16 Version 1.2 (Frozen)

SCRAP

Includes Demonstration Program ScrapPascal

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.

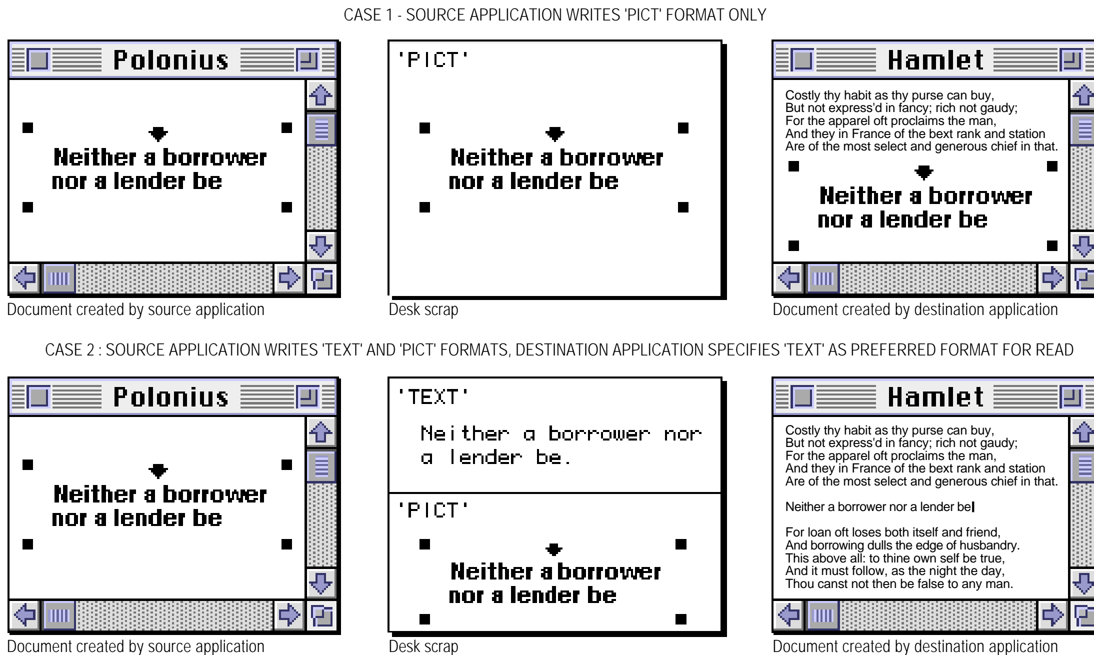

FIG 1 - SPECIFYING FORMATS TO WRITE TO AND READ FROM THE DESK SCRAP

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.

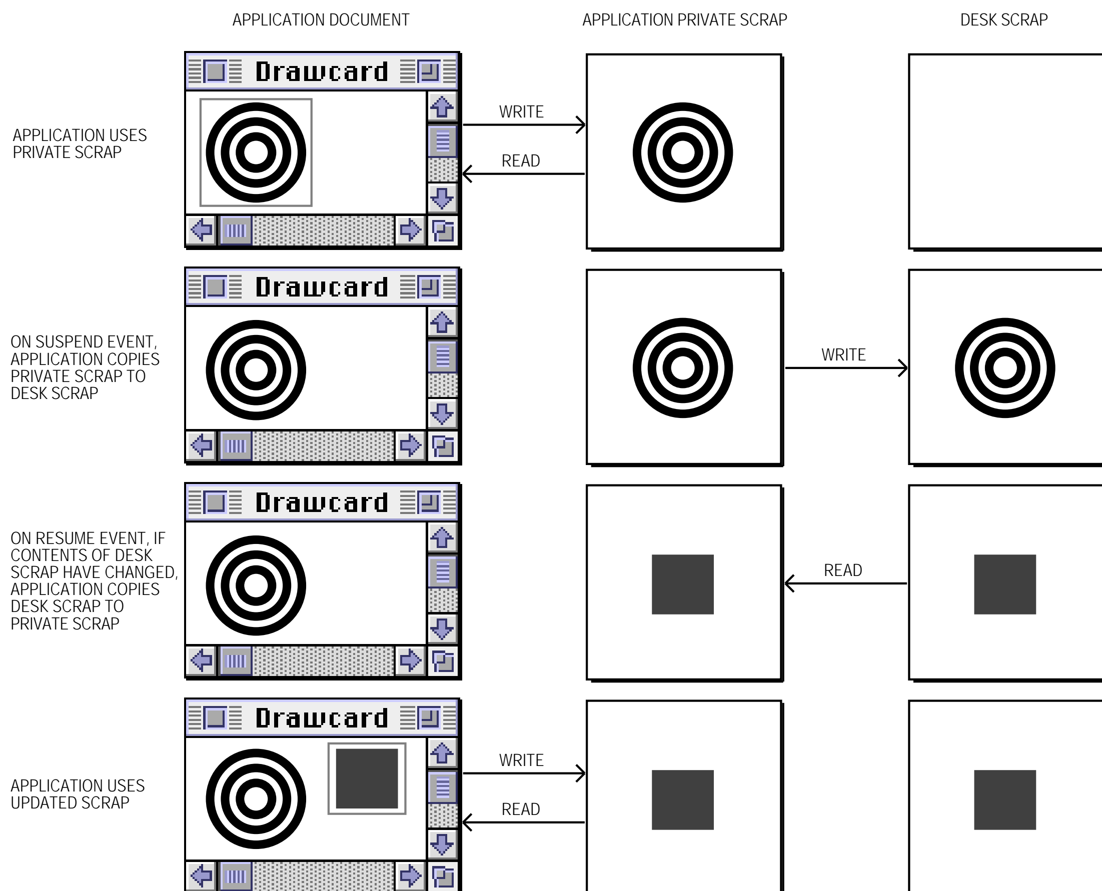

FIG 2 - USING A PRIVATE SCRAP

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 `nil` 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

```
type
  ScrapStuff = record
    scrapSize:     longint;
    scrapHandle:   Handle;
    scrapCount:    integer;
    scrapState:    integer;
    scrapName:     StringPtr;
  end;

  PScrapStuff = ^ScrapStuff;
  ScrapStuffPtr = PScrapStuff;
```

Routines

Getting Information About the Scrap

```
function  InfoScrap: ScrapStuffPtr;
```

Writing Information to the Scrap

```
function  ZeroScrap: longint;
function  PutScrap(length: longint; theType: ResType; source: UNIV Ptr): longint;
```

Reading Information From the Scrap

```
function  GetScrap(hDest: Handle; theType: ResType; var offset: longint): longint;
```

Transferring the Scrap Between Memory and Disk

```
function   UnloadScrap: longint;
function   LoadScrap: longint;
```

Demonstration Program

```
1   { ########################################################################################
2   // ScrapPascal.p
3   // ########################################################################################
4   //
5   // This program utilises the desk scrap and Scrap Manager routines to allow the user to:
6   //
7   // •   Cut, copy and clear pictures from, and paste pictures to, two windows opened by the
8   //     program.
9   //
10  // •   Paste pictures cut or copied from another application to the two windows opened
11  //     by the program.
12  //
13  // •   Open and close a Clipboard window, in which the current contents of the desk scrap
14  //     are displayed.
15  //
16  // In addition to the pictures cut and copied from either the program's windows or from
17  // another application's windows, the Clipboard window will display text copied to the
18  // desk scrap as a result of text cut and copy operations in another application.  The
19  // program, however, does not support the pasting of this text to documents displayed in
20  // the program's windows.  (The demonstration program at Chapter 17 - Text and TextEdit
21  // shows how to cut, copy and paste text from and to a TextEdit edit record using the
22  // desk scrap.)
23  //
24  // The program utilises the following resources:
25  //
26  // •   An 'MBAR' resource, and 'MENU' resources for Apple, File, and Edit menus (preload,
27  //     non-purgeable).
28  //
29  // •   Three 'WIND' resources (purgeable) (initially visible), two for the program's main
30  //     windows and one for the Clipboard window.
31  //
32  // •   A 'PICT' resource (non-purgeable) containing a picture which may be cut, copied,
33  //     and pasted between the windows.
34  //
35  // •   An 'ALRT' resource (purgeable) and associated 'DITL' resource (purgeable) for use
36  //     by an error Alert.
37  //
38  // •   A 'STR#' resource (purgeable) containing strings to be displayed in the error
39  //     Alert.
40  //
41  // •   A 'SIZE' resource with the acceptSuspendResumeEvents, doesActivateOnFGcase, and
42  //     is32BitCompatible flags set.
43  //
44  // ####################################################################################### }
45
46  program ScrapPascal(input, output);
47
48  { ........................................................ include the following Universal Interfaces }
49
50  uses
51
52    Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
53    Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Scrap, SegLoad, Sound;
54
55  { ...................................................................... define the following constants }
56
57  const
58
59  mApple = 128;
60    iAbout = 1;
61  mFile = 129;
62    iClose = 4;
63    iQuit = 11;
64  mEdit = 130;
65    iCut = 3;
66    iCopy = 4;
67    iPaste = 5;
```

```
68      iClear = 6;
69      iClipboard = 9;
70
71    rMenubar = 128;
72    rWindow = 128;
73    rClipboardWindow = 130;
74    rPicture = 128;
75    rAlertBox = 128;
76    rErrorStrings = 128;
77      eFailMenu = 1;
78      eFailWindow = 2;
79      eFailDocRec = 3;
80      eZeroScrap = 4;
81      ePutScrap = 5;
82      eNoPictInScrap = 6;
83
84    kDocumentType = 1;
85    kClipboardType = 2;
86
87    { ...................................................................... user defined types }
88
89    type
90
91    DocRec = record
92      pictureHdl : PicHandle;
93      selectFlag : boolean;
94      windowType : integer;
95      end;
96
97    DocRecPtr = ^DocRec;
98    DocRecHandle = ^DocRecPtr;
99
100   { ...................................................................... global variables }
101
102   var
103
104   gDone : boolean;
105   gInBackground : boolean;
106   gWindowPtrs : array[0..1] of WindowPtr;
107   gClipboardWindowPtr : WindowPtr;
108   gClipboardShowing : boolean;
109   gMarqueePattern : Pattern;
110   menubarHdl : Handle;
111   menuHdl : MenuHandle;
112   gotEvent : boolean ;
113   theEvent : EventRecord;
114
115   { ####################################################################### DoInitManagers }
116
117   procedure DoInitManagers;
118
119     begin
120     MaxApplZone;
121     MoreMasters;
122
123     InitGraf(@qd.thePort);
124     InitFonts;
125     InitWindows;
126     InitMenus;
127     TEInit;
128     InitDialogs(nil);
129
130     InitCursor;
131     FlushEvents(everyEvent, 0);
132     end;
133       {of procedure DoInitManagers}
134
135   { ######################################################################### DoErrorAlert }
136
137   procedure DoErrorAlert(errorCode : integer);
138
139     var
140     errorString : string;
141     ignored : OSErr;
142
143     begin
144     GetIndString(errorString, rErrorStrings, errorCode);
```

```
145    ParamText(errorString, '', '', '');
146
147    if (errorCode < ePutScrap) then
148      begin
149      ignored := StopAlert(rAlertBox, nil);
150      ExitToShell;
151      end
152    else
153      ignored := CautionAlert(rAlertBox, nil);
154    end;
155      {of procedure DoErrorAlert}
156
157  { ######################################################################## DoOpenWindows }
158
159  procedure DoOpenWindows;
160
161    var
162    a : integer;
163    myWindowPtr : WindowPtr;
164    docRecHdl : DocRecHandle;
165
166    begin
167    for a := 0 to 1 do
168      begin
169      myWindowPtr := GetNewWindow(rWindow + a, nil, WindowPtr(-1));
170      if (myWindowPtr = nil) then
171        DoErrorAlert(eFailWindow);
172      gWindowPtrs[a] := myWindowPtr;
173
174      docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
175      if (docRecHdl = nil) then
176        DoErrorAlert(eFailDocRec);
177      SetWRefCon(myWindowPtr, longint(docRecHdl));
178
179      docRecHdl^^.pictureHdl := nil;
180      docRecHdl^^.windowType := kDocumentType;
181      docRecHdl^^.selectFlag := false;
182      end;
183
184    SetPort(myWindowPtr);
185
186    docRecHdl^^.pictureHdl := GetPicture(rPicture);
187    end;
188      {of procedure DoOpenWindows}
189
190  { ######################################################################## DoSetDestRect }
191
192  function DoSetDestRect(var picFrame : Rect; myWindowPtr : WindowPtr) : Rect;
193
194    var
195    destRect : Rect;
196    diffX, diffY : integer;
197
198    begin
199    destRect := picFrame;
200
201    OffsetRect(destRect, -(picFrame.left), -(picFrame.top));
202
203    diffX := (myWindowPtr^.portRect.right - myWindowPtr^.portRect.left) -
204            (picFrame.right - picFrame.left);
205    diffY := (myWindowPtr^.portRect.bottom - myWindowPtr^.portRect.top) -
206            (picFrame.bottom - picFrame.top);
207
208    OffsetRect(destRect, diffX div 2, diffY div 2);
209
210    DoSetDestRect := destRect;
211    end;
212      {of procedure DoSetDestRect}
213
214  { ################################################################## DoDrawPictureWindow }
215
216  procedure DoDrawPictureWindow(myWindowPtr : WindowPtr);
217
218    var
219    oldPort : GrafPtr;
220    destRect : Rect;
221    docRecHdl : DocRecHandle;
```

```
222
223     begin
224     GetPort(oldPort);
225     SetPort(myWindowPtr);
226
227     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
228     destRect := DoSetDestRect(docRecHdl^^.pictureHdl^^.picFrame, myWindowPtr);
229
230     DrawPicture(docRecHdl^^.pictureHdl, destRect);
231
232     if (docRecHdl^^.selectFlag) then
233       begin
234       InsetRect(destRect, -2, -2);
235       PenPat(gMarqueePattern);
236       FrameRect(destRect);
237       end;
238
239     SetPort(oldPort);
240     end;
241       {of procedure DoDrawPictureWindow}
242
243   { ################################################################ DoDrawClipboardWindow }
244
245   procedure DoDrawClipboardWindow;
246
247     var
248     oldPort : GrafPtr;
249     sizeOfPictData, sizeOfTextData, scrapOffset : longint;
250     tempHdl : Handle;
251     destRect : Rect;
252
253     begin
254     GetPort(oldPort);
255     SetPort(gClipboardWindowPtr);
256
257     EraseRect(gClipboardWindowPtr^.portRect);
258
259     MoveTo(0, 18);
260     LineTo(505, 18);
261     MoveTo(0, 20);
262     LineTo(505, 20);
263
264     TextFont(applFont);
265     TextSize(9);
266     MoveTo(4, 13);
267     DrawString('Clipboard contents:');
268
269     sizeOfPictData := GetScrap(nil, 'PICT', scrapOffset);
270     if (sizeOfPictData > 0) then
271       begin
272       MoveTo(95, 13);
273       DrawString('picture');
274
275       tempHdl := NewHandle(Size(sizeOfPictData));
276       HLock(tempHdl);
277
278       sizeOfPictData := GetScrap(tempHdl, 'PICT', scrapOffset);
279
280       destRect := PicHandle(tempHdl)^^.picFrame;
281       OffsetRect(destRect, -(PicHandle(tempHdl)^^.picFrame.left - 2),
282                   -(PicHandle(tempHdl)^^.picFrame.top - 22));
283       DrawPicture(PicHandle(tempHdl), destRect);
284
285       HUnlock(tempHdl);
286       DisposeHandle(tempHdl);
287       end;
288
289     sizeOfTextData := GetScrap(nil, 'TEXT', scrapOffset);
290     if (sizeOfTextData > 0) then
291       begin
292       MoveTo(95, 13);
293       DrawString('text');
294
295       tempHdl := NewHandle(Size(sizeOfTextData));
296       HLock(tempHdl);
297
298       sizeOfTextData := GetScrap(tempHdl, 'TEXT', scrapOffset);
```

```
299
300       destRect := gClipboardWindowPtr^.portRect;
301       destRect.top := destRect.top + 20;
302       InsetRect(destRect, 2, 2);
303
304       TETextBox(tempHdl^, sizeOfTextData, destRect, 0);
305
306       HUnlock(tempHdl);
307       DisposeHandle(tempHdl);
308       end;
309
310     SetPort(oldPort);
311     end;
312       {of procedure DoIdle}
313
314   { ################################################################### DoClipboardCommand }
315
316   procedure DoClipboardCommand;
317
318     var
319     editMenuHdl : MenuHandle;
320     docRecHdl : DocRecHandle;
321
322     begin
323     editMenuHdl := GetMenu(mEdit);
324
325     if (gClipboardWindowPtr = nil) then
326       begin
327       gClipboardWindowPtr := GetNewWindow(rClipboardWindow, nil, WindowPtr(-1));
328       if (gClipboardWindowPtr = nil) then
329         DoErrorAlert(eFailWindow);
330
331       docRecHdl := DocRecHandle(NewHandle(sizeof(DocRec)));
332       if (docRecHdl = nil) then
333         DoErrorAlert(eFailDocRec);
334       SetWRefCon(gClipboardWindowPtr, longint(docRecHdl));
335       docRecHdl^^.windowType := kClipboardType;
336
337       gClipboardShowing := true;
338
339       SetMenuItemText(editMenuHdl, iClipboard, 'Hide Clipboard');
340       end
341
342     else begin
343       if (gClipboardShowing) then
344         begin
345         HideWindow(gClipboardWindowPtr);
346         gClipboardShowing := false;
347         SetMenuItemText(editMenuHdl, iClipboard, 'Show Clipboard');
348         end
349       else
350         begin
351         ShowWindow(gClipboardWindowPtr);
352         gClipboardShowing := true;
353         SetMenuItemText(editMenuHdl, iClipboard, 'Hide Clipboard');
354         end;
355       end;
356     end;
357       {of procedure DoClipboardCommand}
358
359   { ####################################################################### DoClearCommand }
360
361   procedure DoClearCommand;
362
363     var
364     myWindowPtr : WindowPtr;
365     docRecHdl : DocRecHandle;
366     oldPort : GrafPtr;
367
368     begin
369     myWindowPtr := FrontWindow;
370     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
371
372     GetPort(oldPort);
373     SetPort(myWindowPtr);
374
375     DisposeHandle(Handle(docRecHdl^^.pictureHdl));
```

```
376    docRecHdl^^.pictureHdl := nil;
377    docRecHdl^^.selectFlag := false;
378    EraseRect(myWindowPtr^.portRect);
379
380    SetPort(oldPort);
381    end;
382      {of procedure DoClearCommand}
383
384  { ##################################################################### DoPasteCommand }
385
386  procedure DoPasteCommand;
387
388    var
389    myWindowPtr : WindowPtr;
390    docRecHdl : DocRecHandle;
391    oldPort : GrafPtr;
392    sizeOfPictData, scrapOffset : longint;
393    tempHdl : Handle;
394    destRect : Rect;
395
396    begin
397    myWindowPtr := FrontWindow;
398    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
399
400    GetPort(oldPort);
401    SetPort(myWindowPtr);
402
403    sizeOfPictData := GetScrap(nil, 'PICT', scrapOffset);
404    if (sizeOfPictData > 0) then
405      begin
406      tempHdl := NewHandle(Size(sizeOfPictData));
407      HLock(tempHdl);
408
409      sizeOfPictData := GetScrap(tempHdl, 'PICT', scrapOffset);
410
411      EraseRect(myWindowPtr^.portRect);
412      docRecHdl^^.selectFlag := false;
413      destRect := DoSetDestRect(PicHandle(tempHdl)^^.picFrame, myWindowPtr);
414
415      DrawPicture(PicHandle(tempHdl), destRect);
416
417      if (docRecHdl^^.pictureHdl <> nil) then
418        DisposeHandle(Handle(docRecHdl^^.pictureHdl));
419
420      docRecHdl^^.pictureHdl := PicHandle(NewHandle(Size(sizeOfPictData)));
421      BlockMoveData(tempHdl^, docRecHdl^^.pictureHdl^, Size(sizeOfPictData));
422
423      HUnlock(tempHdl);
424      DisposeHandle(tempHdl);
425      end;
426
427    SetPort(oldPort);
428    end;
429      {of procedure DoPasteCommand}
430
431  { ################################################################### DoCutCopyCommand }
432
433  procedure DoCutCopyCommand(cutFlag : boolean);
434
435    var
436    myWindowPtr : WindowPtr;
437    docRecHdl : DocRecHandle;
438    dataLength : Size;
439    errorCode : longint;
440    oldPort : GrafPtr;
441
442    begin
443    myWindowPtr := FrontWindow;
444    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
445
446    if (docRecHdl^^.selectFlag = false) then
447      Exit(DoCutCopyCommand);
448
449    if (ZeroScrap = noErr) then
450      begin
451      dataLength := GetHandleSize(Handle(docRecHdl^^.pictureHdl));
452      HLock(Handle(docRecHdl^^.pictureHdl));
```

```
453
454        errorCode := PutScrap(longint(dataLength), 'PICT', Handle(docRecHdl^^.pictureHdl)^);
455        if (errorCode <> noErr) then
456          DoErrorAlert(ePutScrap);
457
458        HUnlock(Handle(docRecHdl^^.pictureHdl));
459        end
460      else
461        DoErrorAlert(eZeroScrap);
462
463      if (cutFlag) then
464        begin
465        GetPort(oldPort);
466        SetPort(myWindowPtr);
467
468        DisposeHandle(Handle(docRecHdl^^.pictureHdl));
469        docRecHdl^^.pictureHdl := nil;
470        docRecHdl^^.selectFlag := false;
471        EraseRect(myWindowPtr^.portRect);
472
473        SetPort(oldPort);
474        end;
475
476      if (gClipboardWindowPtr <> nil) then
477        DoDrawClipboardWindow;
478      end;
479        {of procedure DoIdle}
480
481    { ########################################################################## DoInContent }
482
483    procedure DoInContent(mouseXY : Point);
484
485      var
486      myWindowPtr : WindowPtr;
487      docRecHdl : DocRecHandle;
488      oldPort : GrafPtr;
489      pictRect : Rect;
490
491      begin
492      myWindowPtr := FrontWindow;
493      docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
494
495      if (docRecHdl^^.windowType = kClipboardType) then
496        Exit(DoInContent);
497
498      GetPort(oldPort);
499      SetPort(myWindowPtr);
500
501      if (docRecHdl^^.pictureHdl <> nil) then
502        begin
503        pictRect := DoSetDestRect(docRecHdl^^.pictureHdl^^.picFrame, myWindowPtr);
504        InsetRect(pictRect, -2, -2);
505
506        GlobalToLocal(mouseXY);
507
508        if (PtInRect(mouseXY, pictRect)) then
509          docRecHdl^^.selectFlag := true
510        else
511          begin
512          docRecHdl^^.selectFlag := false;
513
514          PenPat(qd.black);
515          ForeColor(whiteColor);
516          FrameRect(pictRect);
517          ForeColor(blackColor);
518          end;
519        end;
520
521      SetPort(oldPort);
522      end;
523        {of procedure DoInContent}
524
525    { ######################################################################## DoCloseWindow }
526
527    procedure DoCloseWindow;
528
529      var
```

```
530     myWindowPtr : WindowPtr;
531     docRecHdl : DocRecHandle;
532     editMenuHdl : MenuHandle;
533 
534     begin
535     myWindowPtr := FrontWindow;
536     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
537 
538     if (docRecHdl^^.windowType = kClipboardType) then
539       begin
540       DisposeWindow(myWindowPtr);
541       gClipboardWindowPtr := nil;
542       gClipboardShowing := false;
543       editMenuHdl := GetMenu(mEdit);
544       SetMenuItemText(editMenuHdl, iClipboard, 'Show Clipboard');
545       end;
546     end;
547       {of procedure DoCloseWindow}
548 
549   { ########################################################################## DoEditMenu }
550 
551   procedure DoEditMenu(menuItem : integer);
552 
553     begin
554     case (menuItem) of
555 
556       iCut:
557         begin
558         DoCutCopyCommand(true);
559         end;
560 
561       iCopy:
562         begin
563         DoCutCopyCommand(false);
564         end;
565 
566       iPaste:
567         begin
568         DoPasteCommand;
569         end;
570 
571       iClear:
572         begin
573         DoClearCommand;
574         end;
575 
576       iClipboard:
577         begin
578         DoClipboardCommand;
579         end;
580       end;
581         {of case statement}
582     end;
583       {of procedure DoEditMenu}
584 
585   { ######################################################################## DoMenuChoice }
586 
587   procedure DoMenuChoice(menuChoice : longint);
588 
589     var
590     menuID, menuItem : integer;
591     itemName : string;
592     daDriverRefNum : integer;
593 
594     begin
595     menuID := HiWord(menuChoice);
596     menuItem := LoWord(menuChoice);
597 
598     if (menuID = 0) then
599       Exit(DoMenuChoice);
600 
601     case (menuID) of
602 
603       mApple:
604         begin
605         if (menuItem = iAbout) then
606           SysBeep(10)
```

```
607         else begin
608           GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
609           daDriverRefNum := OpenDeskAcc(itemName);
610           end;
611         end;
612 
613       mFile:
614         begin
615         if (menuItem = iClose) then
616           DoCloseWindow
617         else if (menuItem = iQuit) then
618           gDone := true;
619         end;
620 
621       mEdit:
622         begin
623         DoEditMenu(menuItem);
624         end;
625       end;
626         {of case statement}
627 
628     HiliteMenu(0);
629     end;
630       {of procedure DoMenuChoice}
631 
632   { ######################################################################## DoAdjustMenus }
633 
634   procedure DoAdjustMenus;
635 
636     var
637     fileMenuHdl, editMenuHdl : MenuHandle;
638     docRecHdl : DocRecHandle;
639     scrapOffset : longint;
640 
641     begin
642     fileMenuHdl := GetMenuHandle(mFile);
643     editMenuHdl := GetMenuHandle(mEdit);
644 
645     docRecHdl := DocRecHandle(GetWRefCon(FrontWindow));
646 
647     if (docRecHdl^^.windowType = kClipboardType) then
648       EnableItem(fileMenuHdl, iClose)
649     else
650       DisableItem(fileMenuHdl, iClose);
651 
652     if ((docRecHdl^^.pictureHdl <> nil) & (docRecHdl^^.selectFlag)) then
653       begin
654       EnableItem(editMenuHdl, iCut);
655       EnableItem(editMenuHdl, iCopy);
656       EnableItem(editMenuHdl, iClear);
657       end
658     else
659       begin
660       DisableItem(editMenuHdl, iCut);
661       DisableItem(editMenuHdl, iCopy);
662       DisableItem(editMenuHdl, iClear);
663       end;
664 
665     if ((GetScrap(nil, 'PICT', scrapOffset) <> 0) & (docRecHdl^^.windowType <> kClipboardType))
666       then EnableItem(editMenuHdl, iPaste)
667       else DisableItem(editMenuHdl, iPaste);
668 
669     DrawMenuBar;
670     end;
671       {of procedure DoAdjustMenus}
672 
673   { ############################################################################ DoOSEvent }
674 
675   procedure DoOSEvent(var theEvent : EventRecord);
676 
677     var
678     myWindowPtr : WindowPtr;
679 
680     begin
681     myWindowPtr := FrontWindow;
682 
683     case BAnd(BSR(theEvent.message, 24), $000000FF) of
```

```
684
685       suspendResumeMessage:
686         begin
687         gInBackground := (BAnd(theEvent.message, resumeFlag) = 0);
688         if ((gClipboardWindowPtr <> nil) & gClipboardShowing) then
689           begin
690           if (gInBackground) then
691             HideWindow(gClipboardWindowPtr)
692           else
693             ShowWindow(gClipboardWindowPtr);
694           end;
695         end;
696
697       mouseMovedMessage:
698         begin
699         end;
700       end;
701         {of case statement}
702     end;
703       {of procedure DoOSEvent}
704
705   { ############################################################################ DoUpdate }
706
707   procedure DoUpdate(var theEvent : EventRecord);
708
709     var
710     myWindowPtr : WindowPtr;
711     docRecHdl : DocRecHandle;
712     windowType : longint;
713
714     begin
715     myWindowPtr := WindowPtr(theEvent.message);
716     docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
717     windowType := docRecHdl^^.windowType;
718
719     BeginUpdate(myWindowPtr);
720
721     if (windowType = kDocumentType) then
722       begin
723       if (docRecHdl^^.pictureHdl <> nil) then
724         DoDrawPictureWindow(myWindowPtr);
725       end
726
727     else if (windowType = kClipboardType) then
728       DoDrawClipboardWindow;
729
730     EndUpdate(myWindowPtr);
731     end;
732       {of procedure DoUpdate}
733
734   { ######################################################################### DoMouseDown }
735
736   procedure DoMouseDown(var theEvent : EventRecord);
737
738     var
739     partCode : integer;
740     myWindowPtr : WindowPtr;
741
742     begin
743     partCode := FindWindow(theEvent.where, myWindowPtr);
744
745     case (partCode) of
746
747       inMenuBar:
748         begin
749         DoAdjustMenus;
750         DoMenuChoice(MenuSelect(theEvent.where));
751         end;
752
753       inSysWindow:
754         begin
755         SystemClick(theEvent, myWindowPtr);
756         end;
757
758       inContent:
759         begin
760         if (myWindowPtr <> FrontWindow)
```

```
761          then SelectWindow(myWindowPtr)
762          else DoInContent(theEvent.where);
763        end;
764
765      inDrag:
766        begin
767        DragWindow(myWindowPtr, theEvent.where, qd.screenBits.bounds);
768        end;
769
770      inGoAway:
771        begin
772        if (TrackGoAway(myWindowPtr, theEvent.where) = true) then
773          DoCloseWindow;
774        end;
775      end;
776        {of case statement}
777    end;
778      {of procedure DoMouseDown}
779
780  { ############################################################################## DoEvents }
781
782  procedure DoEvents(var theEvent : EventRecord);
783
784    var
785    charCode : char;
786
787    begin
788    case (theEvent.what) of
789
790      mouseDown:
791        begin
792        DoMouseDown(theEvent);
793        end;
794
795      keyDown, autoKey:
796        begin
797        charCode := chr(BAnd(theEvent.message, charCodeMask));
798        if (BAnd(theEvent.modifiers, cmdKey) <> 0) then
799          begin
800          DoAdjustMenus;
801          DoMenuChoice(MenuKey(charCode));
802          end;
803        end;
804
805      updateEvt:
806        begin
807        DoUpdate(theEvent);
808        end;
809
810      osEvt:
811        begin
812        DoOSEvent(theEvent);
813        HiliteMenu(0);
814        end;
815      end;
816        {of case statement}
817    end;
818      {of procedure DoEvents}
819
820  { ################################################################################ DoIdle }
821
822  procedure DoIdle;
823    var
824    myWindowPtr : WindowPtr;
825    docRecHdl : DocRecHandle;
826    oldPort : GrafPtr;
827    marqueeRect : Rect;
828    lastByte : SInt8;
829    a : integer;
830
831    begin
832    myWindowPtr := FrontWindow;
833    docRecHdl := DocRecHandle(GetWRefCon(myWindowPtr));
834
835    if ((docRecHdl^^.windowType = kClipboardType) or (docRecHdl^^.selectFlag = false)) then
836      begin
837      Exit(DoIdle);
```

```
838        end
839
840      else begin
841        GetPort(oldPort);
842        SetPort(myWindowPtr);
843
844        marqueeRect := DoSetDestRect(docRecHdl^^.pictureHdl^^.picFrame, myWindowPtr);
845        InsetRect(marqueeRect, -2, -2);
846
847        lastByte := gMarqueePattern.pat[7];
848        for a := 7 downto 1 do
849          gMarqueePattern.pat[a] := gMarqueePattern.pat[a - 1];
850        gMarqueePattern.pat[0] := lastByte;
851
852        PenPat(gMarqueePattern);
853        FrameRect(marqueeRect);
854
855        SetPort(oldPort);
856        end;
857      end;
858        {of procedure DoIdle}
859
860    { ################################################################ start of main program }
861
862    begin
863    gClipboardWindowPtr := nil;
864    gClipboardShowing := false;
865    {$R-}
866    gMarqueePattern.pat[0] := SInt8($1F);
867    gMarqueePattern.pat[1] := SInt8($3E);
868    gMarqueePattern.pat[2] := SInt8($7C);
869    gMarqueePattern.pat[3] := SInt8($F8);
870    gMarqueePattern.pat[4] := SInt8($F1);
871    gMarqueePattern.pat[5] := SInt8($E3);
872    gMarqueePattern.pat[6] := SInt8($C7);
873    gMarqueePattern.pat[7] := SInt8($8F);
874    {$R+}
875      { ............................................................................ initialize managers }
876
877      DoInitManagers;
878
879      { ...................................................................... set up menu bar and menus }
880
881      menubarHdl := GetNewMBar(rMenubar);
882      if (menubarHdl = nil) then
883        DoErrorAlert(eFailMenu);
884      SetMenuBar(menubarHdl);
885      DrawMenuBar;
886
887      menuHdl := GetMenuHandle(mApple);
888      if (menuHdl = nil) then
889        DoErrorAlert(eFailMenu)
890      else
891        AppendResMenu(menuHdl, 'DRVR');
892
893      { ........................................................................................ open windows }
894
895      DoOpenWindows;
896
897      { ..................................................................................... enter eventLoop }
898
899      gDone := false;
900
901      while not (gDone) do
902        begin
903        gotEvent := WaitNextEvent(everyEvent, theEvent, 2, nil);
904
905        if (gotEvent)
906          then DoEvents(theEvent)
907          else DoIdle;
908        end;
909
910    end.
911      {of main program block}
912
913    { ######################################################################################### }
```

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.

The constant declaration block

Lines 59-69 establish constants relating to Menu IDs and menu item numbers. Lines 71-82 establish constants relating to various resources. Lines 77-82 are constants which index strings in a 'STR#' resource. Lines 84-85 establish constants which will enable the program to distinguish between the two "document" windows opened by the program and the Clipboard window.

The type declaration block

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

The variable declaration block

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.

The procedure 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 routine depending on the severity of the error.

The procedure 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 167-182). The graphics port of the second window created is then set as the current port (Line 184) 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 186).

The procedure 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 199 makes a local Rect variable equal to the rectangle in the picFrame field. Line 201 then offsets this rectangle to the left and top of the port rectangle. Lines 203-206 calculate the differences between the widths and heights of the rectangle and the window's port rectangle. This is used at Line 208 to further offset the rectangle to the middle of the port rectangle. The rectangle is then returned to the calling function (Line 210).

The procedure DoDrawPictureWindow

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

Lines 224-225 save the current graphics port and make the graphics port associated with the front window the current graphics port.

Line 227 gets the handle to the window's document record. Line 228 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 230 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 232), Lines 234-236 draw a dotted rectangle two pixels outside the picture.

Line 239 resets the current graphics port to the port saved at function entry.

The procedure DoDrawClipboardWindow

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

Lines 254-255 save the current graphics port and make the graphics port associated with the front window the current graphics port.

Line 257 erases the window's port rectangle. Lines 259-267 draw a panel at the top of the window in which the type of data in the desk scrap will be displayed.

Line 269, in which nil is passed as the destHandle parameter of the GetScrap call, checks whether data of type 'PICT' exists in the desk scrap. If so (Line 270), the following occurs. The word "picture" is drawn in the panel at the top of the window (Lines 272-273). A relocatable block the size of the 'PICT' data is created and locked (Lines 275-276) and GetScrap is called once again to copy the 'PICT' data from the scrap into the newly-created block (Line 278). 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 280-282). The picture is then drawn in this destination rectangle (Line 283), following which the relocatable block created at Line 275 is unlocked and disposed of (Lines 285-286).

Lines 289 checks whether data of type 'TEXT' exists in the desk scrap. If so (Line 290), 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 293), the destination rectangle is set to two pixels inside the port rectangle less the panel (Lines 300-302), and the text is drawn in this rectangle using TETextBox (Line 304). (TETextBox is a TextEdit routine, and is described at Chapter 17 – Text and TextEdit.)

Line 310 resets the current graphics port to the port saved at function entry.

The procedure DoClipboardCommand

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

Line 323 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 325 checks whether the Clipboard window has been opened. If not, the Clipboard window is opened (Line 327), a document record is created and attached to the window (Lines 331-334), the windowType field of the document record is set to indicate that the window is of the Clipboard type (Line 335), a global variable which keeps track of whether the Clipboard window is currently showing or hidden is set to true (Line 337), and the text of the menu item is set to **Hide Clipboard** (Line 339).

If the Clipboard window has previously been opened (Line 342), and if the window is currently showing (Line 343), the window is hidden, the Clipboard-showing flag is set to false, and the item's text is set to **Show Clipboard** (Lines 345-347). If the window is not currently showing (Line 349), the window is made visible, the Clipboard-showing flag is set to true, and the item's text is set to **Hide Clipboard** (Lines 351-353)

The procedure DoClearCommand

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

Note that, as is 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 369-370 get a pointer to the front window and the handle to that window's document record. Lines 372-373 save the current graphics port and make the graphics port associated with the front window the current graphics port.

Lines 375-378 dispose of the picture record, set the pictureHdl field of the document record to nil, set the selectionFlag field of the document record to false, and erase the window's port rectangle.

Line 380 resets the current graphics port to the port saved at function entry.

The procedure 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 397-398 get a pointer to the front window and the handle to that window's document record. Lines 400-401 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 403 calls GetScrap with the destHandle parameter set to nil. The following occurs if data of type 'PICT' is present in the desk scrap (Line 404).

Lines 406-407 create and lock a relocatable block of a size equivalent to the 'PICT' data in the scrap. GetScrap is called again (Line 409) to copy the 'PICT' data in the scrap to the newly-created relocatable block. Line 411 erases the front window and Line 412 sets the selectionFlag field of the document record associated with the front window to false. Line 413 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 415 draws the picture in this rectangle.

If the document record currently contains a picture, the picture record is disposed of (Lines 417-418). Line 420 creates a new relocatable block the size of the 'PICT' data and assigns its handle to the pictureHdl field of the document record. Line 421 then copies the bytes in the relocatable block created at Line 406 to this new relocatable block. Lines 423-424 unlock and dispose of the block created at Line 406.

Line 427 resets the current graphics port to the port saved at function entry.

The procedure DoCutCopyCommand

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

Lines 443-444 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 446-447). (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 449 purges the desk scrap. If the call is successful, Line 451 gets the size of the picture record, Line 452 locks the picture record, Line 454 copies the picture to the desk scrap, and Line 458 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 455-456 and Line 461).

If the menu choice was the **Cut** item (Line 463), 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 465-466). Lines 468-470 then dispose of the picture record and set the document record's pictureHdl and selectionFlag fields to NIL and false respectively. Line 471 erases the picture from the window and Line 473 resets the saved graphics port.

Finally, and importantly, if the Clipboard window has previously been opened by the user (Line 476), an application defined function is called to draw the current contents of the desk scrap in the Clipboard window (Line 477).

The procedure 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 492-493 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 495-496). Lines

498-499 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 501) the following occurs. Line 503 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 504 expands this rectangle by two pixels all around. Line 506 converts the mouse-down coordinates to local coordinates. If the mouse-down occurred within the rectangle (Line 508), the document record's selectionFlag field is set to true. If the mouse-down occurred outside that rectangle (Line 510), the document record's selectionFlag field is set to false, and the rectangle is erased (Lines 512-517).

Line 521 resets the current graphics port to that saved at function entry.

The procedure DoCloseWindow

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

Lines 535-536 get a pointer to the front window and the handle to its document record. If the window is the Clipboard window (Line 538), The window is disposed of, the global variable which contains its pointer is set to NIL, 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**.

The procedures DoEditMenu, and DoMenuChoice

DoMenuChoice and DoEditMenu handle menu choices.

The procedure DoAdjustMenus

DoAdjustMenus adjusts the menus.

Lines 642-643 get handles to the **File** and **Edit** menus. Line 645 gets the handle to the document record for the front window.

If the front window is the Clipboard window (Line 647), the **Close** item is enabled, otherwise it is disabled.

If the document contains a picture and that picture is currently selected (Line 652), the **Cut**, **Copy**, and **Clear** items are enabled, otherwise they are disabled (Lines 653-663).

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 665-667).

Line 669 redraws the menu bar.

The procedure DoOSEvent

DoOSEvent handles suspend/resume events. Line 687 sets gInBackground according to whether the event is a suspend or resume event.

Line 688 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 690), the window is hidden (Line 691). If the event is a resume event, the window is shown (Line 693).

The procedure DoUpdate

DoUpdate handles update events.

Lines 715-717 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 721-723), an application-defined function is called to draw that picture (Line 724). If the window is the Clipboard window, an application-defined function is called to draw the Clipboard window (Lines 727-728).

The procedure 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 procedure DoInContent is called (Line 762).

The procedure DoEvents

DoEvents performs initial handling of events.

The procedure 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 832-833 get a handle to the front window's document record.

If Line 835 determines that the window is the Clipboard window or the window does not contain a selected object, the function returns immediately (Line 837); otherwise, the following occurs.

Lines 841-842 save the current graphics port and set the front window's port as the current port. Line 844 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 845 expands that rectangle by 2 pixels all around.

Line 847 saves the byte in the last element of gMarqueePattern. Lines 848-849 move each byte down one element in the array. Line 850 assigns the saved byte to the first element. Line 852 assigns gMarqueePattern to the pen, whose size remains at the default one pixel throughout the program, and Line 853 draws the rectangle in the specified pattern.

Line 855 restores the save graphics port.

The main program block

The main function initialises the system software managers (Line 877), sets up the menus (Lines 881-891), opens the two document windows (Line 895), and enters the main event loop (Lines 899-908). Note that the sleep parameter in the WaitNextEvent call (Line 903) is set to 2 and that a null event will result in the application-defined function DoIdle being called (Line 907).