
Size           textSize;
Ptr            textDataPtr;
SInt32         additionalChars;

if(!gCanAcceptItems) || (gInsertPosition == -1))
    return dragNotAcceptedErr;

docStrucPtr = (docStructurePointer) handlerRefCon;
textEditStrucHdl = docStrucPtr->textEditStrucHdl;

// ... set graphics port to this window's port and, if necessary, activate text edit structure
SetPortWindowPort(windowRef);

wasActive = (*textEditStrucHdl)->active != 0;
if(!wasActive)
    TEActivate(textEditStrucHdl);

// ..... get drag attributes and keyboard modifiers
GetDragAttributes(dragRef,&dragAttributes);
GetDragModifiers(dragRef,0L,&mouseDownModifiers,&mouseUpModifiers);

// ... .. in case their are multiple items, save first insertion point for later TETSetSelect
totalTextStart = gInsertPosition;
totalTextSize = 0;

// ... .. for all items in drag, get 'TEXT' data, insert into this window's text edit structure
CountDragItems(dragRef,&numberOfDragItems);

for(index=1;index <= numberOfDragItems;index++)
{
    GetDragItemReferenceNumber(dragRef,index,&itemReference);

    osError = GetFlavorDataSize(dragRef,itemReference,'TEXT",&textSize);
    if(osError == noErr)
    {
        // ... if addition of drag to the text edit structure would exceed TextEdit limit, return
        if(((textEditStrucHdl)->teLength + textSize) > kMaxTELength)
            return dragNotAcceptedErr;

        // ... .. create nonrelocatable block and get the 'TEXT' data into it
        textDataPtr = NewPtr(textSize);
        if(textDataPtr == NULL)
            return dragNotAcceptedErr;

        GetFlavorData(dragRef,itemReference,'TEXT',textDataPtr,&textSize,0);

        // ... .. if caret or highlighting is on screen, remove it
        if(gCaretOffset != -1)
        {
            doDrawCaret(gCaretOffset,textEditStrucHdl);
            gCaretOffset = -1;
        }

        if(dragAttributes & kDragHasLeftSenderWindow)
            HideDragHilite(dragRef);

        // save current text and selection start/end for Undo, and set Redo/Undo menu item flags
        if(dragAttributes & kDragInsideSenderWindow)

```

```

{
    gotUndoMemory = doSavePreInsertionText(docStrucPtr);
    if(gotUndoMemory)
    {
        gEnableDragUndoRedoItem = true;
        gUndoFlag = true;
    }
}
else
    gEnableDragUndoRedoItem = false;

// ... .. if in sender window, ensure selected text is deleted if option key not down

moveText = (dragAttributes & kDragInsideSenderWindow) &&
            (!(mouseDownModifiers & optionKey) | (mouseUpModifiers & optionKey));

if(moveText)
{
    selStart = (*textEditStrucHdl)->selStart;
    selEnd   = (*textEditStrucHdl)->selEnd;

    // ... .. extend selection by one chara if space charas just before and just after

    if(doIsWhiteSpaceAtOffset(selStart - 1,textEditStrucHdl) &&
        !doIsWhiteSpaceAtOffset(selStart,textEditStrucHdl) &&
        !doIsWhiteSpaceAtOffset(selEnd - 1,textEditStrucHdl) &&
        doIsWhiteSpaceAtOffset(selEnd,textEditStrucHdl))
    {
        if(doGetCharAtOffset(selEnd,textEditStrucHdl) == ' ')
            (*textEditStrucHdl)->selEnd++;
    }

    // if insertion is after selected text, move insertion point back by size of selection

    if(gInsertPosition > selStart)
    {
        selEnd = (*textEditStrucHdl)->selEnd;
        gInsertPosition -= (selEnd - selStart);
        totalTextStart -= (selEnd - selStart);
    }

    // ... .. delete the selection

    TDelete(textEditStrucHdl);
}

// ... .. insert the 'TEXT' data at the insertion point

additionalChars = doInsertTextAtOffset(gInsertPosition,textDataPtr,textSize,
                                       textEditStrucHdl);

// ... .. if inserting multiple blocks of text, update insertion point for next block

gInsertPosition += textSize + additionalChars;
totalTextSize += textSize + additionalChars;

// ... .. dispose of nonrelocatable block

DisposePtr(textDataPtr);
}
}

// ..... select total inserted text and adjust scrollbar

TESetSelect(totalTextStart,totalTextStart + totalTextSize,textEditStrucHdl);
doAdjustScrollbar(windowRef);

// ..... set window's "touched" flag, and save post-insert selection start and end for Redo

```

```

docStrucPtr->windowTouched = true;

if(dragAttributes & kDragInsideSenderWindow)
{
    docStrucPtr->postDropSelStart = totalTextStart;
    docStrucPtr->postDropSelEnd = totalTextStart + totalTextSize;
}

// ..... if text edit structure had to be activated earlier, deactivate it

if(!wasActive)
    TEDeactivate(textEditStrucHdl);

return noErr;
}

// ***** doIsWhiteSpaceAtOffset

Boolean doIsWhiteSpaceAtOffset(SInt16 offset,TEHandle textEditStrucHdl)
{
    char theChar;

    if((offset < 0) || (offset > (*textEditStrucHdl)->teLength - 1))
        return true;

    theChar = ((char *) *((*textEditStrucHdl)->hText))[offset];

    return (doIsWhiteSpace(theChar));
}

// ***** doIsWhiteSpace

Boolean doIsWhiteSpace(char theChar)
{
    return ((theChar == ' ') || (theChar == 0x0D));
}

// ***** doGetCharAtOffset

char doGetCharAtOffset(SInt16 offset,TEHandle textEditStrucHdl)
{
    if(offset < 0)
        return 0x0D;

    return (((char *) *((*textEditStrucHdl)->hText))[offset]);
}

// ***** doInsertTextAtOffset

SInt16 doInsertTextAtOffset(SInt16 textOffset,Ptr textDataPtr,SInt32 textSize,
                           TEHandle textEditStrucHdl)
{
    SInt16 charactersAdded = 0;

    if(textSize == 0)
        return charactersAdded;

    // ..... if inserting at end of word, and selection does not begin with a space, insert a space

    if(!doIsWhiteSpaceAtOffset(textOffset - 1,textEditStrucHdl) &&
        doIsWhiteSpaceAtOffset(textOffset,textEditStrucHdl) &&
        !doIsWhiteSpace(textDataPtr[0]))
    {
        TETSetSelect(textOffset,textOffset,textEditStrucHdl);
        TEKey(' ',textEditStrucHdl);
        ++textOffset;
        ++charactersAdded;
    }
}

```

```

// ... if inserting at beginning of word and selection does not end with a space, insert space

if(doIsWhiteSpaceAtOffset(textOffset - 1,textEditStrucHdl) &&
    !doIsWhiteSpaceAtOffset(textOffset,textEditStrucHdl) &&
    !doIsWhiteSpace(textDataPtr[textSize - 1]))
{
    TETSetSelect(textOffset,textOffset,textEditStrucHdl);
    TEKey(' ',textEditStrucHdl);
    ++charactersAdded;
}

// ..... before inserting, set selection range to a zero

TETSetSelect(textOffset,textOffset,textEditStrucHdl);
TEInsert(textDataPtr,textSize,textEditStrucHdl);

return charactersAdded;
}

// ***** doSavePreInsertionText

Boolean doSavePreInsertionText(docStructurePointer docStrucPtr)
{
    OSErr osError;
    Size tempSize;
    Handle tempTextHdl;

    if(docStrucPtr->preDragText == NULL)
        docStrucPtr->preDragText = NewHandle(0);

    tempTextHdl = (*(docStrucPtr->textEditStrucHdl))->hText;
    tempSize = GetHandleSize(tempTextHdl);
    SetHandleSize(docStrucPtr->preDragText,tempSize);
    osError = MemError();
    if(osError != noErr)
    {
        doErrorAlert(eDragUndo);
        return false;
    }

    BlockMove(*tempTextHdl,*(docStrucPtr->preDragText),tempSize);

    docStrucPtr->preDragSelStart = (*((docStrucPtr->textEditStrucHdl))->selStart);
    docStrucPtr->preDragSelEnd = (*((docStrucPtr->textEditStrucHdl))->selEnd);

    return true;
}

// ***** doUndoRedoDrag

void doUndoRedoDrag(WindowRef windowRef)
{
    docStructurePointer docStrucPtr;
    Handle tempTextHdl;
    Rect portRect;

    docStrucPtr = (docStructurePointer) (GetWRefCon(windowRef));

    tempTextHdl = (*(docStrucPtr->textEditStrucHdl))->hText;
    (*(docStrucPtr->textEditStrucHdl))->hText = docStrucPtr->preDragText;
    docStrucPtr->preDragText = tempTextHdl;

    if(gUndoFlag)
    {
        (*((docStrucPtr->textEditStrucHdl))->selStart) = docStrucPtr->preDragSelStart;
        (*((docStrucPtr->textEditStrucHdl))->selEnd) = docStrucPtr->preDragSelEnd;
    }
    else
    {

```

```
    *((docStrucPtr)->textEditStrucHdl)->selStart = docStrucPtr->postDropSelStart;
    *((docStrucPtr)->textEditStrucHdl)->selEnd = docStrucPtr->postDropSelEnd;
}

gUndoFlag = !gUndoFlag;

TECallText(docStrucPtr->textEditStrucHdl);
GetWindowPortBounds(windowRef, &portRect);
InvalWindowRect(windowRef, &portRect);
}

// *****
```

Demonstration Program Drag Comments

When this program is run, the user should open the included document "Drag Document" and drag selections to other locations within the document, to other demonstration program windows, to the windows of other applications that accept 'TEXT' format data, and to the desktop and Finder windows (to create text clippings). The user should also drag text from the windows of other applications to the demonstration program's windows.

The user should note the following:

- The highlighting of the demonstration program's windows, and of the proxy icon, when items containing data of the 'TEXT' flavour are dragged over those windows.
- The movement of the insertion point caret with the cursor when items are dragged within the demonstration program's windows, and the "hiding" of the caret when the drag originates in a demonstration program window and the cursor is moved over the selection.
- When dragging and dropping within the demonstration program's windows:
 - The program's implementation of "smart drag and drop" (For example, if there is a space character immediately to the left and right of the selection, the deletion is extended to include the second space character, thus leaving a single space character between the two words which previously bracketed the selection.)
 - The availability and effect of the Undo/Redo item in the Edit menu.

The non-drag and drop aspects of this program are based on the demonstration program MonoTextEdit (Chapter 21), and the contents of Drag.h and Drag.c are very similar to the contents of MonoTextEdit.c. Accordingly, comments on the content of Drag.h and Drag.c are restricted to those areas where modifications have been made to the code contained in MonoTextEdit.c.

Drag.h

defines

Three additional constants are established for drag and drop errors.

typedefs

The docStructure data type has been extended to include fields to store the owning window's window reference, a Boolean which is set to true when the contents of the window have been modified, and five fields to support drag and drop undo/redo.

Drag.c

Global Variables

gDragTrackingHandlerUPP and gDragTrackingReceiverUPP will be assigned universal procedure pointers to the tracking and receive handlers. gEnableDragUndoRedoItem and gUndoFlag will be used to control enabling/disabling of the Drag and Drop Undo/Redo item in the Edit menu.

main

Universal procedure pointers are created for the drag tracking and receive handlers (dragTrackingHandler and dragReceiveHandler).

windowEventHandler

When the kWindowEventClickContentRgn event type is received, ConvertEventReftoEventRecord is called and the address of the event record is passed in the call to doInContent, in addition to the mouse location and Shift key position. The address of an event record will be required by the call to doStartDrag in doInContent, and ultimately by the Drag Manager function TrackDrag.

doKeyEvent

The global variable gEnableDragUndoRedoItem is set to false. This causes the Drag and Drop Undo/Redo item in the Edit menu to be disabled.

doInContent

If the mouse-down was within the TextEdit view rectangle, TEGetHiliteRgn is called to attempt to get the highlight region. If there is a highlight region (that is, a selection), and if the mouse-down was within

that region, `WaitMouseMoved` is called. `WaitMouseMoved` waits for either the mouse to move from the given initial mouse location or for the mouse button to be released. If the mouse moves away from the initial mouse location before the mouse button is released, `WaitMouseMoved` returns true, in which case the function `doStartDrag` is called.

doNewDocWindow

A nonrelocatable block is created for the window's document structure.

`SetWindowProxyCreatorAndType` is called with 0 passed in the `fileCreator` parameter and 'TEXT' passed in the `fileType` parameter to cause the system's default icon for a document file to be displayed as the proxy icon. In this program, the proxy icon is used solely for the purpose of demonstrating proxy icon highlighting when `ShowDragHilite` is called to indicate that the window is a valid drag-and-drop target.

After `gNumberOfWindows` is incremented, four of the fields of the document structure are initialised.

Following the call to `TEFeatureFlag`, the drag tracking and receive handlers are installed on the window. (Note that the pointer to the window's document structure is passed in the `handlerRefCon` parameter of the installer functions.) If either installation is unsuccessful, the window and document structure are disposed of, the tracking handler also being removed in the case of a failure to install the receive handler.

doAdjustMenus

When the global variable `gEnableDragUndoRedoItem` is set to true, the Drag and Drop Undo/Redo item is enabled, otherwise it is disabled. If the item is enabled, the global variable `gUndoFlag` controls the item text, setting it to either Undo or Redo.

doEditMenu

If the Drag and Drop Undo/Redo item is chosen from the Edit menu, the function `doUndoRedoDrag` is called.

doAdjustCursor

After the first call to `DiffRgn` (which establishes the arrow and IBeam regions), a pointer to the window's document structure is retrieved. This allows the handle to the window's `TextEdit` structure to be passed in a call to `TEGetHiliteRgn`. The region returned by `TEGetHiliteRgn` is in local coordinates, so the next three lines change it to global coordinates preparatory to a call to `DiffRgn`. The `DiffRgn` call, in effect, cuts the equivalent of the highlight region out of the IBeam region.

If the location of the mouse (returned by the call to `GetGlobalMouse`) is within the highlight region, the cursor is set to the arrow shape.

doCloseWindow

If the `preDragText` field of the window's document structure does not contain NULL, `DisposeHandle` is called to release memory assigned in support of drag and drop redo/undo.

The calls to `RemoveTrackingHandler` and `RemoveReceiveHandler` remove the tracking and receive handlers before the window is disposed of.

StartAndTrackDrag.c

doStartDrag

The call to `NewDrag` allocates a new drag object.

The call to `AddDragItemFlavor` creates a drag item and adds a data flavour (specifically 'TEXT') to that item. Note that the item reference number passed is 1. If additional flavours were to be added to the item, this same item reference number would be passed in the additional calls to `AddDragItemFlavor`.

The next block gets and sets the drag image for translucent dragging, but executes only if the program is running on Mac OS 8/9. `GetRegionBounds` gets the bounding rectangle of the highlight region and `OffsetRect` adjusts the coordinates so that top left is 0,0. The call to `NewGWorld` creates an 8-bit deep offscreen graphics world the same size as this rectangle. `CopyBits` is then called to copy the highlight region area to the offscreen graphics world. The calls to `CopyRgn` and `OffsetRgn` create a region the same shape as the highlight region and adjusted so that the top left of the bounding rectangle is 0,0. The next two lines establish the offset point required by the following call to `SetDragImage`. This offset is required to move the pixel map in the offscreen graphics world to the global coordinates where the drag image is to initially appear. Finally, the handles to the offscreen graphics world and mask, the offset, and a constant specifying the required level of translucency are passed in the call to `SetDragImage`, which associates the image with the drag reference.

Any errors which might occur in setting up the translucent drag are not critical on Mac OS 8/9 because the next block creates the drag region for the alternative visual representation (a gray outline). On Mac OS X, this is the block that creates the gray outline. The received highlight region (which is in local coordinates) is copied to `dragRgnHdl`, which is then converted to global coordinates. This region, in turn, is copied to `tempRgnHdl`, which is then inset by one pixel. The call to `DiffRgn` subtracts the inset region from `dragRgnHdl`, leaving the latter with the same outline as the highlight region but only one pixel thick. The newly defined drag region is passed in the `theRegion` parameter of the call to `TrackDrag`.

`TrackDrag` performs the drag. During the drag, the Drag Manager follows the cursor on the screen with the translucent image (Mac OS 8/9) or gray outline (Mac OS X) and sends tracking messages to applications that have registered drag tracking handlers. When the user releases the mouse button, the Drag Manager calls any receive drop handlers that have been registered on the destination window.

The `TrackDrag` function returns `noErr` in situations where the user selected a destination for the drag and the destination received data from the Drag Manager. If the user drops over a non-aware application or the receiver does not accept any data from the Drag Manager, the Drag Manager automatically provides a "zoom back" animation and returns `userCanceledErr`. Thus the first return will execute only if an error other than `userCanceledErr` was returned.

dragTrackingHandler

`dragTrackingHandler` is the drag tracking handler. It is a callback function.

Firstly, if the message received is not the enter handler message, and if it was determined at the time of receipt of the enter handler message that the drop cannot be accepted, the function returns immediately.

The pointer received in the `handlerRefCon` formal parameter is cast to a pointer to the window's document structure so that certain fields in this structure can later be accessed.

`GetDragAttributes` gets the current set of drag attribute flags. `GetCaretTime` gets the insertion point caret blink interval. `TickCount` gets the current system tick count. These latter two values will be used by the caret drawing code.

enter handler

Within the switch, the "enter handler" message is processed at the first case.

`CountDragItems` returns the number of items in the drag. Then, for each of the items in the drag, `GetDragItemReferenceNumber` is called to retrieve the item reference number, allowing the call to `GetFlavorFlags` to determine whether the item contains data of the 'TEXT' flavour. (`GetFlavorFlags` will return an error if the flavour does not exist.) `gCanAcceptItems` is assigned false if any item does not contain 'TEXT' data, meaning that the drag will only be accepted if all items contain data of the 'TEXT' flavour.

enter window

If the message is the "enter window" message, several global variables are initialised. These globals will be used when the "in window" message is received to control insertion point caret drawing and window highlighting.

in window

Each time the "in window" message is received, `GetDragMouse` is called to get the current mouse location in global coordinates. The location is copied to a local `Point` variable, which is then converted from global to local coordinates.

The first if block executes if the drag has left the sender window. If the mouse cursor is within the window's `TextEdit` view rectangle, and if `gCursorInContent` has previously been set to false (recall that it is set to false at the "enter window" message), `ShowDragHilite` is called to draw the standard drag and drop highlight around the view rectangle. `gCursorInContent` is then set to true to defeat further `ShowDragHilite` calls while the drag remains in this window.

If the mouse cursor is not within the window's `TextEdit` view rectangle, and if `gCursorInContent` has previously been set to true (causing a highlight draw), `HideDragHilite` is called to remove the highlighting, and `gCursorInContent` is set to false again so that `ShowDragHilite` is called if the drag moves back inside a view rectangle again.

With window highlighting attended to, the next task is to perform insertion point caret drawing.

The function `doGetOffset` takes the mouse location in global coordinates and the window's `TextEdit` structure handle and returns the offset into the text corresponding to the mouse location. (Note that `doGetOffset` will return -1 if the cursor is not within the content region and the view rectangle of the

window under the cursor. Note also that, if the cursor is within the content region of the window under the cursor, doGetOffset sets that window's graphics port as the current port.)

if the drag is currently inside the sender window, and the offset returned by doGetOffset indicates that the cursor is within the TextEdit selection, theOffset is set to -1. As will be seen, this defeats the drawing of the caret when the cursor is within the selection.

The offset returned by doGetOffset is then saved to a global variable. As will be seen, the value assigned to this variable the last time the "in window" case executes (when the user releases the mouse button) will be used by the receive handler dragReceiveHandler.

If the offset has changed since the last time the "in window" case executed, the caret flashing timer is reset to the time the handler was entered this time around and the caret show/hide flag is set to "show". The current offset is then assigned to the global gLastOffset preparatory to the execution of the "in window" case.

If the caret flashing interval has elapsed, the show/hide flag is toggled and the caret flashing timer is reset.

If the caret show/hide flag indicates that the caret should be "hidden", theOffset will be set to -1 to defeat caret drawing.

If the offset has changed since the last execution of the "in window" case, the function for drawing/erasing the caret is called in certain circumstances. Firstly, if this is not the first execution of the "in window" case since entering the window, doDrawCaret is called to erase the caret previously drawn at the old offset. Secondly, if the show/hide flag indicates that the caret should be drawn, doDrawCaret is called to draw the caret at the current offset.

The global which stores the old offset is assigned the current offset before the case exits.

leave window

When the "leave window" message is received, if the caret is on the screen, it is erased. If the window highlighting has previously been drawn, HideDragHilite is called to remove the highlighting.

doGetOffset

doGetOffset is called by dragTrackingHandler to return the offset into the text corresponding to the specified mouse location. doGetOffset also sets the graphics port to the port associated with the window under the cursor.

If the part code returned by FindWindow indicates that the cursor is within the content region of a window, that window's graphics port is set as the current port and the cursor location is converted to local coordinates preparatory to the call to PtInRect.

If the cursor is within the view rectangle of the TextEdit structure associated with the window, TEGetOffset is called to get the offset into the text corresponding to the cursor location and TEGetPoint is called to get the local coordinates of the character immediately before the offset.

If the offset is at a TextEdit line start, and the character immediately before the offset is not a carriage return, and the horizontal coordinate of the character immediately before the offset is less than the horizontal coordinate of the cursor, theOffset is decremented by one. In the situation where the cursor is dragged to the right of the rightmost character of a line, this will cause the caret to continue to be drawn immediately to the right of that character instead of "jumping" to the beginning of the next line.

isOffsetAtLineStart

isOffsetAtLineStart is called by doGetOffset. It returns true if the specified offset is at a TextEdit line start.

doDrawCaret

doDrawCaret is called by dragTrackingHandler to draw and erase the caret. The transfer mode is set to patXor so that two successive calls will, in effect, draw and erase the image.

The call to TEGetPoint gets the coordinates of the bottom left of the character at the specified offset. The call to the function doGetLine returns the TextEdit line number that contains the specified offset.

The next block accommodates a quirk of TextEdit. For some reason, TextEdit does not return the proper coordinates of the last offset in the field if the last character in the record is a carriage return. TEGetPoint returns a point that is one line higher than expected. This block fixes the problem.

Following the call to PenMode, TEGetHeight is called to get the height of a single line of text. A line is then drawn from a point one pixel to the left and above the bottom left of the character at the offset to a point vertically above the starting point. The length of the line is one pixel less than the line height.

doGetLine

doGetLine is called by doDrawCaret. It returns the TextEdit line number that contains the specified offset.

ReceiveAndUndoDrag.c

dragReceiveHandler

dragReceiveHandler is the drag receive handler. It is a callback function.

Recall that dragTrackingHandler sets gCanAcceptItems to true if all items in the drag contained data of the 'TEXT' flavour. Recall also that dragTrackingHandler sets gInsertPosition to -1 if the cursor is over the selection. Thus, if at least one item does not contain data of the 'TEXT' flavour, or if the cursor is over the selection at the time the mouse button is released, the function exits and dragNotAcceptedErr is returned to the Drag Manager. dragNotAcceptedErr causes the Drag Manager to execute a "zoomback" animation of the drag region to the source location.

The pointer received in the handlerRefCon formal parameter is cast to a pointer to the window's document structure. The handle to the TextEdit structure associated with the window is then retrieved from the document structure.

SetPortWindowPort sets the window's graphics port as the current port. The next line saves whether the TextEdit structure is currently active or inactive so that that state can later be restored. If the TextEdit structure is currently not active, it is made active.

GetDragAttributes and GetDragModifiers are called to get the drag's attributes and the modifier keys that were pressed at mouse-down and mouse-up time.

If there are multiple items in the drag, the initial insertion point, which is set in the drag tracking handler, is saved to a local variable for later use by TEGetSelect. (If there are multiple items to insert, the offset in gInsertPosition will be updated each time the main if block executes.)

CountDragItems returns the number of items in the drag and the main for loop executes for each item. Within the loop, the first call (to GetDragItemReferenceNumber) retrieves the item's item reference number, which is passed in the call to GetFlavorDataSize to get the size of the 'TEXT' data. If GetFlavorDataSize does not return an error, the following occurs:

- The data size returned by GetFlavorDataSize is added to the current size of the text in the TextEdit structure. If adding the 'TEXT' data to the current text would exceed the TextEdit limit, the handler exits, returning dragNotAcceptedErr to the Drag Manager.
- A nonrelocatable block the size of the 'TEXT' data is created and GetFlavorData is called to get the 'TEXT' data into that block.
- If the insertion point caret is on the screen, it is removed. (Recall that the global variables used here are set in the drag tracking handler.) If the window is currently highlighted, the highlighting is removed.
- If the drop is within the sender window, the function doSavePreInsertionText is called to save the current TextEdit text and the current selection start and end. This is to support drag undo/redo. If doSavePreInsertionText returns true, flags are set to cause the Undo/Redo item in the Edit menu to be enabled and to cause the initial item text to be set to "Undo".
- If there are multiple items in the drag, the initial insertion point is saved for later use by TEGetSelect. If there are multiple items to insert, the offset in gInsertPosition will be updated each time around.
- The variable moveText is assigned true if the drop is inside the sender window and the option key was not down when the mouse button went down or was released. If moveText is true, the current selection must be deleted, so the if block executes.
- Firstly, the current selection start and end are saved to two local variables.

- The next block implements "smart drag and drop" If the character just before selection start is a space or CR character, and if the first character in the selection is not such a character, and if the last character in the selection is not such a character, and if the character just after the selection is such a character, then, if the character just after the selection is a space character, the current end of the selection is extended to include that space character. This means that if, for example, just the characters of a word are currently selected, the space character immediately after the selection is added to the selection so that only one space character will remain between the words which bracket the selection when the selection is deleted by TDelete.
- If the current drop insertion offset is after the selection start offset, the local variable holding the selection end offset is updated (it may have been increased by the "smart drag and drop" code), and gInsertPosition and totalTextStart offsets are moved back by the length of the selection.
- TDelete then deletes the selected text (perhaps extended by one by the "smart drag and drop" code) from the TextEdit structure and redraws the text.
- The doInsertTextAtOffset function is called to insert the 'TEXT' data at the current insertion point. This function implements another aspect of "smart drag and drop" which can possibly add additional characters to the TextEdit structure. The number of additional characters added (if any) is returned by the function.
- If the main if block executes again (meaning that there are multiple items in the drag), gInsertPosition must be updated to the offset for the next insertion. This is achieved by adding the size of the last insertion, plus the number of any additional characters added by doInsertTextAtOffset to the current value in gInsertPosition. The value in totalTextSize is increased by the same amount.

When all items have been inserted, the main if block exits. TSetSelect is then called to set the selection range to the total inserted text. This call unhighlights the previous selection and highlights the new selection range. doAdjustScrollbar is called to adjust the scroll bars.

The windowTouched field in the document structure is set to true to record the fact that the contents of the window have been modified. The post-insertion selection start and selection end offsets are saved to the relevant fields of the document structure for possible use in the application's Undo/Redo function.

Finally, if the TextEdit structure was not active when dragReceiveHandler was entered, it is restored to that state.

isWhiteSpaceAtOffset

isWhiteSpaceAtOffset is called by dragReceiveHandler and doInsertTextAtOffset. Given an offset into a TextEdit structure, it determines if the character at that offset is a "white space" character, that is, a space character or a carriage return character.

isWhiteSpace

isWhiteSpace is called by isWhiteSpaceAtOffset and doInsertTextAtOffset. Given a character, it returns true if that character is either a space character or a carriage return character.

doGetCharAtOffset

doGetCharAtOffset is called by dragReceiveHandler. Given an offset and a handle to a TextEdit structure, it returns the character at that offset.

doInsertTextAtOffset

doInsertTextAtOffset is called by dragReceiveHandler to insert the drag text at the specified offset. In addition to inserting the text, dragReceiveHandler implements additional aspects of "smart drag and drop", returning the number of space characters, if any, added to the TextEdit structure text by this process.

If there is no text in the buffer, the function simply returns.

If the character to the left of the insertion offset is not a space, and if the character at the offset is a space, the insertion is at the end of a word. If, in addition, the first character in the drag is not a space, TSetSelect is called to collapse the selection range to an insertion point at the received offset, TEKey is called to insert a space character in the TextEdit structure at that offset, the offset is incremented by one to accommodate that added character, and the variable which keeps track of the number of added characters is incremented.

The next block is similar, except that it inserts a space character if the insertion is at the beginning of a word and the text to be inserted does not end with a space character. Also, in this block, the offset is not incremented.

With the "smart drag and drop" segment completed, `TESetSelect` is called to ensure that the selection range is collapsed to an insertion point at the (possibly incremented) offset, and `TEInsert` is called to insert the text into the `TextEdit` structure at that insertion point.

doSavePreInsertionText

`doSavePreInsertionText` is called by `dragReceiveHandler` to save the document's pre-drop text and selection in support of drag and drop undo/redo.

If the `preDragText` field of the window's document structure currently contains `NULL`, a new empty relocatable block is created and its handle assigned to the `preDragText` field.

The next block copies the handle to the `TextEdit` structure's text to a local variable, gets the size of the `TextEdit` structure's text, and expands the newly-created block to that size.

`BlockMove` is then called to copy the `TextEdit` structure's text to the newly-created block. In addition, the current (pre-drop) selection start and end are saved to the window's document structure.

doUndoRedoDrag

`doUndoRedoDrag` is called by `doEditMenu` in the event of the user choosing the Drag and Drop Undo/Redo item in the Edit menu. That item will only be enabled when the user has released the mouse button and the drop is within the sender window.

The first block means that, each time this function is called, the text block whose handle is assigned to the `TextEdit` structure will toggle between the pre-drop text and the post-drop text. The first time this function is called, the item text in the Edit Menu will be "Drag and Drop Undo" and the `TextEdit` structure will be assigned the handle to pre-drop text saved in the document structure's `preDragText` field. The next time the function is called, the item text in the Edit menu will be "Drag and Drop Redo" and the `TextEdit` structure will be assigned the handle to post-drop text, and so on.

The global variable `gUndoFlag` is toggled by this function. At the next block, and depending on the value in `gUndoFlag`, either the pre-drop or post-drop selection start and end offsets are assigned to the `TextEdit` structure, as appropriate. (`gUndoFlag` also controls the text in the Drag and Drop Undo/Redo item in the Edit menu. See `doAdjustMenus`.)

With the appropriate text and selection assigned to the window's `TextEdit` structure, `TECalcText` is called to wrap the text to the width of the view rectangle and re-calculate the line-starts. Finally, `InvalWindowRect` is called to force a call to `TEUpdate` (in the `doDrawContent` function) to redraw the text.