Technical Note TB41

Translation Manager 1.1

CONTENTS

This Technical Note discusses changes to the Translation Manager which are available in Macintosh Easy Open version 1.1 and later. The information contained here is in addition to what is discussed in *Inside Macintosh* More Macintosh Toolbox, Translation Manager chapter, as well as in the *APDA* Macintosh Easy Open Developers Kit.

[Jun 01 1994]

Introduction

The document assumes that you are somewhat familiar with the Translation Manager API.

Some of the new API's are available only in Translation Manager 1.1 and some have always been available. Use `Gestalt()` as appropriate to check for the existence of some of the new API's (discussed later in this document).

```
gestaltTranslationMgrHintOrder = 1
```

The previous bit will be set if the Translation Manager hint order fix is in effect. In Translation Manager version 1.0.1 (fixed in version 1.0.2 and later) there is a bug where the hints in `DoTranslateScrap()` are reversed - the `dstTypeHint` is actually the `srcTypeHint` and vice-versa. If you want your Translation Extension to work with the early versions of the Translation Manager, and you're these using these scrap hints, then you need to check this bit to see where your destination and source hints are.

`gestaltTranslationPPCAvail = 2`

If `gestaltTranslationPPCAvail` bit is set then that is an indicator that the native PowerPC Translation Manager library is available. This means that it's safe to make a call to the Translation Manager from native PowerPC code.

`gestaltTranslationGetPathAPIAvail = 3`

The new API's `GetFileTranslationPath()` and `GetPathTranslationDialog()` calls are available if the `gestaltTranslationGetPathAPIAvail` bit is set.

`GetDocumentKindString()` is available in all versions of the Translation Manager, therefore no gestalt selector is needed to check for its existence.

The new Translation Extension routine `DoGetTranslatedFilename()` does not have a gestalt selector since there is no compatibility problem running a Translation Extension that supports the call on an earlier version of the Translation Manager. In that case, the function will simply not be called.

The New Translation Manager API's

One of the most glaring limitations of the 1.0 Translation Manager API was that the function `CanDocBeOpened()` couldn't be used in the case where a translation preference did not exist. You would have to set the translation preference manually using the automatic translation user interface in the Finder or Standard File. This meant for all purposes, that you could not programmatically use the Translation Manager.

Starting with Translation Manager 1.1 this limitation has been removed with two new API's `GetPathFromTranslationDialog()` and `GetFileTranslationPaths()`. The function `GetPathFromTranslationDialog()` allows you to post the Translation Dialog box programmatically while the function `GetFileTranslationPaths()` is a low level access routine allowing you to get all the translation capabilities of the Translation Manager.

Additional API's have been added to provide access to the kind strings, the name of Translation Extensions, as well as to programmatically execute scrap translations. These will be discussed later in this section.

Displaying The Translation Dialog Box Programmatically

```
pascal OSErr GetPathFromTranslationDialog(
    const FSSpec* theDocument,
    const FSSpec* theApplication,
    const TypesBlockPtr typeList,
    DocOpenMethod* howToOpen,
    FileTranslationSpec* howToTranslate)
```

As mentioned in the earlier, `GetPathFromTranslationDialog()` is used to programmatically display the Translation Dialog box. This has the "side-effect" of setting a preference so next time `CanDocBeOpened()` is called it will have the preference and succeed.

`GetPathFromTranslationDialog()` targets the file specified by the `theDocument` and attempts to generate a list of translation paths resulting in a document readable by the target application specified by the `theApplication`. The parameter `typeList` specifies, in a `0L` terminated form, a list of file types to translate the target document into. The order in the list is important. The first item should be the file type most desired and the last item should be the file type least wanted. The following two parameters `howToOpen` and `howToTranslate` are returned once the user has interacted with the Translation Dialog box and will contain the translation open method and the translation specification. It's important to point out the `howToTranslate` is only valid if `howToOpen` is equal to `domTranslateFirst`, otherwise it's undefined.

Note:
None of the parameters are optional and `NULL` cannot be passed in place of them.

A common way of using this routine along with `CanDocBeOpened()` can be seen in the following snippet of code:

```
/* Translate

   This routine translates a file.  It assumes that the
   application has the signature 'ttxt' and can only can
   read the document types 'ttro' and 'TEXT' */

OSErr Translate(const FSSpec* targetDocument,
                const FSSpec* destinationDocument,
                const FSSpec* theApplication)
{
OSType              typeList[3] = { 'ttro', 'TEXT', 0L };
DocOpenMethod       howToOpen;
FileTranslationSpec  howToTranslate;
OSErr               result;

   /* Try to get the translation path based on the
   preference (if one is set) */

result = CanDocBeOpened(targetDocument,
                        theApplication->vRefNum,
                        'ttxt',
                        typeList,
                        false,
                        &howToOpen,
                        &howToTranslate);

   /* Did it work? */

if (result == noPrefAppErr) {

       /* Couldn't find a path, run the Translation
       Dialog box to get the path and set the preference */

  result = GetPathFromTranslationDialog(targetDocument,
                                        theApplication,
                                        typeList,
                                        &howToOpen,
                                        &howToTranslate);
  }

       /* Did we get a path from either CanDocBeOpened
       GetPathFromTranslationDialog, and does the
       translation path specified require translation? */

if ((result == noErr)
&& (howToOpen == domTranslateFirst)) {

       /* Translate the file */

  result = TranslateFile(targetDocument,
                         destinationDocument,
                         &howToTranslate);
```

```
      }

    return result;
    }
```

Getting All The Translation Paths

At the low level, the new routine `GetFileTranslationPaths()` can be used to get raw translation paths. The paths are each of the translation paths that allow a specific document to be translated to the target type (under some constraints that are discussed later). A specific translation may have one or many paths - that depends on the translation itself and the capabilities of the Translation Extensions installed.

`GetFileTranslationPaths()` is defined as:

```
     pascal short GetFileTranslationPaths(
       FSSpec* srcDocument,
       FileType dstDocType,
       unsigned short maxResultCnt,
       FileTranslationSpecArrayPtr resultBuffer)
```

Both `srcDocument` and `dstDocType` are optional parameters. `srcDocument` is the source document (if any) and `dstDocType` is the desired document type to which you would like `srcDocument` translated. Depending on what is passed, the routine returns a different set of translation paths, as seen in Figure 1.

```
srcDocumentdstDocType
```

Translation Paths Returned

valid	valid	All paths to translate `srcDocument` to `dstDocType`
NULL	valid	All paths to tranlsate to `dstDocType`
valid	0	All paths from `srcDocument`
NULL	0	All translation paths

Figure 1

The parameter `maxResultCnt` is the maximum number of entries your `resultBuffer` can hold.

The final parameter `resultBuffer` is returned with the requested translation information. This buffer's type is defined as:

```
    struct FileTranslationSpec {
      OSType          componentSignature;
   const void*      translationSystemInfo;
   FileTypeSpec     src;
   FileTypeSpec     dst;
      };

    typedef struct FileTranslationSpec FileTranslationSpec;

    typedef FileTranslationSpec *FileTranslationSpecArrayPtr;
```

The function returns the number of translation paths, or a result code if the value is negative.

The following example shows a how to get the list of translation paths to open a specific document and how to translate using the first path in the list.

```
/* TranslateUsingFirstPath

This routine translates a file to 'SYLK' using the
first translation path in the Translation Manager
path list. */

OSErr Translate(FSSpec* targetDocument,
                FSSpec* destinationDocument,
                FSSpec* theApplication) {

FileTranslationSpec ts[10];
OSErr               result;
short               numberPaths;

   /* Try to get the translation path */
numberPaths =  GetFileTranslationPaths(targetDocument,
                                       'SYLK',
                                       10,
                                       &ts[0]);

if (numberPaths > 0)
  {
  result = TranslateFile(targetDocument, destinationDocument, ts);
  }
else
  result = noTypeErr;

return result;
}
```

Getting Kind Strings

Kind strings describe documents; for instance "FreeHand graphic". Previously the kind strings were displayed in the Finder, but there was no programmatic way of accessing them. The Translation Manager now provides the means to get to the kind strings, as well as the names of the Translation Extensions installed.

The routine to get a kind string looks like:

```
OSErr GetDocumentKindString(short  docVRefNum,
                            OSType docType,
                            OSType docCreator,
                            Str63  kindString)
```

This routine takes in the `docVRefNum` parameter, the volume containing the document. This is a hint to the Translation Manager where to look for the kind string. If it doesn't find the string on that volume, it will use an internal search path to look on other volumes for the string. In `docType` and `docCreator` you pass the type and creator of the document you want to query. When the function returns, `kindString` contains the kind string to display for that specific document.

If you have a `FileTranslationSpec` and you want to find out the name of the Translation Extension that's performing the translation, you call:

```
pascal OSErr GetTranslationExtensionName(
    const FileTranslationSpec* translationMethod,
    Str31 extensionName)
```

This routine takes a `FileTranslationSpec` (returned from `CanDocBeOpened()` or `GetFileTranslationPaths()`) and returns, in `extensionName`, the name of the Translation Extension performing the translation.

Both of these routines can be used, for example when using `GetFileTranslationPaths()` to create your own Translation Dialog box. Using these will allow you to generate strings like "MacWrite II document with XYZZY translation" and so forth.

Scrap Translation

An additional routine has been made public in the Translation Manager allowing you to perform Scrap translation. The name scrap translation is somewhat misleading; rather it's in-memory translation. Scrap translation is used by the Scrap Manager, Edition Manager, Drag Manager, and OpenDoc to name a few clients for in-memory translation. Scrap translation

can be used any time you want to translate a buffer of information.

The routine to call when you want to perform scrap translation is:

```
pascal OSErr TranslateScrap(
  GetScrapDataProcPtr sourceDataGetter,
  void*               sourceDataGetterRefCon,
  ScrapType           destinationFormat,
  Handle              destinationData,
  short               progressDialogID)
```

The routine is designed in a way to use a callback you provide to get the source data to translate. That information is then translated and placed into a destination handle.

The parameter `sourceDataGetter` and `sourceDataGetterRefCon` are the two parameters dealing with your callback routine. `sourceDataGetterRefCon` is for your own use - allowing you to pass information to your callback. The parameter `souceDataGetter` is defined as:

```
typedef pascal OSErr (*GetScrapDataProcPtr)(
  ScrapType requestedFormat,
  Handle    dataH,
  void* srcDataGetterRefCon);
```

The callback routine has two responsibilities. The first is to tell the caller what source types are available for translating (if you are the Scrap Manager for example you would pass all the different formats already on the desk scrap). The other responsibility is to actually provide the data requested.

For the first case, if the parameter `requestedFormat` is `'fmts'`, then it's the responsibility of the callback routine to return in `dataH` a list of pairs containing the `ScrapType` and the size of the that `ScrapType`'s data. The handle should be re-sized accordingly.

For the second case, the Translation Manager will call the callback routine with one of the types it had provided earlier in response to `'fmts'`. The callback in that case is responsible for re-sizing the `dataH` and placing in it the data of type `requstedFormat`.

In the callback, the parameter `sourceDataRefCon` is the same as what you had passed in the `sourceDataGetterRefCon` field in `TranslateScrap()`.

Back in `TranslateScrap()`, the third parameter, `destinationFormat` is the desired format you would like the information translated into. `destinationData` is a handle you provide. The Translation Extension will automatically re-size it as necessary during translation. Upon exit, if the routine successfully executes, it will contain the translated information.

The final parameter is `progressDialogID`. At this time that parameter should always be assigned the value `TranslationScrapProgressDialogID`.

```
struct FmtsRecord
{
  ScrapType theType;
  Size      dataSize;
};
typedef struct FmtsRecord FmtsRecord;
typedef FmtsRecord *FmtsRecordPtr;

pascal OSErr PStringGetter(
      ScrapType requestedFormat,
      Handle    dataH,
      void*     srcDataGetterRefCon)
{
OSErr      result;
Str63      thePString;

/* Build an internal buffer to the PString we can get
BlockMove("\pHello there, this is lower case",
      &thePString, sizeof(Str63));

  /* See if we are being requested to tell what
   source format's we have available */

if (requestedFormat == 'fmts') {

  /* Size the handle to contain our one source
  format */

  SetHandleSize(dataH, sizeof(FmtsRecord));
  if ((result = MemError()) == noErr) {

      /* Stuff our data into the fmts handle */

    ((FmtsRecordPtr)*dataH)->theType = 'PSTR';
    ((FmtsRecordPtr)*dataH)->dataSize = thePString[0]+1;
    }
} else {

      /* If we're here, we've been asked to get the
      source data.   Stuff data into handle */

  SetHandleSize(dataH, thePString[0]+1);
  if ((result = MemError()) == noErr)
    BlockMove(&thePString, *dataH, thePString[0]+1);
  }

return result;
 }
void TranslatePSTRToUPPR(void)
{
     Handle      destinationData;
     OSErr       result;

     /* A handle to fill with the translated data */

     if (destinationData = NewHandle(0))
         result = TranslateScrap(PStringGetter,
             0,
             'UPPR',
             destinationData,
             TranslationScrapProgressDialogID);

     /* Do something with the translated information */

}
```

The preceding example shows how to use the `TranslateScrap()` routine and how to implement a data-getter. In the above example, `TranslateScrap()` is called requesting type `'UPPR'` (upper case) and providing the source data through the data-getter referenced in `PStringGetter`.

New Translation Extension Capabilities

Beginning with Translation Manager 1.1, Translation Extensions now have the capability of generating the filenames of documents created by a document converter. This is useful, for example, if your Translation Extension wants to provide a DOS compatible filename for generated documents. Name generation is done by implementing the new Translation Extension routine (selector `kTranslateGetTranslatedFilename`).

```
pascal ComponentResult DoGetTranslatedFilename(
  ComponentInstance self,
  FileType dstType,
  long dstTypeHint,
  FSSpec* theDocument);
```

This routine is called after `DoIdentifyFile()`, but before `DoTranslateFile()`.

The first parameter, like all other references to self in the Component Manager, is a reference to this instance of the component. `dstType` is the target type and `dstTypeHint` is the hint that goes with that type - they are the same that will be passed to `DoTranslateFile()` when that call is made by the Translation Manager. The final parameter, `theDocument` is the document to generate a name on - and where to store your generated name.

It's important to not modify `theDocument` unless your routine successfully completes because whatever is returned will be used (even if you return an error).

Your Translation Extension should verify the uniqueness of the filename in the target location before returning.

The Translation Manager will not call your Translation Extension with this message unless you have the translation flags correctly set. That is done by modifying the `'thng'` resources `Flags` field, and setting the `kTranslatorCanGenerateFilename` bit.

PowerPC Translation Extensions

Native Translation Extensions are pretty much the same as 68K ones. The biggest difference is putting a wrapper around the code. You have several options when writing a translator. Your translator can be 68K, your translator can be PowerPC only, or your translator can be fat (both 68K and PowerPC).

The best of both worlds is the third case. Your translator is a bit bigger, but a user can simply drag it from machine to machine without worrying about the platform. To implement a native PowerPC translator, simply replace your code referred by your `'thng'` resource (usually type `'xlat'`) with a reference to a resource wrapped by a `'sdes'` resource (see MixedMode.r). An `'sdes'` includes both your 68K and PowerPC code into a single resource and automatically dispatches to the correct 68K or PowerPC code depending on what platform you are on.

For specific examples on putting together a fat resource, please consult *Inside Macintosh* , specifically, the chapter on the Mixed Mode Manager.

Back to top

References

Inside Macintosh , More Macintosh Toolbox, Translation Manager

Inside Macintosh , More Macintosh Toolbox, Component Manager

Inside Macintosh , PowerPC System Software, Mixed Mode Manager

APDA , Macintosh Easy Open Developers Kit

Back to top

Downloadables

Acrobat version of this Note (K). Download