Scrap Manager
External Reference Specification
Steven Glass
June 26, 1986

June 26, 1986	**Initial Release.**

THE SCRAP MANAGER

The Scrap Manager lets an application handle cutting and pasting. From the user's point of view, all data that's cut or copied resides in the Clipboard. The Cut command deletes data from a document and places it in the Clipboard; the Copy command copies data into the Clipboard without deleting it from the document. The next Paste command—whether applied to the same document or another, in the same application or another—inserts the contents of the Clipboard at a specified place. An application that supports cutting and pasting may also provide a Clipboard window for displaying the current contents of the scrap; it may show the Clipboard window at all times or only when requested via the toggled command Show (or Hide) Clipboard.

> Note: The Scrap Manager was designed to transfer *small* amounts of data; attempts to transfer very large amounts of data may fail due to lack of memory.

The nature of the data to be transferred varies according to the application. For example, in a word processor or in the Calculator desk accessory, the data is text; in a graphics application it's a picture. The amount of information retained about the data being transferred also varies. Between two text applications, text can be cut and pasted without any loss of information; however, if the user of a graphics application cuts a picture consisting of text and then pastes it into a word processor document, the text in the picture may not be editable in the word processor, or it may be editable but not look exactly the same as it the graphics application. The Scrap Manager allows for a variety of data types and provides a mechanism whereby applications have some control over how much information is retained when data is transferred.

The desk scrap is usually stored in memory, but can be stored on the disk (in the file **clipboard** in the system subdirectory of the boot volume) if there's not enough room for it in memory. There is no requirement on where the scrap must be when an application starts or stops (as there is on the Macintosh).

MEMORY AND THE DESK SCRAP

A large scrap can prevent an application from loading. The application can get the size of the scrap by making a ScrapManager call. An application concerned about whether there's room for the desk scrap in memory could be set up so that a small initial segment of the application is loaded in just to check the scrap size. After a decision is made about whether to keep the scrap in memory or on the disk, the remaining segments of the application can be loaded in as needed. Of course, if there is not room for the scrap at application load time, there probably won't be room for it later when a user tries to paste its contents into a document.

There are other disadvantages to keeping the desk scrap on the disk. The disk may be locked, it may not have enough room for the scrap, or it may be removed during use of the application. If the application can't write the scrap to the disk, it should put up an alert box informing the user, who may want to abort the operation at that point.

DESK SCRAP DATA TYPES

From the user's point of view there can be only one thing in the Clipboard at a time, but the application may store more than one version of the information in the scrap, each representing the same Clipboard contents in a different form. For example, text cut with a word processor may be stored in the desk scrap both as text and as a QuickDraw picture.

Why would an application want to do this you ask? The answer is somewhat complex. Applications like to keep information in their own internal format, but they also want to be able

to communicate via the clipboard with other applications. So when a user cuts or copies something to the clipboard, the application can put it there two different ways.

The internal way so that a subsequent paste can easily be dealt with

The public way so that if the user trys to paste this into another application or desk accessory, the other application can deal with it.

What is the public way? There are two public scrap types defined:

Text	Type = 0
Picture	Type = 1

Applications must write at least one of these standard types of data to the desk scrap and must be able to read *both* types. Most applications will prefer one of these types over the other; for example, a word processor prefers text while a graphics application prefers pictures. An application should write at least its preferred standard type of data to the desk scrap, and may write both types (to pass the most information possible on to the receiving application, which may prefer the other type).

An application reading the desk scrap will look for its preferred data type. If its preferred type isn't there, or if it's there but was written by an application having a different p eferred type, the receiving application may or may not be able to convert the data to the type it needs. If not, some information may be lost in the transfer process. For example, a graphics application can easily convert text to a picture, but the reverse isn't true.

USING THE SCRAP MANAGER

If your application supports display of the Clipboard, you can call GetScrapCount each time through your main event loop to check this count: If the Clipboard window is visible, it needs to be updated whenever the count changes.

When a Cut or Copy command is given, you need to write the cut or copied data to the desk scrap. First call ZeroScrap to clear its previous contents, and then PutScrap to put the data into the scrap. (You can call PutScrap more than once, to put the data in the scrap in different forms.)

Call GetScrap when a Paste command is given, to access data of a particular type in the desk scrap and to get information about the data.

> **Note:** ZeroScrap, PutScrap, and GetScrap all keep track of whether the scrap is in memory or on the disk, so you don't have to worry about loading it first. After any of these calls the scrap will be in memory again.

PRIVATE SCRAPS

Instead of using the desk scrap for storing data that's cut and pasted within an application, advanced programmers may want to set up a private scrap for this purpose. In applications that use the standard text or picture data types, it's simpler to use the desk scrap, but if your application defines its own private type of data, or if it's likely that very large amounts of data will be cut and pasted, using a private scrap may result in faster cutting and pasting within the application.

The format of a private scrap can be whatever the application likes, since no other application will use it. For example, an application can simply maintain a pointer to data that's been cut or

copied. The application must, however, be able to convert data between the format of its private scrap and the format of the desk scrap.

> Note: The LineEdit scrap is a private scrap for applications that use LineEdit. LineEdit provides routines for accessing this scrap; you'll need to transfer data between the LineEdit scrap and the desk scrap.

If you use a private scrap, you must be sure that the right data will always be pasted when the user gives a Paste command (the right data being whatever was most recently cut or copied in any application or desk accessory), and that the Clipboard, if visible, always shows the current data. You should copy the contents of the desk scrap to your private scrap at application startup and whenever a desk accessory is deactivated (call GetScrap to access the desk scrap). When the application is terminated or when a desk accessory is activated, you should copy the contents of the private scrap to the desk scrap: Call ZeroScrap to clear its previous contents, and PutScrap to write data to the desk scrap.

If transferring data between the two scraps means converting it, and possibly losing information, you can copy the scrap only when you actually need to, at the time something is cut or pasted. The desk scrap needn't be copied to the private scrap unless a Paste command is given before the first Cut or Copy command since the application started up or since a desk accessory that changed the scrap was deactivated. Until that point, you must keep the contents of the desk scrap intact, displaying it instead of the private scrap in the Clipboard window if that window is visible. Thereafter, you can ignore the desk scrap until a desk accessory is activated or the application is terminated; in either of these cases, you must copy the private scrap back to the desk scrap. Thus whatever was last cut or copied within the application will be pasted if a Paste command is given in a desk accessory or in the next application. If no Cut or Copy commands are given within the application, you never have to change the desk scrap.

To find out whether a desk accessory has changed the desk scrap, you can check the ScrapCount. Save the value of this field when one of your application's windows is deactivated and a system window is activated. Check each time through the main event loop to see whether its value has changed; if so, the contents of the desk scrap have changed.

If the application encounters problems in trying to copy one scrap to another, it should alert the user. The desk scrap may be too large to copy to the private scrap, in which case the user may want to leave the application or just proceed with an empty Clipboard. If the private scrap is too large to copy to the desk scrap, either because it's disk-based and too large to copy into memory or because it exceeds the maximum size allowed for the desk scrap, the user may want to stay in the application and cut or copy something smaller.

SCRAP MANAGER ROUTINES

ScrapBootInit — Internal routine called at load time to initialize the scrap manager.

No Stack Parameters

ScrapStartup — Call made by an application before it makes any other scrap manager calls.

No Stack Parameters

ScrapShutdown — Call made by an application before shutdown if it has called ScrapStartup.

No Stack Parameters

ScrapVersion — Returns the version number of the Scrap manager

Stack Before Call

previous contents	
space for version	
	<-SP

Stack After Call

previous contents	
version number	
	<-SP

ScrapReset — Resets the Scrap Manager.

No Stack Parameters

This is called when the system reset (CONTROL-RESET is pressed).

ScrapActive — Always returns true: the scrap manager is always active.

Stack Before Call

previous contents	
space for boolean	
	<-SP

Stack After Call

previous contents	
boolean result	
	<-SP

UnloadScrap — UnloadScrap writes the desk scrap from memory to the scrap file, and releases the memory it occupied.

No Stack Parameters

If the desk scrap is already on the disk, UnloadScrap does nothing.

LoadScrap — LoadScrap reads the desk scrap from the scrap file into memory.

No Stack Parameters

If the desk scrap is already in memory, it does nothing. If the clipboard file cannot be found, no error is returned. This is just like loading an empty clipboard file.

ZeroScrap — Clears the contents of the scrap.

No Stack Parameters

ZeroScrap does not care if the scrap is memory or on disk. You must call ZeroScrap before the first time you call PutScrap.

ZeroScrap also changes ScrapCount .

PutScrap — Appends specified data to data in the scrap of the same type.

```
Stack Before Call
    |   previous contents   |
    |       length          | LONGINT
    |     ScrapType         | WORD
    |      SrcPtr           | POINTER
    |                       |<-SP

Stack After Call
    |   previous contents   |
    |                       |<-SP
```

If the scrap is on the disk it loads first. The length parameter indicates the number of bytes to write, and the ScrapType is the data type.

Warning: Don't forget to call ZeroScrap if you want to clear its previous contents.

Note: To copy the LineEdit scrap to the desk scrap, use the LineEdit function LEToScrap.

GetScrap — Copies the scrap information of the appropriate type to the specified handle setting the handle to the correct size.

```
Stack Before Call
    |   previous contents   |
    |     DestHandle        | HANDLE
    |     ScrapType         | WORD
    |                       |<-SP

Stack After Call
    |   previous contents   |
    |                       |<-SP
```

Note: To copy the desk scrap to the LineEdit scrap, use the LineEdit function LEFromScrap.

GetScrapCount — Returns the current scrap count.

```
Stack Before Call
    |   previous contents   |
    |   space for integer   |
    |                       |<-SP

Stack After Call
```

```
|  previous contents  |
|    ScrapCount       | INTEGER
|                     |<-SP
```

ScrapCount is a count that changes every time ZeroScrap is called. You can use this count for testing whether the contents of the desk scrap have changed, since if ZeroScrap has been called, presumably PutScrap was also called. This may be useful if your application supports display of the Clipboard or has a private scrap.

GetScrapState — Returns the scrapState flag. This is zero if the scrap is currently on the disk and non-zero if it is in memory.

Stack Before Call

```
|  previous contents  |
|  space for integer  |
|                     |<-SP
```

Stack After Call

```
|  previous contents  |
|    ScrapCount       | INTEGER
|                     |<-SP
```

ScrapState is actually 0 if the scrap *should* be on the disk. This may not be the case because a user can delete the Clipboard file.

GetScrapHandle — Returns a copy of the handle for the scrap of the specified type.

Stack Before Call

```
|  previous contents  |
|  Space for handle   | LONG
|    ScrapType        | WORD
|                     |<-SP
```

Stack After Call

```
|  previous contents  |
|   HandleToScrap     | LONG
|                     |<-SP
```

This call is provided so that users can access the scrap without making a copy of it. This may be important in situations where memory is in short supply.

GetScrapSize — Returns the size of the specified scrap.

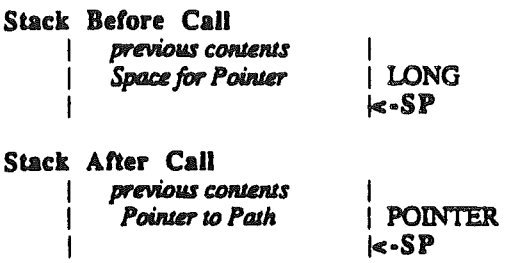
Stack Before Call

```
|  previous contents  |
|  Space for Size     | LONG
|    ScrapType        | WORD
|                     |<-SP
```

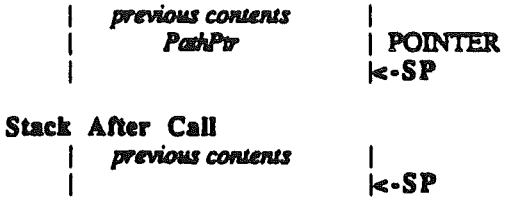
Stack After Call

```
|  previous contents  |
|       Size          | LONG
|                     |<-SP
```

GetScrapPath — Returns a pointer to the pathname used for clipboard file.



SetScrapPath — Sets the internal pointer to the clipboard file to the specified value.



Differences from the Macintosh Scrap Manager

It appears that the Macintosh Scrap Manager is capable of writing to the scrap while it is on disk without bringing it into memory. I don't see an easy way to do this so all calls to PutScrap, GetScrap and ZeroScrap do a LoadScrap if the scrap is on disk.

The Macintosh Scrap Manager's internal data structures are public. The Cortland scrap manager data structures are private. There are individual calls to provide access to any of the internal variables available on the Macintosh.

The data structures used by the Macintosh are different from those used on the Cortland.

Questions to be Answered

How will we assign scrap types? On the Macintosh, the types are four ASCII characters (equivalent to long integers). This ERS uses integers for the type (sixteen bit integers). Is there any reason not to do this?