CHAPTER 2

Scrap Manager

2

Contents

This chapter describes how your application can allow the user to cut, copy, and paste data between documents or within a document by using the Scrap Manager. When you copy data, your application writes the data to a specific location, and your application writes the data using a standard format. The Scrap Manager makes this data available to other applications. Furthermore, when your application copies data such as text or graphics, you write the data using the standard formats that all Macintosh applications should support. By using standard formats, the user can copy and paste data between documents created by your application and others.

The Scrap Manager supports the sharing of static data between applications. That is, once the data is pasted into another document, there is no connection between the data that was pasted and the original source of the data. To support dynamic sharing of data, where the user can copy data from one document into another document and receive automatic updating of the information when the data in the original document changes, use the Edition Manager. See *Inside Macintosh: Interapplication Communication* for information on the Edition Manager.

You can also support the copying and pasting of sounds, movies, publishers or subscribers, and other formats. For specific information on supporting sounds and movies, see *Inside Macintosh: Sound* and *Inside Macintosh: QuickTime*, respectively. For information on supporting publishers and subscribers, see the chapter "Edition Manager" in *Inside Macintosh: Interapplication Communication*.

If the Translation Manager is available, the Scrap Manager uses its services as necessary to translate data in one format into another format. For specific information on the Translation Manager, see the chapter "Translation Manager" in this book.

If your application uses only TextEdit for all text input, you can use TextEdit routines to cut, copy, and paste data. For complete information on TextEdit, see the chapter "TextEdit" in *Inside Macintosh: Text*.

To support the copying and pasting of data in dialog boxes, use Dialog Manager routines. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to create and handle dialog boxes.

This chapter discusses the Edit menu commands Cut, Copy, and Paste. For specific information on how to create and handle menus in your application, see the chapter "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

To use this chapter, you should be familiar with the Event Manager, in particular, how to handle suspend and resume events. See the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for additional information on the Event Manager.

This chapter begins by describing how the copy-and-paste operation works and the user interface behind it. The chapter then discusses how you can

- get information about the current contents of the scrap
- read data from the scrap
- write data to the scrap

Introduction to the Scrap Manager

You can use the Scrap Manager 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

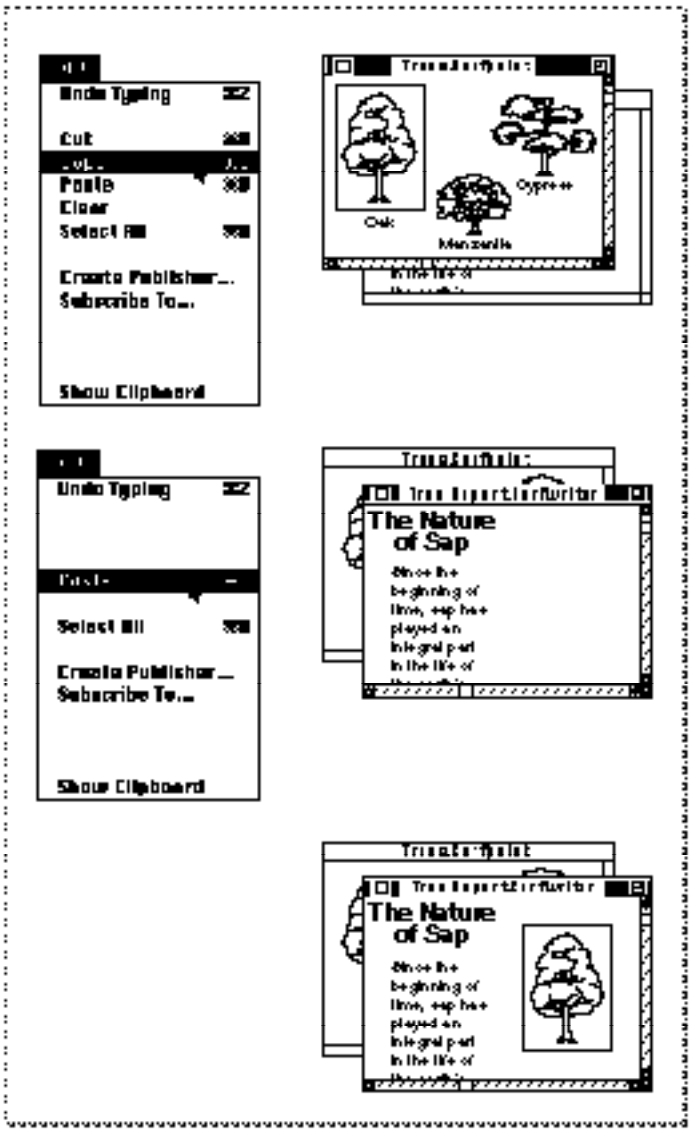
Figure 2-1 shows two documents from two applications (SurfPaint and SurfWriter) that the user currently has open. The user can select the data to copy from the SurfPaint document, choose Copy from the Edit menu, activate the SurfWriter document, then choose Paste from the Edit menu.

In the example shown in Figure 2-1, when the user chooses Copy, the SurfPaint application writes the selected data to the scrap. When the user chooses Paste, the SurfWriter application reads any data from the scrap and inserts the data at the current insertion point.

For each application, the Scrap Manager maintains a storage area to hold the last data cut or copied by the user. The area that is available to an application for this purpose is referred to as the **scrap.** The scrap can reside either in memory or on disk. All applications that support copy-and-paste operations read data from and write data to the scrap.

Whenever the user cuts or copies data, your application should write the data to the scrap (replacing the previous contents of the scrap); and whenever the user pastes data, your application should read the data from the scrap. Alternatively, your application can choose to use its own private scrap, and only write data to and read data from the scrap when necessary. If you use a private scrap, you must copy the data from your private scrap to the scrap upon receiving a suspend event. Upon receiving a resume event you should determine whether the data in the scrap has changed and, if so, either immediately copy the data from the scrap to your private scrap or copy the data from the scrap to your private scrap when the user next chooses the Paste command.

Figure 2-1 Copying and pasting data between two applications using the scrap



You use the Edit menu commands Cut, Copy, and Paste to support cutting, copying, and pasting of data within a document and between documents. Table 2-1 describes the actions your application should perform to support these three commands.

Table 2-1 Actions your application perfoms in response to editing commands

Edit command	Actions your application performs
Cut	Remove the data in the current selection (if any) and save the data, either in the scrap or in your application's private scrap.
Copy	Copy the data in the current selection (if any) and save the copied data, either in the scrap or in your application's private scrap.
Paste	Paste the last data (if any) that the user cut or copied (you get the data to paste by reading the scrap or your application's private scrap). Paste the data at the insertion point, replacing any current selection.

You should implement the editing commands as described in Table 2-1 so that when the user chooses the Paste command—whether applied to the same document or another, in the same application or another—the data last operated upon by the user (cut or copied) can be inserted into the current document. Note that if your application implements the Clear command, in response to the Clear command your application should remove the data in the current selection but should not save the data into the scrap.

The nature of the data that the user can transfer varies between the application that the user copies data from and the application that the user pastes data into. The amount of information retained also depends on the capabilities of the applications supporting the copy-and-paste operation. For example, an application that allows a user to record and edit sounds may write a copied sound to the scrap both in `'snd '` and `'TEXT'` formats. Other applications choose which format to read from the scrap. A word processor that attempts to paste the sound data may not be able to read the sound in the `'snd '` format but should be able to read the data in the `'TEXT'` format.

You write data to the scrap using the standard formats that all Macintosh applications should support: `'PICT'` and `'TEXT'`. These scrap format types are defined as follows:

- `'TEXT'`: a series of ASCII characters
- `'PICT'`: a QuickDraw picture, which is a saved sequence of drawing commands that can be played back with the `DrawPicture` procedure

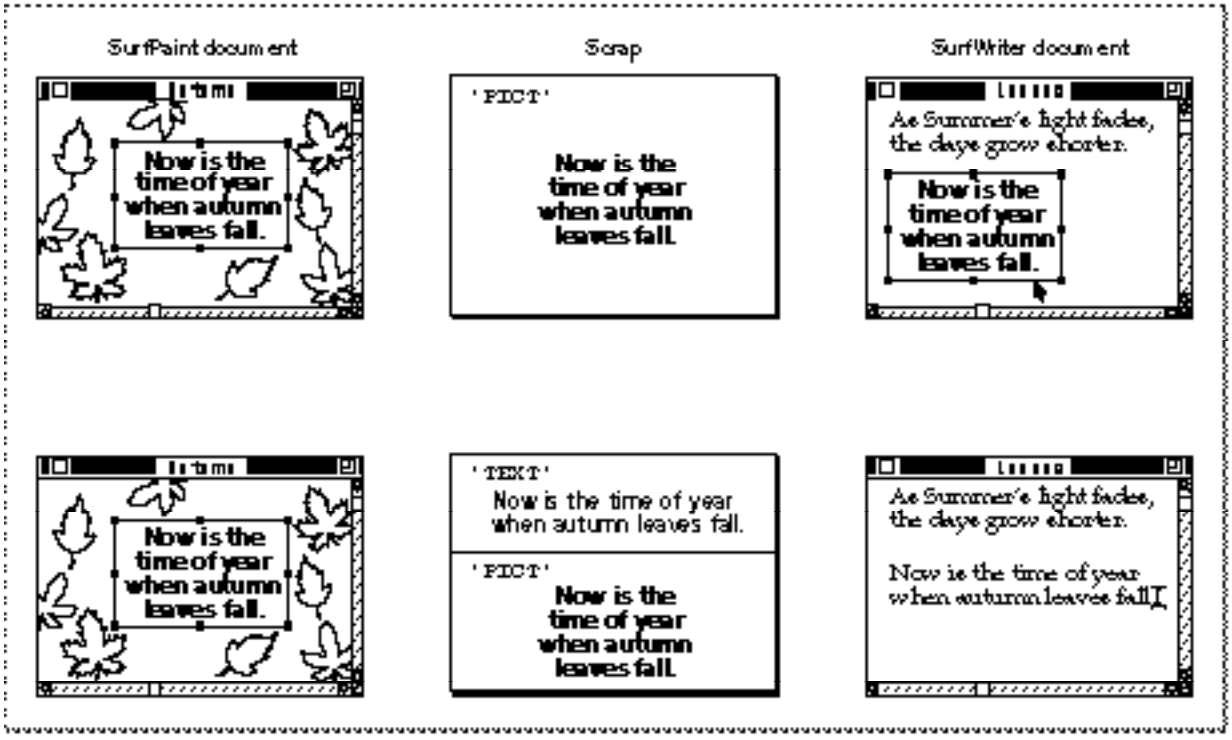
Your application must be able to write at least one standard format (`'PICT'` or `'TEXT'`) to the scrap and should be able to read both. In addition, your application can support other optional popular scrap format types (such as `'snd '` or `'movv'`). Your application can also write its own private format to the scrap, but must always write one of the standard formats as well.

When your application requests data from the scrap, it must specify the scrap format type that the Scrap Manager should retrieve from the scrap. Your application typically requests its preferred scrap format first; if that format isn't available, it requests the data specifying another format type.

If you request a scrap format type that isn't in the scrap and the Translation Manager is available, the Scrap Manager uses the Translation Manager to attempt to convert the data of a scrap format type that does exist in the scrap into the scrap format type requested by your application. For example, if the SurfWriter application requests data from the scrap in the `'SURF'` scrap format type, and the data in the scrap is available in the format types `'TEXT'`, `'PICT'`, and `'SDBS'`(SurfDB's private scrap format type), the Scrap Manager uses the Translation Manager to convert any one of the scrap format types `'TEXT'`, `'PICT'`, or `'SDBS'` into the `'SURF'` scrap format type. The Translation Manager looks in the Extensions folder for a translator that can perform one of these translations. If such a translator is available (for example, a translator that can translate the `'SDBS'` scrap format type into the `'SURF'` scrap format type), the Translation Manager uses the translator to translate the data in the scrap into the requested scrap format type. If the translation is successful, the Scrap Manager returns to your application the data from the scrap in the requested scrap format type.

Whenever possible, your application should write both of the standard data types to the scrap. For example, a graphics application, such as SurfPaint, can choose to write both `'PICT'` and `'TEXT'` formats to the scrap when the user copies a picture consisting of text. Figure 2-2 shows a SurfPaint document and a SurfWriter document. The user copies, then pastes, a picture consisting of text. The SurfPaint application can choose to write only the `'PICT'` format; if it does so, then SurfWriter reads the data from the scrap in `'PICT'` format and inserts the data as a picture in the SurfWriter document. If the SurfPaint document writes both `'PICT'` and `'TEXT'` formats to the scrap, SurfWriter can choose which format to read. In this case, SurfWriter can choose to read the `'TEXT'` format of the data and insert the data as editable text into the document.

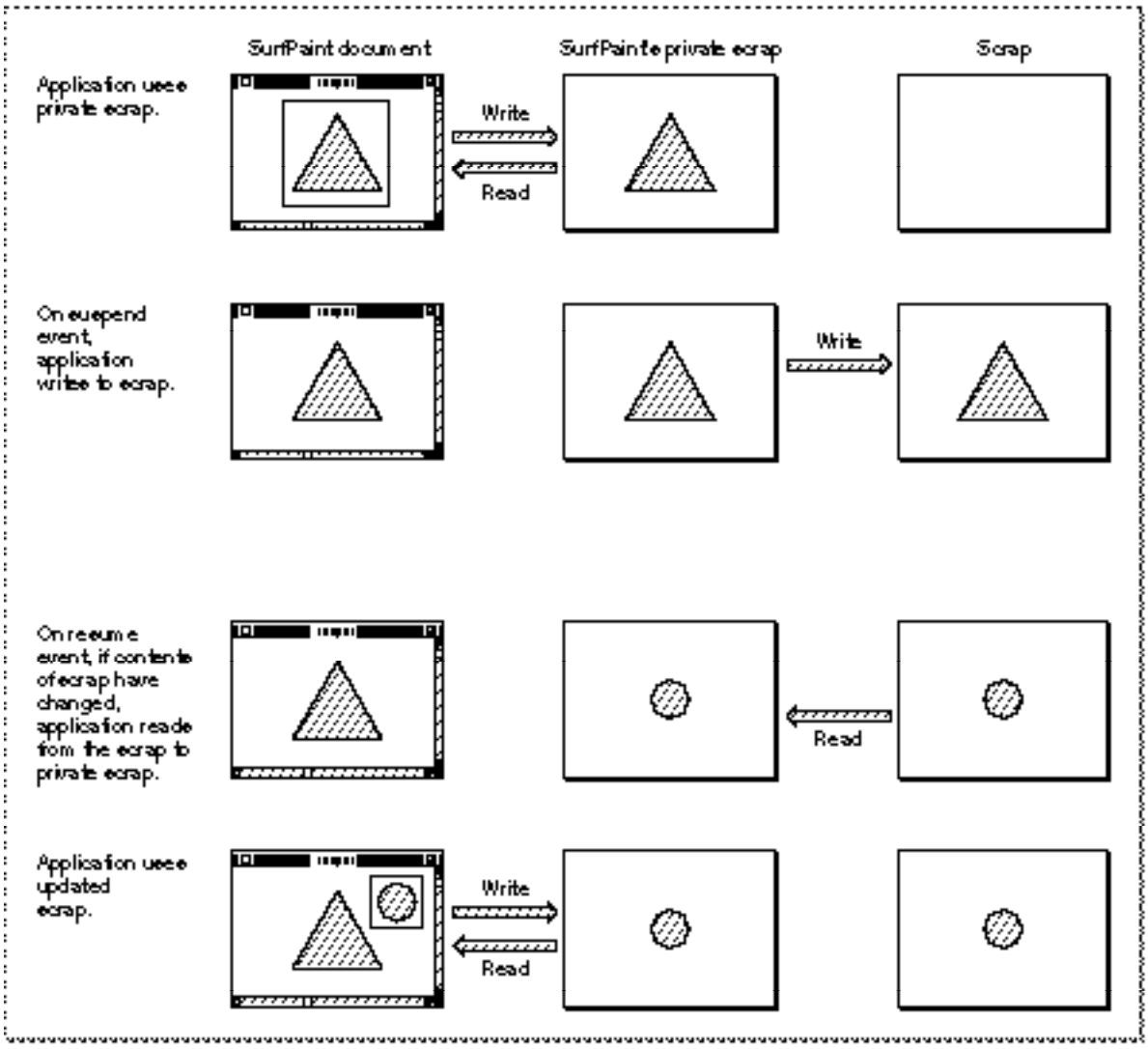
Figure 2-2 Writing both standard formats to the scrap



The SurfPaint application uses an application-defined data type to describe the data in its documents. It uses this same format in its private scrap; this implementation works well as long as the user is working exclusively with SurfPaint documents. When the SurfPaint application receives a suspend event, indicating that another application is about to become the foreground process, SurfPaint copies the data from its private scrap to the scrap. SurfPaint writes data to the scrap in its own private format (`'SFPN'`), in `'PICT'` format, and if the picture contains text, it writes the data to the scrap in `'TEXT'`

format as well. Upon receiving a resume event, SurfPaint determines whether the contents of the scrap have changed and if so, copies the new data from the scrap into its private scrap. Figure 2-3 shows how the SurfPaint application uses its own private scrap.

Figure 2-3 Using a private scrap



Note that when your application receives a resume event, it should determine whether the contents of the scrap have changed. If your application uses a private scrap, either you can choose to copy the data from the scrap to your private scrap immediately or you can delay copying until the data is needed.

If your application writes data to the scrap in more than one format, it should write the data in its order of preference. For example, the SurfPaint application writes its preferred scrap format type first (its own private format, `'SFPN'`), then writes the data in `'PICT'` format, and then, if appropriate, writes the data in `'TEXT'` format. However, the size of the scrap is limited; therefore, when your application needs to write a large amount of data to the scrap and there isn't enough room in the scrap for both your application's private scrap format type and one of the standard formats, write the data in the standard format.

As previously described, the Scrap Manager uses the Translation Manager (if it's available) to convert data in one scrap format type into another. If your application writes its own private scrap format type to the scrap, you may want to provide one or more translators that translate your private scrap format type into other format types. See the chapter "Translation Manager" in this book for information on how to write translators.

The Clipboard

The Clipboard refers to what the user views as residing in the scrap. Your application can provide a Show Clipboard / Hide Clipboard command to show or hide a window, referred to as the Clipboard window. When the user chooses this command, your application should display in its Clipboard window the current contents of the scrap. Although multiple scrap format types may reside in the scrap, applications that support a Clipboard window typically display the data in only one format.

If your application provides this command, your application should hide its Clipboard window (if it's showing) whenever it receives a suspend event. It can show the Clipboard window again when it receives a resume event.

Intelligent Cut and Paste

When the user selects text and then chooses the Cut command, or sets the insertion point and then chooses Paste, your application should apply "intelligent cut and paste," that is, discard extra spaces or add spaces, as outlined here. In general, your application should follow these rules to provide intelligent cut and paste:

- If the user selects a word or range of words, highlight the selection but not any adjacent spaces.
- When the user chooses the Cut command, if the character to the left of the selection is a space, discard it. Otherwise, if the character to the right of the selection is a space, discard it.
- When the user chooses the Paste command, if the character to the left or right of the current selection or if the character to the left or right of the insertion point is part of a word, insert a space before pasting the text.

Figure 2-4 shows examples of intelligent cut and paste.

Figure 2-4 Intelligent cut and paste

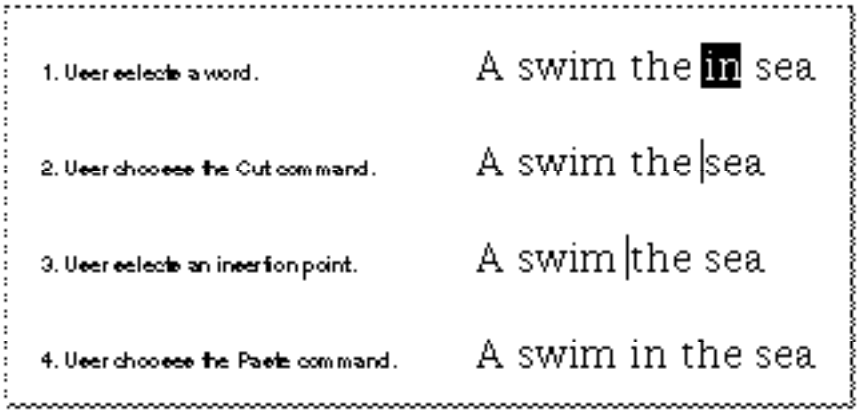
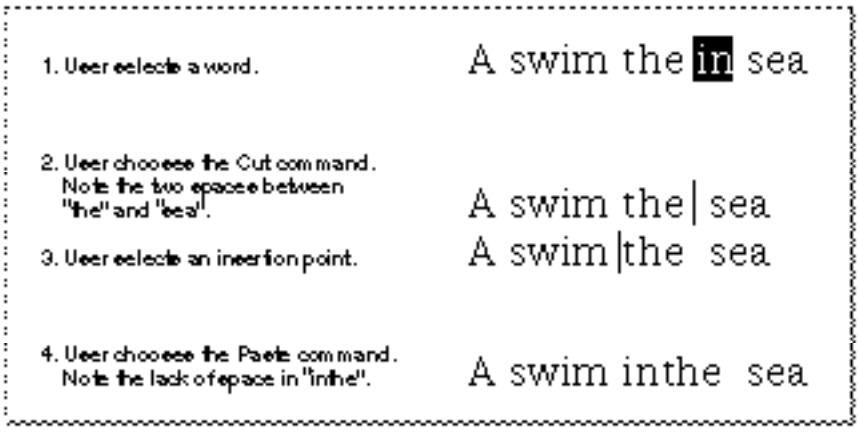


Figure 2-5 shows the results of applying the same operations in an application that doesn't support intelligent cut and paste.

Figure 2-5 Non-intelligent cut and paste



See *Macintosh Human Interface Guidelines* for details of selection techniques and guidelines for selecting words and paragraphs.

About the Scrap Manager

You can use the Scrap Manager to support copying and pasting of data. If your application uses TextEdit (in its windows or dialog boxes), be aware that TextEdit also maintains its own private scrap. You use TextEdit routines to copy data from the TextEdit scrap (if any) to the scrap. See "Converting Data Between the TextEdit Scrap and the Scrap" beginning on page 2-28 for information on TextEdit's scrap.

The next section describes the location of the scrap. "Using the Scrap Manager" beginning on page 2-14 provides specific information on how you can use Scrap Manager routines in your application.

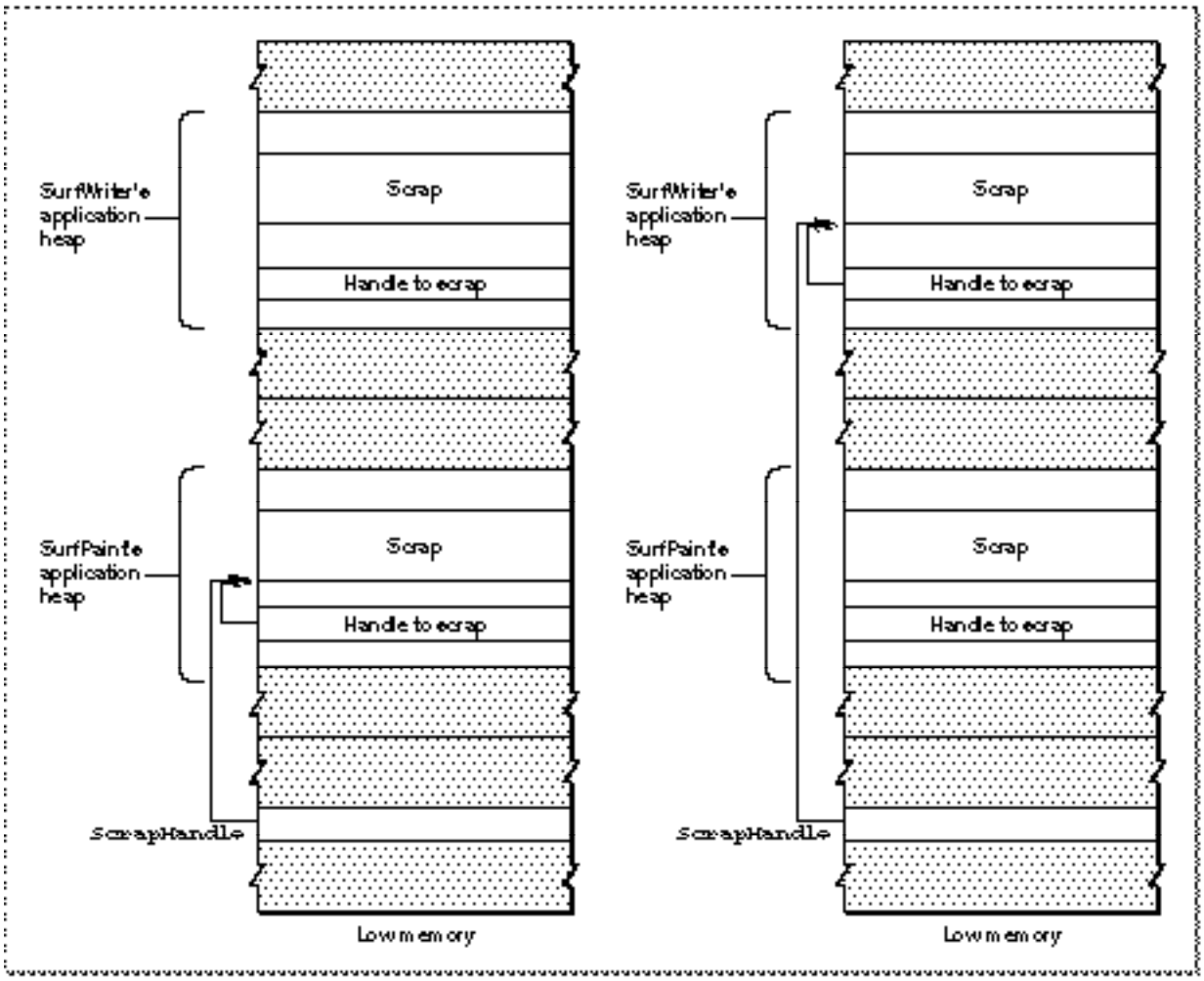
Location of the Scrap

System software allocates space in each application's heap for the scrap and allocates a handle to reference the scrap. The system global variable `ScrapHandle` contains a handle to the 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 heap to `NIL` to indicate that the scrap is on disk.

Figure 2-6 shows two applications (SurfWriter and SurfPaint) that are in memory and shows the handles and allocated space for the scrap in each application's heap. In this example, SurfPaint was the previously active application and the user switches to the SurfWriter application. At this moment, the system global variable `ScrapHandle` references the scrap in SurfPaint's application heap. SurfPaint's application heap contains a handle to the scrap in its application heap.

System software sends SurfPaint a suspend event to begin the switch to the SurfWriter application. Because SurfPaint uses a private scrap, upon receiving the suspend event it copies data from its private scrap to the scrap. After SurfPaint responds to the suspend event, system software copies the data from the scrap in SurfPaint's application heap to SurfWriter's application heap, resizing the scrap in SurfWriter's application heap as necessary. System software sets the handle in SurfWriter's application heap to reference the new scrap and sets the system global variable `ScrapHandle` to reference the scrap in SurfWriter's application heap. System software sends SurfWriter a resume event and sets the `convertClipboardFlag` bit in the `message` field of the event record. System software sets this bit when the contents of the scrap have changed since the previous suspend event, indicating to the application that it should copy the scrap to its private scrap.

Figure 2-6 Location of the scrap in memory



You can get the size of the scrap and a handle to the scrap in your application's heap by calling the `InfoScrap` function.

Although the scrap is usually located in memory, your application can write the contents of the scrap in memory to a scrap file using the `UnloadScrap` function. After writing the contents of the scrap to disk, the `UnloadScrap` function releases the memory previously occupied by the scrap in your application's heap; 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 the `LoadScrap` function to read the contents of the scrap file back into memory. The `LoadScrap` function allocates memory in your application's heap for the scrap and reads the contents of the scrap on disk into memory; thereafter, any operations your application performs on data in the scrap affect the scrap in memory.

The Scrap Manager keeps track of whether the scrap is in memory or on disk and always reads data from and writes data to the scrap's current location. As a result, your application seldom needs to know the location of the scrap. Your application should use the `UnloadScrap` function only if the scrap in memory isn't large enough to hold the data you need to write to the scrap.

If your application transfers the scrap from memory to disk and is then switched to the background, system software reads the scrap from disk into the newly active application's heap. When your application returns to the foreground, system software writes the scrap from the previous application's application heap back to disk.

Using the Scrap Manager

This section explains how you can use the Scrap Manager to support copy-and-paste operations in your application. In particular, this section explains how you can

- get information about the current contents of the scrap
- handle the Cut and Copy commands
- respond to suspend events
- handle the Paste command
- respond to resume events
- use TextEdit to support the editing commands
- support copying and pasting of data in dialog boxes

The Scrap Manager uses the services of the Translation Manager (if it's available). To determine whether the Scrap Manager can use the Translation Manager, call the `Gestalt` function with the `gestaltScrapMgrAttr` selector and check the value of the `response` parameter. If the bit indicated by the constant `gestaltScrapMgrTranslationAware` is set, then the Scrap Manager uses the Translation Manager when needed to translate scrap format types.

```
CONST
gestaltScrapMgrAttr              = 'scra';{Gestalt selector for }
                                          { Scrap Mgr attributes}
gestaltScrapMgrTranslationAware  = 0;     {check this bit in the }
                                          { response parameter }
```

Getting Information About the Scrap

To get information about the scrap, you can use the `InfoScrap` function. The `InfoScrap` function returns a pointer to a scrap information record. (See "The Scrap Information Record" on page 2-32 for detailed information on the fields of this record.) The information in the scrap information record provides

- the size (in bytes) of the scrap
- a handle to the scrap if it's in memory
- a count, or number that your application can use to determine whether the contents of the scrap have changed
- the location of the scrap (whether in memory or on disk)
- the filename of the scrap when it is on the disk

For example, this code uses the `InfoScrap` function to get the size of the scrap.

```
VAR
   curScrapInfoPtr:  PScrapStuff;
   curScrapSize:     LongInt;

   curScrapInfoPtr := InfoScrap;
   curScrapSize := curScrapInfoPtr^.scrapSize;
```

Putting Data in the Scrap

Your application should write data to the scrap (or to its own private scrap) whenever the user chooses the Cut or Copy command and the document the user is working with contains a selection. In addition, if your application uses a private scrap, your application must copy the contents of its private scrap to the scrap upon receiving a suspend event. The next sections explain how to perform these tasks.

Handling the Cut Command

When the user chooses the Cut command and the document the user is working with contains a selection, your application should remove the data from the selection and save the data (either in the scrap or in your application's private scrap).

The SurfWriter application doesn't use a private scrap; whenever the user performs a cut operation, SurfWriter writes the current selection to the scrap. The SurfWriter application does define its own private scrap format type and writes this format to the scrap, along with one of the standard scrap formats. Listing 2-1 shows SurfWriter's routine for handling the Cut command (it also uses this routine for the Copy command).

Listing 2-1 Writing data to the scrap

```
PROCEDURE DoCutOrCopyCmd (cut: Boolean);
VAR
   window:              WindowPtr;
   windowType:          Integer;
   isText:              Boolean;
   ptrToScrapData:      Ptr;
   length, myLongErr:   LongInt;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      ptrToScrapData := NewPtr(kDefaultSize);
      isText := MyIsSelectionText;
      IF isText THEN {selection contains text}
      BEGIN
         MyGetSelection('SURF', ptrToScrapData, length);
         myLongErr := ZeroScrap;
         myLongErr := PutScrap(length, 'SURF', ptrToScrapData);
         IF myLongErr <> noErr THEN DoError(myLongErr);
         MyGetSelection('TEXT', ptrToScrapData, length);
         myLongErr := PutScrap(length, 'TEXT', ptrToScrapData);
         IF myLongErr <> noErr THEN DoError(myLongErr);
      END
      ELSE           {selection contains graphics}
      BEGIN
         MyGetSelection('PICT', ptrToScrapData, length);
         myLongErr := ZeroScrap;
         myLongErr := PutScrap(length, 'PICT', ptrToScrapData);
         IF myLongErr <> noErr THEN DoError(myLongErr);
      END;
      DisposePtr(ptrToScrapData);
      IF cut THEN
         MyDeleteSelection;
   END
```

```
   ELSE
   IF windowType <> kNIL THEN
   BEGIN       {window is a dialog box}
      IF cut THEN
         DialogCut(window)
      ELSE
         DialogCopy(window);
   END;
END;
```


The `DoCutOrCopyCmd` procedure first determines the type of window that is frontmost. If the frontmost window is a document window, `DoCutOrCopyCmd` uses another application-defined routine, `MyIsSelectionText`, to determine whether the current selection contains text or graphics. If the selection contains only text, SurfWriter writes the data to the scrap using two formats: its own private format (`'SURF'`) and the standard format `'TEXT'`. The `DoCutOrCopyCmd` procedure uses another application-defined routine, `MyGetSelection`, to return the current selection in a particular format. `DoCutOrCopyCmd` then calls the `ZeroScrap` function to clear the contents of the scrap. After calling `ZeroScrap`, `DoCutOrCopyCmd` calls `PutScrap`, specifying the length of the data, a pointer to the data, and identifying the scrap format type as `'SURF'`. `DoCutOrCopyCmd` then uses the `MyGetSelection` routine again, this time to return the current selection in the `'TEXT'` format type. `DoCutOrCopyCmd` calls `PutScrap` to write the data to the scrap, specifying a pointer to the data and identifying the scrap format type as `'TEXT'`.

If the selection contains a picture, `DoCutOrCopyCmd` uses the `MyGetSelection` routine to return the current selection using the `'PICT'` format type. After calling `ZeroScrap`, `DoCutOrCopyCmd` calls `PutScrap` to write the data to the scrap, specifying a pointer to the data and identifying the scrap format type as `'PICT'`.

Finally, if `DoCutOrCopyCmd` was called as a result of the user performing a cut operation, `DoCutOrCopyCmd` deletes the selection from the current document.

If the frontmost window is a dialog box, `DoCutOrCopyCmd` uses the Dialog Manager's `DialogCut` (or `DialogCopy`) procedure to write the selected data to the scrap.

Note that you should always call `ZeroScrap` before writing data to the scrap. If you write multiple formats to the scrap, call `ZeroScrap` once before you write the first format to the scrap.

You should always write data to the scrap in your application's preferred order of formats. For example, SurfWriter's preferred format for text data is its own private format (`'SURF'`), so it writes that format first and then writes the standard format `'TEXT'`.

If your application uses TextEdit in its document windows, then use the TextEdit routine `TECut` (or `TECopy`) instead of `ZeroScrap` and `PutScrap`. See Listing 2-8 on page 2-29 for an application-defined routine that uses TextEdit routines to help handle the Cut and Copy commands.

If your application uses a private scrap, then copy the selected data to your private scrap rather than to the scrap. For example, the SurfPaint application uses a private scrap. Listing 2-2 shows SurfPaint's application-defined routine that handles the Cut command by writing the selected data to its private scrap.

Listing 2-2 Writing data to a private scrap

```
PROCEDURE DoCutOrCopyCmd (cut: Boolean);
VAR
   window:           WindowPtr;
   windowType:       Integer;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      MyWriteDataToPrivateScrap;
      {reset gScrapNewData to indicate that this app's private }
      { scrap now contains the most recent data}
      IF gScrapNewData THEN
         gScrapNewData := FALSE;
      IF cut THEN
         MyDeleteSelection;
   END
   ELSE
   IF windowType <> kNil THEN
   BEGIN          {window is a dialog window}
      IF cut THEN
         DialogCut(window)
      ELSE
         DialogCopy(window);
   END;
END;
```

The application-defined `DoCutOrCopyCmd` procedure shown in Listing 2-2 calls another application-defined procedure, `MyWriteDataToPrivateScrap`, to write the data in the current selection to the application's private scrap. SurfPaint uses the application-defined global variable `gScrapNewData` to indicate when data should be read from the scrap instead of its own private scrap as a result of the user choosing the Paste command. Upon receiving a resume event, if the contents of the scrap have changed, SurfPaint sets the `gScrapNewData` global variable to `TRUE`. If the user chooses Paste and `gScrapNewData` is `TRUE`, SurfPaint reads the scrap to get the data to paste; otherwise SurfPaint reads its own private scrap to get the data to paste.

If the user chooses Cut or Copy before the next Paste command, SurfPaint writes the newly selected data to its private scrap, eliminating the need to read the previous contents of the scrap, and thus the `DoCutOrCopyCmd` procedure resets the `gScrapNewData` global variable to `FALSE`.

Handling the Copy Command

When the user chooses the Copy command and the document the user is working with contains a selection, your application should copy the selected data (without deleting it) and save the copied data (either in the scrap or in your application's private scrap). See Listing 2-1 on page 2-16, Listing 2-2 on page 2-18, and Listing 2-8 on page 2-29 for application-defined routines that handle the Copy command.

Handling Suspend Events

As previously described, if your application uses a private scrap, your application must copy the contents of its private scrap to the scrap upon receiving a suspend event. In addition, if your application supports the Show Clipboard command, it should hide the Clipboard window if it's currently showing (because the contents of the scrap may change while your application yields time to another application).

Listing 2-3 shows SurfPaint's routine that responds to suspend events (and resume events).

Listing 2-3 Copying data from the scrap in response to suspend events

```
PROCEDURE DoSuspendResumeEvent (event: EventRecord);
VAR
   currentFrontWindow: WindowPtr;
BEGIN
   currentFrontWindow := FrontWindow;
   IF (BAnd(event.message, resumeFlag) <> 0) THEN
   BEGIN                        {it's a resume event; }
   END                          { handle as shown in Listing 2-6}
   ELSE
   BEGIN                        {it's a suspend event}
                                {copy private scrap to the scrap}
      MyConvertScrap(kPrivateToClipboard);
      gInBackground := TRUE;
                                {deactivate front window}
      DoActivate(currentFrontWindow, NOT gInBackground, event);
      MyHideClipboardWindow;  {hide Clipboard window if showing}
      MyHideFloatingWindows;  {hide any floating windows}
   END;
END;
```

Listing 2-3 shows a procedure that responds to suspend and resume events. The `DoSuspendResumeEvent` procedure first gets a pointer to the front window using the Window Manager function `FrontWindow`. It then examines bit 0 of the `message` field of the event record to determine whether the event is a suspend or resume event. See Listing 2-6 on page 2-25 for details on handling resume events.

For suspend events, the `DoSuspendResumeEvent` procedure calls the application-defined `MyConvertScrap` procedure to copy the contents of its private scrap to the scrap. (See Listing 2-7 on page 2-27 for the `MyConvertScrap` procedure.) It then sets the private global flag `gInBackground` to `TRUE` to indicate that the application is in the background. It calls another application-defined routine to deactivate the application's front window. It also calls the application-defined routine `MyHideClipboardWindow` to hide the Clipboard window if it's currently showing.

Getting Data From the Scrap

Your application should read data from the scrap (or from its own private scrap) whenever the user chooses the Paste command. In addition, if your application uses a private scrap, upon receiving a resume event your application should determine whether the contents of the scrap have changed since the previous resume event, and if so, it should take the appropriate actions. The next sections explain how to perform these tasks.

Handling the Paste Command

When the user chooses the Paste command, your application should paste the data last cut or copied by the user. You should insert the new data at the current insertion point or, if a selection exists, replace the selection with the new data. You get the data to paste by reading the data from the scrap or from your application's private scrap.

When you read data from the scrap, your application should request the data in its preferred scrap format type. If that type of format doesn't exist in the scrap, then request the data in another format. For example, SurfWriter's preferred format type is `'surf'`, so it requests data from the scrap in this format. If this format isn't in the scrap, SurfWriter requests its next preferred type, `'TEXT'`. Finally, if the `'TEXT'` format isn't in the scrap, SurfWriter requests the data in the `'PICT'` format.

If your application doesn't have a preferred scrap format type, then read from the scrap each format type your application supports. Along with a pointer to the data of the requested format type, the `GetScrap` function returns an offset, a value that indicates the relative offset of the start of that format of data in the scrap. (Note that the returned value for the offset is valid only if the Translation Manager isn't available; if the Translation Manager is available, then your application should not rely on the offset value.) The format type with the lowest offset is the preferred format type of the application that put the data in the scrap; thus a format with a lower offset is more likely to contain more information than formats in the scrap with higher offsets. So when the Translation Manager isn't available, use the format with the lowest offset when your application doesn't have a particular scrap format that it prefers.

If you request a scrap format type that isn't in the scrap and the Translation Manager is available, the Scrap Manager uses the Translation Manager to convert any one of the scrap format types currently in the scrap into the scrap format type requested by your application. The Translation Manager looks for a translator that can perform one of these translations. If such a translator is available (for example, a translator that can translate the `'SDBS'` scrap format type into the `'SURF'` scrap format type), the Translation Manager uses the translator to translate the data in the scrap into the requested scrap format type. If the translation is successful, the Scrap Manager returns to your application the data from the scrap in the requested scrap format type.

Listing 2-4 shows SurfWriter's routine for handling the Paste command. The SurfWriter application doesn't use a private scrap; whenever the user performs a paste operation, SurfWriter reads the data that is to be pasted from the scrap.

For document windows, the SurfWriter application first determines whether the data in the scrap exists in its own private scrap format (`'SURF'`) by using the `GetScrap` function. If you specify a `NIL` handle as the location 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 of the specified format that exists in the scrap. If data of this format does exist, SurfWriter reads the data in this format. SurfWriter allocates the handle to hold any returned data but does not need to size the handle; `GetScrap` automatically resizes the handle passed to it to the required size to hold the retrieved data. Once the data is retrieved in `'SURF'` format, SurfWriter pastes the data into the current document.

If the scrap does not contain data in `'SURF'` format (and the available translators can't convert any of the scrap format types in the scrap to the `'SURF'` format), SurfWriter determines whether any data in `'TEXT'` format exists in the scrap. If so, SurfWriter uses `GetScrap` to retrieve the data. Once the data is retrieved in `'TEXT'` format, SurfWriter pastes the data into the current document.

If the scrap does not contain data in `'TEXT'` format, SurfWriter determines whether any data in `'PICT'` format exists in the scrap. If so, SurfWriter uses `GetScrap` to retrieve the data. Once the data is retrieved in `'PICT'` format, SurfWriter determines the destination rectangle, that is, the rectangle where the picture should be displayed, then uses the QuickDraw `DrawPicture` procedure to draw the picture in the window. SurfWriter stores a handle to this picture and sets other application-defined variables as needed.

Listing 2-4 Handling the Paste command using the scrap

```
PROCEDURE DoPasteCommand;
VAR
  window:              windowPtr;
  windowType:          LongInt;
  offset:              LongInt;
  sizeOfSurfData:      LongInt;
  sizeOfPictData:      LongInt;
  sizeOfTextData:      LongInt;
```

```
   hDest:               Handle;
   myData:              MyDocRecHnd;
   teHand:              TEHandle;
   destRect:            Rect;
   myErr:               OSErr;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN {handle Paste command in document window. Check }
         { whether the scrap contains any data. This app }
         { checks for its preferred format type, 'SURF', first}
      sizeOfSurfData := GetScrap(NIL, 'SURF', offset);
      IF sizeOfSurfData > 0 THEN
      BEGIN
         {allocate handle to hold data from scrap--GetScrap }
         hDest := NewHandle(0);  { automatically resizes it}
         HLock(hDest);
         {put data into memory referenced thru hDest handle}
         sizeOfSurfData := GetScrap(hDest, 'SURF', offset);
         {paste the data into the current document}
         MyPasteSurfData(hDest);
         HUnlock(hDest);
         DisposeHandle(hDest);
      END
      ELSE
      BEGIN    {if no 'SURF' data in scrap, check for 'TEXT'}
         sizeOfTextData := GetScrap(NIL, 'TEXT', offset);
         IF sizeOfTextData > 0 THEN
         BEGIN
            {allocate handle to hold data from scrap--GetScrap }
            hDest := NewHandle(0);  { automatically resizes it}
            HLock(hDest);
            {put data into memory referenced thru hDest handle}
            sizeOfTextData := GetScrap(hDest, 'TEXT', offset);
            {paste the text into the current document}
            MyPasteText(hDest);
            HUnlock(hDest);
            DisposeHandle(hDest);
         END
         ELSE  {if no 'TEXT' data in scrap, check for 'PICT'}
         BEGIN
            sizeOfPictData := GetScrap(NIL, 'PICT', offset);
```

```
            IF sizeOfPictData > 0 THEN
            BEGIN
               {allocate handle to hold scrap data--GetScrap }
               hDest := NewHandle(0);  { automatically resizes it}
               HLock(hDest);
               {put data into memory referenced thru hDest handle}
               sizeOfPictData := GetScrap(hDest, 'PICT', offset);
               {calculate destination rectangle for plotting the }
               { picture}
               MyGetDestRect(hDest, destRect);
               DrawPicture(PicHandle(hDest), destRect);
               {save information about the picture}
               myData := MyDocRecHnd(GetWRefCon(window));
               myData^^.pictNum := myData^^.pictNum +1;
               myData^^.pictDestRect[myData^^.pictNum] :=
                                                     destRect;
               IF myData^^.windowPicHndl[myData^^.pictNum] = NIL
               THEN
                  myData^^.windowPicHndl[myData^^.pictNum] :=
                     PicHandle(NewHandle(Size(sizeOfPictData)));
               myData^^.windowPicHndl[myData^^.pictNum] :=
                                              PicHandle(hDest);
               myErr := HandToHand(Handle
                     (myData^^.windowPicHndl[myData^^.pictNum]));
               HUnlock(hDest);
               DisposeHandle(hDest);
            END;     {of sizeOfPictData > 0}
         END;     {of "if no 'TEXT' data, check for 'PICT'"}
      END;     {of "if no 'surf' data, check for 'TEXT'"}
   END      {of "if windowType = kMyDocWindow"}
   ELSE     {window is not a document window}
   BEGIN
      IF windowType <> kNil THEN
      BEGIN {handle Paste command in dialog box, }
            { DialogPaste checks whether the dialog box has any }
            { editText items and if so, uses TEPaste to paste }
            { any text from the scrap to the currently selected }
            { editText item, if any}
         DialogPaste(window);
      END;
   END;
END;
```

If your application uses TextEdit in its document windows, then use the TextEdit routine `TEPaste` instead of `GetScrap` to read the data to paste. See Listing 2-9 on page 2-30 for an application-defined routine that uses TextEdit to help handle the application's Paste command.

If your application uses a private scrap, then read the data from your private scrap rather than from the scrap (unless the scrap contains the more recent data). Listing 2-5 shows SurfPaint's application-defined routine that handles the Paste command by reading the desired data from its private scrap.

Listing 2-5 Handling the Paste command using a private scrap

```
PROCEDURE DoPasteCmd;
VAR
   window:           WindowPtr;
   windowType:       Integer;
   dataToPaste:      Ptr;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      IF gNewScrap THEN    {if new data in scrap, }
      BEGIN                { copy to private scrap}
         MyConvertScrap(kClipboardToPrivate);
         gNewScrap := FALSE;
      END;
      {get the data to paste from app's private scrap}
      dataToPaste := NewPtr(kDefaultSize);
      MyReadDataFromPrivateScrap(dataToPaste);
      MyPasteData(dataToPaste);
      DisposePtr(dataToPaste);
   END
   ELSE
   IF windowType <> kNil THEN
   BEGIN {window is a dialog box}
      DialogPaste(window);
   END;
END;
```

The SurfPaint application uses a private scrap, and when it receives a resume event, it determines whether the contents of the scrap have changed. If so, SurfPaint sets an application-defined global variable, `gScrapNewData`, but does not immediately read in the contents of the scrap. Instead, whenever the user chooses the Paste command, SurfPaint checks the value of this global variable. If `gScrapNewData` is `TRUE` SurfPaint

reads the new data from the scrap to its private scrap, resets the gScrapNewData global variable to FALSE, and then performs the paste operation. SurfPaint also resets the value of the gScrapNewData global variable to FALSE whenever the user chooses the Cut or Copy command. By using this method, SurfPaint reads in new data from the scrap only when necessary and avoids reading in data that the user might not use. This method also decreases the time it takes for the application to return to the foreground, as the application avoids or delays any lengthy translation of data from the scrap.

Handling Resume Events

As previously described, when your application receives a resume event (and your application uses a private scrap), your application should determine whether the contents of the scrap have changed since the previous suspend event. If the contents of the scrap have changed, your application must be sure to use the new data in the scrap for the user's next Paste command (unless the user chooses Cut or Copy before choosing Paste).

In addition, if your application supports the Show Clipboard command and the Clipboard window was showing at the time of the previous suspend event, your application should update its Clipboard window to show the new contents of the scrap.

Listing 2-6 shows SurfPaint's procedure for handling resume events (and suspend events).

Listing 2-6 Handling resume events

```
PROCEDURE DoSuspendResumeEvent (event: EventRecord);
VAR
   currentFrontWindow: WindowPtr;
BEGIN
   currentFrontWindow := FrontWindow;
   IF (BAnd(event.message, resumeFlag) <> 0) THEN
   BEGIN                {it's a resume event}
      IF (BAnd(event.message, convertClipboardFlag) <> 0) THEN
      BEGIN
         {set flag to indicate there's new data in the scrap}
         gNewScrap := TRUE;
      END;
      gInBackground := FALSE; {app no longer in background}
                              {activate front window}
      DoActivate(currentFrontWindow, NOT gInBackground, event);
      {show Clipboard window if it was showing at last suspend }
      { event and update its contents to match scrap}
      MyShowClipboardWindow(gNewScrap);
      MyShowFloatingWindows;  {show any floating windows}
   END
```

```
   ELSE
   BEGIN                {it's a suspend event, }
                        { handle as shown in Listing 2-3}
   END;
END;
```

Listing 2-6 shows a procedure that responds to suspend and resume events. The `DoSuspendResumeEvent` procedure first gets a pointer to the front window using the Window Manager function `FrontWindow`. It then examines bit 0 of the `message` field of the event record to determine whether the event is a suspend or resume event. If the event is a resume event, the code examines bit 1 of the `message` field of the event record to determine whether it needs to read in the contents of the scrap. If so, the code sets an application-defined global variable, `gNewScrap`, to indicate that new data exists in the scrap. When the user next chooses the Paste command, SurfPaint checks the value of the `gNewScrap` global variable and, if it's `TRUE`, reads the data from the scrap to its private scrap and then performs the paste operation. If the user chooses the Cut or Copy command before choosing Paste, then SurfPaint resets the `gNewScrap` global variable to `FALSE` to indicate that its private scrap contains the most recent data for the Paste command. This technique allows SurfPaint to delay or avoid any lengthy translation of data from the scrap to its private scrap and decreases the time it takes for SurfPaint to return to the foreground.

The `DoSuspendResumeEvent` procedure then sets a private global flag, `gInBackground`, to `FALSE`, to indicate that the application is not in the background. It then calls another application-defined routine, `DoActivate`, to activate the application's front window. It also calls the application-defined routine `MyShowClipboardWindow` to show the Clipboard window and update its contents if it was showing at the time of the previous suspend event.

Converting Data Between a Private Scrap and the Scrap

If you use a private scrap, you need to copy the data from your private scrap to the scrap upon receiving a suspend event. Upon receiving a resume event, you need to determine whether the contents of the scrap have changed since the previous suspend event. If so, your application must be sure to use the new data in the scrap for the user's next Paste command (unless the user chooses Cut or Copy before choosing Paste). In addition, your application needs to update the contents of its Clipboard window, if necessary.

Listing 2-7 shows the application-defined procedure `MyConvertScrap`. This procedure is called either indirectly as a result of a resume event (indicated by the `kClipboardToPrivate`, constant) or directly as a result of a suspend event (indicated by the `kPrivateToClipboard` constant). If the `whichWay` parameter contains `kClipboardToPrivate`, then the contents of the scrap have changed. In this case, `MyConvertScrap` uses `GetScrap` to read the contents of the scrap. The `MyConvertScrap` procedure checks the scrap for `'PICT'` data first, and then for `'TEXT'` data if the scrap doesn't contain any data in `'PICT'` format. `MyConvertScrap` then copies this data to its private scrap.

If the `MyConvertScrap` procedure is called as a result of a suspend event, the procedure copies the data from its private scrap to the scrap. It writes the data to the scrap in its own private format, in `'PICT'` format, and, if appropriate, in `'TEXT'` format.

Listing 2-7 Converting data between the scrap and a private scrap

```
PROCEDURE MyConvertScrap (whichWay: Integer);
VAR
   sizeOfTextData:   LongInt;
   sizeOfPictData:   LongInt;
   offset:           LongInt;
   hDest:            Handle;
   ptrToScrapData:   Ptr;
   length:           LongInt;
   myLongErr:        LongInt;
BEGIN
   IF whichWay = kClipboardToPrivate THEN
   BEGIN {copy scrap to private scrap}
      sizeOfPictData := GetScrap(NIL, 'PICT', offset);
      IF sizeOfPictData > 0 THEN
      BEGIN
         {get handle to hold data from scrap, GetScrap }
         hDest := NewHandle(0);  { automatically resizes it}
         HLock(hDest);
         {put data into memory referenced by hDest handle}
         sizeOfPictData := GetScrap(hDest, 'PICT', offset);
         MyCopyToPrivateScrap(hDest);
         HUnlock(hDest);
         DisposeHandle(hDest);
      END
      ELSE  {if no 'PICT' data on scrap, check for 'TEXT'}
      BEGIN
         sizeOfTextData := GetScrap(NIL, 'TEXT', offset);
         IF sizeOfTextData > 0 THEN
         BEGIN
            {allocate handle to hold scrap data--GetScrap }
            hDest := NewHandle(0);  { automatically resizes it}
            HLock(hDest);
            {put data into memory referenced by hDest handle}
            sizeOfTextData := GetScrap(hDest, 'TEXT', offset);
            {copy data to private scrap}
            MyCopyToPrivateScrap(hDest);
            HUnlock(hDest);
```

```
            DisposeHandle(hDest);
         END
      END;
   END
   ELSE
   BEGIN {copy private scrap into scrap}
      IF MyGetPrivateScrapSize > 0 THEN   {if private scrap }
         myLongErr := ZeroScrap; { not empty, clear the scrap}
      ptrToScrapData := NewPtr(kDefaultSize);
      {retrieve data from private scrap in private format}
      IF (MyGetScrap('SURF', ptrToScrapData, length) > 0) THEN
      BEGIN    {copy data to the scrap}
         myLongErr := PutScrap(length, 'SURF', ptrToScrapData);
         IF myLongErr <> noErr THEN DoError(myLongErr);
      END;
      {retrieve data from private scrap in 'PICT' format}
      IF (MyGetScrap('PICT', ptrToScrapData, length) > 0) THEN
      BEGIN    {copy data to the scrap}
         myLongErr := PutScrap(length, 'PICT', ptrToScrapData);
         IF myLongErr <> noErr THEN DoError(myLongErr);
      END;
      {retrieve data from private scrap in 'TEXT' format}
      IF (MyGetScrap('TEXT', ptrToScrapData, length) > 0) THEN
      BEGIN    {copy data to the scrap}
         myLongErr := PutScrap(length, 'TEXT', ptrToScrapData);
         IF myLongErr <> noErr THEN DoError(myLongErr);
      END;
      DisposePtr(ptrToScrapData);
   END;
END;
```

Converting Data Between the TextEdit Scrap and the Scrap

If your application uses TextEdit to handle text in its document windows, then use TextEdit routines instead of Scrap Manager routines to implement editing commands. For example, use the TextEdit procedures `TECut`, `TECopy`, and `TEPaste` to implement the Cut, Copy, and Paste commands. Upon receiving a suspend event, use `TEToScrap` instead of `PutScrap` to write the data to the scrap (always call `ZeroScrap` before calling `TEToScrap`). Upon receiving a resume event, use `TEFromScrap` instead of `GetScrap` to read data from the scrap. TextEdit uses a private scrap and handles copying data between its private scrap and the scrap. See *Inside Macintosh: Text* for complete information on TextEdit.

To implement the Cut (or Copy) commands, use the TextEdit routines `TECut` (or `TECopy`) instead of `ZeroScrap` and `PutScrap`. The TextEdit procedures `TECut` and `TECopy` copy the data in the current selection to TextEdit's private scrap. For example, Listing 2-8 shows an application-defined routine that uses TextEdit to help handle the application's Cut command (assuming the application uses TextEdit to handle text editing in its document windows).

Listing 2-8 Using TextEdit to handle the Cut command

```
PROCEDURE DoCutOrCopyCmd (cut: Boolean);
VAR
   window:           WindowPtr;
   windowType:       Integer;
   myData:           MyDocRecHnd;
   teHand:           TEHandle;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      myData := MyDocRecHnd(GetWRefCon(window));
      teHand := myData^^.editRec;
      IF cut THEN
         TECut(teHand)
      ELSE
         TECopy(teHand);
   END
   ELSE
   IF windowType <> kNIL THEN
   BEGIN       {window is a dialog box}
      IF cut THEN
         DialogCut(window)
      ELSE
         DialogCopy(window);
   END;
END;
```

Use the TextEdit routine `TEPaste` instead of `GetScrap` to read the data to paste. The `TEPaste` procedure reads the data to paste from TextEdit's private scrap. Listing 2-9 shows an application-defined routine that uses TextEdit to help handle the application's Paste command (assuming the application uses TextEdit to handle text editing in its document windows).

Listing 2-9 Using TextEdit to handle the Paste command

```
PROCEDURE DoPasteCmd;
VAR
   window:           WindowPtr;
   windowType:       Integer;
   myData:           MyDocRecHnd;
   teHand:           TEHandle;
BEGIN
   window := FrontWindow;
   windowType := MyGetWindowType(window);
   IF windowType = kMyDocWindow THEN
   BEGIN
      myData := MyDocRecHnd(GetWRefCon(window));
      teHand := myData^^.editRec;
      TEPaste(teHand);
   END
   ELSE
   IF windowType <> kNIL THEN
   BEGIN       {window is a dialog box}
      DialogPaste(window);
   END;
END;
```

Upon receiving a suspend event, use `ZeroScrap` and then the TextEdit procedure `TEToScrap` to copy data from TextEdit's private scrap to the scrap. Upon receiving a resume event, use the TextEdit procedure `TEFromScrap` to copy data from the scrap to TextEdit's private scrap. As with any other private scrap and as explained in "Handling Resume Events" on page 2-25, either you can choose to immediately copy the data from the scrap to TextEdit's private scrap or you can delay performing the copy until the data is needed. See Listing 2-5 on page 2-24 and Listing 2-6 on page 2-25 for code that uses this approach.

Handling Editing Operations in Dialog Boxes

You can use the Dialog Manager to handle most editing operations in dialog boxes. In general, use the procedures `DialogCut`, `DialogCopy`, and `DialogPaste` to support the Cut, Copy, and Paste commands in editable text items in your dialog boxes. As shown in Listing 2-2 on page 2-18 and Listing 2-5 on page 2-24, when the user chooses the Cut, Copy, or Paste command, the application-defined routine uses Dialog Manager routines to perform the editing operation.

The Dialog Manager uses TextEdit to perform the editing operation. TextEdit copies data between its private scrap and the editable text item in the dialog box. TextEdit uses a private scrap, which allows the user to copy and paste data between dialog boxes. However, your application must make sure the user can copy and paste data between your application's dialog boxes and its document windows. That is, when the user selects text in a document window and chooses Copy, then activates a dialog box and chooses Paste, the data previously copied from the document window should appear in the active editable text item. Your application is responsible for maintaining consistency between the scrap (or your application's private scrap) and TextEdit's private scrap.

If your application uses TextEdit for all text editing in its document windows, then you can easily allow the user to copy and paste between your application's document windows and its dialog boxes, as your application uses `TECut`, `TECopy`, and `TEPaste` for its document windows and `DialogCut`, `DialogCopy`, and `DialogPaste` (which in turn use TextEdit routines) for its dialog boxes. These routines all use TextEdit's private scrap, which maintains consistency of data between editing operations.

If your application does not use TextEdit for text handling in your document windows and uses a private scrap, then when the user activates a dialog box you should copy any text data in your private scrap to TextEdit's private scrap. When a document window becomes active, if there's data in TextEdit's private scrap, you should copy the data to your private scrap (or the scrap if your application doesn't 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 scrap. The Standard File Package reads the data from the scrap when the user chooses an editing operation and a standard file dialog box is active. So your application needs to put the text data (if any) from the last Cut or Copy command in the scrap before calling `StandardPutFile`.

Scrap Manager Reference

This section describes the data structures and routines that are specific to the Scrap Manager. The "Data Structures" section describes the scrap information record and scrap format types. The "Routines" section describes the routines that your application can use to read and write data to the scrap and to get information about data in the scrap.

Data Structures

This section describes the scrap information record and the standard scrap format types.

The Scrap Information Record

The Scrap Manager returns information about the scrap in a scrap information record. The scrap information record is defined by the `ScrapStuff` data type.

```
TYPE
   ScrapStuff =                  {scrap information record}
   RECORD
      scrapSize:     LongInt;    {size (in bytes) of scrap}
      scrapHandle:   Handle;     {handle to scrap}
      scrapCount:    Integer;    {indicates whether the contents }
                                 { of the scrap have changed}
      scrapState:    Integer;    {indicates state and location }
                                 { of scrap}
      scrapName:     StringPtr;  {filename of the scrap}
   END;
   PScrapStuff = ^ScrapStuff;    {pointer to scrap info record}
```

Field descriptions

`scrapSize` The size of the scrap in bytes.

`scrapHandle` A handle to the scrap if it's in memory; otherwise, this field is `NIL`.

`scrapCount` A number that changes each time your application (or another application) calls the `ZeroScrap` function. When your application receives a suspend event, it should copy any data from its private scrap to the scrap and it can save the value of the `scrapCount` field. Upon receiving a resume event, your application can use the `InfoScrap` function to examine the current value of the `scrapCount` field. If the value in the `scrapCount` field is different from the previous value, the contents of the scrap have changed and your application should copy the data from the scrap to its private scrap.

Alternatively, rather than saving and examining the value of the `scrapCount` field, your application can check the `convertClipboardFlag` bit of the event record for a resume event. If this bit is set, the contents of the scrap have changed and your application should take the appropriate actions.

`scrapState` The location and state of the scrap. This field is positive if the scrap data is in memory, 0 if the scrap data is on the disk, or negative if the scrap hasn't been initialized.

Note

In unusual circumstances the value of `scrapState` might be 0 when the scrap is actually in memory. This can occur if the user deletes the scrap file on disk and then performs a cut or copy operation. ◆

`scrapName` The filename of the scrap when the scrap is stored on disk. Usually the scrap file is named "Clipboard". The scrap file is always stored on the startup volume.

The Scrap Format Types

Data in the scrap is defined by a scrap format type, a four-character sequence that defines the type of data.

```
TYPE  ResType = PACKED ARRAY[1..4] OF Char;
```

The standard scrap format types are

- `'TEXT'`: a series of ASCII characters
- `'PICT'`: a QuickDraw picture, which is a saved sequence of QuickDraw commands that can be displayed using the `DrawPicture` procedure

Optional scrap format types include

- `'styl'`: a series of bytes that have the same format as a TextEdit `'styl'` resource and that describe styled text data
- `'snd '`: a series of bytes that have the same format as an `'snd '` resource and that define a sound
- `'movv'`: a series of bytes that have the same format as an `'movv'` resource and that define a movie

Your application should support the `'TEXT'` and `'PICT'` scrap format types and should optionally support any other scrap format types (such as `'snd '`) that are appropriate to your application.

In general, when your application writes data to the scrap, the Scrap Manager appends the data to the scrap in this format:

Number of bytes	Contents
4	Scrap format type
4	Length of following data in bytes
n	Data; *n* must be even

Routines

This section describes the routines you use to

- get information about the scrap
- write data to the scrap
- read data from the scrap
- store the scrap in memory onto disk
- read the scrap from disk into memory

Getting Information About the Scrap

You can get information about the scrap using the `InfoScrap` function.

InfoScrap

You can use the `InfoScrap` function to get information about the scrap.

```
FUNCTION InfoScrap: PScrapStuff;
```

DESCRIPTION

The `InfoScrap` function returns a pointer to a scrap information record. The information in the scrap information record provides

- the size (in bytes) of the scrap
- a handle to the scrap if it's in memory
- a count, or number, that your application can use to determine whether the contents of the scrap have changed
- the location of the scrap (whether in memory or on disk)
- the filename of the scrap when it is on the disk

ASSEMBLY-LANGUAGE INFORMATION

You can also access the same information as that stored in the scrap information record using system global variables that have the same names as the fields of the scrap information record.

SEE ALSO

See "Getting Information About the Scrap" on page 2-15 for an example that uses the `InfoScrap` function to get information about the scrap. See page 2-32 for information on the fields of the scrap information record.

Writing Information to the Scrap

To write information to the scrap, first use the `ZeroScrap` function to clear the contents of the scrap, and then use the `PutScrap` function to write data in a specific format to the scrap. You can use the `PutScrap` function multiple times to place data in more than one format in the scrap.

ZeroScrap

You use the `ZeroScrap` function to clear the contents of the scrap before writing data to the scrap.

```
FUNCTION ZeroScrap: LongInt;
```

DESCRIPTION

If the scrap already exists (in memory or on the disk), the `ZeroScrap` function clears its contents; otherwise, `ZeroScrap` initializes the scrap in memory. Whenever your application needs to write data to the scrap as a result of a cut or copy operation by the user, you should call `ZeroScrap` before calling `PutScrap`. Whenever your application needs to write data in one or more formats to the scrap, you should call `ZeroScrap` before the first time you call `PutScrap`.

If your application uses `TEToScrap` to write TextEdit's scrap to the scrap, your application should call `ZeroScrap` to clear the contents of the scrap first. However, note that your application does not have to call `ZeroScrap` before calling `TECut` or `TECopy`.

The `ZeroScrap` function returns a long integer with the value `noErr` if `ZeroScrap` successfully clears the contents of or initializes the scrap. Otherwise, the `ZeroScrap` function returns a nonzero value, whose value corresponds to a result code.

SPECIAL CONSIDERATIONS

Your application should not call the `ZeroScrap` function at interrupt time.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`memFullErr`	–108	Not enough memory in heap zone

PutScrap

You can use the `PutScrap` function to write data in a specific format to the scrap.

```
FUNCTION PutScrap (length: LongInt; theType: ResType; source: Ptr)
                  : LongInt;
```

`length` The number of bytes of data to write to the scrap.

`theType` The scrap format type of the data to be written to the scrap. The scrap format type is a four-character sequence that refers to the particular data format, such as `'TEXT'`, `'PICT'`, `'styl'`, `'snd '`, or `'movv'`.

`source` A pointer to the data that the `PutScrap` function should write to the scrap.

DESCRIPTION

The `PutScrap` function writes the specified number of bytes of data from the memory location pointed to by the `source` parameter to the scrap. The Scrap Manager writes the data to the current location of the scrap (your application's heap or disk).

Whenever your application needs to write data to the scrap as a result of a cut or copy operation, your application uses the `PutScrap` function to write a representation of the data to the scrap. If your application uses a private scrap, it should copy data from its private scrap to the scrap using the `PutScrap` function whenever it receives a suspend event. Your application can use the `PutScrap` function multiple times to write different formats of the same data to the scrap.

IMPORTANT

Whenever your application needs to write data in one or more formats to the scrap, you should call `ZeroScrap` before the first time you call `PutScrap`. ▲

If your application writes multiple formats to the scrap, you should write your application's preferred scrap format type first. For example, if the SurfWriter application's preferred scrap format type is a private scrap format type called `'SURF'` and SurfWriter also supports the scrap format types `'TEXT'` and `'PICT'`, then SurfWriter should write the data to the scrap using the `'SURF'` scrap format type first, and then write any other scrap format types that it supports in subsequent order of preference.

▲ **WARNING**
Do not write data to the scrap that has the same scrap format type as any data already in the scrap. If you do so, the new data is appended to the scrap. Note that when you request data from the scrap using the `GetScrap` function, `GetScrap` returns the first data that it finds with the requested scrap format type; thus you cannot retrieve any appended data of the same format type using `GetScrap`. ▲

If your application uses TextEdit to handle text in its documents, use TextEdit routines to implement cut and copy operations and to write the TextEdit scrap to the scrap. If your application uses the Dialog Manager to handle editable text in your application's dialog boxes and a dialog box is the frontmost window, use the Dialog Manager procedure `DialogCut` or `DialogCopy` to copy the data from the current editable text item to the scrap.

If the scrap does not already exist (in memory or on the disk), the `PutScrap` function returns a long integer with the value `noScrapErr`. The `PutScrap` function returns other nonzero values corresponding to result codes if an error occurs.

SPECIAL CONSIDERATIONS

The `PutScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

`noErr`	0	No error
`noScrapErr`	–100	Scrap does not exist (not initialized)

SEE ALSO

See Listing 2-1 on page 2-16, Listing 2-2 on page 2-18, and Listing 2-7 on page 2-27 for examples that write data to the scrap. If your application uses a private scrap, see "Handling Editing Operations in Dialog Boxes" on page 2-31 for information on maintaining consistency of the scrap when copying and pasting data between document windows and dialog boxes. See *Inside Macintosh: Text* for information on TextEdit. See *Inside Macintosh: Imaging With QuickDraw* for information on the QuickDraw `'PICT'` format.

Reading Information From the Scrap

To read information from the scrap, use the `GetScrap` function.

GetScrap

You can use the `GetScrap` function to read data of a specific format from the scrap.

```
FUNCTION GetScrap (hDest: Handle; theType: ResType;
                   VAR offset: LongInt): LongInt;
```

`hDest` A handle to the memory location where the `GetScrap` function should place the data from the scrap. If you specify `NIL` in this parameter, the `GetScrap` function does not read in the data but does return the offset of the data in the scrap and the number of bytes of the requested scrap data type if the requested type exists in the scrap.

`theType` The scrap format type of the data to be read from the scrap.

`offset` The `GetScrap` function returns in this parameter the location of the data in the scrap. This value is expressed as an offset (in bytes) from the beginning of the scrap. If the Translation Manager is available, the value of the `offset` parameter is undefined.

DESCRIPTION

The `GetScrap` function looks in the scrap for any data of the requested scrap format type and returns the first data of the requested type that it finds.The `GetScrap` function writes the data to the memory location specified by the `hDest` parameter.

The `GetScrap` function reads the data from the scrap, makes a copy of it in memory, and sets the handle specified by the `hDest` parameter to refer to this copy. The `GetScrap` function resizes the handle specified by the `hDest` parameter if necessary.

Your application can use the `GetScrap` function multiple times to read different formats of the same data from the scrap. If more than one format of the same scrap format type exists in the scrap, the `GetScrap` function returns the first occurrence of that format type that it finds. For example, if data of type `'TEXT'`, `'PICT'`, and `'TEXT'` exist on the scrap, and your application requests the data in the scrap with scrap format type `'TEXT'`, the `GetScrap` function returns the first data of type `'TEXT'` that it finds.

If your application supports more than one scrap format type, your application should attempt to read its preferred scrap format type first. If your application doesn't prefer one scrap format type over any other type, it should try reading each of the scrap format types that it supports and use the type that returns the lowest offset. The scrap format type with the lowest offset indicates that this format type was written before any of the others and therefore was preferred by the application that wrote it.

Note
The returned value for the `offset` parameter is valid only if the Translation Manager isn't available; if the Translation Manager is available, then your application should not rely on the offset value. ◆

If you request a scrap format type that isn't in the scrap and the Translation Manager is available, the Scrap Manager uses the Translation Manager to convert the data of a scrap format type that does exist in the scrap into the scrap format type requested by your application. For example, if the SurfWriter application requests data from the scrap in the `'SURF'` scrap format type, and the data in the scrap is available in the format types `'TEXT'`, `'PICT'`, and `'SDBS'`(SurfDB's private scrap format type), the Scrap Manager uses the Translation Manager to convert any one of the scrap format types `'TEXT'`, `'PICT'`, or`'SDBS'` into the `'SURF'` scrap format type. The Translation Manager looks for a translator that can perform one of these translations. If such a translator is available (for example, a translator that can translate the `'SDBS'` scrap format type into the `'SURF'` scrap format type), the Translation Manager uses the translator to translate the data in the scrap into the requested scrap format type. If the translation is successful, the Scrap Manager returns to your application the data from the scrap in the requested scrap format type.

If your application uses TextEdit to handle text in its documents, use TextEdit routines to implement the paste operation and to copy data from the scrap to the TextEdit scrap. If your application uses the Dialog Manager to handle editable text items in your application's dialog boxes and a dialog box is the frontmost window, use the Dialog Manager procedure `DialogPaste` to copy data from the scrap to the current editable text item.

If the `GetScrap` function successfully reads the data of the requested scrap format type from the scrap, `GetScrap` returns as its function result the length (in bytes) of the data. Otherwise, `GetScrap` returns a negative function result that indicates the error. If `GetScrap` returns the constant `noTypeErr`, then the data in the scrap isn't available in the scrap format type requested by your application. If the Translation Manager is available and `GetScrap` returns the constant `noTypeErr`, this value also indicates that the Translation Manager could not find any translators to convert the data into the scrap format type requested by your application.

```
CONST noTypeErr = -102;{no data of the requested scrap format type}
```

SPECIAL CONSIDERATIONS

The `GetScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

SEE ALSO

See Listing 2-4 on page 2-21, Listing 2-5 on page 2-24, and Listing 2-7 on page 2-27 for examples that read data from the scrap. If your application uses a private scrap, see "Handling Editing Operations in Dialog Boxes" on page 2-31 for information on maintaining consistency of the scrap when copying and pasting data between document windows and dialog boxes. See *Inside Macintosh: Text* for information on TextEdit. See *Inside Macintosh: Imaging With QuickDraw* for information on the QuickDraw `'PICT'` format.

Transferring Data Between the Scrap in Memory and the Scrap on Disk

When system software launches your application, it initially allocates space in your application's heap for the scrap. To write the scrap from memory to the scrap file, use the `UnloadScrap` function. To read data from a scrap file into memory, use the `LoadScrap` function.

UnloadScrap

You can use the `UnloadScrap` function to write the scrap from memory to the scrap file.

```
FUNCTION UnloadScrap: LongInt;
```

DESCRIPTION

The `UnloadScrap` function writes the scrap in memory to the scrap file and releases the memory occupied by the scrap in your application's heap. The scrap file is located in the System Folder of the startup volume and has the filename as indicated by the `scrapName` field of the scrap information record (usually "Clipboard"). If the scrap is already on the disk, the `UnloadScrap` function does nothing.

`UnloadScrap` returns as its function result a long integer corresponding to a result code.

SPECIAL CONSIDERATIONS

The `UnloadScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error

LoadScrap

You can use the `LoadScrap` function to read the scrap from the scrap file into memory.

```
FUNCTION LoadScrap: LongInt;
```

DESCRIPTION

The `LoadScrap` function allocates memory in your application's heap to hold the scrap and then reads the scrap from the scrap file into memory. The scrap file is located in the System Folder of the startup volume and has the filename (usually "Clipboard") as indicated by the `scrapName` field of the scrap information record. If the scrap is already in memory, `LoadScrap` does nothing.

`LoadScrap` returns as its function result a long integer corresponding to a result code.

SPECIAL CONSIDERATIONS

The `LoadScrap` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`memFullErr`	–108	Not enough memory in heap zone

Summary of the Scrap Manager

Pascal Summary

Constants

```
gestaltScrapMgrAttr              = 'scra';{Gestalt selector for }
                                          { Scrap Mgr attributes}
gestaltScrapMgrTranslationAware  = 0;     {check this bit in the }
                                          { response parameter to see }
                                          { whether Scrap Mgr supports }
                                          { Translation Mgr}
```

Data Types

```
TYPE
   ScrapStuff =                  {scrap information record}
   RECORD
      scrapSize:     LongInt;    {size (in bytes) of scrap}
      scrapHandle:   Handle;     {handle to scrap}
      scrapCount:    Integer;    {indicates whether the contents }
                                 { of the scrap have changed}
      scrapState:    Integer;    {indicates state and location }
                                 { of scrap}
      scrapName:     StringPtr;  {filename of the scrap}
   END;
   PScrapStuff = ^ScrapStuff;    {pointer to a scrap information record}
```

Routines

Getting Information About the Scrap

```
FUNCTION InfoScrap          : PScrapStuff;
```

Writing Information to the Scrap

```
FUNCTION ZeroScrap          : LongInt;
FUNCTION PutScrap           (length: LongInt; theType: ResType; source: Ptr)
                             : LongInt;
```

Reading Information From the Scrap

```
FUNCTION GetScrap           (hDest: Handle; theType: ResType;
                             VAR offset: LongInt): LongInt;
```

Transferring Data Between the Scrap in Memory and the Scrap on Disk

```
FUNCTION UnloadScrap        : LongInt;
FUNCTION LoadScrap          : LongInt;
```


C Summary

```
enum {
#define gestaltScrapMgrAttr       'scra'     /*Gestalt selector for */
                                             /* Scrap Mgr attributes*/
gestaltScrapMgrTranslationAware  = 0         /*check this bit in the */
                                             /* response parameter to see */
                                             /* whether Scrap Mgr supports */
                                             /* Translation Mgr*/
};
```

Data Types

```
   struct ScrapStuff {           /*scrap information record*/
      long           scrapSize;  /*size (in bytes) of scrap*/
      Handle         scrapHandle;/*handle to scrap*/
      short          scrapCount; /*indicates whether the contents */
                                 /* of the scrap have changed*/
      short          scrapState; /*indicates state and location */
                                 /* of scrap*/
      StringPtr      scrapName;  /*filename of the scrap*/
   };
   typedef struct ScrapStuff ScrapStuff;
   typedef ScrapStuff *PScrapStuff;
```

Routines

Getting Information About the Scrap

```
pascal PScrapStuff InfoScrap
                            (void);
```

Writing Information to the Scrap

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

Reading Information From the Scrap

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

Transferring Data Between the Scrap in Memory and the Scrap on Disk

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

Assembly-Language Summary

Data Structures

Scrap Information Data Structure

0	`ScrapSize`	long	size (in bytes) of the scrap
4	`ScrapHandle`	long	handle to scrap
8	`ScrapCount`	2 bytes	indicates whether the contents of the scrap have changed
10	`ScrapState`	2 bytes	indicates state and location of scrap
12	`ScrapName`	long	pointer to the filename of the scrap

Result Codes

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`noScrapErr`	–100	Scrap does not exist (not initialized)
`noTypeErr`	–102	No data of the requested scrap format type in scrap
`memFullErr`	–108	Not enough memory in heap zone