New Technical Notes

Macintosh ®

Developer Support

AO 01 - AOCE SMPReadContent Function

AOCE

Written by: Steve Falkenburg, Scott Kuechle September 1994

This Technical Note attempts to clarify certain aspects of the AOCE Standard Mail Package SMPReadContent routine, as described in *Inside Macintosh: AOCE Application Interfaces*, pages 3-98 through 3-102, and also discusses some undocumented features of the call.

Topics

- Correct usage of the AOCE SMPReadContent function.

Introduction

The AOCE Standard Mail Package `SMPReadContent` routine is used to read a segment from a letter's standard-interchange-format block. The routine is defined as follows (refer to *Inside Macintosh: AOCE Application Interfaces*, pages 3-81 through 3-84 for a complete description):

```
pascal OSErr SMPReadContent(WindowPtr window,
                                    MailSegmentMask segmentTypeMask,
                                    void *buffer,
                                    unsigned long bufferSize,
                                    unsigned long *dataSize,
                                    StScrpRec *textScrap,
                                    ScriptCode *script,
                                    MailSegmentType *segmentType,
                                    Boolean *endOfScript,
                                    Boolean *endOfSegment.
                                    Boolean *endOfContent
                                    long *segmentLength,
                                    long *segmentID);
```

Some Sample Code, Explained

Shown below is some sample code that demonstrates the proper usage of the call. The code reads all of the styled text blocks from a letter's standard-interchange-format block.

```
#define kBufferSize 1024

OSErr DoReadMyContent(WindowPtr window);
```

```
OSErr DoReadMyContent(WindowPtr window)
{
Ptr dataBuffer;
unsigned long bufferSize,dataSize;
StScrpRec *textScrap;
ScriptCode script;
MailSegmentType segmentType;
Boolean endOfScript,endOfSegment,endOfContent;
long segmentLength,segmentID;
OSErr err;

   /* allocate data buffer */

   dataBuffer = NewPtr(kBufferSize);
   if (MemError()!=noErr)
     return MemError();
   bufferSize = kBufferSize;

   /* allocate scrap record */

   textScrap = (StScrpRec *)NewPtr(sizeof(StScrpRec));
   if (MemError()!=noErr)
     return MemError();

   /* read all of the styled text blocks */

   do
   {
     textScrap->scrpNStyles = sizeof(ScrpSTTable)/sizeof(ScrpSTElement);

     segmentID = 0;
     err = SMPReadContent(window,kMailStyledTextSegmentMask,dataBuffer,bufferSize
                          &dataSize,textScrap,&script,&segmentType,&endOfScript,
                          &endOfSegment,&endOfContent,&segmentLength,&segmentID);
     if (dataSize>0)
     {
       /* we got a styled text block */
       /* process the styled text block here */
     }

   } while ((err==noErr) && (endOfContent==false));

   DisposPtr((Ptr)textScrap);
   DisposPtr(dataBuffer);

   return err;
}
```

The `segmentID` parameter represents the ID of the segment. It is important to note that this parameter is passed by reference. It is both an input and an output. Basically, `segmentID` is used to provide random-access reference to the content in a letter. If you repeatedly pass 0 for `segmentID`, `SMPReadContent` will sequentially give you all of the content blocks in the letter. Upon return from each of the `SMPReadContent` calls, `segmentID` will be set to the ID of the segment returned. This is so you can later call `SMPReadContent` to read a specific block of content out of a letter.

This is useful in case you want to know what all of the blocks are in a letter without actually reading the data associated with all of the blocks. For example, you might only want to

download a QuickTime movie in a letter if the user clicks on the spot where the movie is in the letter.

Also, it is possible to determine the size of a given segment without actually having to read the data into a buffer. Simply pass 0 in the `bufferSize` parameter and the length of the segment will be returned in the `segmentLength` parameter. This is nice in that it enables you to to see ahead of time how much buffer space you will need to allocate for a given segment (in case you need to retrieve it at a later time, for example).

In our example above we always set `segmentID` to 0, since we just want to index through all of the available content.

Some other things to note in the above code:

- it's necessary to re-set `segmentID` to 0 for each call, otherwise, the system will treat the non-zero `segmentID` as an index into the content you just read, and you'll get the same content over and over

- it's necessary to set `textScrap->scrpNStyles` for each call to `SMPReadContent` that returns scrap information. Again, this field is both an input and output. On input, it specifies how many styles your scrap record can hold, and on output, it returns how many were read

- even though we're only requesting data on styled text content blocks (`kMailStyledTextSegmentMask`), `SMPReadContent` will return the `segmentID`, `segmentLength`, and `segmentType` for other non-styled text blocks in the letter. However, `SMPReadContent` will always return a `dataSize` of 0 for these blocks, and will always set `endOfBlock` to true for these blocks (since you don't want to read them at all). This is useful, since you could set the `segmentTypeMask` to 0 and still get a list of all the blocks in a letter without actually reading any information

- the `segmentID`, `segmentLength`, and `segmentType` fields are only valid for the first `SMPReadContent` call *within* a segment. Our buffer size in the above code is 1024 bytes. If the styled text block we were reading were bigger than 1024 bytes, `SMPReadContent` would return information out of this larger block across multiple calls. Only in the first call for the block are these three parameters valid as outputs. You should still always reset `segmentID` to 0 (or the segment you want to read from), however.

- this last point seems obvious, but don't declare your scrap record as a non-dynamically allocated local variable. This structure is *huge* and will eat up all of your stack space. Use `NewPtr` to allocate the structure instead.

Further Reference:

- *Inside Macintosh : AOCE Application Interfaces*