20

CARBON SCRAP

Demonstration Program: CarbonScrap

The Carbon Scrap Manager and the Scrap

Introduction

The inclusion of the word "Carbon" in the title of this chapter is quite deliberate, reflecting the fact that, in Carbon, the original Scrap Manager has been redesigned to fully support the needs of the preemptively scheduled Mac OS X.

Applications which support cut, copy, and paste operations write data to, and read data from, the **scrap**. The scrap is a storage area, maintained by the **Scrap Manager**, which holds the last text, graphics, sounds, etc., cut or copied by the user.

The various data formats in which data may be written to, and read from, the scrap are called **scrap flavours**. A scrap flavour is a self-contained, self-describing stream of bytes which represent a discreet object such as a picture or text selection. Each scrap flavour has a **scrap flavour type** and a set of **scrap flavour flags**. The scrap may contain data in one or more flavours, each flavour being a different representation of the same object.

Your application specifies the scrap flavour, or flavours, to be read from, and written to, the scrap. The ultimate aim is to allow the user to copy and paste documents:

- Within a document created by your application.
- Between different documents created by your application.
- Between documents created by your application and documents created by other applications.

Location of the Scrap

On Mac OS 8/9, space is allocated for the scrap in each application's heap. The system software stores a handle to the scrap of the current process in the system global variable `ScrapHandle`. When an application is launched, data is copied to the newly activated application's heap from the previously active application's heap. If the scrap is too large to fit in the application's heap, the scrap is copied to disk. In this event, the handle to the scrap is set to `NULL` to indicate that the scrap is on disk.

On Mac OS X, the scrap is held by the pasteboard server.

Scrap Reference

A scrap is referred to by a scrap reference. The data type `ScrapRef` is defined as a pointer to a scrap reference:

```
typedef struct OpaqueScrapRef *ScrapRef;
```

Note that, although there is only one scrap, there may be multiple `ScrapRef` values. A `ScrapRef` value is valid only until the scrap is cleared.

Scrap Flavours

Standard Scrap Flavours

Your application should be capable of writing at least one of the following **standard scrap flavours** to the scrap and should be capable of reading both:

- `'TEXT'` (a series of ASCII characters in the same format as a `'TEXT'` resource).
- `'PICT'` (a QuickDraw picture in the same format as a `'PICT'` resource).

Optional Flavours

Your application may also choose to support the following optional scrap flavours:

- `'styl'` (a series of bytes which describe styled text data, and which have the same format as a TextEdit `'styl'` resource).
- `'movv'` (a series of bytes which define a movie, and which have the same format as a `'movv'` resource).

Private Flavours

It is also possible for your application to use its own private flavour, or flavours, but this should be in addition to at least one of the standard flavours.

Preferred Flavour

Recall that each flavour in the scrap (assuming there is more than one) is simply a different representation of the same object.

Your application should have a **preferred scrap flavour**. When reading data from the scrap, your application should request its preferred flavour first and only request its next preferred flavour if the preferred flavour does not exist in the scrap. When writing data to the scrap, your application should write its preferred flavour first. Any additional flavours should be written in the preferred order.

Implementing Edit Menu Commands

You use the **Edit** menu **Cut**, **Copy**, and **Paste** commands to implement cutting, copying, and pasting of data within or between documents. The following are the actions your application should perform to support these three commands:

Edit Command	Actions Performed by Your Application
Cut	If there is a current selection range, write the data in the selection range to the scrap and remove the data from the document.
Copy	If there is a current selection range, write the data in the selection range to the scrap.
Paste	Read the scrap and insert the data (if any) at the insertion point, replacing any current selection.[1]

If your application implements a **Clear** command, it should remove the data in the current selection range but should not save the data to the scrap.

Cut and Copy — Putting Data in the Scrap

A typical approach to a basic implementation of the **Cut** and **Copy** commands is as follows:

- Determine whether the frontmost window is a document window or a dialog.
- If the frontmost window is a document window:

[1] The insertion point in a text document is represented by the blinking vertical bar known as the **caret.** There is a close relationship between the selection range and the insertion point in that the insertion point is, in effect, an empty selection range.

- Call `ClearCurrentScrap` to purge the current contents of the scrap.
- Call `GetCurrentScrap` to obtain a reference to the current scrap.
- Determine whether the current selection contains text or a picture.
- If the current selection is text, get a pointer to the selected text and get the size of the selection. If the current selection is a picture, get a pointer to the picture structure and get the size of that structure.
- Call `PutScrapFlavor` to write the data to the scrap, passing the appropriate flavour type in the `flavorType` parameter.
- If the command was the **Cut** command, delete the selection from the current document.

- If the frontmost window is a dialog, use the Dialog Manager functions `DialogCut` or `DialogCopy`, as appropriate, to write the selected data to the scrap.

Paste — Getting Data From the Scrap

When you read the data from the scrap, your application should request the data in the application's preferred flavour type. If your application determines that that flavour does not exist in the scrap, it should then request the data in another flavour. If your application does not have a preferred flavour type, it should read each flavour type that your application supports.

If you request a scrap format that is not in the scrap, the Scrap Manager uses the Translation Manager to convert any one of the scrap flavour types currently in the scrap into the scrap flavour requested by your application. The Translation Manager looks for a translator that can perform one of these translations. If such a translator is available, the Translation Manager uses the translator to translate the data in the scrap into the requested flavour.

A typical approach to an implementation of the **Paste** command, for an application that prefers a flavour type of `'TEXT'` as its first preference, is as follows:

- Determine whether the frontmost window is a document window or a dialog.
- If the frontmost window is a document window:
  - Call `GetCurrentScrap` to obtain a reference to the current scrap.
  - Call `GetScrapFlavorFlags` to determine whether the preferred flavour exists in the scrap.
  - If the preferred flavour type (`'TEXT'`) does exist, call `GetScrapFlavorSize` to get the size of the text data, allocate a relocatable block of that size, and call `GetScrapFlavorData` to read the data into that block. Copy the data in the relocatable block to the current document at the insertion point.
  - If the preferred flavour type does not exist, call `GetScrapFlavorFlags` again to determine whether the next preferred flavour (say, `'PICT'`) exists in the scrap. If it does, call `GetScrapFlavorSize` to get the size of the picture data, allocate a relocatable block of that size, and call `GetScrapFlavorData` to read the data into that block. Call `DrawPicture` to draw the picture described by the data in the relocatable block in the current document at the insertion point.
- If the frontmost window is a , use the Dialog Manager function `DialogPaste` to paste the text from the scrap in the dialog.

Enabling the Paste Menu Item

Your application can determine whether to enable the **Paste** item in the **Edit** menu by calling `GetScrapFlavorFlags` to determine whether the scrap contains data of the flavour type specified in that call. `GetScrapFlavorFlags` returns `noErr` if the specified flavour exists.

Example

Fig 1 illustrates two cases, both of which deal with a user copying a picture consisting of text from a source document created by one application to a destination document created by another application.

In the first case, the source application has chosen to write only the `'PICT'` flavour to the scrap, and the destination application has pasted the data, in that flavour, to its document.

In the second case, the source application has chosen to write both the `'TEXT'` and `'PICT'` flavours to the scrap, and the destination application has chosen the `'TEXT'` flavour as the preferred flavour for the paste. The data is thus inserted into the document as editable text.

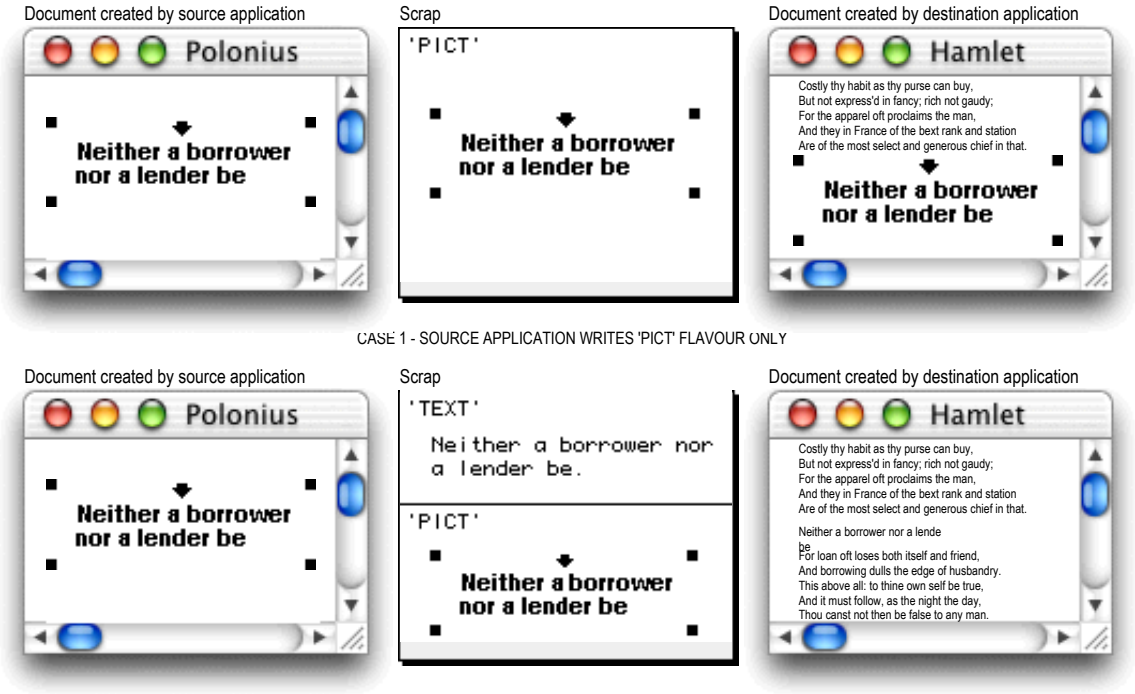

FIG 1 - SPECIFYING FLAVOURS TO WRITE TO AND READ FROM THE DESK SCRAP

Clipboard Windows

Your application can provide a **Show Clipboard** command in the **Edit** menu which, when chosen, shows a window which displays the current contents of the scrap. Such a window is known as a **Clipboard** window. The **Show Clipboard** command should be toggled with a **Hide Clipboard** command to allow the user to hide the Clipboard window when required.

Although the scrap may contain multiple scrap flavours, your Clipboard window should ordinarily display the data in the application's preferred flavour only.

If the user has chosen to open the Clipboard window, your application should hide the window on receipt of a suspend event (Classic event model) or `kEventAppActivated` event type (Carbon event model) and show it when a resume event (Classic event model) or `kEventAppDeactivated` event type (Carbon event model) is received. This is necessary because the contents of the scrap could change while the application is in the background.

Transferring the Scrap to Disk — Mac OS 8/9

Although, on Mac OS 8/9, the scrap is usually located in memory, your application can write the contents of the scrap in memory to a scrap file using `UnloadScrap`. You should do this only if memory is not large enough to hold the data you need to write to the scrap. After writing the contents of the scrap to disk, `UnloadScrap` releases the memory previously occupied by the scrap. Thereafter, any operations your application performs on data in the scrap affect the scrap as stored in the scrap file on disk. You can use `LoadScrap` to read the contents of the scrap file back into memory.

On Mac OS X, calls to `LoadScrap` and `UnloadScrap` are ignored.

Main Carbon Scrap Manager Functions

The main Carbon Scrap Manager functions are as follows:

Function	Description
`GetCurrentScrap`	Gets a reference to the current scrap. (Note that this reference will become invalid and unusable after the scrap is cleared.)
`GetScrapFlavorFlags`	Determines whether the scrap contains data for a particular flavour and provides information about that flavour if it exists. (Amongst other things, this function is useful for deciding whether to enable the **Paste** item in your **Edit** menu.)
`GetScrapFlavorSize`	Gets the size of the data of the specified flavour from the specified scrap.
`GetScrapFlavorData`	Gets the data of the specified flavour from the specified scrap.
`ClearCurrentScrap`	Clears the current scrap. This function should be called immediately the user requests a **Copy** or **Cut** operation.
`PutScrapFlavor`	Puts data on the scrap. Also promises data to the specified scrap (see below).

Associated Constants and Data Types

The following constants and data types are associated with the main Scrap Manager functions:

Scrap Flavour Type Constants

Constant	Flavour Type	Description
`kScrapFlavorTypePicture`	`'PICT'`	Picture
`kScrapFlavorTypeText`	`'TEXT'`	Text
`kScrapFlavorTypeTextStyle`	`'styl'`	Text style
`kScrapFlavorTypeMovie`	`'moov'`	Movie

Scrap Flavour Flag Constants

In the following, the first two constants may be passed in the `flavorFlags` parameter in calls to `PutScrapFlavour`, and the third is received in the `flavorFlags` parameter in calls to `GetScrapFlavorFlags`:

Constant	Meaning
`kScrapFlavorMaskNone`	No flags required.
`kScrapFlavorMaskSenderOnly`	Only the process which puts the flavour on the scrap can see it. If another process puts a flavour with this flag on the scrap,your process will never see the flavour. Accordingly, there is no point in testing for this flag. This flag is typically used to save a private flavour to the scrap so that other promised (see below) public flavours can be derived from it on demand.
`kScrapFlavorMaskTranslated`	The flavour was translated, by the Translation Manager, from some other flavour in the scrap. (Most callers should not care about this flag.)

ScrapFlavorInfo Data Type

The `ScrapFlavorInfo` data type describes a single flavour within a scrap and is used by those functions which get information about the current scrap (`GetScrapFlavorFlags`, `GetScrapFlavorSize`, and `GetScrapFlavorData`):

```
struct ScrapFlavorInfo
{
  ScrapFlavorType  flavorType;
  ScrapFlavorFlags flavorFlags;
};
typedef struct ScrapFlavorInfo ScrapFlavorInfo;
```

Private Scrap

As an alternative to writing to and reading from the scrap whenever the user cuts, copies and pastes data, your application can choose to use its own **private scrap**. An application which uses a private scrap copies data to its private scrap when the user chooses the **Cut** or **Copy** command and pastes data from the private scrap when the user chooses the **Paste** command.

Additional Actions — Old Scrap Manager

In the old pre-Carbon Scrap Manager, an application which used a private scrap had to take the following additional actions whenever it received suspend and resume events:

- ***Suspend Event.*** On receipt of a suspend event, the application had to copy data from the private scrap to the scrap.
- ***Resume Event.*** On receipt of a resume event, the application had to first examine the `convertClipboardFlag` bit in the `message` field of the resume event structure to determine if the data in the scrap had changed since the previous suspend event. If the data in the scrap had changed, the application had to copy the data from the scrap to its private scrap. The application's menu adjustment function enabled the **Paste** item if the data copied to the private scrap was of the preferred, or other acceptable, type.

The process is illustrated at Fig 2.

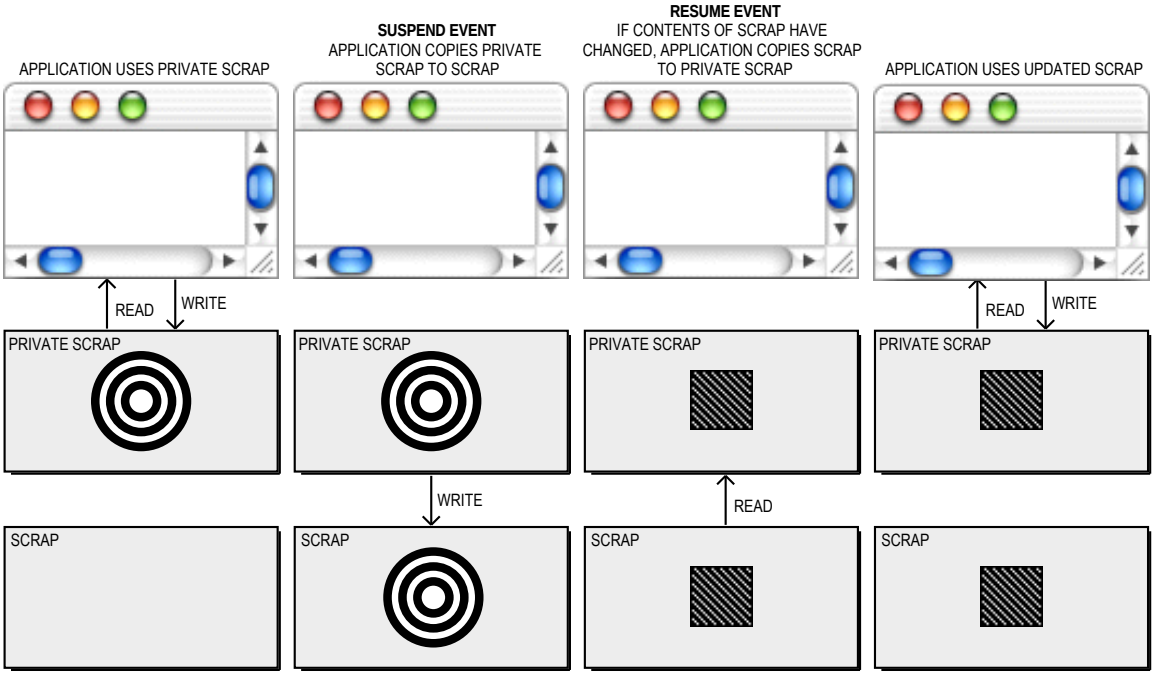

FIG 2 - PRIVATE SCRAP AND SUSPEND AND RESUME EVENTS - OLD SCRAP MANAGER

Additional Actions — Carbon Scrap Manager

In the preemptively scheduled Mac OS X, this rather straightforward approach is no longer feasible. Consider the following scenario on Mac OS X:

- Application B, which has a private scrap, is the frontmost application. The user clicks in a window belonging to application A to make application A the frontmost application. Application B receives a suspend event and begins to convert its private scrap.
- While application B is still converting its private scrap, application A has become the frontmost application, and the user clicks in its menu bar. Application A, needing to decide whether to enable

the **Paste** item in its **Edit** menu, looks at the scrap to determine what flavours it contains. Because application B has not finished converting its private scrap, and thus has not put anything onto the scrap, application A finds nothing it wants on the scrap and, accordingly, disables the **Paste** item.

The situation in which application A finds itself with regard to the **Paste** item is not acceptable in terms of human interface. The user cannot be expected to know that application B is still converting its scrap and that application A's **Paste** item will be enabled in due course.

Making Promises

The Carbon Scrap Manager eliminates this problem using the concept of **promised flavours**. If, in the above example, application B calls `PutScrapFlavor` with `NULL` passed in the `flavorData` parameter whenever the user chooses **Cut** or **Copy**, a promise is made that data of the flavour specified in the `flavorType` parameter will later be placed on the scrap. On checking the scrap, application A will see the promise and can thus enable its **Paste** item in the expectation that the actual data will eventually appear in the scrap. The actual data can then be provided by application B through a subsequent call to `PutScrapFlavor` during the execution of a **scrap promise keeper (callback) function**. (Scrap promise keeper callback functions are called by the Carbon Scrap Manager as required to keep an earlier promise of a particular scrap flavour.)

In the first (promise-making) call to `PutScrapFlavor`, passing a non-zero size in the `flavorSize` parameter is optional; however, providing the size is advisable because callers of `GetScrapFlavorSize` will then be able to avoid blocking. If the size is provided, the subsequent call to `PutScrapFlavor` must provide the same amount of data as was promised. If the size is unknown at the time of the promise, your application should pass `kScrapFlavorSizeUnknown` in the `flavorSize` parameter.

Note that the promise-making `PutScrapFlavor` call cannot be made when your application receives a suspend event. This is because of the fundamental difference between the receipt of suspend events in Carbon applications as compared with Classic applications (see Chapter 2). Making the promise each time the user chooses **Cut** or **Copy** involves very little overhead, since only the promise, not the data, is being placed on the scrap.

Calling In Promises

In applications that use the Classic event model, your application should invariably call `CallInScrapPromises` on exit to cater for the possibility that it may have made promises that, after it quits, it cannot possibly honour. `CallInScrapPromises` forces all promises to be kept. On Mac OS X, this action is necessary even if your application has itself made no promises, the reason being that it is possible that, unbeknown to the application, promises could have been made on its behalf. For example, when you copy `TEXT` data (which has ASCII 13 for line endings) onto the scrap, the Carbon Scrap Manager promises other flavours which have different line endings and/or text encodings so that Cocoa applications can paste.

Calling `CallInScrapPromises` is not necessary in applications which use the Carbon event model.

TextEdit, Dialogs, and Scrap

TextEdit and Scrap

TextEdit is a collection of functions and data structures which you can use to provide your application with basic text editing capabilities.

If your application uses TextEdit in its windows, be aware that TextEdit maintains its own private scrap. Accordingly:

- The special TextEdit functions `TECut`, `TECopy`, and `TEToScrap` are used in the processes of cutting text from the document and copying text to the TextEdit private scrap and to the scrap.
- The special TextEdit functions `TEPaste`, `TEStylePaste`, and `TEFromScrap` are used in the processes of pasting text from the TextEdit private scrap and copying text from the scrap to the TextEdit private scrap.

Chapter 21 describes TextEdit, including the TextEdit private scrap and the TextEdit scrap-related functions.

Dialogs and Scrap

Dialogs may contain edit text items, and the Dialog Manager uses TextEdit to perform the editing operations within those items.

You can use the Dialog Manager to handle most editing operations within dialogs. The Dialog Manager functions `DialogCut`, `DialogCopy`, and `DialogPaste` may be used to implement **Cut**, **Copy** and **Paste** commands within edit text items in dialogs. (See the demonstration program at Chapter 8.)

TextEdit's private scrap facilitates the copying and pasting of data between dialogs. However, your application itself must ensure that the user can copy and paste data between your application's dialogs and its document windows. If your application uses TextEdit for all editing operations within its document windows, this is easily achieved because TextEdit's `TECut`, `TECopy`, `TEPaste`, and `TEStylePaste` functions and the Dialog Manager's `DialogCut`, `DialogCopy`, and `DialogPaste` functions all use TextEdit's private scrap.

Main Carbon Scrap Manager Constants, Data Types, and Functions

Constants

Scrap Flavour Types

```
kScrapFlavorTypePicture     = FOUR_CHAR_CODE('PICT')  Picture
kScrapFlavorTypeText        = FOUR_CHAR_CODE('TEXT')  Text
kScrapFlavorTypeTextStyle   = FOUR_CHAR_CODE('styl')  Text style
kScrapFlavorTypeMovie       = FOUR_CHAR_CODE('moov')  Movie
```

Scrap Flavour Flags

```
kScrapFlavorMaskNone        = 0x00000000
kScrapFlavorMaskSenderOnly  = 0x00000001
kScrapFlavorMaskTranslated  = 0x00000002
```

Promising Flavours

```
kScrapFlavorSizeUnknown     = -1
```

Result Codes

```
internalScrapErr            = -4988
duplicateScrapFlavorErr     = -4989
badScrapRefErr              = -4990
processStateIncorrectErr    = -4991
scrapPromiseNotKeptErr      = -4992
noScrapPromiseKeeperErr     = -4993
nilScrapFlavorDataErr       = -4994
scrapFlavorFlagsMismatchErr = -4995
scrapFlavorSizeMismatchErr  = -4996
illegalScrapFlavorFlagsErr  = -4997
illegalScrapFlavorTypeErr   = -4998
illegalScrapFlavorSizeErr   = -4999
scrapFlavorNotFoundErr      = -102
needClearScrapErr           = -100
```

Data Types

```
typedef struct OpaqueScrapRef  *ScrapRef;
typedef FourCharCode           ScrapFlavorType;
typedef UInt32                 ScrapFlavorFlags;
```

ScrapFlavorInfo

```
struct ScrapFlavorInfo
{
  ScrapFlavorType  flavorType;
  ScrapFlavorFlags flavorFlags;
};
typedef struct ScrapFlavorInfo ScrapFlavorInfo;
```

Functions

Obtaining a Reference to the Current Scrap

```
OSStatus  GetCurrentScrap(ScrapRef *scrap);
```

Obtaining Information About a Specific Scrap Flavour

```
OSStatus  GetScrapFlavorFlags(ScrapRef scrap,ScrapFlavorType flavorType,
          ScrapFlavorFlags *flavorFlags);
```

Obtaining the Size of Data of a Specified Scrap Flavour

```
OSStatus  GetScrapFlavorSize(ScrapRef scrap,ScrapFlavorType flavorType,Size *byteCount);
```

Obtaining the Data of a Specified Scrap Flavour

```
OSStatus  GetScrapFlavorData(ScrapRef scrap,ScrapFlavorType flavorType,Size *byteCount,
          void *destination);
```

Writing Data to the Scrap and Clearing the Scrap

```
OSStatus  PutScrapFlavor(ScrapRef scrap,ScrapFlavorType flavorType,
          ScrapFlavorFlags flavorFlags, Size flavorSize,const void *flavorData);
OSStatus  ClearCurrentScrap(void);
```

Scrap Promise Keeping

```
ScrapPromiseKeeperUPP  NewScrapPromiseKeeperUPP(ScrapPromiseKeeperProcPtr userRoutine);
void                   DisposeScrapPromiseKeeperUPP(ScrapPromiseKeeperUPP userUPP);
OSStatus  SetScrapPromiseKeeper(ScrapRef scrap,ScrapPromiseKeeperUPP upp,
          const void *userData);
OSStatus  CallInScrapPromises(void);
```

Application-Defined (Callback) Function

```
OSStatus  myScrapPromiseKeeperFunction(ScrapRef scrap,ScrapFlavorType flavorType,
                                       void *userData);
```

Transferring the Scrap Between Memory and Disk (Mac OS 8/9)

```
SInt32    UnloadScrap(void);  // Does nothing when called on Mac OS X
SInt32    LoadScrap(void);    // Does nothing when called on Mac OS X
```

Demonstration Program CarbonScrap Listing

```
// ****************************************************************************************
// CarbonScrap.c                                                          CARBON EVENT MODEL
// ****************************************************************************************
//
// This program utilises Carbon Scrap Manager functions to allow the user to:
//
// •  Cut, copy, clear, and paste text and pictures from and to two document windows opened by
//    the program.
//
// •  Paste text and pictures cut or copied from another application to the two document
//    windows.
//
// •  Open and close a Clipboard window, in which the current contents of the scrap are
//    displayed.
//
// The program's preferred scrap flavour type is 'TEXT'.  Thus, if the scrap contains data in
// both the 'TEXT' and 'PICT' flavour types, only the 'TEXT' flavour will be used for pastes
// to the document windows and display in the Clipboard window.
//
// In order to keep that part of the source code that is not related to the Carbon Scrap
// Manager to a minimum, the windows do not display insertion points, nor can the pictures be
// dragged within the windows.  The text and pictures are not inserted into a document as
// such.  Rather, when the Paste item in the Edit menu is chosen:
//
// •  The text or picture on the Clipboard is simply drawn in the centre of the active window.
//
// •  A handle to the text or picture is assigned to fields in a document structure associated
//    with the window.  (The demonstration program MonoTextEdit (Chapter 21) shows how to cut,
//    copy, and paste text from and to a TextEdit structure using the scrap.)
//
// For the same reason, highlighting the selected text or picture in a window is simplified by
// simply inverting the image.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, and Edit menus (preload,
//    non-purgeable).
//
// •  A 'TEXT' resource (non-purgeable) containing text displayed in the left window at
//    program  start.
//
// •  A 'PICT' resource (non-purgeable) containing a picture displayed in the right window at
//    program start.
//
// •  A 'STR#' resource (purgeable) containing strings to be displayed in the error Alert.
//
// •  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  iClose           4
#define  iQuit            12
#define mEdit             130
```

```
#define  iCut             3
#define  iCopy            4
#define  iPaste           5
#define  iClear           6
#define  iClipboard       8
#define rText             128
#define rPicture          128
#define rErrorStrings     128
#define  eFailMenu        1
#define  eFailWindow      2
#define  eFailDocStruc    3
#define  eFailMemory      4
#define  eClearScrap      5
#define  ePutScrapFlavor  6
#define  eGetScrapSize    7
#define  eGetScrapData    8
#define kDocumentType     1
#define kClipboardType    2
#define MAX_UINT32        0xFFFFFFFF

// ..... typedefs

typedef struct
{
  PicHandle pictureHdl;
  Handle    textHdl;
  Boolean   selectFlag;
  SInt16    windowType;
} docStructure, **docStructureHandle;

// ..... global variables

Boolean   gRunningOnX         = false;
WindowRef gClipboardWindowRef = NULL;
Boolean   gClipboardShowing   = false;

// ..... function prototypes

void            main                   (void);
void            doPreliminaries        (void);
OSStatus        appEventHandler        (EventHandlerCallRef,EventRef,void *);
OSStatus        docWindowEventHandler  (EventHandlerCallRef,EventRef,void *);
OSStatus        clipWindowEventHandler (EventHandlerCallRef,EventRef,void *);
void            doAdjustMenus          (void);
void            doMenuChoice           (MenuID,MenuItemIndex);
void            doErrorAlert           (SInt16);
void            doOpenDocumentWindows  (void);
EventHandlerUPP doGetHandlerUPP        (void);
void            doCloseWindow          (void);
void            doInContent            (Point);
void            doCutCopyCommand       (Boolean);
void            doPasteCommand         (void);
void            doClearCommand         (void);
void            doClipboardCommand     (void);
void            doDrawClipboardWindow  (void);
void            doDrawDocumentWindow   (WindowRef);
Rect            doSetDestRect          (Rect *,WindowRef);

// ********************************************************************************** main

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

```
  // ..... do preliminaries

  doPreliminaries();

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

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    doErrorAlert(eFailMenu);
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }

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

  // ..... open document windows

  doOpenDocumentWindows();

  // ..... install application event handler

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

  // ..... run application event loop

  RunApplicationEventLoop();
}

// ********************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  MoreMasterPointers(96);
  InitCursor();
}

// ********************************************************************** appEventHandler

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

  eventClass = GetEventClass(eventRef);
```

```
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassApplication:
      if(eventKind == kEventAppActivated)
      {
        SetThemeCursor(kThemeArrowCursor);
        if(gClipboardWindowRef && gClipboardShowing)
          ShowWindow(gClipboardWindowRef);
      }
      else if(eventKind == kEventAppDeactivated)
      {
        if(gClipboardWindowRef && gClipboardShowing)
          ShowHide(gClipboardWindowRef,false);
      }
    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)
        doAdjustMenus();
      result = noErr;
      break;
  }

  return result;
}

// ********************************************************************** docWindowEventHandler

OSStatus  docWindowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                                void * userData)
{
  OSStatus           result = eventNotHandledErr;
  UInt32             eventClass;
  UInt32             eventKind;
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  Point              mouseLocation;

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

  switch(eventClass)
  {
    case kEventClassWindow:
      GetEventParameter(eventRef,kEventParamDirectObject,typeWindowRef,NULL,sizeof(windowRef),
                    NULL,&windowRef);
      switch(eventKind)
      {
        case kEventWindowDrawContent:
          docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
          if((*docStrucHdl)->pictureHdl != NULL || (*docStrucHdl)->textHdl != NULL)
```

```
            doDrawDocumentWindow(windowRef);
          result = noErr;
          break;

        case kEventWindowClickContentRgn:
          GetEventParameter(eventRef,kEventParamMouseLocation,typeQDPoint,NULL,
                            sizeof(mouseLocation),NULL,&mouseLocation);
          SetPortWindowPort(windowRef);
          GlobalToLocal(&mouseLocation);
          doInContent(mouseLocation);
          result = noErr;
          break;
      }
      break;
  }

  return result;
}

// ********************************************************************* clipWindowEventHandler

OSStatus  clipWindowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                                 void * userData)
{
  OSStatus result = eventNotHandledErr;
  UInt32   eventClass;
  UInt32   eventKind;
  MenuRef  editMenuRef;

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

  if(eventClass == kEventClassWindow)
  {
    switch(eventKind)
    {
      case kEventWindowActivated:
      case kEventWindowDeactivated:
      case kEventWindowDrawContent:
        doDrawClipboardWindow();
        result = noErr;
        break;

      case kEventWindowClose:
        DisposeWindow(gClipboardWindowRef);
        gClipboardWindowRef = NULL;
        gClipboardShowing = false;
        editMenuRef = GetMenuRef(mEdit);
        SetMenuItemText(editMenuRef,iClipboard,"\pShow Clipboard");
        break;
    }
  }

  return result;
}

// ********************************************************************************** doAdjustMenus

void  doAdjustMenus(void)
{
  MenuRef            fileMenuRef, editMenuRef;
  docStructureHandle docStrucHdl;
  ScrapRef           scrapRef;
  OSStatus           osError;
  ScrapFlavorFlags   scrapFlavorFlags;
  Boolean            scrapHasText = false, scrapHasPicture = false;

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

```
  docStrucHdl = (docStructureHandle) GetWRefCon(FrontWindow());

  if((*docStrucHdl)->windowType == kClipboardType)
    EnableMenuItem(fileMenuRef,iClose);
  else
    DisableMenuItem(fileMenuRef,iClose);

  if(((*docStrucHdl)->pictureHdl || (*docStrucHdl)->textHdl) && ((*docStrucHdl)->selectFlag))
  {
    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)
    scrapHasText = true;

  osError = GetScrapFlavorFlags(scrapRef,kScrapFlavorTypePicture,&scrapFlavorFlags);
  if(osError == noErr)
    scrapHasPicture = true;

  if((scrapHasText || scrapHasPicture) && ((*docStrucHdl)->windowType != kClipboardType))
    EnableMenuItem(editMenuRef,iPaste);
  else
    DisableMenuItem(editMenuRef,iPaste);

  DrawMenuBar();
}

// ********************************************************************************* doMenuChoice

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

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

    case mFile:
      if(menuItem == iClose)
        doCloseWindow();
      break;

    case mEdit:
      switch(menuItem)
      {
        case iCut:
          doCutCopyCommand(true);
          break;

        case iCopy:
          doCutCopyCommand(false);
          break;
```

```
        case iPaste:
          doPasteCommand();
          break;

        case iClear:
          doClearCommand();
          break;

        case iClipboard:
          doClipboardCommand();
          break;
      }
      break;
  }
}

// ********************************************************************************* doErrorAlert

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

  GetIndString(errorString,rErrorStrings,errorCode);
  StandardAlert(kAlertStopAlert,errorString,NULL,NULL,&itemHit);
  ExitToShell();
}

// ************************************************************************ doOpenDocumentWindows

void  doOpenDocumentWindows(void)
{
  SInt16             a;
  OSStatus           osError;
  WindowRef          windowRef;
  Rect               contentRect = { 43,7,223,297 }, theRect;
  Str255             title1      = "\pDocument A";
  Str255             title2      = "\pDocument B";
  docStructureHandle docStrucHdl;
  EventTypeSpec      windowEvents[] = { { kEventClassWindow,  kEventWindowDrawContent     },
                                        { kEventClassWindow,  kEventWindowClickContentRgn } };

  for(a=0;a<2;a++)
  {
    osError = CreateNewWindow(kDocumentWindowClass,kWindowStandardHandlerAttribute,
                              &contentRect,&windowRef);
    if(osError != noErr)
      doErrorAlert(eFailWindow);

    if(a == 0)
    {
      SetWTitle(windowRef,"\pDocument A");
      OffsetRect(&contentRect,305,0);
    }
    else
      SetWTitle(windowRef,"\pDocument B");

    if(!(docStrucHdl = (docStructureHandle) NewHandle(sizeof(docStructure))))
      doErrorAlert(eFailDocStruc);
    SetWRefCon(windowRef,(SInt32) docStrucHdl);

    (*docStrucHdl)->pictureHdl = NULL;
    (*docStrucHdl)->textHdl    = NULL;
    (*docStrucHdl)->windowType = kDocumentType;
    (*docStrucHdl)->selectFlag = false;

    SetPortWindowPort(windowRef);

    if(gRunningOnX)
```

```
    {
      GetWindowPortBounds(windowRef,&theRect);
      InsetRect(&theRect,40,40);
      ClipRect(&theRect);
    }
    else
      UseThemeFont(kThemeSmallSystemFont,smSystemScript);

    if(a == 0)
      (*docStrucHdl)->textHdl = (Handle) GetResource('TEXT',rText);
    else
      (*docStrucHdl)->pictureHdl = GetPicture(rPicture);

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

    ShowWindow(windowRef);
  }
}

// ********************************************************************** doGetHandlerUPP

EventHandlerUPP  doGetHandlerUPP(void)
{
  static EventHandlerUPP  windowEventHandlerUPP;

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

  return windowEventHandlerUPP;
}

// ********************************************************************** doCloseWindow

void  doCloseWindow(void)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  MenuRef            editMenuRef;

  windowRef = FrontWindow();
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  if((*docStrucHdl)->windowType == kClipboardType)
  {
    DisposeWindow(windowRef);
    gClipboardWindowRef = NULL;
    gClipboardShowing = false;
    editMenuRef = GetMenuRef(mEdit);
    SetMenuItemText(editMenuRef,iClipboard,"\pShow Clipboard");
  }
}

// ********************************************************************** doInContent

void  doInContent(Point mouseLocation)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  GrafPtr            oldPort;
  Rect               theRect;

  windowRef = FrontWindow();
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  if((*docStrucHdl)->windowType == kClipboardType)
    return;

  GetPort(&oldPort);
```

```
  SetPortWindowPort(windowRef);

  if((*docStrucHdl)->textHdl != NULL || (*docStrucHdl)->pictureHdl != NULL)
  {
    if((*docStrucHdl)->textHdl != NULL)
    {
      GetWindowPortBounds(windowRef,&theRect);
      InsetRect(&theRect,40,40);
    }
    else if((*docStrucHdl)->pictureHdl != NULL)
    {
      theRect = doSetDestRect(&(*(*docStrucHdl)->pictureHdl)->picFrame,windowRef);
    }

    if(PtInRect(mouseLocation,&theRect) && (*docStrucHdl)->selectFlag == false)
    {
      (*docStrucHdl)->selectFlag = true;
      InvertRect(&theRect);
    }
    else if(!PtInRect(mouseLocation,&theRect) && (*docStrucHdl)->selectFlag == true)
    {
      (*docStrucHdl)->selectFlag = false;
      InvertRect(&theRect);
    }
  }

  SetPort(oldPort);
}

// ********************************************************************** doCutCopyCommand

void  doCutCopyCommand(Boolean cutFlag)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  OSStatus           osError;
  ScrapRef           scrapRef;
  Size               dataSize;
  GrafPtr            oldPort;
  Rect               portRect;

  windowRef = FrontWindow();
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  if((*docStrucHdl)->selectFlag == false)
    return;

  osError = ClearCurrentScrap();
  if(osError == noErr)
  {
    GetCurrentScrap(&scrapRef);

    if((*docStrucHdl)->textHdl != NULL)  // ..........'TEXT'
    {
      dataSize = GetHandleSize((Handle) (*docStrucHdl)->textHdl);
      HLock((*docStrucHdl)->textHdl);

      osError = PutScrapFlavor(scrapRef,kScrapFlavorTypeText,kScrapFlavorMaskNone,
                               dataSize,*((*docStrucHdl)->textHdl));
      if(osError != noErr)
        doErrorAlert(ePutScrapFlavor);
    }
    else if((*docStrucHdl)->pictureHdl != NULL)  // .......... 'PICT'
    {
      dataSize = GetHandleSize((Handle) (*docStrucHdl)->pictureHdl);
      HLock((Handle) (*docStrucHdl)->pictureHdl);

      osError = PutScrapFlavor(scrapRef,kScrapFlavorTypePicture,kScrapFlavorMaskNone,
                               dataSize,*((Handle) (*docStrucHdl)->pictureHdl));
```

```
      if(osError != noErr)
        doErrorAlert(ePutScrapFlavor);
    }

    if((*docStrucHdl)->textHdl != NULL)
      HUnlock((*docStrucHdl)->textHdl);
    if((*docStrucHdl)->pictureHdl != NULL)
      HUnlock((Handle) (*docStrucHdl)->pictureHdl);
  }
  else
    doErrorAlert(eClearScrap);

  if(cutFlag)
  {
    GetPort(&oldPort);
    SetPortWindowPort(windowRef);

    if((*docStrucHdl)->pictureHdl != NULL)
    {
      DisposeHandle((Handle) (*docStrucHdl)->pictureHdl);
      (*docStrucHdl)->pictureHdl = NULL;
      (*docStrucHdl)->selectFlag = false;
    }
    if((*docStrucHdl)->textHdl != NULL)
    {
      DisposeHandle((*docStrucHdl)->textHdl);
      (*docStrucHdl)->textHdl = NULL;
      (*docStrucHdl)->selectFlag = false;
    }

    GetWindowPortBounds(windowRef,&portRect);
    EraseRect(&portRect);

    SetPort(oldPort);
  }

  if(gClipboardWindowRef != NULL)
    doDrawClipboardWindow();
}

// ********************************************************************************* doPasteCommand

void  doPasteCommand(void)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  GrafPtr            oldPort;
  ScrapRef           scrapRef;
  OSStatus           osError;
  ScrapFlavorFlags   flavorFlags;
  Size               sizeOfPictData = 0, sizeOfTextData = 0;
  Handle             newTextHdl, newPictHdl;
  CFStringRef        stringRef;
  Rect               destRect, portRect;

  windowRef = FrontWindow();
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  GetPort(&oldPort);
  SetPortWindowPort(windowRef);

  GetCurrentScrap(&scrapRef);

  // ..................................................................................................... 'TEXT'

  osError = GetScrapFlavorFlags(scrapRef,kScrapFlavorTypeText,&flavorFlags);
  if(osError == noErr)
  {
    osError = GetScrapFlavorSize(scrapRef,kScrapFlavorTypeText,&sizeOfTextData);
```

```
  if(osError == noErr && sizeOfTextData > 0)
  {
    newTextHdl = NewHandle(sizeOfTextData);
    osError = MemError();
    if(osError == memFullErr)
      doErrorAlert(eFailMemory);

    HLock(newTextHdl);

    osError = GetScrapFlavorData(scrapRef,kScrapFlavorTypeText,&sizeOfTextData,*newTextHdl);
    if(osError != noErr)
      doErrorAlert(eGetScrapData);

    // ..... draw text in window

    GetWindowPortBounds(windowRef,&portRect);
    EraseRect(&portRect);
    InsetRect(&portRect,40,40);

    if(!gRunningOnX)
    {
      TETextBox(*newTextHdl,sizeOfTextData,&portRect,teFlushLeft);
    }
    else
    {
      stringRef =  CFStringCreateWithBytes(NULL,(UInt8 *) *newTextHdl,sizeOfTextData,
                                           smSystemScript,false);
      DrawThemeTextBox(stringRef,kThemeSmallSystemFont,kThemeStateActive,true,&portRect,
                       teFlushLeft,NULL);
      if(stringRef != NULL)
        CFRelease(stringRef);
    }

    HUnlock(newTextHdl);

    (*docStrucHdl)->selectFlag = false;

    // ..... assign handle to new text to window's document structure

    if((*docStrucHdl)->textHdl != NULL)
      DisposeHandle((*docStrucHdl)->textHdl);
    (*docStrucHdl)->textHdl = newTextHdl;

    if((*docStrucHdl)->pictureHdl != NULL)
      DisposeHandle((Handle) (*docStrucHdl)->pictureHdl);
    (*docStrucHdl)->pictureHdl = NULL;
  }
  else
    doErrorAlert(eGetScrapSize);
}

// ..... ' PICT'

else
{
  (osError = GetScrapFlavorFlags(scrapRef,kScrapFlavorTypePicture,&flavorFlags));
  if(osError == noErr)
  {
    osError = GetScrapFlavorSize(scrapRef,kScrapFlavorTypePicture,&sizeOfPictData);
    if(osError == noErr && sizeOfPictData > 0)
    {
      newPictHdl = NewHandle(sizeOfPictData);
      osError = MemError();
      if(osError == memFullErr)
        doErrorAlert(eFailMemory);

      HLock(newPictHdl);

      osError = GetScrapFlavorData(scrapRef,kScrapFlavorTypePicture,&sizeOfPictData,
```

```
                                    *newPictHdl);
        if(osError != noErr)
          doErrorAlert(eGetScrapData);

        // ..... draw picture in window

        GetWindowPortBounds(windowRef,&portRect);
        EraseRect(&portRect);
        (*docStrucHdl)->selectFlag = false;
        destRect = doSetDestRect(&(*(PicHandle) newPictHdl)->picFrame,windowRef);
        DrawPicture((PicHandle) newPictHdl,&destRect);

        HUnlock(newPictHdl);

        (*docStrucHdl)->selectFlag = false;

        // ..... assign handle to new picture to window's document structure

        if((*docStrucHdl)->pictureHdl != NULL)
          DisposeHandle((Handle) (*docStrucHdl)->pictureHdl);
        (*docStrucHdl)->pictureHdl = (PicHandle) newPictHdl;


        if((*docStrucHdl)->textHdl != NULL)
          DisposeHandle((Handle) (*docStrucHdl)->textHdl);
        (*docStrucHdl)->textHdl = NULL;
      }
      else
        doErrorAlert(eGetScrapSize);
    }
  }

  SetPort(oldPort);
}

// ********************************************************************************* doClearCommand

void  doClearCommand(void)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  GrafPtr            oldPort;
  Rect               portRect;

  windowRef = FrontWindow();
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  GetPort(&oldPort);
  SetPortWindowPort(windowRef);

  if((*docStrucHdl)->textHdl != NULL)
  {
    DisposeHandle((*docStrucHdl)->textHdl);
    (*docStrucHdl)->textHdl = NULL;
  }

  if((*docStrucHdl)->pictureHdl != NULL)
  {
    DisposeHandle((Handle) (*docStrucHdl)->pictureHdl);
    (*docStrucHdl)->pictureHdl = NULL;
  }

  (*docStrucHdl)->selectFlag = false;

  GetWindowPortBounds(windowRef,&portRect);
  EraseRect(&portRect);

  SetPort(oldPort);
}
```

```
// ********************************************************************** doClipboardCommand

void  doClipboardCommand(void)
{
  MenuRef            editMenuRef;
  OSStatus           osError;
  Rect               contentRect = { 254,7,384,603 };
  docStructureHandle docStrucHdl;
  EventTypeSpec      windowEvents[] = { { kEventClassWindow, kEventWindowActivated   },
                                        { kEventClassWindow, kEventWindowDeactivated },
                                        { kEventClassWindow, kEventWindowDrawContent },
                                        { kEventClassWindow, kEventWindowClose       } };
  editMenuRef = GetMenuRef(mEdit);

  if(gClipboardWindowRef == NULL)
  {
    osError = CreateNewWindow(kDocumentWindowClass,kWindowStandardHandlerAttribute |
                              kWindowCloseBoxAttribute,&contentRect,&gClipboardWindowRef);
    if(osError != noErr)
      doErrorAlert(eFailWindow);

    SetWTitle(gClipboardWindowRef,"\pClipboard");
    SetPortWindowPort(gClipboardWindowRef);

    if(!(docStrucHdl = (docStructureHandle) NewHandle(sizeof(docStructure))))
      doErrorAlert(eFailDocStruc);

    SetWRefCon(gClipboardWindowRef,(SInt32) docStrucHdl);
    (*docStrucHdl)->windowType = kClipboardType;

    SetMenuItemText(editMenuRef,iClipboard,"\pHide Clipboard");

    InstallWindowEventHandler(gClipboardWindowRef,
                              NewEventHandlerUPP((EventHandlerProcPtr) clipWindowEventHandler),
                              GetEventTypeCount(windowEvents),windowEvents,0,NULL);

    ShowWindow(gClipboardWindowRef);
    gClipboardShowing = true;
  }
  else
  {
    if(gClipboardShowing)
    {
      HideWindow(gClipboardWindowRef);
      gClipboardShowing = false;
      SetMenuItemText(editMenuRef,iClipboard,"\pShow Clipboard");
    }
    else
    {
      ShowWindow(gClipboardWindowRef);
      gClipboardShowing = true;
      SetMenuItemText(editMenuRef,iClipboard,"\pHide Clipboard");
    }
  }
}

// ********************************************************************** doDrawClipboardWindow

void  doDrawClipboardWindow(void)
{
  GrafPtr          oldPort;
  Rect             theRect, textBoxRect;
  ScrapRef         scrapRef;
  OSStatus         osError;
  ScrapFlavorFlags flavorFlags;
  CFStringRef      stringRef;
  Handle           tempHdl;
  Size             sizeOfPictData = 0, sizeOfTextData = 0;
```

```
RGBColor         blackColour = { 0x0000, 0x0000, 0x0000 };

GetPort(&oldPort);
SetPortWindowPort(gClipboardWindowRef);

GetWindowPortBounds(gClipboardWindowRef,&theRect);
EraseRect(&theRect);

SetRect(&theRect,-1,-1,597,24);
DrawThemeWindowHeader(&theRect,gClipboardWindowRef == FrontWindow());

if(gClipboardWindowRef == FrontWindow())
  TextMode(srcOr);
else
  TextMode(grayishTextOr);

SetRect(&textBoxRect,10,5,120,20);
DrawThemeTextBox(CFSTR("Clipboard Contents:"),kThemeSmallSystemFont,0,true,&textBoxRect,
                 teJustLeft,NULL);

GetCurrentScrap(&scrapRef);

// ..... 'TEXT'

osError = GetScrapFlavorFlags(scrapRef,kScrapFlavorTypeText,&flavorFlags);
if(osError == noErr)
{
  osError = GetScrapFlavorSize(scrapRef,kScrapFlavorTypeText,&sizeOfTextData);
  if(osError == noErr && sizeOfTextData > 0)
  {
    SetRect(&textBoxRect,120,5,597,20);
    DrawThemeTextBox(CFSTR("Text"),kThemeSmallSystemFont,0,true,&textBoxRect,teJustLeft,
                     NULL);

    tempHdl = NewHandle(sizeOfTextData);
    osError = MemError();
    if(osError == memFullErr)
      doErrorAlert(eFailMemory);

    HLock(tempHdl);

    osError = GetScrapFlavorData(scrapRef,kScrapFlavorTypeText,&sizeOfTextData,*tempHdl);
    if(osError != noErr)
      doErrorAlert(eGetScrapData);

    // ..... draw text in clipboard window

    GetWindowPortBounds(gClipboardWindowRef,&theRect);
    theRect.top += 22;
    InsetRect(&theRect,2,2);

    if(sizeOfTextData >= 965)
      sizeOfTextData = 965;
    stringRef =  CFStringCreateWithBytes(NULL,(UInt8 *) *tempHdl,sizeOfTextData,
                                         CFStringGetSystemEncoding(),true);
    DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,true,&theRect,teFlushLeft,NULL);
    if(stringRef != NULL)
      CFRelease(stringRef);

    HUnlock(tempHdl);
    DisposeHandle(tempHdl);
  }
  else
    doErrorAlert(eGetScrapSize);
}

// ..... 'PICT'

else
```

```
  {
    osError = GetScrapFlavorFlags(scrapRef,kScrapFlavorTypePicture,&flavorFlags);
    if(osError == noErr)
    {
      osError = GetScrapFlavorSize(scrapRef,kScrapFlavorTypePicture,&sizeOfPictData);
      if(osError == noErr && sizeOfPictData > 0)
      {
        SetRect(&textBoxRect,120,5,597,20);
        DrawThemeTextBox(CFSTR("Picture"),kThemeSmallSystemFont,0,true,&textBoxRect,
                         teJustLeft,NULL);

        tempHdl = NewHandle(sizeOfPictData);
        osError = MemError();
        if(osError == memFullErr)
          doErrorAlert(eFailMemory);

        HLock(tempHdl);

        osError = GetScrapFlavorData(scrapRef,kScrapFlavorTypePicture,&sizeOfPictData,
                                     *tempHdl);
        if(osError != noErr)
          doErrorAlert(eGetScrapData);

        // ..... draw picture in clipboard window

        theRect = (*(PicHandle) tempHdl)->picFrame;
        OffsetRect(&theRect,-((*(PicHandle) tempHdl)->picFrame.left - 2),
                            -((*(PicHandle) tempHdl)->picFrame.top - 26));
        DrawPicture((PicHandle) tempHdl,&theRect);

        HUnlock(tempHdl);
        DisposeHandle(tempHdl);
      }
      else
        doErrorAlert(eGetScrapSize);
    }
  }

  TextMode(srcOr);
  SetPort(oldPort);
}

// ********************************************************************** doDrawDocumentWindow

void  doDrawDocumentWindow(WindowRef windowRef)
{
  GrafPtr            oldPort;
  docStructureHandle docStrucHdl;
  Rect               destRect;
  CFStringRef        stringRef;

  GetPort(&oldPort);
  SetPortWindowPort(windowRef);

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  if((*docStrucHdl)->textHdl != NULL)
  {
    GetWindowPortBounds(windowRef,&destRect);
    EraseRect(&destRect);
    InsetRect(&destRect,40,40);

    stringRef =  CFStringCreateWithBytes(NULL,(UInt8 *) *(*docStrucHdl)->textHdl,
                                         GetHandleSize((*docStrucHdl)->textHdl),
                                         smSystemScript,false);
    DrawThemeTextBox(stringRef,kThemeSmallSystemFont,0,true,&destRect,teFlushLeft,NULL);
    if(stringRef != NULL)
      CFRelease(stringRef);
```

```
    if((*docStrucHdl)->selectFlag)
      InvertRect(&destRect);
  }
  else if((*docStrucHdl)->pictureHdl != NULL)
  {
    destRect = doSetDestRect(&(*(*docStrucHdl)->pictureHdl)->picFrame,windowRef);
    DrawPicture((*docStrucHdl)->pictureHdl,&destRect);
    if((*docStrucHdl)->selectFlag)
      InvertRect(&destRect);
  }

  SetPort(oldPort);
}

// ************************************************************************** doSetDestRect

Rect  doSetDestRect(Rect *picFrame,WindowRef windowRef)
{
  Rect   destRect, portRect;
  SInt16 diffX, diffY;

  destRect = *picFrame;
  GetWindowPortBounds(windowRef,&portRect);

  OffsetRect(&destRect,-(*picFrame).left,-(*picFrame).top);

  diffX = (portRect.right - portRect.left) - ((*picFrame).right - (*picFrame).left);
  diffY = (portRect.bottom - portRect.top) - ((*picFrame).bottom - (*picFrame).top);

  OffsetRect(&destRect,diffX / 2,diffY / 2);

  return destRect;
}

// ********************************************************************************************
```

Demonstration Program CarbonScrap Comments

When this program is run, the user should choose the Edit menu's Show Clipboard item to open the Clipboard window. The user should then cut, copy, clear and paste the supplied text or picture from/to the two document windows opened by the program, noting the effect on the scrap as displayed in the Clipboard window. (To indicate selection, the text or picture inverts when the user clicks on it with the mouse. The text and picture can be deselected by clicking outside their boundaries.)

The user should also copy text and pictures from another application's window, observing the changes to the contents of the Clipboard window when the demonstration program is brought to the front, and paste that text and those pictures to the document windows. (On Mac OS 8/9, a simple way to get a picture into the scrap is to use Command-Shift-Control-4 to copy an area of the screen to the scrap.)

The program's preferred scrap flavour type is 'TEXT'. Thus, if the scrap contains data in both the 'TEXT' and 'PICT' flavour types, only the 'TEXT' flavour will be used for pastes to the document windows and for display in the Clipboard window. The user can prove this behaviour by copying a picture object containing text in an application such as Adobe Illustrator, bringing the demonstration program to the front, noting the contents of the Clipboard window, pasting to one of the document windows, and noting what is pasted.

The user should note that, when the Clipboard window is open and showing, it will be hidden when the program is sent to the background and shown again when the program is brought to the foreground.

defines

kDocumentType and kClipboardType will enable the program to distinguish between the "document" windows opened by the program and the Clipboard window.

typedefs

Document structures will be attached to the two document windows and the Clipboard window. docStructure is the associated data type. The windowType field will be used to differentiate between the document windows and the Clipboard window.

Global Variables

gClipBoardWindowRef will be assigned a reference to the Clipboard window when it is opened by the user. gClipBoardShowing will keep track of whether the Clipboard window is currently hidden or showing.

appEventHandler

When the kEventAppActivated event type is received, if the Clipboard window has been opened and was showing when the program was sent to the background, ShowWindow is called to show the Clipboard window. When the kEventAppActivated event type is received, if the Clipboard window has been opened and is currently showing, ShowHide is called to hide the Clipboard window. ShowHide, rather than HideWindow is used in this instance to prevent activation of the first document window in the list when the Clipboard window is in front and the application is switched out.

windowEventHandler

windowEventHandler is the handler for the document windows. When the kEventWindowClickContentRegion event type is received, the function doInContent is called.

clipWindowEventHandler

clipWindowEventHandler is the handler for the Clipboard window. The function doDrawClipboardWindow is called when the window receives the event types kEventWindowActivated, kEventWindowDeactivated, kEventWindowDrawContent are received. When the kEventWindowClose event type is received, the Clipboard window is disposed of and the text of its item in the Edit menu is changed.

doAdjustMenus

If the front window is the Clipboard window, the Close item is enabled, otherwise it is disabled. If the document contains a picture and that picture is currently selected, the Cut, Copy, and Clear items are enabled, otherwise they are disabled.

If the scrap contains data of flavour type 'PICT' or flavour type 'TEXT', and the front window is not the Clipboard window, the Paste item is enabled, otherwise it is disabled. In this section, GetCurrentScrap is called to obtain a reference to the current scrap. This reference is then passed in two calls to GetScrapFlavorFlags, which determine whether the scrap contains data of the flavour type 'PICT' and/or flavour type 'TEXT'. If it does, and if the front window is not the Clipboard window, the Paste item is enabled.

doOpenDocumentWindows

doOpenDocumentWindows opens the two document windows, creates document structures for each window, attaches the document structures to the windows and initialises the fields of the document structures.

The textHdl field of the first window's document structure is assigned a handle to a 'TEXT' resource and the textHdl field of the second window's document structure is assigned a handle to a 'PICT' resource.

doCloseWindow

doCloseWindow closes the Clipboard window (the only window that can be closed from within the program).

If the window is the Clipboard window, the window is disposed of, the global variable which contains its reference is set to NULL, the global variable which keeps track of whether the window is showing or hidden is set to false, and the text of the Show/Hide Clipboard menu item is set to "Show Clipboard".

doInContent

doInContent handles mouse-down events in the content region of a document window. If the window contains text or a picture, and if the mouse-down was inside the text or picture, the text or picture is selected. If the window contains a text or picture, and if the mouse-down was outside the text or picture, the text or picture is deselected.

The first two lines get a reference to the front window and a handle to its document structure. If the front window is the Clipboard window, the function returns immediately.

doCutCopyCommand

doCutCopyCommand handles the user's choice of the Cut and Copy items in the Edit menu.

The first two lines get a reference to the front window and a handle to that window's document structure.

If the selectFlag field of the document structure contains false (meaning that the text or picture has not been selected), the function returns immediately. (Note that no check is made as to whether the front window is the Clipboard window because the menu adjustment function disables the Cut and Copy items when the Clipboard window is the front window, meaning that this function can never be called when the Clipboard window is in front.)

ClearCurrentScrap attempts to clear the current scrap. (This call should always be made immediately the user chooses Cut or Copy.) If the call is successful, GetCurrentScrap then gets a reference to the scrap.

If the selected item is text, GetHandleSize gets the size of the text from the window's document structure. (In a real application, code which gets the size of the selection would appear here.) PutScrapFlavor copies the "selected" text to the scrap. If the call to PutScrapFlavor is not successful, an alert is displayed to advise the user of the error.

If the selected item is a picture, GetHandleSize gets the size of the picture from the window's document structure. PutScrapFlavor copies the selected picture to the scrap. If the call to PutScrapFlavor is not successful, an alert is displayed to advise the user of the error.

If the menu choice was the Cut item, additional action is taken. Preparatory to a call to EraseRect, the current graphics port is saved and the front window's port is made the current port. DisposeHandle is called to dispose of the text or picture and the document structure's textHdl or pictureHdl field, and selectFlag field, are set to NULL and false respectively. EraseRect then erases the port rectangle. (In a real application, the action taken in this block would be to remove the selected text or picture from the document.)

Finally, and importantly, if the Clipboard window has previously been opened by the user, doDrawClipboardWindow is called to draw the current contents of the scrap in the Clipboard window.

doPasteCommand

doPasteCommand handles the user's choice of the Paste item from the Edit menu. Note that no check is made as to whether the front window is the Clipboard window because the menu adjustment function disables the Paste item when the Clipboard window is the front window, meaning that this function can never be called when the Clipboard window is in front.

GetCurrentScrap gets a reference to the scrap.

The first call to GetScrapFlavorFlags determines whether the scrap contains data of flavour type 'TEXT'. If so, GetScrapFlavorSize is called to get the size of the 'TEXT' data. NewHandle creates a relocatable

block of a size equivalent to the 'TEXT' data. GetScrapFlavorData is called to copy the 'TEXT' data in the scrap to this block.

TETextBox or DrawThemeTextBox is called to draw the text in a rectangle equal to the port rectangle minus 40 pixels all round. If the textHdl field of the window's document structure does not contain NULL, the associated block is disposed of, following which the handle to the block containing the new 'TEXT' data is then assigned to the textHdl field. (In a real application, this block would copy the text into the document at the insertion point.) (The last three lines in this section simply ensure that, if the window's "document" contains text, it cannot also contain a picture. This is to prevent a picture overdrawing the text when the window contents are updated.)

If the scrap does not contain data of flavour type 'TEXT', GetScrapFlavorFlags is called again to determine whether the scrap contains data of flavour type 'PICT'. If it does, much the same procedure is followed, except that rectangle in which the picture is drawn is extracted from the 'PICT' data itself and adjusted to the middle of the window via a call to the function doSetDestRec.

It is emphasized that the scrap is only checked for flavour type 'PICT' if the scrap does not contain flavour type 'TEXT'. Thus, if both flavours exist in the scrap, only the 'TEXT' flavour will be used to draw the Clipboard.

doClearCommand

doClearCommand handles the user's choice of the Clear item in the Edit menu.

Note that no check is made as to whether the front window is the Clipboard window because the menu adjustment function disables the Clear item when the Clipboard window is the front window.

If the front window's document structure indicates that the window contains text or a picture, the block containing the TextEdit structure or Picture structure is disposed of and the relevant field of the document structure is set to NULL. In addition, the selectFlag field of the document structure is set to false and the window's port rectangle is erased.

doClipboardCommand

doClipboardCommand handles the user's choice of the Show/Hide Clipboard item in the Edit menu.

The first line gets a reference to the Edit menu. This will be required in order to toggle the Show/Hide Clipboard item's text between Show Clipboard and Hide Clipboard.

The if statement checks whether the Clipboard window has been created. If not, the Clipboard window is created by the call to GetNewCWindow, a document structure is created and attached to the window, the windowType field of the document structure is set to indicate that the window is of the Clipboard type, the Show/Hide Clipboard menu item text is set, the window's special window event handler is installed, the window is shown, and a global variable which keeps track of whether the Clipboard window is currently showing or hidden is set to true.

If the Clipboard window has previously been created, and if the window is currently showing, the window is hidden, the Clipboard-showing flag is set to false, and the Show/Hide Clipboard item's text is set to "Show Clipboard". If the window is not currently showing, the window is made visible, the Clipboard-showing flag is set to true, and the Show/Hide Clipboard item's text is set to "Hide Clipboard".

doDrawClipboardWindow

doDrawClipboardWindow draws the contents of the scrap in the Clipboard window. It supports the drawing of both 'PICT' and 'TEXT' flavour type data.

The first four lines save the current graphics port, make the Clipboard window's graphics port the current graphics port and erase the window's content region.

DrawThemeWindowHeader draws a window header in the top of the window. Text describing the type of data in the scrap will be drawn in this header.

The text mode for text drawing is set at the next four lines, following which "Clipboard Contents:" is drawn in the header.

The code for getting a reference to the current scrap, checking for the 'TEXT' and 'PICT' flavour types, getting the flavour size, getting the flavour data, and drawing the text and picture in the window is much the same as in the function doPasteCommand. The differences are: the rectangle in which the text is drawn is the port rectangle minus two pixels all round and with the top further adjusted downwards by the height of the window header; the left/top of the rectangle in which the picture is drawn is two pixels inside the

left side of the content region and two pixels below the window header respectively; the words "Text" or "Picture" are drawn in the window header as appropriate.

Note that, as was the case in the function doPasteCommand, the scrap is only checked for flavour type 'PICT' if the scrap does not contain flavour type 'TEXT'. Thus, if both flavours exist in the scrap, only the 'TEXT' flavour will be used to draw the Clipboard.

doDrawDocumentWindow

doDrawDocumentWindow draws the text or picture belonging to that window in the window. It is called when the kEventWindowDrawContent event type is received for the window.