Text Encoding Converter ERS

Version 7
Wednesday, February 12, 1997

Tom Naughton

Text Encoding Converter ERS Friday February 14, 1997

Text and International Engineering
Extension: 4-2076
eMail: naughton@apple.com

1.0 Background

The Unicode converter provides table-based conversion to or from Unicode. It does not directly provide general conversion between any two encodings (although it can be used as part of this process), nor does it handle algorithmic code conversions (such as JIS to Shift-JIS) or multi-encoding streams (using ISO-2022 or EUC, for example).

The High Level Text Encoding Converter (a.k.a. HLTEC, HLEC, and TEC) is a layer above the Unicode converter to provide these services in a general way. If not:

- many applications may have to provide their own routines for this (thus resulting in much duplicate code and bigger applications)
- specific APIs may be provided as necessary in certain localized versions of system software, resulting in a proliferation of localization-specific—something we definitely want to avoid, since it makes it difficult for developers to write a single application that works everywhere.

1.1 Dependencies of this Document

For a description of the TextEncoding data type see the ERS for the Unicode converter. Important concepts that need to be understood to fully understand this document include Text encoding bases, variants, and formats.

2.0 Requirements

This layer provides conversion between arbitrary encodings. This involves a combination of the following techniques:

- Table-based conversion to or from Unicode, using the Unicode converter. A single X-to-Y conversion may involve converting X to Unicode, and then converting Unicode to Y. Intermediate storage for the Unicode form is handled by the Text Encoding Converter.

- Algorithmic conversion, using plug-in code modules. These plug-ins are implemented as code fragments.

- Maintaining and updating the current state (the current encoding and other relevant

information) for multi-encoding streams. It also handles detecting escape sequences, special control characters, and any other tags which change the current encoding state. This information is stored in the converter object and maintained by the plug-ins performing the conversion.

2.1 Performance Goals

One immediate requirement for the converter is to perform code conversions for mail and other text sent and received over the internet. Conversions should be fast enough that a user doesn't notice the time delay for an average sized web page or email to be converted.

2.1.1 Code Size

The Text Encoding Converter represents entirely new capabilities for the Macintosh, so there are no memory or disk footprint requirements based on existing use by applications. Until such requirements emerge, we will use worst-case requirements.
The Text Encoding Converter, when compiled for the PowerPC and not counting any plugins, must occupy less than 100 kilobytes in memory and on disk. The fat version for PowerPC and 68K must be less than 150 kilobytes.

Here are the current approximate code sizes in bytes for the various components.

	PowerPC	68K
Text Encoding Converter	24,288	15,078
Japanese Encodings	11,312	7,074
Chinese Encodings	8,140	3,998
Korean Encodings	6,580	2,450
Unicode plugin	11,700	7,278

The Unicode plugin will also pull in the entire low level (Unicode) text encoding converter code fragments. See that documentation for approximate memory requirements.

2.1.2 Speed Measurements

Like the footprint requirements, the performance requirements are worst-case and are not intended to serve as realistic performance estimates. In most cases we expect to do much better than these requirements.

On average, an application must be able to perform a round-trip conversion on a 255 character MacRoman string in less than half a millisecond on a 6100/60 with 8 megabytes

of RAM, if the converter and plugins are already in memory.

Here are timings in microseconds of the same five conversions being done three times in a row on a 9500/132. The later times are faster because certain resources have been cached. Conversions involving the Unicode converter tend to take a bit longer than those performed entirely through the high level converter.

Conversion	Create	Convert	In	Out
First Try				
Latin1toMacRoman	2703	1743	32	32
2022toSJIS	845	865	454	334
EUCtoSJIS	293	109	230	230
UTF7toMacJapanese	1192	1478	126	92
Latin1toUTF7	1989	804	67	90
Second Try				
Latin1toMacRoman	2070	1008	32	32
2022toSJIS	845	845	454	334
EUCtoSJIS	293	109	230	230
UTF7toMacJapanese	1212	1437	126	92
Latin1toUTF7	2213	886	67	90
Third Try				
Latin1toMacRoman	2070	1029	32	32
2022toSJIS	845	845	454	334
EUCtoSJIS	293	89	230	230
UTF7toMacJapanese	1396	1437	126	92
Latin1toUTF7	2009	1008	67	90

Create - Time in microseconds to create the converter object. This creation time includes the conversion path search.
Convert - Time in microseconds to perform encoding conversion.
In - Number of text bytes in input buffer

Out - Number of text bytes placed in output buffer

3.0 Architecture

Conversion routines are provided as plug-in components implemented as code fragments. The main export symbol of each fragment is a routine which returns a pointer to a table containing a plug-in signature, table version information, and hooks to each of the plug-ins methods. Each plug-in can be polled for the encodings it supports and is responsible for handling all conversions between its supported encodings. The High Level Encoding Converter will decide how best to meet a callers conversion requirements using the conversion resources available to it. This may involve stringing together multiple encoding converters. The caller is shielded from this complexity and treats an encoding converter object as a single entity regardless of its actual structure.

When using the converter an application does not need to be aware of the plug-ins available. Each of the APIs that returns information about the available text encoding conversion services polls all plug-ins and makes them appear as one large plug-in.

3.1 SOM vs. CFM

The plug-in model is based on CFM rather than SOM. This is because SOM was not felt to offer any substantial benefits over CFM for this application, while it would impose a definite cost in runtime overhead, and an unknown but potentially significant cost in implementation complexity. The TextEncoding plug-ins are modeled after those of the Mac OS 8 I/O families.

3.2 Error Codes

All APIs that call through to plug-ins return an OSStatus. Though some may never return an error, this allows plug-ins the opportunity to return errors as necessary. For example a plug-in may need to return an error status code when performing a simple action requires disk or network access and an error occurs.

3.2.1 Errors

The High Level Text Encoding Converter will use the following error codes:

ERROR CODE INFORMATION IN THE REST OF THE ERS NEED TO BE UPDATED

kTextUnsupportedEncodingErr	None of the currently insalled plugins recognizes an encoding
kTECCorruptConverterErr	Invalid converter object reference was passed to a TEC API

TECMalformedInputErr	Error in input text (bad escape sequence, illegal multibyte character)
TECNoConversionPathErr	Could not find a conversion path between the source and destination encodings
kTECBufferBelowMinimumSizeErr	Conversion did not take place because the output buffer is too small to contain the next character
kTECPartialCharErr	Conversion cannot continue because the input buffer has been truncated in the middle of a multibyte character
kTextUndefinedElementErr	Conversion was stopped because a character that could not be converted to the output encoding was found in the input
kTECNoConversionPathErr	A converter could not be created because there was no direct conversion routine available and no path of multiple conversions routines.
kTECBufferBelowMinimumSizeErr	The output buffer is too small for the next text element
kTECNeedFlushStatus	Conversion will not be complete until the converter is flushed using TECFlushText
kTECOutputBufferFullStatus	There is no more room in the output buffer for the next text element.

The following errors can occur during single to multiple/multiple to single encoding conversions.

TECBadTextRunErr	Conversion was stopped because text in the input buffer lies outside of all encoding runs
TECTextRunBuffFullErr	Conversion was stopped because there was not enough room in the encoding runs output buffer to continue

ERROR CODE INFORMATION IN THE REST OF THE ERS NEEDS TO BE UPDATED

4.0 APIs

4.1 Available encoding Information

TECCountAvailableTextEncodings returns the largest number of encodings that will be returned by the routine TECGetAvailableTextEncodings. This will allow the caller to supply a buffer of adequate size to TECGetAvailableTextEncodings.

```
OSStatus TECCountAvailableTextEncodings(ItemCount *numberEncodings);

errors:
   plug-in defined errors
```

TECGetAvailableTextEncodings will fill in an array of type TextEncoding passed in by the user with information about the types of encodings the current configuration of the encoder can handle. The number of encodings may be slightly smaller than the number returned by TECCountAvailableTextEncodings, since it removes duplicate encodings supported by different plug-ins. This actual number can't be known until the list is constructed.

```
OSStatus TECGetAvailableTextEncodings(TextEncoding
   availableEncodings[], ItemCount maxAvailableEncodings, ItemCount
   *actualAvailableEncodings);

errors:
   plug-in defined errors
```

TECGetSubTextEncodings and TECCountSubTextEncodings are used to get a list of encodings that are contained in an encoding scheme. For example EUC_JP contains ISO 8859-1, JIS 208, JIS 212, and half-width katakana. Not every encoding that can be broken down into multiple encodings will necessarily support this routine. It's up to the plug-in developer to decide with encodings might be useful to break up.

```
OSStatus TECCountSubTextEncodings(TextEncoding inputEncoding,
   ItemCount *numberOfEncodings);

errors:
   UnknownEncodingErr

OSStatus TECGetSubTextEncodings(TextEncoding inputEncoding,
   TextEncoding subEncodings[], ItemCount maxSubEncodings, ItemCount
   *actualSubEncodings);

errors:
   UnknownEncodingErr
```

TECGetTextEncodingLocalizedName returns a string representing the name of the encoding in a caller-specified language and encoding, if the specified language is not available, the converter will try to return something universal. These strings could be displayed as part of the calling application's user interface.

The name of an encoding could be returned in a number of different encodings. For

example in Japanese the name could be either in MacJapanese or Unicode. For this reason the caller supplies a preferred encoding. This could allow a plugin to store names in multiple encodings and choose the encoding that a caller requests.

```
extern pascal OSStatus TECGetTextEncodingLocalizedName(TextEncoding
   textEncoding, LocaleIdentifier locale, TextEncoding
   preferedEncoding, ByteCount bufLen, TextEncoding *nameEncoding,
   ByteCount *nameLength, Byte encodingName[])

errors:
   UnknownEncodingErr
```

TECGetEncodingInternetName is used to return a string containing the encoding names approved for use in HTML encoding tags. TECGetEncodingFromInternetName performs the reverse function. All names are returned as a Str255 in 7 bit US-ASCII since that is the encoding they are defined and most commonly found in.

```
OSStatus TECGetTextEncodingInternetName(TextEncoding textEncoding,
   Str255 encodingName)

OSStatus TECGetTextEncodingFromInternetName(TextEncoding
   *textEncoding, ConstStr255Param encodingName)

errors:
   UnknownEncodingErr
```

4.1.1 Text Encoding Conversion Info

A TECConversionInfois a structure used to describe conversion services available in a plug-in converter. Each plug-in is required to implement a routine that returns the number of pairs it supports, and a second routine to return the actual pairs in a buffer provided by the caller.

Each pair contains a source and destination encoding which describes a kind of conversion the plug-in can perform. There are also two sets of 16 bits reserved for future use. These bits could provide more information about the given conversions routine such as the relative speed of the routine, whether the conversion is a lossy one, or the conversion options it supports.

```
struct TECConversionInfo{
   TextEncoding      sourceEncoding;
   TextEncoding      destinationEncoding;
   UInt16            reserved1;
   UInt16            reserved2;
};
```

TECCountDirectTextEncodingConversions returns the number of encoding pair structures that will be returned by the routine TECGetAvailableTextEncodingConversions. This will allow the caller to supply a buffer of the correct size to TECGetDirectTextEncodingConversions.

```
OSStatus TECCountDirectTextEncodingConversions (ItemCount
   *numberOfEncodings);

errors:
   plug-in defined errors
```

TECGetDirectTextEncodingConversions will fill in an array of type TextEncodingConversionPair passed in by the caller with information about the types of conversions the current configuration of the encoder can handle.

```
OSStatus TECGetDirectTextEncodingConversions
   (TECConversionInfoavailableConversions[], ItemCount
   maxAvailableConversions, ItemCount *actualAvailableConversions);

errors:
   plug-in defined errors
```

Given a text encoding, TECCountDestinationTextEncodings returns the number of encodings it can be converted into using a single-stage (direct) converter.

```
OSStatus TECCountDestinationTextEncodings(TextEncoding inputEncoding,
   ItemCount *numberOfEncodings);

errors:
   plug-in defined errors
```

Given a TextEncoding, TECGetDestinationTextEncodings will fill in an array of type TextEncoding passed in by the caller with information about the Encodings the current configuration of the encoder can convert the original TextEncoding into using a single-stage converter.

```
OSStatus TECGetDestinationTextEncodings(TextEncoding inputEncoding,
   TextEncoding destinationEncodings[], ItemCount
   maxDestinationEncodings, ItemCount *actualDestinationEncodings);


errors:
   plug-in defined errors
```

4.2 Investigating Encodings

Given a stream of bytes in an unknown encoding and an array of possible encodings, TECSniffTextEncoding will return counts of "errors" and "features" for each of the encodings. Each error indicates a code point or sequence that is illegal in a specified encoding while a feature indicates the presence of a sequence that is characteristic of that encoding.

For example the string "äøéö" could legally be interpreted as MacRoman or MacJapanese. Both sniffers would return zero errors, but the MacJapanese sniffer would also return 2 features of MacJapanese (representing two legal two byte characters.)

Before returning, the passed-in arrays are sorted in order of probability of being the encoding of the sample buffer. The results are sorted first by fewest number of errors, then by greatest number of features, and then by the original order in the list. Upon return from the routine, the caller can assume the correct encoding is in testEncodings[0], or possibly testEncodings[1].

```
pascal OSStatus TECCountSniffers(ItemCount *numberOfSniffers);
pascal OSStatus TECGetSniffers(TextEncoding sniffers[], ItemCount
   maxSniffers, ItemCount *actualSniffers);
pascal OSStatus TECCreateSniffer(TECSniffertRef *encodingSniffer,
   TextEncoding testEncodings[], ItemCount numTextEncodings,);
pascal OSStatus TECSniffTextEncoding(TECObjectRef *encodingSniffer,
   TextPtr inputBuffer, ByteCount inputBufferLength,  ItemCount
   numErrsArray[], ItemCount maxErrs, ItemCount numFeaturesArray[],
   ItemCount maxFeatures);
pascal OSStatus TECDisposeSniffer(TECSniffertRef encodingSniffer);
```

4.3 Creating a Converter Object

When the caller supplies valid source and destination encodings, TECCreateConverter will create a conversion object with the required default characteristics, the defaults should reflect what most users would need in each specific conversion. These defaults can be changed with the option getter and setter functions described in section 4.6.

```
OSStatus TECCreateConverter(TECObjectRef *newEncodingConverter,
   TextEncoding inputEncoding, TextEncoding outputEncoding);

errors:
   TECNoConversionPathErr
   TECUnknownEncodingErr
   Memory Manager errors
   plug-in defined errors
```

While TECCreateConverter will find a path from the source to the destination encoding of one exists in the current plug-in configuration, TECCreateConverterFromPath can be used

to specify a specific path

```
OSStatus TECCreateConverterFromPath(TECObjectRef
   *newEncodingConverter, TextEncoding inPath[], ItemCount
   inEncodings);

errors:
   TECUnknownEncodingErr
   Memory Manager errors
   plug-in defined errors
```

TECDisposeConverter disposes of an encoding converter object. The object should no longer be used after this call is made.

```
OSStatus TECDisposeConverter(TECObjectRef newEncodingConverter);

errors:
   TECCorruptConverterErr
   plug-in defined errors
```

4.4 Converting Encodings

During encoding conversion the Text Encoding Converter may need to create temporary buffers to contain intermediate results, in the current implementation these buffers are one third the size of the output buffer. These buffers are only allocated while a conversion is being performed, and will always be deallocated upon return from TECConvertText.

TECClearContext clears any internal status information saved in a converter object. Calling this function will return a converter object back to the same state it was in when it was created. Multiple conversions can be performed with the same converter object by calling this function between conversions.

```
OSStatus TECClearContext(TECObjectRef encodingConverter);

errors:
   TECCorruptConverterErr
   plugin defined errors
```

TECConvertText converts the specified stream of bytes to the desired encoding according to the parameters in the specified encoding converter object. On exit the actualInputLength will indicate how much of the input buffer was consumed and actualOutputLength will indicate how much text was placed in the output buffer. This can be useful in cases where the output buffer is not large enough to contain the entire converted text. In such cases TECConvertText will return TECNotDoneStatus and can be called repeatedly until the entire text has been converted.

```
OSStatus TECConvertText(TECObjectRef encodingConverter, TextPtr
   inputBuffer, ByteCount inputBufferLength, ByteCount
   *actualInputLength, TextPtr outputBuffer, ByteCount
   outputBufferLength, ByteCount *actualOutputLength)

errors:
   TECCorruptConverterErr
   TECBuffTooSmallErr
   TECPartialCharErr
   TECBadCharErr
   TECMalformedInputErr
   TECNotDoneStatus
   plugin defined errors
   Memory Manager errors
```

TECFlushText is called at the end of the conversion process to flush out any data that may be stored in a converter's temporary buffers or perform other end of encoding conversion functions. Encodings such as ISO-2022 that need to shift back to a certain default state at the end of a conversion can do so when this API is called.

```
OSStatus TECFlushText(TECObjectRef encodingConverter, TextPtr
   outputBuffer, ByteCount outputBufferLength, ByteCount
   *actualOutputLength);

errors:
   TECBuffTooSmallErr
   plug-in defined errors
```

4.5 Multiple Encoding Run Conversions

NEEDS UPDATE

The following APIs allow conversion to and from multiple encoding runs. This will make possible conversion of Unicode to multiple Mac encodings and vice versa for display using the Script Manager under System 7. It could also be used to break up a multiple encoding packaging format such as ISO 2022 or EUC into runs of constituent encodings.

TECCreateOneToManyConverter takes an input encoding and a list of destination encodings and returns a converter object that will break the input encoding into runs of any of the destination encodings.

```
extern pascal OSStatus TECCreateOneToManyConverter(TECObjectRef
   *newEncodingConverter, TextEncoding inputEncoding, ItemCount
   numOutputEncodings, const TextEncoding outputEncodings[]);
```

```
errors:
   TECNoConversionPathErr
   TECUnknownEncodingErr
   Memory Manager errors
   plug-in defined errors
```

TECCreateManyToOneConverter takes a destination encoding and a list of input encodings and returns a converter object that will convert all runs of the input encodings into a single run of the destination encoding.

```
extern pascal OSStatus TECCreateManyToOneConverter(TECObjectRef
   *newEncodingConverter, ItemCount numInputEncodings, const
   TextEncoding inputEncodings[], TextEncoding outputEncoding);

errors:
   TECNoConversionPathErr
   TECUnknownEncodingErr
   Memory Manager errors
   plug-in defined errors
```

The following two APIs convert text from or to multiple text runs. The runs are delimited using an array of structures of type TextEncodingRun. Each run contains an offset from the beginning of the text buffer and the text encoding of the run.

```
struct TextEncodingRun {
   ByteOffset                                  offset;
   TextEncoding                                textEncoding;
};
```

TECConvertTextToMultipleEncodings takes an input text buffer, an output text buffer, and an output buffer for the text encoding runs. On return it returns information about how much of the input buffer was consumed and how much of the output buffers were used.

```
OSStatus TECConvertTextToMultipleEncodings(TECObjectRef
   encodingConverter, ConstTextPtr inputBuffer, ByteCount
   inputBufferLength, ByteCount *actualInputLength, TextPtr
   outputBuffer, ByteCount outputBufferLength, ByteCount
   *actualOutputLength, TextEncodingRunPtr outEncodingsBuffer,
   ByteCount outEncodingsBufferLength, ByteCount
   *actualOutEncodingsLength);

errors:
   TECCorruptConverterErr
   TECBuffTooSmallErr
   TECTextRunBuffFullErr
```

```
    TECPartialCharErr
    TECBadCharErr
    TECMalformedInputErr
    TECNotDoneStatus
    plugin defined errors
    Memory Manager errors
```

TECConvertTextFromMultipleEncodings takes a text input buffer an input text encoding run buffer, and an output text buffer. The multiple runs of input text are converted into the single destination encoding and placed in the output text buffer.

```
OSStatus TECConvertTextFromMultipleEncodings(TECObjectRef
    encodingConverter, ConstTextPtr inputBuffer, ByteCount
    inputBufferLength, ByteCount *actualInputLength,
    ConstTextEncodingRun inEncodingsBuffer[], ByteCount
    inEncodingsBufferLength, ByteCount *actualInEncodingsLength,
    TextPtr outputBuffer, ByteCount outputBufferLength, ByteCount
    *actualOutputLength);

errors:
    TECBadTextRunErr
    TECCorruptConverterErr
    TECBuffTooSmallErr
    TECPartialCharErr
    TECBadCharErr
    TECMalformedInputErr
    TECNotDoneStatus
    plugin defined errors
    Memory Manager errors
```

4.6 Specifying Conversion Options

Conversion options might include such information as whether to convert half width Japanese Katakana characters to full width or whether an error should be generated when a conversion fallback occurs.

Each plug-in can specify options unique to itself. Options are specified by providing a plug-in signature, an option selector, and a 32 bit RefCon. The selector and RefCon are defined completely by the plugin. The RefCon could either contain actual data, such as a Boolean to turn an option on or off, or it could be a pointer to a plug-in specific structure.

When TECSetOption is called on a converter object, the encoding converter loops through each stage of the converter asking the corresponding plugin to set options for that stage.

Any plugin that can't handle the specified option will return paramErr to the TEC. If no plugin handles the specified option, paramErr will be returned to the caller.

```
OSStatus TECSetOption(TECObjectRef encodingConverter, TECPluginSig
   pluginSig, TECOptionSelector selector, TECOptionRefCon options);

errors:
   paramErr
   noErr
```

TECGetOption returns the current settings of a particular option.

```
OSStatus TECGetOption(TECObjectRef encodingConverter, TECPluginSig
   pluginSig, TECOptionSelector selector, TECOptionRefCon *options);

errors:
   paramErr
   noErr
```

See Section 7.0 "Deliverables" for descriptions and code examples of the supported plug-in-specific conversion options.

5.0 Using the Converter

5.1 Converting Encodings

Below is a code sample showing how to use the TextEncodingConverter to convert text encoded in ISO 2022JP into MacJapanese.

Calling TECCreateConverter will return in JIStoSJISConverter a newly created converter object set up to convert ISO2022 to MacJapanese. In order to convert from encoding X to encoding Z, the Text Encoding Converter may need to convert to intermediate encodings (X to Y to .. to Z). In the current implementation there is no algorithm that converts directly from ISO2022 to MacJapanese, but there are algorithms that convert ISO2022 to EUC and EUC to MacJapanese. The converter object JIStoSJISConverter will automatically be wired to use these two routines to perform the conversion. If at a later date, an ISO2022 to MacJapanese algorithm is added, this example code will take advantage of it without modification.

TECConvertText takes a converter object, and input /output buffer information. On return, actualInputLength will be modified to indicate how much of the input text has been converted, the conversion could have stopped either for lack of space in the output buffer or because an error occurred. The number of bytes placed in the outputBuffer are returned in actualOutputLength.

Because the converter object retains context information, TECConvertText can be called

repeatedly on successive portions of the text buffer in the case of constrained memory. TECDisposeConverter destroys the converter object and releases any memory it had reserved.

```
OSStatus TestISO2022toMacJapanese(void)
{

   TECObjectRef      JIStoSJISConverter;
   ByteCount         actualOutputLength;
   ByteCount         outBuffLen = 5000;
   TextPtr           outputBuffer =(TextPtr)NewPtr(outBuffLen);
   TextPtr           inputBuffer;
   ByteCount         inputLength;
   OSStatus          status;

   char  *testJIS = "$BAa9b(J [$B$O$d$?$+(J] /Hayataka”;

   inputBuffer = (TextPtr)testJIS;
   inputLength = strlen(testJIS);

   // Make a converter object

   status =  TECCreateConverter(&JIStoSJISConverter,
         kTextEncodingISO_2022_JP, kTextEncodingMacJapanese);

   if (!status) {

         // Convert text

         status =  TECConvertText(JIStoSJISConverter,
                     inputBuffer, inputLength, &actualInputLength,
                     outputBuffer, outBuffLen, &actualOutputLength);

         // Dispose converter

         status =  TECDisposeConverter(JIStoSJISConverter);

   }

   DisposePtr((Ptr)outputBuffer);
   return status;
}
```

6.0 Writing Text Encoding Plug-Ins

The High Level Text Encoding Converter was designed to make writing plug-ins to add new functionality as quick and painless as possible. Their functions include implementing a mechanism of informing the encoding converter about their conversion and encoding analysis capabilities, setting up converters, tearing them down, performing conversions, handling caller options, and examining text encodings.

The plug-ins are implemented as code fragments. The main export symbol of which is a routine which returns the address of a record of type TECPluginDispatchTable containing a dispatch table version number, a signature for the plug-in, and hooks for the methods each plug-in needs to support.

6.1 Plug-in dispatch Table

```
TECPluginDispatchTable KoreanPluginDispatchTable = {
   kTECPluginDispatchTableCurrentVersion,
   kTECPluginDispatchTableCurrentVersion,
   kTECKoreanPluginSignature,

   &ConverterPluginNewEncodingConverter,
   &ConverterPluginSetOption,
   &ConverterPluginGetOption,
   &ConverterPluginClearContextInfo,
   &ConverterPluginConvertTextEncoding,
   &ConverterPluginPluginFlushConversion;
   &ConverterPluginDisposeEncodingConverter,

   &ConverterPluginNewEncodingSniffer,
   &ConverterPluginClearSnifferContextInfo,
   &ConverterPluginSniffTextEncoding,
   &ConverterPluginDisposeEncodingSniffer,

   /* The following functionality is handled by providing resources.
    * When these resources are present in a plugin, the TEC
    * will look through them instead of calling the method.
    * Unless you need to dynamically determine the information
    * provided by the following hooks, using resources is the
    * way to go
    */

   nil, // &ConverterPluginGetAvailableTextEncodings,
   nil, // &ConverterPluginGetAvailableTextEncodingPairs,
   nil, // &ConverterPluginGetDestinationTextEncodings,
   nil, // &ConverterPluginGPluginGetSubTextEncodings,
   nil, // &ConverterPluginPluginGetSniffers;
   nil, // &ConverterPluginPluginGetTextEncodingMIMEName,
   nil, // &ConverterPluginPluginGetTextEncodingFromMIMEName,
   nil, // &ConverterPluginPluginGetTextEncodingInterfaceName,
   nil, // &ConverterPluginPluginGetWebTextEncodings;
```

```
    nil, // &ConverterPluginPluginGetMailTextEncodings;


};
```

6.2 Required plug-in methods

6.2.1 Creation, Destruction, and Conversion

A plug-in's TECPlugInNewEncodingConverter routine is called to allow the plug-in to set up its TECPlugInContextRec structure. The TECPlugInContextRec structure needs to contain all the information the plug-in requires to perform conversions between the encodings specified in inputEncoding and outputEncoding.

```
OSStatus TECPlugInNewEncodingConverter(
   TECObjectRef *newEncodingConverter,
   TECPlugInContextRec *plugContext,
   TextEncoding inputEncoding,
   TextEncoding outputEncoding)
```

6.3 Plug-in data structures

The Text Encoding Converter communicates with it's plugins through the PluginContextRec. This record contains information about the state of a conversion stage.

The public section

Most of the public section of the PluginContextRec record will be maintained by the Text Encoding Converter and modified by the plug-in. The bufferContext will be set up to point to the input and output buffers before TECPlugInConvertTextEncoding is called. On exit from that routine, the plug-in will update this structure to indicate how much of the input buffer was consumed, and how much text was placed in the output buffer.

The private section is intended for the plug-in's private use and will not be modified by the Text Encoding Converter. The plug-in uses this area to store whatever state information it needs to keep track of the type of conversion it is performing and to save any context sensitive state information required during a multi-pass encoding conversion. If a plug-in requires more space than is provided in this record to keep its local data, it can maintain a pointer to its data in the contextRefCon field.

All current Apple plugins use these fields in the following way: The conversionProc field points to a routine within the plug-in that performs a specific conversion, from EUC to ISO 2022 for example. The clearContextProc can point either to a generic routine that clears all state information in the PluginPrivateRec, or could point to routines custom tailored to clear the conversion context for each specific conversion routine. The contextRefCon can be

used to store a handle to additional information handled by the plugin. The pluginData structure is available for storing converter state information.

```
struct TECConverterContextRec {

                                                                    /*
   public - manipulated externally*/
   Ptr                          pluginRec;
   TextEncoding                 sourceEncoding;
   TextEncoding                 destEncoding;
   UInt32                       pluginOptions;
   UInt32                       converterOptions;
   TECBufferContextRec          bufferContext;

                                                                    /*
   private - manipulated only within Plugin*/
   UInt32                       contextRefCon;
   ProcPtr                      conversionProc;
   ProcPtr                      clearContextInfoProc;

   UInt32                       options1;
   UInt32                       options2;
   UInt32                       options3;
   UInt32                       options4;
                                                                    /*
   state information */
   TECPluginStateRec            pluginState;

};
typedef struct TECConverterContextRec TECConverterContextRec;
```

TECPlugInDisposeEncodingConverter will be called for each plug-in referenced in a converter object when it is disposed. The plug-in is responsible for disposing of any memory or other resources such as conversion tables it may have created or loaded from disk in TECPlugInNewEncodingConverter.

```
OSStatus TECPlugInDisposeEncodingConverter(
   TECObjectRef newEncodingConverter,
   TECPlugInContextRec *plugContext);
```

TECPlugInConvertTextEncoding will be called to perform the actual conversion. The BufferContextRec will point to the start and end of the input and output buffers. The plug-in will convert the text in the input buffer to the desired encoding and place it in the output buffer, deciding how much of the input text it will consume to fit in the output buffer. Upon exit, the plug-in needs to update the inputBuffer and outputBuffer pointers to reflect how much of the text was converted and how large the output was. The plug-in

should save all necessary state information so that it can continue the conversion where it left off in the event that the entire converted input text could not fit in the output buffer.

```
OSStatus TECPlugInConvertTextEncoding(
   TECObjectRef encodingConverter,
   TECPlugInContextRec *plugContex);
```

6.2.2 Providing Information about Encodings and Plug-in Services

The following routines provide the mechanism that the Text Encoding Converter uses to find out what services are available to it in each of its plug-ins. These services include which encodings the plug-in knows about and which conversions it can perform on those encodings.

Each of these mechanisms consist of a count routine which returns the number of elements that will be returned, and a routine that returns the actual elements in a preallocated array. The array should be large enough to accommodate the number of elements returned by the count function.

TECPlugInCountAvailableTextEncodings and TECPlugInGetAvailableTextEncodings are used to fill in an array of type TextEncoding with the encodings supported by the plug-in.

```
OSStatus TECPlugInCountAvailableTextEncodings(
   ItemCount *numberOfMappings)

OSStatus TECPlugInGetAvailableTextEncodings(
   TextEncoding *availableEncodings,
   ItemCount maxAvailableEncodings,
   ItemCount *actualAvailableEncodings)
```

TECPlugInCountAvailableTextEncodingPairs and TECPlugInGetAvailableTextEncodingPairs are used to fill in an array of type TextEncodingConversionPair with the types of conversions a plug-in supports. The data type TextEncodingConversionPair is explained in section 3.1.1 above.

```
OSStatus TECPlugInCountAvailableTextEncodingPairs(
   ItemCount *numberOfMappings)

OSStatus TECPlugInGetAvailableTextEncodingPairs(
   TextEncodingConversionPair *availableEncodings,
   ItemCount maxAvailableEncodings,
   ItemCount *actualAvailableEncodings)
```

TECPlugInCountDestinationTextEncodings and TECPlugInGetDestinationTextEncodings are used to fill in an array of type TextEncoding with all of the text encodings that the

parameter inputEncoding can be converted to in one step. This routine is used by the Text Encoding Converter to find and evaluate paths from one encoding to another.

```
OSStatus TECPlugInCountDestinationTextEncodings(
   TextEncoding inputEncoding, ItemCount *numberOfMappings)

OSStatus TECPlugInGetDestinationTextEncodings(
   TextEncoding inputEncoding,
   TextEncoding *destinationEncodings, ItemCount
   maxDestinationEncodings,
   ItemCount *actualDestinationEncodings)
```

Since encodings supported on a user's system may need to be displayed in menus or option dialogs, and these encoding names may not be known to an application at compile time, each plug-in provides a mechanism which returns the names of the encodings it knows about. This routine takes a text encoding a desired language, a desired encoding for the text name data, and returns a string with the name of the encoding in the best language and encoding available.

```
OSStatus TECPluginGetTextEncodingLocalizedName(TextEncoding
   inputEncoding, LocaleIdentifier locale, TextEncoding
   preferedEncoding, ByteCount bufLen, TextEncoding *nameEncoding,
   ByteCount *nameLength, Byte encodingName[])
```

TECPluginGetTextEncodingInternetName is used to find the name of a text encoding as it would appear in a MIME header and TECPluginGetTextEncodingFromInternetName performs the inverse.

```
OSStatus TECPluginGetTextEncodingInternetName(TextEncoding
   textEncoding, Str255 encodingName)


OSStatus TECPluginGetTextEncodingFromInternetName(TextEncoding
   *textEncoding, ConstStr255Param encodingName)
```

TECPluginCountSubTextEncodings and TECPluginGetSubTextEncodings are used to find out which encodings are packaged within a text encoding. For example EUC_JP and ISO2022_JP both contain JIS 208, JIS 212, JIS Roman, and Half Width Katakana.

```
OSStatus TECPluginCountSubTextEncodings(TextEncoding inputEncoding,
   ItemCount *numberOfEncodings)


OSStatus TECPluginGetSubTextEncodings(TextEncoding inputEncoding,
   TextEncoding subEncodings[], ItemCount maxSubEncodings, ItemCount
   *actualSubEncodings)
```

6.2.2 Plug-in Resources

To facilitate plugin development, avoid duplicate code, and eventually avoid unnecessarily loading a plug-in, certain data access plug-in methods can be implemented as resources. If these resources are present, the corresponding routines will never be called. If this information is not available until run time, such as is the case with the Unicode plugin which needs to find out which conversion tables are available, then the plugin will be loaded and the corresponding function will be called instead.

#define	**Value**
kTECAvailableEncodingsResType	'cvcd'
Replaces Routines	
PluginCountAvailableTextEncodings, PluginCountAvailableTextEncodings	

#define	**Value**
kTECSubEncodingsResType	'cvsb'
Replaces Routines	
PluginCountSubTextEncodings, PluginGetSubTextEncodings	

#define	**Value**
kTECConversionPairsResType	'cvpr'
Replaces Routines	
PluginCountAvailableTextEncodingPairs, PluginGetAvailableTextEncodingPairs	

#define	**Value**
kTECInternetNamesResType	'cvnm'
Replaces Routines	
PluginGetTextEncodingInternetName, PluginGetTextEncodingFromInternetName	

#define	**Value**
kTECLocalizedNamesResType	'cvni'
Replaces Routine	

PluginGetTextEncodingLocalizedName

MORE RESOURCES

7.0 Deliverables

The Text Encoding Converter will consist of a shared library the incorporates the code fragments for the TEC, Unicode Converter, Text Common, and the Unicode plug-in. The rest of the plug-in code fragments will be installed along with the Unicode converter's tables in the Text Encodings folder in the system folder.

7.1 Unicode Plug-In

Though the Unicode converter code fragment is always present, the Unicode plug-in provides the interface to make the Unicode converter look like any other plug-in to the High Level Text Encoding Converter.

7.1.1 Supported Unicode Encodings

When its code fragment is loaded and prepared for execution the Unicode plug-in polls the Unicode converter to find out which conversion tables are available. Once initialized, the Unicode plug-in provides conversion support for all the encoding bases, variants, and formats provided by the Unicode converter plus support for UTF7 and UTF8.

7.1.1 Supported Unicode Conversions Options

The current version allows the caller to take full advantage of the conversion option flags available in the low level Unicode converter APIs.

From Unicode – kUnicodeUseFallbacksBit

To Unicode – kUnicodeUseFallbacksBit, kUnicodeDirectionalityMask, kUnicodeVerticalFormBit, kUnicodeLooseMappingsBit, kUnicodeStringUnterminatedBit

Unicode to Multiple Runs – kUnicodeUseFallbacksBit., kUnicodeDirectionalityMask, kUnicodeVerticalFormBit, kUnicodeLooseMappingsBit, kUnicodeStringUnterminatedBit, kUnicodeKeepSameEncodingBit

For more information on these conversion options, see the Unicode converter ERS.

Example:

```
status =  TECSetOption(encodingConverter, 'unic',
```

```
    kTECUnicodeLowLevelFlags, kUnicodeUseFallbacksBit |
    kUnicodeVerticalFormBit );
```

An optional feature of UTF7 is to encode the characters !"#$%&*;<=>@[]^_`{| and } either directly or by shifting into base 64 mode.

```
status =  TECSetOption(encodingConverter, 'unic',
   kTECUnicodeHighLevelFlags, kUnicodeUTF7MustShiftOpt);
```

7.2 Japanese Plug-In

7.2.1 Supported Japanese Encodings

Encodings:	kTextEncodingJIS_X0208_90 kTextEncodingShiftJIS, kTextEncodingISO_2022_JP, kTextEncodingEUC_JP, kTextEncodingMacJapanese,
Conversions:	round trip EUC to ShiftJIS round trip 2022 to EUC ShiftJIS to JIS 208

Ideally, routines t o convert directly from Unicode 1.1 to EUC_JP and back should also be provided since the characters of JIS X 0212-1990 are supported in EUC_JP and Unicode but not in MacJapanese.

7.2.2 Supported Japanese Conversion Options

Selector:	kTECJapaneseOptionFlags
Options flags:	kJapaneseConvertSingleByteKatakana - convert single byte katakana to full width katakana kJapaneseRejectSingleByteKatakana - return an error if single byte katakana is present in input kJapaneseRejectNonJISChars - return an error if nonstandard KanjiTalk extension characters are present in input

Example:

```
status =  TECSetOption(encodingConverter, 'tktn',
   kTECJapaneseOptionFlags, kJapaneseConvertSingleByteKatakana |
   kTokorotenRejectNonJISChars);
```

7.3 Chinese Plug-In

7.3.1 Supported Chinese Encodings

Encodings:	kTextEncodingISO_2022_CN ISO_2022Format, kTextEncodingISO_2022_CN HZ_GB_2312Format, kTextEncodingEUC_CN,
Conversions:	round trip EUC to GB2312 round trip 2022 to HZ round trip 2022 to EUC

7.4.2 Supported Korean Conversion Options

Selector:	not currently defined
Options flags:	