16 Version 1.1

SCRAP

Includes Demonstration Program Scrap

The Scrap Manager and the Desk Scrap

Introduction

For each open application, the Scrap Manager maintains a storage area to hold the last data cut or copied by the user. This area is called the **scrap** or, sometimes, the **desk scrap**. The desk scrap can reside in memory or on disk. All applications which support cut, copy, and paste operations write data to, and read data from, the desk scrap. Typically, that data relates to text, graphics, sounds, or movies.

Your application specifies the format, or formats, in which data is written to, and read from, the desk scrap. Your application should write that data using the so-called **standard formats** (in addition to any other format it might specify), since this ensures that a user can copy and paste data between documents created by your application and other applications as well as within and between documents created by your application. The ultimate aim is to allow the user to:

- Copy and paste data within a document created by your application.
- Copy and paste data between different documents created by your application.
- Copy and paste data between documents created by your application and documents created by other applications.

Scrap Data Formats

Standard Formats

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

- `'TEXT'`, that is, a series of ASCII characters.
- `'PICT'`, that is, a QuickDraw picture.

Optional Formats

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

- `'snd '`, that is, a series of bytes which define a sound, and which have the same format as a `'snd '` resource.
- `'movv'`, that is, a series of bytes which define a movie, and which have the same format as a `'movv'` resource.

- `'styl'`, that is, a series of bytes which have the same format as a TextEdit `'styl'` resource, and which describe styled text data.

Private Formats

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

Location of the Desk Scrap and Getting Information About the Scrap

Location of the Desk Scrap

System software allocates space in each application's heap for the desk scrap and allocates a handle to reference the scrap. The system global variable `Scraphandle` contains a handle to the desk scrap of the current process.

When system software launches an application, it copies the data from the scrap of the previously active application into the application heap of the newly active application. If the scrap is too large to fit in the application's application heap, system software copies the scrap to disk and sets the value of the handle to the scrap in the application's heap to `NULL` to indicate that the scrap is on disk.

Getting Information About the Desk Scrap

To get information about the scrap, you can use `InfoScrap`, which returns a pointer to a **scrap information record**, which is defined by the data type `ScrapStuff`. The information in the scrap information record includes:

- The size, in bytes, of the scrap.
- A handle to the scrap (if it is in memory).
- The location of the scrap (memory or disk).
- The filename of the scrap when it is on disk.

Using the Desk Scrap - 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, copy the data in the selection range to the desk scrap and remove the data from the document.
Copy	If there is a current selection range, copy the data in the selection range to the desk scrap.
Paste	Read the desk scrap and insert the data (if any) at the insertion point, replacing any current selection.[1]

Note that, 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 desk scrap.

Cut and **Copy** — Putting Data in the Scrap

A typical approach to implementing the **Cut** and **Copy** commands is as follows:

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

[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.

- If the frontmost window is a document window:
    - Call an application-defined function which determines whether the current selection contains text or whether it contains graphics.
    - Get a pointer to the selection range data and get the selection length.
    - Call `ZeroScrap` to purge the current contents of the desk scrap.
    - Call `PutScrap` to write the data to the scrap, specifying `'TEXT'` or `'PICT'`, as appropriate, as the format type.
    - If the command was the **Cut** command, delete the selection from the current document.
- If the frontmost window is a dialog box, use the Dialog Manager routines `DialogCut` or `DialogCopy`, as appropriate, to write the selected data to the scrap.

Paste - Getting Data From the Scrap

When the user chooses the **Paste** command, your application should paste the data last cut or copied by the user. Your application gets the data to paste by reading the data from the desk scrap.

When you read the data from the scrap, your application should request the data in the application's preferred format type. If your application determines that that format does not exist in the scrap, it should then request the data in another format. If your application does not have a preferred format type, it should read each format 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 format types currently in the scrap into the scrap format requested by your application. The Translation Manager looks in the Extensions folder 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 format type.

A typical approach, for an application that prefers a data format other than `'TEXT'` or `'PICT'` as its first preference, is as follows:

- Determine whether the frontmost window is a document window or a dialog box.
- If the frontmost window is a document window:
    - Call `GetScrap` to search the scrap for the preferred format type. (If you specify a `NULL` handle as the location to which to return the data, `GetScrap` does not return the data but does return as its function result the number of bytes (if any) of data in the specified format that exists in the scrap. Thus, if `GetScrap` returns a non-positive value, data of that format type does not exist.)
    - If data of the specified format does exist, allocate a handle to hold the data from the scrap and call `GetScrap` again to read in the data in that format. (`GetScrap` automatically resizes the handle passed to it to the required size.)
    - If the scrap does not contain data of the preferred format type, repeat the above process specifying `'TEXT'` as the format type in the calls to `GetScrap`. If this is not successful, repeat the process again specifying `'PICT'` as the format type.
    - Paste the data to the current document.
- If the frontmost window is a dialog box, use the Dialog Manager routine `DialogPaste` to paste the text from the scrap in the dialog.

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'` format to the desk scrap, and the destination application has pasted the data to its document in that format.

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

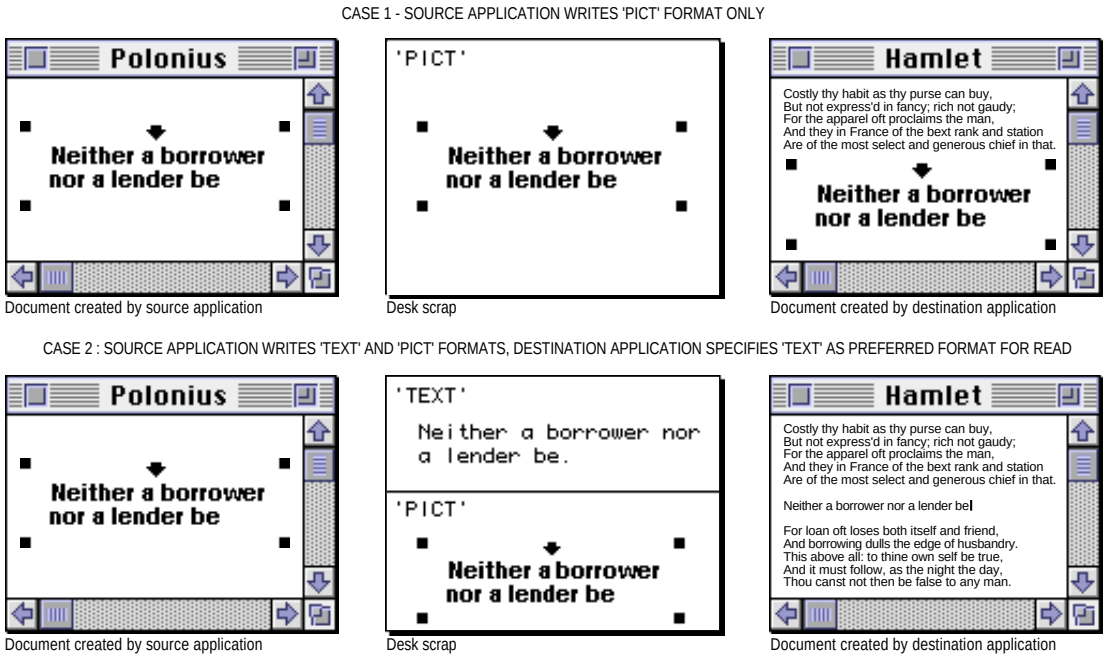

FIG 1 - SPECIFYING FORMATS TO WRITE TO AND READ FROM THE DESK SCRAP

The Clipboard

The **Clipboard** refers to what the user views as residing in the scrap. Your application can provide a **Show Clipboard** command which, when chosen, should show a window which displays the current contents of the desk 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 multiple scrap format types can reside in the desk scrap, applications which support a Clipboard window typically display the data in one format only.

Transferring the Desk Scrap to Disk

Although 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.

Private Scrap

As an alternative to writing to and reading from the desk 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.

In addition, an application which uses a private scrap must take the following actions on receipt of suspend and resume events:

- **Suspend Event.** On receipt of a suspend event, the data from the private scrap must be copied to the desk scrap. If your application supports the **Show Clipboard** command, the Clipboard window must be hidden if it is currently showing (because the contents of the scrap may change while the application yields time to another application).

- **Resume Event.** On receipt of a resume event, your application must determine if the data in the desk scrap has changed since the previous suspend event and, if so, copy the data from the desk scrap to its private scrap either immediately or when the user next chooses the **Paste** command. In addition, if your application supports the **Show Clipboard** command, and if the data in the desk scrap has changed, your application must update the contents of the Clipboard window.

Note that, when the contents of the desk scrap have changed since the last suspend event, system software sets the `convertClipboardFlag` bit in the `message` field of the resume event record.

The process of copying data between an application's document, an application's private scrap, and the desk scrap in response to suspend and resume events is shown diagrammatically at Fig 2.

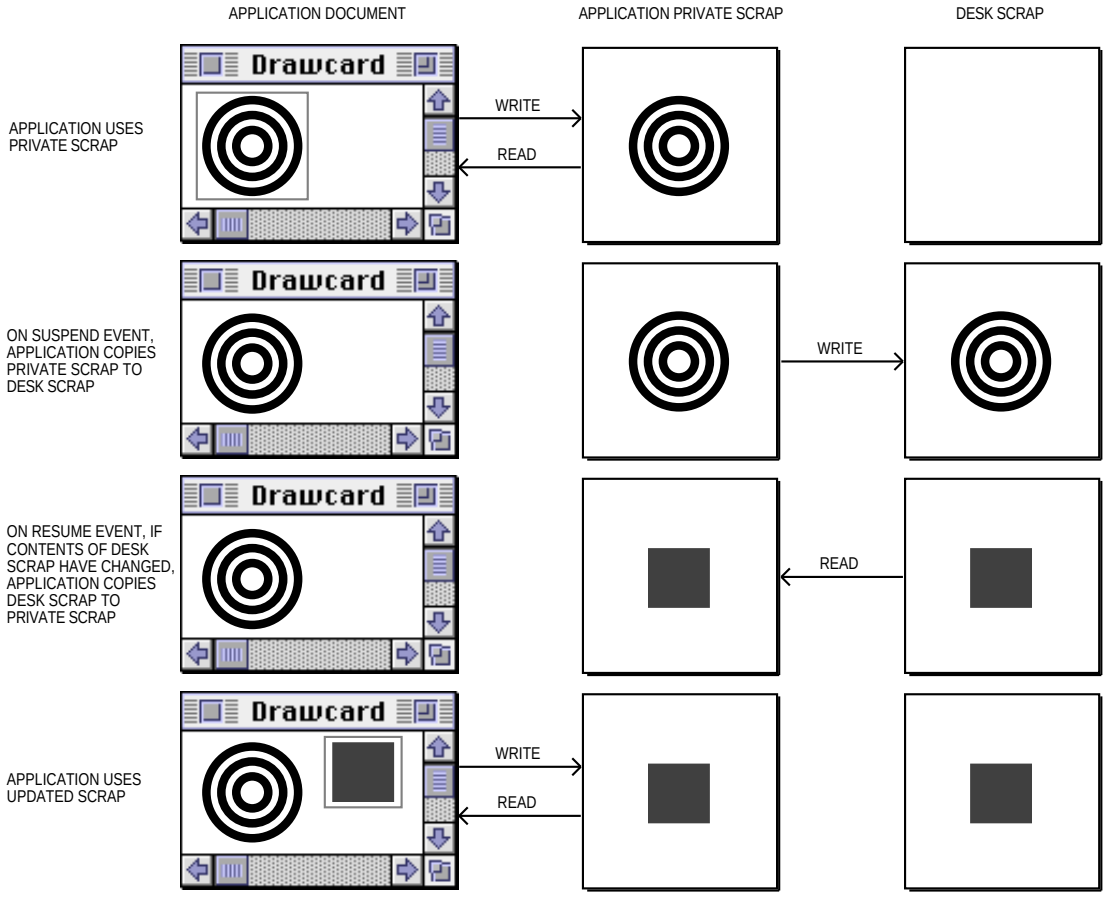

FIG 2 - USING A PRIVATE SCRAP

Copying Data Between Private Scrap and the Desk Scrap

A typical approach to copying data between the private scrap and the desk scrap is as follows:

- **Resume Event.** When a resume event is received, and a check indicates that the contents of the desk scrap have changed since the last suspend event:
    - Call `GetScrap`, with `NULL` passed as the `destHandle` parameter, to determine if the scrap contains data in the `'PICT'` format type. If data of that format type exists:
        - Allocate a handle to hold the data from the scrap and call `GetScrap` again to read in the data.
        - Call an application-defined function to copy the data to the private scrap.
        - Dispose of the handle.
    - If data of the `'PICT'` format type does not exist in the scrap, repeat this process specifying `'TEXT'` as the data format type.
- **Suspend Event.** When a suspend event is received:
    - Call an application-defined function which determines if there is any data in the private scrap. If there is data in the private scrap, call `ZeroScrap` to empty the desk scrap.
    - Create a non-relocatable block to receive the private scrap data.
    - For each appropriate data format type:
        - Determine if data in that format exists in the private scrap.
        - If data in that format type exists in the private scrap, call an application-defined function which gets the data from the private scrap into the nonrelocatable block. Then call `PutScrap` to copy the data from the nonrelocatable block to the scrap.
    - Dispose of the nonrelocatable block.

TextEdit, Dialog Boxes, and Scrap

TextEdit and Scrap

TextEdit is a collection of routines 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:

- `PutScrap` is not used and the special TextEdit routines `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 desk scrap.
- `GetScrap` is not used and the special TextEdit routines `TEPaste`, `TEStylePaste`, and `TEFromScrap` are used in the processes of pasting text from the TextEdit private scrap and copying text from the desk scrap to the TextEdit private scrap.

Chapter 17 — Text and TextEdit describes TextEdit, including the TextEdit private scrap and the TextEdit scrap-related routines.

Dialog Boxes and Scrap

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

You can use the Dialog Manager to handle most editing operations within dialog boxes. The Dialog Manager routines `DialogCut`, `DialogCopy`, and `DialogPaste` may be used to implement **Cut**, **Copy** and **Paste** commands within editable text items in dialog boxes. (See the demonstration program at Chapter 6 — Dialogs and Alerts.)

TextEdit's private scrap facilitates the copying and pasting of data between dialog boxes. However, your application must ensure that the user can copy and paste data between your application's dialog boxes 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` routines and the Dialog Manager's `DialogCut`, `DialogCopy`, and `DialogPaste` routines all use TextEdit's private scrap.

If your application does not use TextEdit for text handling within its document windows, and if it uses a private scrap, then, when the user activates a dialog box, you should copy any data in your private scrap to TextEdit's private scrap. Also, when a document window becomes active, and there is data in TextEdit's private scrap, that data should be copied to your application's private scrap (or to the desk scrap if your application does not use a private scrap).

Similarly, before displaying the Standard File Package's save dialog box, your application should copy any text data in its private scrap to the desk scrap. The Standard File Package reads the data from the desk scrap whenever the user chooses an editing operation and a standard file dialog box is active. Accordingly, your application needs to put the text data (if any) from the last cut or copy in the desk scrap before calling `StandardPutFile`.

Main Scrap Manager Data Types and Routines

Data Types

Scrap Information Record

```
struct ScrapStuff
{
  long       scrapSize;    // Size of scrap in bytes.
  Handle     scrapHandle;  // Handle to scrap.
  short      scrapCount;   // Indicates whether contents of scrap have changed.
  short      scrapState;   // Indicates state and location of scrap
  StringPtr  scrapName;    // Filename of scrap.
};

typedef struct ScrapStuff ScrapStuff;
typedef ScrapStuff *PScrapStuff,*ScrapStuffPtr;
```

Routines

Getting Information About the Scrap

```
ScrapStuffPtr  InfoScrap(void);
```

Writing Information to the Scrap

```
long           ZeroScrap(void);
long           PutScrap(long length,ResType theType,void *source);
```

Reading Information From the Scrap

```
long           GetScrap(Handle hDest,ResType theType,long *offset);
```

Transferring the Scrap Between Memory and Disk

```
long          UnloadScrap(void);
long          LoadScrap(void);
```

Demonstration Program

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// Scrap.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
//
// This program utilises the desk scrap and Scrap Manager routines to allow the user to:
//
// •  Cut, copy and clear pictures from, and paste pictures to, two windows opened by the
//    program.
//
// •  Paste pictures cut or copied from another application to the two windows opened
//    by the program.
//
// •  Open and close a Clipboard window, in which the current contents of the desk scrap
//    are displayed.
//
// In addition to the pictures cut and copied from either the program's windows or from
// another application's windows, the Clipboard window will display text copied to the
// desk scrap as a result of text cut and copy operations in another application.  The
// program, however, does not support the pasting of this text to documents displayed in
// the program's windows.  (The demonstration program at Chapter 17 — Text and TextEdit
// shows how to cut, copy and paste text from and to a TextEdit edit record using the
// desk scrap.)
//
// The program utilises the following resources:
//
// •  An 'MBAR' resource, and 'MENU' resources for Apple, File, and Edit menus (preload,
//    non-purgeable).
//
// •  Three 'WIND' resources (purgeable) (initially visible), two for the program's main
//    windows and one for the Clipboard window.
//
// •  A 'PICT' resource (non-purgeable) containing a picture which may be cut, copied,
//    and pasted between the windows.
//
// •  An 'ALRT' resource (purgeable) and associated 'DITL' resource (purgeable) for use
//    by an error Alert.
//
// •  A 'STR#' resource (purgeable) containing strings to be displayed in the error
//    Alert.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGSwitch, and
//    is32BitCompatible flags set.
//
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

// ………………………………………………………………………………………………………………………………………………………………… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Devices.h>
#include <Scrap.h>

// …………………………………………………………………………………………………………………………………………………………………… defines

#define mApple          128
#define  iAbout         1
#define mFile           129
#define  iClose         4
#define  iQuit          11
#define mEdit           130
#define  iCut           3
#define  iCopy          4
#define  iPaste         5
```

```
#define  iClear           6
#define  iClipboard       9
#define rMenubar          128
#define rWindow           128
#define rClipboardWindow  130
#define rPicture          128
#define rAlertBox         128
#define rErrorStrings     128
#define  eFailMenu        1
#define  eFailWindow      2
#define  eFailDocRec      3
#define  eZeroScrap       4
#define  ePutScrap        5
#define  eNoPictInScrap   6
#define kDocumentType     1
#define kClipboardType    2

// ………………………………………………………………………………………………………………………………………………………… typedefs

typedef struct
{
  PicHandle pictureHdl;
  Boolean   selectFlag;
  SInt16    windowType;
} DocRec, **DocRecHandle;

// …………………………………………………………………………………………………………………………………………… global variables

Boolean   gDone;
Boolean   gInBackground;
WindowPtr gWindowPtrs[2];
WindowPtr gClipboardWindowPtr = NULL;
Boolean   gClipboardShowing   = false;
Pattern   gMarqueePattern     = { 0x1F, 0x3E, 0x7C, 0xF8, 0xF1, 0xE3, 0xC7, 0x8F };

// ………………………………………………………………………………………………………………………………………… function prototypes

void  main                  (void);
void  doInitManagers        (void);
void  doIdle                (void);
void  doEvents              (EventRecord *);
void  doMouseDown           (EventRecord *);
void  doUpdate              (EventRecord *);
void  doOSEvent             (EventRecord *);
void  doAdjustMenus         (void);
void  doMenuChoice          (SInt32);
void  doEditMenu            (SInt16);
void  doCloseWindow         (void);
void  doErrorAlert          (SInt16);
void  doInContent           (Point);
void  doCutCopyCommand      (Boolean);
void  doPasteCommand        (void);
void  doClearCommand        (void);
void  doClipboardCommand    (void);
void  doDrawClipboardWindow (void);
void  doDrawPictureWindow   (WindowPtr);
void  doOpenWindows         (void);
Rect  doSetDestRect         (Rect *,WindowPtr);

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ main

void  main(void)
{
  Handle        menubarHdl;
  MenuHandle    menuHdl;
  Boolean       gotEvent;
  EventRecord   eventRec;

  // ……………………………………………………………………………………………………………………………………… initialise managers

  doInitManagers();

  // ……………………………………………………………………………………………………………………………… set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    doErrorAlert(eFailMenu);
```

```
145     SetMenuBar(menubarHdl);
146     DrawMenuBar();
147
148     menuHdl = GetMenuHandle(mApple);
149     if(menuHdl == NULL)
150       doErrorAlert(eFailMenu);
151     else
152       AppendResMenu(menuHdl,'DRVR');
153
154     // …… open windows
155
156     doOpenWindows();
157
158     // …… enter eventLoop
159
160     gDone = false;
161
162     while(!gDone)
163     {
164       gotEvent = WaitNextEvent(everyEvent,&eventRec,2,NULL);
165
166       if(gotEvent)
167         doEvents(&eventRec);
168       else
169         doIdle();
170     }
171   }
172
173   // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInitManagers
174
175   void  doInitManagers(void)
176   {
177     MaxApplZone();
178     MoreMasters();
179
180     InitGraf(&qd.thePort);
181     InitFonts();
182     InitWindows();
183     InitMenus();
184     TEInit();
185     InitDialogs(NULL);
186
187     InitCursor();
188     FlushEvents(everyEvent,0);
189   }
190
191   // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doIdle
192
193   void  doIdle(void)
194   {
195     WindowPtr     windowPtr;
196     DocRecHandle  docRecHdl;
197     GrafPtr       oldPort;
198     Rect          marqueeRect;
199     UInt8         lastByte;
200     SInt16        a;
201
202     windowPtr = FrontWindow();
203     docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
204
205     if(((*docRecHdl)->windowType == kClipboardType) || (*docRecHdl)->selectFlag == false)
206     {
207       return;
208     }
209     else
210     {
211       GetPort(&oldPort);
212       SetPort(windowPtr);
213
214       marqueeRect = doSetDestRect(&(*(*docRecHdl)->pictureHdl)->picFrame,windowPtr);
215       InsetRect(&marqueeRect,-2,-2);
216
217       lastByte = gMarqueePattern.pat[7];
218       for(a=7;a>0;--a)
219         gMarqueePattern.pat[a] = gMarqueePattern.pat[a - 1];
220       gMarqueePattern.pat[0] = lastByte;
221
```

```
    PenPat(&gMarqueePattern);
    FrameRect(&marqueeRect);

    SetPort(oldPort);
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doEvents

void  doEvents(EventRecord *eventRecPtr)
{
  UInt8 charCode;

  switch(eventRecPtr->what)
  {
    case mouseDown:
      doMouseDown(eventRecPtr);
      break;

    case keyDown:
    case autoKey:
      charCode = eventRecPtr->message & charCodeMask;
      if((eventRecPtr->modifiers & cmdKey) != 0)
      {
        doAdjustMenus();
        doMenuChoice(MenuKey(charCode));
      }
      break;

    case updateEvt:
      doUpdate(eventRecPtr);
      break;

    case osEvt:
      doOSEvent(eventRecPtr);
      HiliteMenu(0);
      break;
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMouseDown

void  doMouseDown(EventRecord *eventRecPtr)
{
  SInt16    partCode;
  WindowPtr windowPtr;

  partCode = FindWindow(eventRecPtr->where,&windowPtr);

  switch(partCode)
  {
    case inMenuBar:
      doAdjustMenus();
      doMenuChoice(MenuSelect(eventRecPtr->where));
      break;

    case inSysWindow:
      SystemClick(eventRecPtr,windowPtr);
      break;

    case inContent:
      if(windowPtr != FrontWindow())
        SelectWindow(windowPtr);
      else
        doInContent(eventRecPtr->where);
      break;

    case inDrag:
      DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
      break;

    case inGoAway:
      if(TrackGoAway(windowPtr,eventRecPtr->where) == true)
        doCloseWindow();
      break;
  }
}
```

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doUpdate

void  doUpdate(EventRecord *eventRecPtr)
{
  WindowPtr     windowPtr;
  DocRecHandle docRecHdl;
  SInt32        windowType;

  windowPtr = (WindowPtr) eventRecPtr->message;
  docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
  windowType = (*docRecHdl)->windowType;

  BeginUpdate(windowPtr);

  if(windowType == kDocumentType)
  {
    if((*docRecHdl)->pictureHdl != NULL)
      doDrawPictureWindow(windowPtr);
  }
  else if(windowType == kClipboardType)
    doDrawClipboardWindow();

  EndUpdate(windowPtr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOSEvent

void  doOSEvent(EventRecord *eventRecPtr)
{
  WindowPtr windowPtr;

  windowPtr = FrontWindow();

  switch((eventRecPtr->message >> 24) & 0x000000FF)
  {
    case suspendResumeMessage:
      gInBackground = (eventRecPtr->message & resumeFlag) == 0;
      if(gClipboardWindowPtr && gClipboardShowing)
      {
        if(gInBackground)
          HideWindow(gClipboardWindowPtr);
        else
          ShowWindow(gClipboardWindowPtr);
      }
      break;

    case mouseMovedMessage:
      break;
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doAdjustMenus

void  doAdjustMenus(void)
{
  MenuHandle    fileMenuHdl, editMenuHdl;
  DocRecHandle  docRecHdl;
  SInt32        scrapOffset;

  fileMenuHdl = GetMenuHandle(mFile);
  editMenuHdl = GetMenuHandle(mEdit);

  docRecHdl = (DocRecHandle) GetWRefCon(FrontWindow());

  if((*docRecHdl)->windowType == kClipboardType)
    EnableItem(fileMenuHdl,iClose);
  else
    DisableItem(fileMenuHdl,iClose);

  if((*docRecHdl)->pictureHdl && (*docRecHdl)->selectFlag)
  {
    EnableItem(editMenuHdl,iCut);
    EnableItem(editMenuHdl,iCopy);
    EnableItem(editMenuHdl,iClear);
  }
  else
```

```
  {
    DisableItem(editMenuHdl,iCut);
    DisableItem(editMenuHdl,iCopy);
    DisableItem(editMenuHdl,iClear);
  }

  if(GetScrap(NULL,'PICT',&scrapOffset)  && (*docRecHdl)->windowType != kClipboardType)
    EnableItem(editMenuHdl,iPaste);
  else
    DisableItem(editMenuHdl,iPaste);

  DrawMenuBar();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
  SInt16  menuID, menuItem;
  Str255  itemName;
  SInt16  daDriverRefNum;

  menuID = HiWord(menuChoice);
  menuItem = LoWord(menuChoice);

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mApple:
      if(menuItem == iAbout)
        SysBeep(10);
      else
      {
        GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
        daDriverRefNum = OpenDeskAcc(itemName);
      }
      break;

    case mFile:
      if(menuItem == iClose)
        doCloseWindow();
      else if(menuItem == iQuit)
        gDone = true;
      break;

    case mEdit:
      doEditMenu(menuItem);
      break;
  }

  HiliteMenu(0);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doEditMenu

void  doEditMenu(SInt16 menuItem)
{
  switch(menuItem)
  {
    case iCut:
      doCutCopyCommand(true);
      break;

    case iCopy:
      doCutCopyCommand(false);
      break;

    case iPaste:
      doPasteCommand();
      break;

    case iClear:
      doClearCommand();
      break;
```

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCloseWindow

void  doCloseWindow(void)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  MenuHandle    editMenuHdl;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);

  if((*docRecHdl)->windowType == kClipboardType)
  {
    DisposeWindow(windowPtr);
    gClipboardWindowPtr = NULL;
    gClipboardShowing = false;
    editMenuHdl = GetMenu(mEdit);
    SetMenuItemText(editMenuHdl,iClipboard,"\pShow Clipboard");
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doErrorAlert

void  doErrorAlert(SInt16 errorCode)
{
  Str255  errorString;

  GetIndString(errorString,rErrorStrings,errorCode);
  ParamText(errorString,NULL,NULL,NULL);

  if(errorCode < ePutScrap)
  {
    StopAlert(rAlertBox,NULL);
    ExitToShell();
  }
  else
    CautionAlert(rAlertBox,NULL);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInContent

void  doInContent(Point mouseXY)
{
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;
  GrafPtr       oldPort;
  Rect          pictRect;

  windowPtr = FrontWindow();
  docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);

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

  GetPort(&oldPort);
  SetPort(windowPtr);

  if((*docRecHdl)->pictureHdl != NULL)
  {
    pictRect = doSetDestRect(&(*(*docRecHdl)->pictureHdl)->picFrame,windowPtr);
    InsetRect(&pictRect,-2,-2);

    GlobalToLocal(&mouseXY);

    if(PtInRect(mouseXY,&pictRect))
    {
      (*docRecHdl)->selectFlag = true;
    }
    else
    {
      (*docRecHdl)->selectFlag = false;
```

```
530
531         PenPat(&qd.black);
532         ForeColor(whiteColor);
533         FrameRect(&pictRect);
534         ForeColor(blackColor);
535       }
536     }
537
538     SetPort(oldPort);
539   }
540
541   // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCutCopyCommand
542
543   void  doCutCopyCommand(Boolean cutFlag)
544   {
545     WindowPtr     windowPtr;
546     DocRecHandle  docRecHdl;
547     Size          dataLength;
548     SInt32        errorCode;
549     GrafPtr       oldPort;
550
551     windowPtr = FrontWindow();
552     docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
553
554     if((*docRecHdl)->selectFlag == false)
555       return;
556
557     if(ZeroScrap() == noErr)
558     {
559       dataLength = GetHandleSize((Handle) (*docRecHdl)->pictureHdl);
560       HLock((Handle) (*docRecHdl)->pictureHdl);
561
562       errorCode = PutScrap((SInt32) dataLength,'PICT',*((Handle) (*docRecHdl)->pictureHdl));
563       if(errorCode != noErr)
564         doErrorAlert(ePutScrap);
565
566       HUnlock((Handle) (*docRecHdl)->pictureHdl);
567     }
568     else
569       doErrorAlert(eZeroScrap);
570
571     if(cutFlag)
572     {
573       GetPort(&oldPort);
574       SetPort(windowPtr);
575
576       DisposeHandle((Handle) (*docRecHdl)->pictureHdl);
577       (*docRecHdl)->pictureHdl = NULL;
578       (*docRecHdl)->selectFlag = false;
579       EraseRect(&windowPtr->portRect);
580
581       SetPort(oldPort);
582     }
583
584     if(gClipboardWindowPtr != NULL)
585       doDrawClipboardWindow();
586   }
587
588   // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doPasteCommand
589
590   void  doPasteCommand(void)
591   {
592     WindowPtr     windowPtr;
593     DocRecHandle  docRecHdl;
594     GrafPtr       oldPort;
595     SInt32        sizeOfPictData, scrapOffset;
596     Handle        tempHdl;
597     Rect          destRect;
598
599     windowPtr = FrontWindow();
600     docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
601
602     GetPort(&oldPort);
603     SetPort(windowPtr);
604
605     sizeOfPictData = GetScrap(NULL,'PICT',&scrapOffset);
606     if(sizeOfPictData > 0)
```

```
607     {
608       tempHdl = NewHandle((Size) sizeOfPictData);
609       HLock(tempHdl);
610
611       sizeOfPictData = GetScrap(tempHdl,'PICT',&scrapOffset);
612
613       EraseRect(&windowPtr->portRect);
614       (*docRecHdl)->selectFlag = false;
615       destRect = doSetDestRect(&(*(PicHandle) tempHdl)->picFrame,windowPtr);
616
617       DrawPicture((PicHandle) tempHdl,&destRect);
618
619       if((*docRecHdl)->pictureHdl != NULL)
620         DisposeHandle((Handle) (*docRecHdl)->pictureHdl);
621
622       (*docRecHdl)->pictureHdl = (PicHandle) NewHandle((Size) sizeOfPictData);
623       BlockMoveData(*tempHdl,*((*docRecHdl)->pictureHdl),(Size) sizeOfPictData);
624
625       HUnlock(tempHdl);
626       DisposeHandle(tempHdl);
627     }
628
629     SetPort(oldPort);
630   }
631
632   // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doClearCommand
633
634   void  doClearCommand(void)
635   {
636     WindowPtr     windowPtr;
637     DocRecHandle  docRecHdl;
638     GrafPtr       oldPort;
639
640     windowPtr = FrontWindow();
641     docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
642
643     GetPort(&oldPort);
644     SetPort(windowPtr);
645
646     DisposeHandle((Handle) (*docRecHdl)->pictureHdl);
647     (*docRecHdl)->pictureHdl = NULL;
648     (*docRecHdl)->selectFlag = false;
649     EraseRect(&windowPtr->portRect);
650
651     SetPort(oldPort);
652   }
653
654   // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doClipboardCommand
655
656   void  doClipboardCommand(void)
657   {
658     MenuHandle    editMenuHdl;
659     DocRecHandle  docRecHdl;
660
661     editMenuHdl = GetMenu(mEdit);
662
663     if(gClipboardWindowPtr == NULL)
664     {
665       if(!(gClipboardWindowPtr = GetNewWindow(rClipboardWindow,NULL,(WindowPtr)-1)))
666         doErrorAlert(eFailWindow);
667       if(!(docRecHdl = (DocRecHandle) NewHandle(sizeof(DocRec))))
668         doErrorAlert(eFailDocRec);
669       SetWRefCon(gClipboardWindowPtr,(SInt32) docRecHdl);
670       (*docRecHdl)->windowType = kClipboardType;
671
672       gClipboardShowing = true;
673
674       SetMenuItemText(editMenuHdl,iClipboard,"\pHide Clipboard");
675     }
676     else
677     {
678       if(gClipboardShowing)
679       {
680         HideWindow(gClipboardWindowPtr);
681         gClipboardShowing = false;
682         SetMenuItemText(editMenuHdl,iClipboard,"\pShow Clipboard");
683       }
```

```
      else
      {
        ShowWindow(gClipboardWindowPtr);
        gClipboardShowing = true;
        SetMenuItemText(editMenuHdl,iClipboard,"\pHide Clipboard");
      }
    }
  }

  // ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDrawClipboardWindow

  void  doDrawClipboardWindow(void)
  {
    GrafPtr oldPort;
    SInt32  sizeOfPictData, sizeOfTextData, scrapOffset;
    Handle  tempHdl;
    Rect    destRect;

    GetPort(&oldPort);
    SetPort(gClipboardWindowPtr);

    EraseRect(&gClipboardWindowPtr->portRect);

    MoveTo(0,18);
    LineTo(505,18);
    MoveTo(0,20);
    LineTo(505,20);

    TextFont(applFont);
    TextSize(9);
    MoveTo(4,13);
    DrawString("\pClipboard contents:");

    sizeOfPictData = GetScrap(NULL,'PICT',&scrapOffset);
    if(sizeOfPictData > 0)
    {
      MoveTo(95,13);
      DrawString("\ppicture");

      tempHdl = NewHandle((Size) sizeOfPictData);
      HLock(tempHdl);

      sizeOfPictData = GetScrap(tempHdl,'PICT',&scrapOffset);

      destRect = (*(PicHandle) tempHdl)->picFrame;
      OffsetRect(&destRect,-((*(PicHandle) tempHdl)->picFrame.left - 2),
                           -((*(PicHandle) tempHdl)->picFrame.top - 22));
      DrawPicture((PicHandle) tempHdl,&destRect);

      HUnlock(tempHdl);
      DisposeHandle(tempHdl);
    }

    sizeOfTextData = GetScrap(NULL,'TEXT',&scrapOffset);
    if(sizeOfTextData > 0)
    {
      MoveTo(95,13);
      DrawString("\ptext");

      tempHdl = NewHandle((Size) sizeOfTextData);
      HLock(tempHdl);

      sizeOfTextData = GetScrap(tempHdl,'TEXT',&scrapOffset);

      destRect = gClipboardWindowPtr->portRect;
      destRect.top += 20;
      InsetRect(&destRect,2,2);

      TextBox(*tempHdl,sizeOfTextData,&destRect,0);

      HUnlock(tempHdl);
      DisposeHandle(tempHdl);
    }

    SetPort(oldPort);
  }
```

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDrawPictureWindow

void  doDrawPictureWindow(WindowPtr windowPtr)
{
  GrafPtr       oldPort;
  Rect          destRect;
  DocRecHandle  docRecHdl;

  GetPort(&oldPort);
  SetPort(windowPtr);

  docRecHdl = (DocRecHandle) GetWRefCon(windowPtr);
  destRect = doSetDestRect(&(*(*docRecHdl)->pictureHdl)->picFrame,windowPtr);

  DrawPicture((*docRecHdl)->pictureHdl,&destRect);

  if((*docRecHdl)->selectFlag)
  {
    InsetRect(&destRect,-2,-2);
    PenPat(&gMarqueePattern);
    FrameRect(&destRect);
  }

  SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOpenWindows

void  doOpenWindows(void)
{
  SInt16        a;
  WindowPtr     windowPtr;
  DocRecHandle  docRecHdl;

  for(a=0;a<2;a++)
  {
    if(!(windowPtr = GetNewWindow(rWindow + a,NULL,(WindowPtr)-1)))
      doErrorAlert(eFailWindow);
    gWindowPtrs[a] = windowPtr;

    if(!(docRecHdl = (DocRecHandle) NewHandle(sizeof(DocRec))))
      doErrorAlert(eFailDocRec);
    SetWRefCon(windowPtr,(SInt32) docRecHdl);

    (*docRecHdl)->pictureHdl = NULL;
    (*docRecHdl)->windowType = kDocumentType;
    (*docRecHdl)->selectFlag = false;
  }

  SetPort(windowPtr);

  (*docRecHdl)->pictureHdl = GetPicture(rPicture);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doSetDestRect

Rect  doSetDestRect(Rect *picFrame,WindowPtr windowPtr)
{
  Rect    destRect;
  SInt16  diffX, diffY;

  destRect = *picFrame;

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

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

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

  return(destRect);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
```

Demonstration Program Comments

When this program is run, the user should choose the **Edit** menu's **Show Clipboard** command to open the Clipboard window. The user should then cut, copy, clear and paste the supplied picture from/to the two windows opened by the program, noting the effect on the desk scrap as displayed in the Clipboard window. The user should also copy some text from another application's window and observe the changes to the contents of the Clipboard window.

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.

The user may also copy pictures from another application's window and paste them in the demonstration program's windows.

#define

Lines 59-69 establish constants relating to Menu IDs and menu item numbers. Lines 70-75 establish constants relating to various resources. Lines 75-80 are constants which index strings in a 'STR#' resource. Lines 82-83 establish constants which will enable the program to distinguish between the two "document" windows opened by the program and the Clipboard window.

#typedef

Document records will be attached to each of the two document windows. This is the associated data type.

Global Variables

gDone controls program termination. gInBackground relates to foreground/background switching. The WindowPtrs for the two document windows will be copied into the elements of gWindowPtrs. gClipBoardWindowPtr will be assigned the WindowPtr for the Clipboard window when it is opened by the user. gClipBoardShowing will keep track of whether the Clipboard window is currently hidden or showing. gMarqueePattern will be used to create an animated marquee-style rectangle around selected objects in the document windows.

main

The main function initialises the system software managers (Line 138), sets up the menus (Lines 142-152), opens the two document windows (Line 156), and enters the main event loop (Lines 160-170). Note that the sleep parameter in the WaitNextEvent call (Line 164) is set to 2 and that a null event will result in the application-defined function doIdle being called (Lines 168-169).

doIdle

doIdle is called when a null event is received (every 2 ticks). If the front window is not the Clipboard window, and if it contains a selected object, doIdle draws a rectangle around that object using the pattern contained in gMarqueePattern. doIdle also manipulates gMarqueePattern so that, with repeated calls to doIdle, the rectangle appears as an animated marquee-style rectangle.

Lines 202-203 get a handle to the front window's document record.

If Line 205 determines that the window is the Clipboard window or the window does not contain a selected object, the function returns immediately (Line 207); otherwise, the following occurs.

Lines 212 save the current graphics port and set the front window's port as the current port. Lines 214 retrieves the picFrame rectangle for the picture in the front window and calls an application-defined function which centres that rectangle in the window' port rectangle. Line 215 expands that rectangle by 2 pixels all around.

Line 217 saves the byte in the last element of gMarqueePattern. Lines 218-219 move each byte down one element in the array. Line 220 assigns the saved byte to the first element. Line 222 assigns gMarqueePattern to the pen, whose size remains at the default one pixel throughout the program, and Line 223 draws the rectangle in the specified pattern.

Line 225 restores the save graphics port.

doEvents

doEvents performs initial handling of events.

doMouseDown

doMouseDown handles mouse-down events. Note that, in the case of a mouse-down in the content region of the active window, the application-defined function doInContent is called (Line 286).

doUpdate

doUpdate handles update events.

Lines 308-310 retrieve the window type of the window in question. The main action occurs between the usual calls to BeginUpdate and EndUpdate. If the window is of the document type (as opposed to the Clipboard type), and if the window's document record currently contains a picture (Lines 314-316), an application-defined function is called to draw that picture (Line 317). If the window is the Clipboard window, an application-defined function is called to draw the Clipboard window (Lines 319-320).

doOSEvent

doOSEvent handles suspend/resume events. Line 337 sets gInBackground according to whether the event is a suspend or resume event.

Line 337 tests whether the Clipboard window has been opened by the user and whether the Clipboard should be showing when the demonstration program is in the foreground. If the window has previously been opened and gClipboardShowing contains true, and if the event is a suspend event (Line 339), the window is hidden (Line 340). If the event is a resume event, the window is shown (Line 342).

doAdjustMenus

doAdjustMenus adjusts the menus.

Lines 359-360 get handles to the **File** and **Edit** menus. Line 362 gets the handle to the document record for the front window.

If the front window is the Clipboard window (Line 364), the **Close** item is enabled, otherwise it is disabled.

If the document contains a picture and that picture is currently selected (Line 369), the **Cut**, **Copy**, and **Clear** items are enabled, otherwise they are disabled (Lines 370-380).

If the desk scrap contains data of type 'PICT' and the front window is not the Clipboard window, the **Paste** item is enabled, otherwise it is disabled (Lines 382-385).

Line 387 redraws the menu bar.

doMenuChoice, doEditMenu

doMenuChoice and doEditMenu handle menu choices.

doCloseWindow

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

Lines 467-468 get a pointer to the front window and the handle to its document record. If the window is the Clipboard window (Line 470), The window is disposed of, the global variable which contains its pointer 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.**

doErrorAlert

doErrorAlert invokes an appropriate alert box in which an error string is displayed. It then either terminates the program or returns to the calling function depending on the severity of the error.

doInContent

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

Lines 507-508 get a pointer the front window and the handle to its document record. If the front window is the Clipboard window, the function returns immediately (Lines 510-511). Lines 513-514 save the current graphics port and make the graphics port associated with the front window the current graphics port.

If the front window contains a picture (Line 516) the following occurs. Line 518 calls an application-defined function which returns a rectangle of the same dimensions as that contained in the picture record's picFrame field, but centred laterally and vertically in the window. Line 519 expands this rectangle by two pixels all around. Line 521 converts the mouse-down coordinates to local coordinates. If the mouse-down occurred within the rectangle (Line 523), the document record's selectionFlag field is set to true. If the mouse-down occurred outside that rectangle (Line 527), the document record's selectionFlag field is set to false, and the rectangle is erased (Lines 529-533).

Line 538 resets the current graphics port to that saved at function entry.

doCutCopyCommand

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

Lines 551-552 get a pointer to the front window and the handle to that window's document record.

If the selectionFlag field of the document record contains false, the function returns immediately (Lines 554-555). (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.)

Line 557 purges the desk scrap. If the call is successful, Line 559 gets the size of the picture record, Line 560 locks the picture record, Line 562 copies the picture to the desk scrap, and Line 566 unlocks the picture record. If the calls to ZeroScrap and PutScrap are not successful, a caution alert is displayed to advise the user of the error (Lines 563-564 and Line 569).

If the menu choice was the **Cut** item (Line 571), 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 (Lines 573-574). Lines 576-578 then dispose of the picture record and set the document record's pictureHdl and selectionFlag fields to NULL and false respectively. Line 579 erases the picture from the window and Line 581 resets the saved graphics port.

Finally, and importantly, if the Clipboard window has previously been opened by the user (Line 584), an application defined function is called to draw the current contents of the desk scrap in the Clipboard window (Line 585).

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.

Lines 599-600 get a pointer to the front window and the handle to that window's document record. Lines 602-603 save the current graphics port and make the graphics port associated with the front window the current graphics port.

In order to determine whether the desk scrap contains data of type 'PICT', Line 605 calls GetScrap with the destHandle parameter set to NULL. The following occurs if data of type 'PICT' is present in the desk scrap (Line 606).

Lines 608-609 create and lock a relocatable block of a size equivalent to the 'PICT' data in the scrap. GetScrap is called again (Line 611) to copy the 'PICT' data in the scrap to the newly-created relocatable block. Line 613 erases the front window and Line 614 sets the selectionFlag field of the document record associated with the front window to false. Line 615 then calls an application-defined function which takes the picFrame field from the picture record and creates a destination rectangle of the same dimensions as the picFrame rectangle but centred in the front window. Line 617 draws the picture in this rectangle.

If the document record currently contains a picture, the picture record is disposed of (Lines 619-620). Line 622 creates a new relocatable block the size of the 'PICT' data and assigns its handle to the pictureHdl field of the document record. Line 623 then copies the bytes in the relocatable block created at Line 608 to this new relocatable block. Lines 625-626 unlock and dispose of the block created at Line 608.

Line 629 resets the current graphics port to the port saved at function entry.

doClearCommand

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

Note that, as was the case in the doCutCopyCommand function, 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.

Lines 640-641 get a pointer to the front window and the handle to that window's document record. Lines 643-644 save the current graphics port and make the graphics port associated with the front window the current graphics port.

Lines 648-649 dispose of the picture record, set the pictureHdl field of the document record to NULL, set the selectionFlag field of the document record to false, and erase the window's port rectangle.

Line 651 resets the current graphics port to the port saved at function entry.

doClipboardCommand

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

Line 661 gets the handle 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**.

Line 663 checks whether the Clipboard window has been opened. If not, the Clipboard window is opened (Line 665), a document record is created and attached to the window (Lines 667-669), the windowType field of the document record is set to indicate that the window is of the Clipboard type (Line 670), a global variable which keeps track of whether the Clipboard window is currently showing or hidden is set to true (Line 672), and the text of the menu item is set to **Hide Clipboard** (Line 674).

If the Clipboard window has previously been opened (Line 676), and if the window is currently showing (Line 678), the window is hidden, the Clipboard-showing flag is set to false, and the item's text is set to **Show Clipboard** (Lines 680-682). If the window is not currently showing (Line 684), the window is made visible, the Clipboard-showing flag is set to true, and the item's text is set to **Hide Clipboard** (Lines 686-688).

doDrawClipboardWindow

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

Lines 702-703 save the current graphics port and make the graphics port associated with the front window the current graphics port.

Line 705 erases the window's port rectangle. Lines 707-715 draw a panel at the top of the window in which the type of data in the desk scrap will be displayed.

Line 717, in which NULL is passed as the destHandle parameter of the GetScrap call, checks whether data of type 'PICT' exists in the desk scrap. If so (Line 718), the following occurs. The word "picture" is drawn in the panel at the top of the window (Lines 720-721). A relocatable block the size of the 'PICT' data is created and locked (Lines 723-724) and GetScrap is called once again to copy the 'PICT' data from the scrap into the newly-created block (Line 726). A destination rectangle, based on the rectangle in the picFrame field of the picture record, is created with its left and top fields set to two pixels right of, and 22 pixels below, the left and top sides of the window (Lines 728-730). The picture is then drawn in this destination rectangle (Line 731), following which the relocatable block created at Line 723 is unlocked and disposed of (Lines 733-734).

Lines 737 checks whether data of type 'TEXT' exists in the desk scrap. If so (Line 738), much the same procedure is followed, the differences being that the word "text" is drawn in the panel at the top of the window (Line 741), the destination rectangle is set to two pixels inside the port rectangle less the panel (Lines 748-750), and the text is drawn in this rectangle using TextBox (Line 752). (TextBox is a TextEdit routine, and is described at Chapter 17 — Text and TextEdit.)

Line 758 resets the current graphics port to the port saved at function entry.

doDrawPictureWindow

doDrawPictureWindow draws the picture belonging to a document window in that window.

Lines 769-771 save the current graphics port and make the graphics port associated with the front window the current graphics port.

Line 772 gets the handle to the window's document record. Line 773 calls an application-defined function which takes the rectangle contained in the picFrame field of the picture record (the handle to which is contained in the pictureHdl field of the document record), and creates a new rectangle of the same dimensions but centred in the window. Line 775 draws the picture specified in the window's document record in this rectangle.

If the selectionFlag field of the document record indicates that the picture is currently selected (Line 777), Lines 779-781 draw a dotted rectangle two pixels outside the picture.

Line 784 resets the current graphics port to the port saved at function entry.

doOpenWindows

doOpenWindows opens the two document windows, creates document records for each window, attaches the document records to the windows and initialises the fields of the document records (Lines 795-808). The graphics port of the second window created is then set as the current port (Line 810) and a picture is read in from a resource, its handle being assigned to the pictureHdl field of the second window's document record (Line 812).

doSetDestRect

doSetDestRect takes the rectangle contained in the picFrame field of a picture record and returns a rectangle of the same dimensions but centred in the window's port rectangle.

Line 822 makes a local Rect variable equal to the rectangle in the picFrame field. Line 824 then offsets this rectangle to the left and top of the port rectangle. Lines 826-829 calculate the differences between the widths and heights of the rectangle and the window's port rectangle. This is used at Line 831 to further offset the rectangle to the middle of the port rectangle. The rectangle is then returned to the calling function (Line 833).