18

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 structure**, which is defined by the data type ScrapStuff. The information in the scrap information structure 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.
- 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 functions 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

[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.

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 function 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.

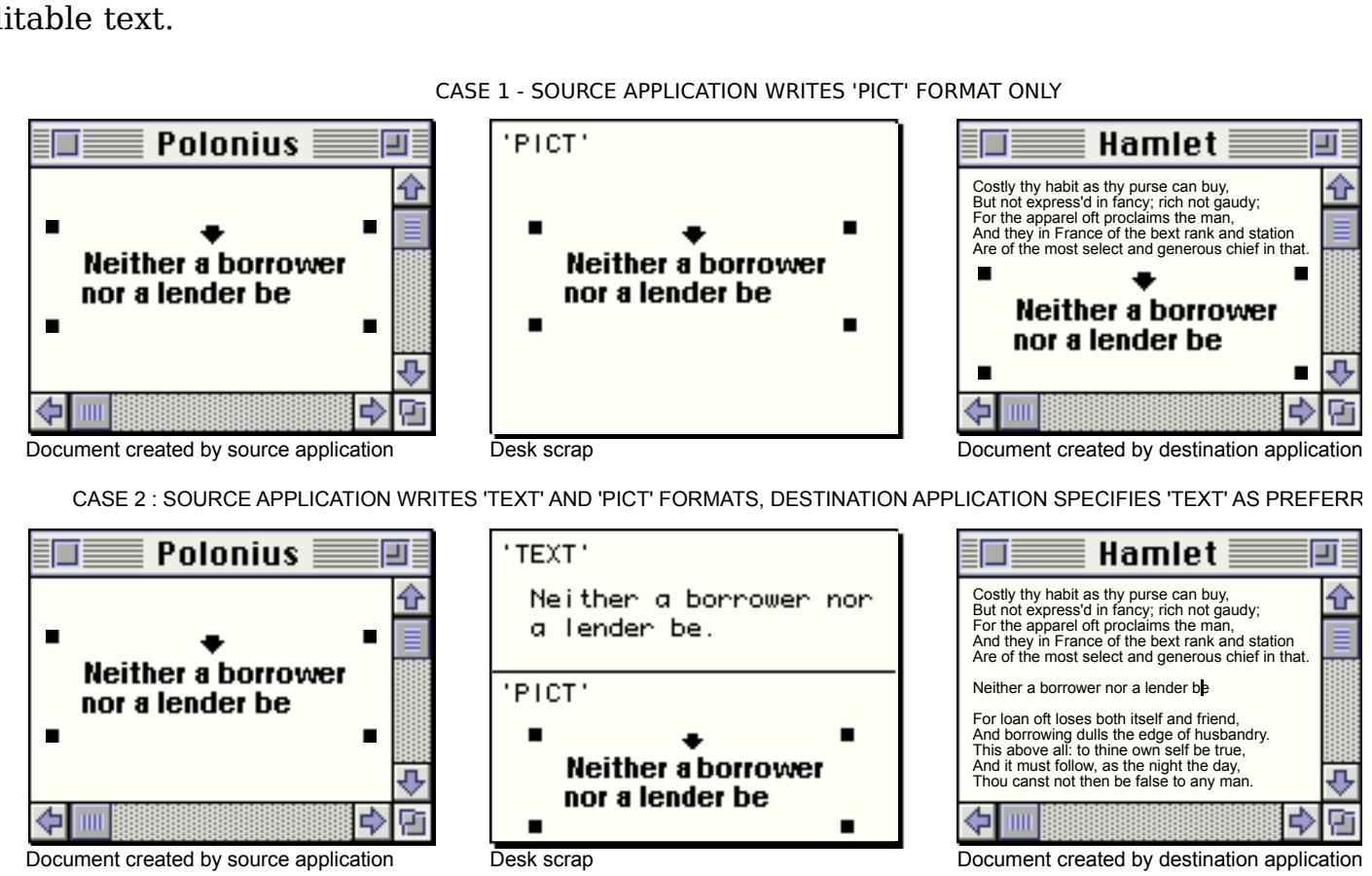

FIG 1 - SPECIFYING FORMATS TO WRITE TO AND READ FROM THE DESK SC

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 structure.

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.

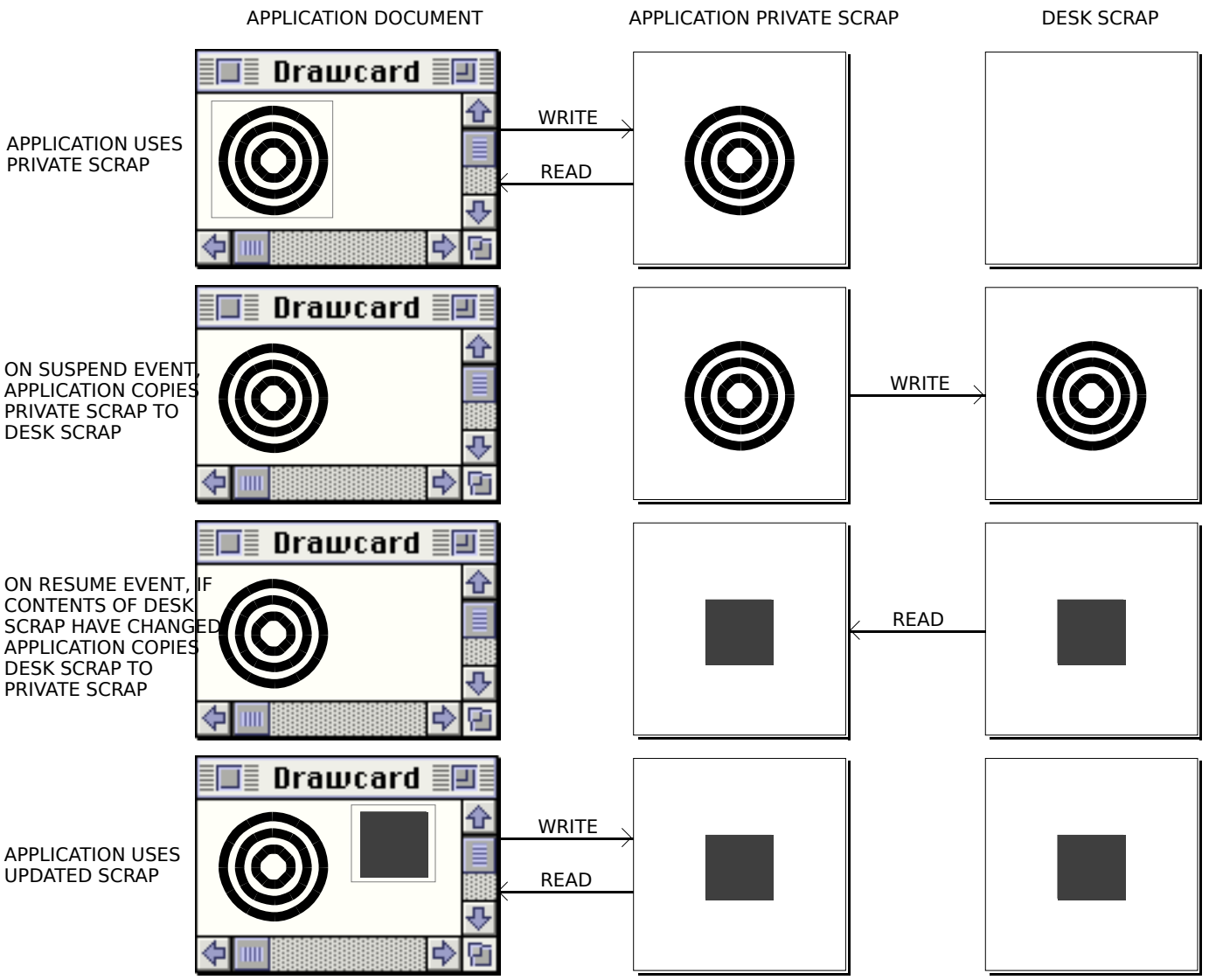

FIG 2 - USING A PRIVATE SCRAP

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 functions and data structures which you can use to provide your application with basic text editing capabilities.

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

- PutScrap is not used and the special TextEdit functions TECut, TECopy, and TEToScrap are used in the processes of cutting text from the document and copying text to the TextEdit private scrap and to the desk scrap.
- GetScrap is not used and the special TextEdit functions TEPaste, TEStylePaste, and TEFromScrap are used in the processes of pasting text from the TextEdit private scrap and copying text from the desk scrap to the TextEdit private scrap.

Chapter 19 — Text and TextEdit describes TextEdit, including the TextEdit private scrap and the TextEdit scrap-related functions.

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 functions 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 8 — 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 functions and the Dialog Manager's DialogCut, DialogCopy, and DialogPaste functions 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 Functions

Data Types

Scrap Information Structure

```
struct ScrapStuff
{
    SInt32        scrapSize;       // Size of scrap in bytes.
    Handle        scrapHandle;     // Handle to scrap.
    SInt16        scrapCount;      // Indicates whether contents of scrap have changed.
    SInt16        scrapState;      // Indicates state and location of scrap
    StringPtr     scrapName;       // Filename of scrap.
};
typedef struct ScrapStuff ScrapStuff;
typedef ScrapStuff *PScrapStuff;
typedef ScrapStuff *ScrapStuffPtr;
```

Functions

Getting Information About the Scrap

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

Writing Information to the Scrap

```
SInt32            ZeroScrap(void);
SInt32            PutScrap(long length,ResType theType,void *source);
```

Reading Information From the Scrap

```
SInt32            GetScrap(Handle hDest,ResType theType,long *offset);
```

Transferring the Scrap Between Memory and Disk

```
SInt32            UnloadScrap(void);
SInt32            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 19 — Text and TextEdit
// shows how to cut, copy and paste text from and to a TextEdit edit structure 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.
//
// •   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 <Appearance.h>
#include <Devices.h>
#include <LowMem.h>
#include <Scrap.h>
#include <Sound.h>
#include <ToolUtils.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 rErrorStrings            128
#define  eFailMenu               1
#define  eFailWindow             2
#define  eFailDocStruc     3
#define  eZeroScrap              4
#define  ePutScrap               5
#define  eNoPictInScrap          6
#define kDocumentType              1
#define kClipboardType           2
#define MAXLONG                        0x7FFFFFFF

// ..........................................................................................................................................................................
.......................................... typedefs

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

// ..........................................................................................................................................................................
.................. global variables
```

```
Boolean        gDone;
Boolean        gInBackground;
SInt16      gPixelDepth;
Boolean        gIsColourDevice           = false;
WindowPtr    gWindowPtrs[2];
WindowPtr    gClipboardWindowPtr   = NULL;
Boolean        gClipboardShowing       = false;

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

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

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

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

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

    doInitManagers();

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

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

    menuHdl = GetMenuHandle(mApple);
    if(menuHdl == NULL)
        doErrorAlert(eFailMenu);
    else
        AppendResMenu(menuHdl,'DRVR');

    //
………………………………………………………………………………………………………………………………………………………………
…………………… open windows

    doOpenWindows();

    // …………………………… get pixel depth and whether colour device for function SetThemeTextColor

    doGetDepthAndDevice();

    //
………………………………………………………………………………………………………………………………………………………………
…………… enter eventLoop

    gDone = false;

    while(!gDone)
```

```
  {
    gotEvent = WaitNextEvent(everyEvent,&eventStructure,MAXLONG,NULL);

    if(gotEvent)
      doEvents(&eventStructure);
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInitManagers

void  doInitManagers(void)
{
  MaxApplZone();
  MoreMasters();

  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(NULL);

  InitCursor();
  FlushEvents(everyEvent,0);

  RegisterAppearanceClient();
}

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

void  doEvents(EventRecord *eventStrucPtr)
{
  WindowPtr    windowPtr;
  SInt16       partCode;
  SInt8        charCode;

  switch(eventStrucPtr->what)
  {
    case mouseDown:
      partCode = FindWindow(eventStrucPtr->where,&windowPtr);

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

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

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

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

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

    case updateEvt:
      doUpdate(eventStrucPtr);
      break;
```

```
        case activateEvt:
            windowPtr = (WindowPtr) eventStrucPtr->message;
            if(windowPtr == gClipboardWindowPtr)
                doDrawClipboardWindow();
            break;

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
    WindowPtr           windowPtr;
    docStructureHandle  docStrucHdl;
    SInt32              windowType;

    windowPtr = (WindowPtr) eventStrucPtr->message;
    docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
    windowType = (*docStrucHdl)->windowType;

    BeginUpdate(windowPtr);

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

    EndUpdate(windowPtr);
}

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

void  doOSEvent(EventRecord *eventStrucPtr)
{
    switch((eventStrucPtr->message >> 24) & 0x000000FF)
    {
        case suspendResumeMessage:
            gInBackground = (eventStrucPtr->message & resumeFlag) == 0;
            if(gClipboardWindowPtr && gClipboardShowing)
            {
                if(gInBackground)
                    ShowHide(gClipboardWindowPtr);
                else
                    ShowWindow(gClipboardWindowPtr);
            }
            break;

        case mouseMovedMessage:
            break;
    }
}

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

void  doAdjustMenus(void)
{
    MenuHandle          fileMenuHdl, editMenuHdl;
    docStructureHandle  docStrucHdl;
    SInt32              scrapOffset;

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

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

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

```
  if((*docStrucHdl)->pictureHdl && (*docStrucHdl)->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)  && (*docStrucHdl)->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:
      switch(menuItem)
      {
        case iCut:
          doCutCopyCommand(true);
          break;

        case iCopy:
          doCutCopyCommand(false);
          break;

        case iPaste:
          doPasteCommand();
          break;

        case iClear:
          doClearCommand();
          break;

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

```
    HiliteMenu(0);
}

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

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

    paramRec.movable          = true;
    paramRec.helpButton       = false;
    paramRec.filterProc       = NULL;
    paramRec.defaultText      = (StringPtr) kAlertDefaultOKText;
    paramRec.cancelText       = NULL;
    paramRec.otherText        = NULL;
    paramRec.defaultButton    = kAlertStdAlertOKButton;
    paramRec.cancelButton     = 0;
    paramRec.position         = kWindowDefaultPosition;

    GetIndString(errorString,rErrorStrings,errorCode);

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

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

void  doOpenWindows(void)
{
    SInt16              a;
    WindowPtr           windowPtr;
    docStructureHandle  docStrucHdl;

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

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

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

    SetPort(windowPtr);

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

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

void  doCloseWindow(void)
{
    WindowPtr           windowPtr;
    docStructureHandle  docStrucHdl;
    MenuHandle          editMenuHdl;

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

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

```
   }
}

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

void  doInContent(Point mouseXY)
{
   WindowPtr            windowPtr;
   docStructureHandle  docStrucHdl;
   GrafPtr              oldPort;
   Rect                 pictRect;

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

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

   GetPort(&oldPort);
   SetPort(windowPtr);

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

      GlobalToLocal(&mouseXY);

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

   SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCutCopyCommand

void  doCutCopyCommand(Boolean cutFlag)
{
   WindowPtr            windowPtr;
   docStructureHandle  docStrucHdl;
   Size                 dataLength;
   SInt32               errorCode;
   GrafPtr              oldPort;

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

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

   if(ZeroScrap() == noErr)
   {
      dataLength = GetHandleSize((Handle) (*docStrucHdl)->pictureHdl);
      HLock((Handle) (*docStrucHdl)->pictureHdl);

      errorCode = PutScrap((SInt32) dataLength,'PICT',*((Handle) (*docStrucHdl)->pictureHdl));
      if(errorCode != noErr)
         doErrorAlert(ePutScrap);

      HUnlock((Handle) (*docStrucHdl)->pictureHdl);
   }
   else
      doErrorAlert(eZeroScrap);

   if(cutFlag)
   {
      GetPort(&oldPort);
      SetPort(windowPtr);

      DisposeHandle((Handle) (*docStrucHdl)->pictureHdl);
      (*docStrucHdl)->pictureHdl = NULL;
```

```
    (*docStrucHdl)->selectFlag = false;
    EraseRect(&windowPtr->portRect);

    SetPort(oldPort);
  }

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doPasteCommand

void  doPasteCommand(void)
{
  WindowPtr              windowPtr;
  docStructureHandle docStrucHdl;
  GrafPtr              oldPort;
  SInt32               sizeOfPictData, scrapOffset;
  Handle               tempHdl;
  Rect                 destRect;

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

  GetPort(&oldPort);
  SetPort(windowPtr);

  sizeOfPictData = GetScrap(NULL,'PICT',&scrapOffset);
  if(sizeOfPictData > 0)
  {
    tempHdl = NewHandle((Size) sizeOfPictData);
    HLock(tempHdl);

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

    EraseRect(&windowPtr->portRect);
    (*docStrucHdl)->selectFlag = false;
    destRect = doSetDestRect(&(*(PicHandle) tempHdl)->picFrame,windowPtr);

    DrawPicture((PicHandle) tempHdl,&destRect);

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

    (*docStrucHdl)->pictureHdl = (PicHandle) NewHandle((Size) sizeOfPictData);
    BlockMoveData(*tempHdl,*((*docStrucHdl)->pictureHdl),(Size) sizeOfPictData);

    HUnlock(tempHdl);
    DisposeHandle(tempHdl);
  }

  SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doClearCommand

void  doClearCommand(void)
{
  WindowPtr              windowPtr;
  docStructureHandle docStrucHdl;
  GrafPtr              oldPort;

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

  GetPort(&oldPort);
  SetPort(windowPtr);

  DisposeHandle((Handle) (*docStrucHdl)->pictureHdl);
  (*docStrucHdl)->pictureHdl = NULL;
  (*docStrucHdl)->selectFlag = false;
  EraseRect(&windowPtr->portRect);

  SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doClipboardCommand

void  doClipboardCommand(void)
```

```
{
   MenuHandle              editMenuHdl;
   docStructureHandle  docStrucHdl;

   editMenuHdl = GetMenu(mEdit);

   if(gClipboardWindowPtr == NULL)
   {
      if(!(gClipboardWindowPtr = GetNewCWindow(rClipboardWindow,NULL,(WindowPtr)-1)))
         doErrorAlert(eFailWindow);
      if(!(docStrucHdl = (docStructureHandle) NewHandle(sizeof(docStructure))))
         doErrorAlert(eFailDocStruc);
      SetWRefCon(gClipboardWindowPtr,(SInt32) docStrucHdl);
      (*docStrucHdl)->windowType = kClipboardType;

      gClipboardShowing = true;

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

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

void  doDrawClipboardWindow(void)
{
GrafPtr     oldPort;
Rect        theRect, destRect;
RGBColor  blackColour = { 0x0000, 0x0000, 0x0000 };
SInt32      sizeOfPictData, sizeOfTextData, scrapOffset;
Handle      tempHdl;

GetPort(&oldPort);
SetPort(gClipboardWindowPtr);

EraseRect(&gClipboardWindowPtr->portRect);

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

if(gClipboardWindowPtr == FrontWindow())
   SetThemeTextColor(kThemeTextColorWindowHeaderActive,gPixelDepth,gIsColourDevice);
else
   SetThemeTextColor(kThemeTextColorWindowHeaderInactive,gPixelDepth,gIsColourDevice);

TextSize(10);
MoveTo(7,16);
DrawString("\pClipboard Contents:");

sizeOfPictData = GetScrap(NULL,'PICT',&scrapOffset);
if(sizeOfPictData > 0)
{
   MoveTo(105,16);
   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 - 26));
   DrawPicture((PicHandle) tempHdl,&destRect);
```

```
        HUnlock(tempHdl);
        DisposeHandle(tempHdl);
    }

    sizeOfTextData = GetScrap(NULL,'TEXT',&scrapOffset);
    if(sizeOfTextData > 0)
    {
        MoveTo(105,16);
        DrawString("\pText");

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

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

        destRect = gClipboardWindowPtr->portRect;
        destRect.top += 24;
        InsetRect(&destRect,2,2);

        RGBForeColor(&blackColour);

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

        HUnlock(tempHdl);
        DisposeHandle(tempHdl);
    }

    SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDrawPictureWindow

void  doDrawPictureWindow(WindowPtr windowPtr)
{
    GrafPtr             oldPort;
    Rect                destRect;
    docStructureHandle  docStrucHdl;

    GetPort(&oldPort);
    SetPort(windowPtr);

    docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
    destRect = doSetDestRect(&(*(*docStrucHdl)->pictureHdl)->picFrame,windowPtr);

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

    if((*docStrucHdl)->selectFlag)
        InvertRect(&destRect);

    SetPort(oldPort);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ 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);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doGetDepthAndDevice

void doGetDepthAndDevice(void)
{
    GDHandle deviceHdl;
```

```
    deviceHdl = LMGetMainDevice();
    gPixelDepth = (*(*deviceHdl)->gdPMap)->pixelSize;
    if(BitTst(&(*deviceHdl)->gdFlags,gdDevType))
      gIsColourDevice = true;
}

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

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 picture inverts when the user selects it with the mouse.) 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

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

#typedef

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

Global Variables

gDone controls program termination. gInBackground relates to foreground/background switching. The values assigned to gPixelDepth and gIsColourDevice will be used by the Appearance Manager function SetThemeTextColor.

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.

doEvents

Note that, in the case of an activate event for the Clipboard window, the application-defined function doDrawClipBoardWindow is called.

doUpdate

If the window is of the document type (as opposed to the Clipboard type), and if the window's document structure currently contains a picture, the application-defined function doDrawPicture is called to draw that picture. If the window is the Clipboard window, the application-defined function doDrawClipboardWindow is called to draw the Clipboard window.

doOSEvent

The if statement tests whether the Clipboard window has been opened by the user and whether it 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, the window is hidden. If the event is a resume event, the window is shown. Note that ShowHide, rather than HideWindow is called to hide the window. This is to prevent activation of the first document window in the list when the Clipboard window is in front and the application is switched out.

doAdjustMenus

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

doOpenWindows

doOpenWindows opens the two document windows, creates document structures for each window, attaches the document structures to the windows and initialises the fields of the document structures. The graphics port of the second window

created is then set as the current port and a picture is read in from a resource, its handle being assigned to the pictureHdl field of the second window's document structure.

doCloseWindow

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

If the window is the Clipboard window, The window is disposed of, the global variable which contains its 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.

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.

The first two lines get a pointer the front window and a handle to its document structure. If the front window is the Clipboard window, the function returns immediately. GetPort and SetPort 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 the following occurs. The application-defined function doSetDestRect is called to return a rectangle of the same dimensions as that contained in the picture structure's picFrame field, but centred laterally and vertically in the window. GlobalToLocal converts the mouse-down coordinates to local coordinates preparatory to the call to PtInRect. If the mouse-down occurred within the rectangle and the picture has not yet been selected, the document structure's selectFlag field is set to true and the picture is inverted. If the mouse-down occurred outside the rectangle and the picture is currently selected, the document structure's selectFlag field is set to false, and the picture is re-inverted.

doCutCopyCommand

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

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

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

ZeroScrap attempts to purge the desk scrap. If the call is successful, GetHandleSize gets the size of the picture structure, HLock locks the picture structure, PutScrap copies the picture to the desk scrap, and HUnlock unlocks the picture structure. If the calls to ZeroScrap and PutScrap are not successful, a caution alert is displayed to advise the user of the error.

If the menu choice was the Cut item, additional action is taken. Preparatory to a call to EraseRect, the current graphics port is saved and the front window's port is made the current port. DisposeHandle is called to dispose of the picture structure and the document structure's pictureHdl and selectFlag fields are set to NULL and false respectively. EraseRect then erases the picture.

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

doPasteCommand

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

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

NewHandle and HLock create and lock a relocatable block of a size equivalent to the 'PICT' data in the scrap. GetScrap is called again to copy the 'PICT' data in the scrap to the newly-created relocatable block. EraseRect erases the front window and the next line sets the selectFlag field of the document structure associated with the front window to false. The call to the application-defined function doSetDestRect returns a destination rectangle of the same dimensions as the picFrame rectangle but centred in the front window. DrawPicture draws the picture in this rectangle.

If the document structure currently contains a picture, DisposeHandle is called to dispose of the picture structure. NewHandle creates a new relocatable block the size of 'PICT' data and assigns its handle to the pictureHdl field of the document structure. BlockMoveData then copies the bytes in the relocatable block created at the first line in the if block to this new relocatable block. HUnlock and DisposeHandle then dispose of the block created at the first line in the if block.

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.

DisposeHandle dispose of the picture structure. The next two three lines set the pictureHdl field of the document structure to NULL, set the selectFlag field of the document structure to false, and erase the window's port rectangle.

doClipboardCommand

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

The first line 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.

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

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

doDrawClipboardWindow

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

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

DrawThemeWindowHeader draws a window header in the top of the window. Text describing the type of data in the desk scrap will be drawn in this header. The theme-compliant colour for this text is set at the next four lines. The following three lines set the text size to 10pt and draw some text in the header.

The call to GetScrap, with NULL passed as the destHandle parameter, checks whether data of type 'PICT' exists in the desk scrap. If so, the following occurs. The word "Picture" is drawn in the window header. A relocatable block the size of the 'PICT' data is created and locked and GetScrap is called once again to copy the 'PICT' data from the scrap into the newly-created block. A destination rectangle, based on the rectangle in the picFrame field of the picture structure, is created with its left and top fields set to two pixels inside that part of the content area not occupied by the window header. The picture is then drawn in this destination rectangle, following which the relocatable block created earlier is unlocked and disposed of.

The next call to GetScrap checks whether data of type 'TEXT' exists in the desk scrap. If so, much the same procedure is followed, the differences being that the word "Text" is drawn in the window header, the destination rectangle is set to two pixels inside that part of the content area not occupied by the window header., and the text is drawn in this rectangle using TETextBox. (TETextBox is a TextEdit routine, and is described at Chapter 19 — Text and TextEdit.)

doDrawPictureWindow

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

The third line gets the handle to the window's document structure. The call to the application-defined function doSetDestRect returns a rectangle of the same dimensions as that contained in the picFrame field of the picture structure (the handle to which is contained in the pictureHdl field of the document structure), but centred in the window. DrawPicture draws the picture specified in the window's document structure in this rectangle.

If the selectFlag field of the document structure indicates that the picture is currently selected, the call to InvertRect inverts the picture.

doSetDestRect

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

The first line makes a local Rect variable equal to the rectangle in the picFrame field. OffsetRect then offsets this rectangle so that its left and top fields both contain 0. The next four lines calculate the differences between the widths and heights of the rectangle and the window's port rectangle. This is used at the next call to OffsetRect to further offset the rectangle to the middle of the port rectangle. The rectangle is then returned to the calling function.