ERS for Unicode Converter & TextCommon

Version 23 for TEC 1.1 & 1.2 (Tempo): Feb. 10, 1997; external version—parts deleted

Authors: Peter Edberg & Julio González

1 Introduction

This document is the ERS for the Unicode Converter project. It should contain everything about the project that someone from outside the project team needs to know.

1.1 Unicode

Unicode is a new standard character set encoding that tries to encode all the characters in use in the world today. Each character is a 16-bit quantity and multiple characters can combine to form a single text element for a particular operation such as sorting or printing.

Many developers have chosen Unicode as the basis for their internationalization strategy. For example, a subset of Unicode is the primary encoding for NT, and there is some support for Unicode in Windows 95. The NeXT text system uses Unicode as the basic text encoding, so Rhapsody will also be Unicode-based.

The global nature of the Internet has also pushed developers to support more international features and has made them aware of the complexities. The myriad of encodings that need to be supported has given developers a compelling reason to push Unicode as a standard. In Java for example, strings are Unicode-only.

Unicode should make it easier to develop internationalized applications, ease data interchange with other platforms, and provide a better foundation for modern, multilingual systems. Also because Unicode contains a vast assortment of technical, typographic, and other symbols, Unicode offers advantages even to English-only users.

Appendix A specifies the format and conventions for Unicode plain text documents and strings in the Mac OS.

1.X [Others deleted]

2 Stage 1 Deliverables

There are three planned ways of distributing the Unicode Converter and related items: as part of the Tempo system release, as an SDK, and bundled with Cyberdog and MrJ releases.

2.1 The Unicode Converter

The Unicode Converter provides the functions necessary for applications and system software to convert text between Unicode and other encodings. It does not provide other Unicode-related text processing services; these will be handled by other libraries.

For the SDK, we will provide a FAT version of the converter as a shared library. The library is named "TextEncodingConverter" and it will also include the high-level Text Encoding Converter, which has its own ERS, and some TextCommon functionality. This library will work on versions of system 7 beginning with version 7.5. For 68K, the converter will come in two flavors: as a CFM-68K shared library in the Text Encoding

Converter file (which requires 68020 or greater) or as a static library (non A5 dependent) which developers may link directly to their applications.

2.2 Unicode Conversion Tables

The conversion tables provide the information about a particular mapping that the Unicode Converter uses. The tables exist as resources in files of type 'utbl' (or 'ecpg', if they also contain plugin code for the high-level text Encoding Converter) which are kept in the "Text Encodings:" folder. The path to this is "{SystemFolder}Text Encodings:". The folder names may be localized. Each file may contain tables for one or more base encodings.

Appendix B lists the encodings that the design must support, when the table will be available, and the mnemonic symbols for that TextEncodingBase.

2.3 Installation Script

The SDK installer script will install the TextEncodingConverter library and the TextEnxoding tables. If the user selects Easy Install, then they will obtain the conversion tables needed for the Macintosh scripts installed on their system. If they select Custom Install, then they may choose which additional conversion tables they want.

2.4 Mapping Table Compiler

Building the mapping tables will be a major part of this project. To ease table construction for ourselves, localizers, and third party application developers we will provide a mapping table compiler.

2.5 Documentation and Interface Files

We will work with Tech Pubs to develop documentation suitable for Inside Macintosh that describes how to use the Unicode functions and how to add support for additional encodings.

We will distribute Unicode.[hp] and TextCommon.[hp] with the SDK release.

3 Requirements

The requirements are ordered by priority.

3.1 Fidelity

3.1.1 Round-trips

A round-trip conversion occurs when an application uses the Unicode Converter facilities to convert a string to Unicode and then converts that string back to the original encoding. If the original and final strings are identical, then we say the round-trip conversion is perfect.

Perfect round-trip conversion is not always possible. For example if the original string contains a ligature that is not represented in Unicode, then that information may be lost during the round-trip unless we can make use of Unicode private-use characters. Some characters, for example the Apple logo, have no counterpart in Unicode. Instead, we will map these characters to the Unicode Corporate Use Zone. Although mappings to the Corporate Use Zone are not portable to other systems, it allows us to guarantee perfect round-trips for several encodings.

3.1.2 Supported Encodings

The Unicode Converter must allow perfect round-trip conversion, with use of the Corporate Use Zone, for at least the encodings listed in Appendix B.3.

The Converter must also be able to handle commercially important encodings for which perfect round-trip conversion is not always possible. In particular, it must be able to handle all Mac OS encodings.

The Converter need not handle any rich text or compound encoding mechanism such as ISO 2022 which embeds multiple encodings within a single stream. These encodings are handled by the Text Encoding Converter.

3.1.3 Variants

The Unicode Converter must work well with all popular variants, such as exist for certain fonts, of the basic Mac OS encodings. Also, the Converter design must work well with different versions of ISO 10646 and Unicode, including vendor subsets and revisions such as UTF-16 (also known as UCS-2E).

3.1.4 Interchange

When converting from existing Mac OS character sets to Unicode, the Unicode Converter must make the resulting text as interchangeable as possible with other Unicode systems. This means minimizing the use of corporate characters except when absolutely necessary for round-trip fidelity.

3.X [Others deleted]

4 [Deleted]

5 Scenarios

This section describes some typical uses of the Unicode Converter.

5.1 Importing Plain Text

A Macintosh user in the US of a desktop publishing application wishes to import a plain text document from a Windows NT™ system into their document. The user first copies the document to the Macintosh and then imports the document using the application’s import facilities.

The import facility first calls CreateUnicodeToTextInfo to load the Unicode-to-MacRoman conversion table and then loads the entire plain text into memory. Finally it calls ConvertFromUnicodeToText, with the kUnicodeUseFallbacks bit set, to convert the text to MacRoman. Any non-Roman characters in the NT plain text document are mapped to fallback characters.

5.2 Using Unicode as a Converter Hub

A Japanese company wants to build a Macintosh application for accessing text files on an existing DEC VMS® file server system. The Macintosh uses MacJapanese and the VMS system uses DECKanji.

The application initially calls CreateTextToUnicodeInfo and CreateUnicodeToTextInfo to load the DECKanji and MacJapanese conversion tables. When the user copies a file from the VMS system to the Macintosh, the application first calls ConvertFromTextToUnicode with the DECKanji-to-Unicode table resource and then ConvertFromUnicodeToText with the Unicode-to-MacJapanese table resource. Copying a file from the Mac to the VMS system is the mirror-image. Thus the application uses Unicode as a conversion hub to convert the file from DECKanji to MacJapanese.

5.3 Styled Text

A US application developer is porting a Unicode-based desk top publishing system to the Macintosh. The application must preserve the style run information in the Unicode documents when calling the Macintosh services. To accomplish this, the developer copies the style run offsets into offsetLen and offsetArray before calling ConvertFromUnicodeToText. The converter maps each offset to the corresponding offset in the MacRoman string.

5.4 Packet Conversion

A bulletin board provider in Egypt wishes to add a Macintosh client for accessing their service, which runs on machines using the ISO 8859-6 encoding. The Macintosh client must convert the text packets on-the-fly as they arrive from the BBS. The packets are not block delimited, that is they usually do not begin on paragraph boundaries.

This is similar to the scenario in section 5.2 except that the encodings are different and, what is more important, the converter must resolve the directionality of the characters without the full block-delimited context. To accomplish this, the application calls TruncateForUnicodeToText with the kUnicodeRestartSafeBit set for each packet. The application prefixes any truncated portion of the packet to the next packet. Once it has truncated the string, the application calls ConvertFromUnicodeToText and ConvertFromTextToUnicode as in the previous scenario, except that it sets kUnicodeRestartContextBit for each packet after the first. Also, because the global line direction for this client is right-to-left, the application sets the kUnicodeRightToLeft control flag.

5.5 Toolbox Rewrite

An AppleSoft engineer is rewriting the File Manager to use Unicode internally. The new File Manager must call the converter before invoking a Macintosh service that does not yet support Unicode.

For some operations, such as maintaining the file hierarchy on the disk, it is essential that the results of the conversion never vary. Hence the File Manager calls ConvertFromTextToUnicode and ConvertFromUnicodeToText with the kUnicodeUseFallbacksBit clear. Also, if they call CreateTextToUnicodeInfo or CreateUnicodeToTextInfo, they supply a specific version number, never kUnicodeUseLatestMapping, which loads the latest version.

6 Interface file organization

6.1 Public interfaces

6.1.1 TextCommon.h

This header file defines the public data types, constants, and prototypes related to TexttEncodings themselves, including the functions to convert or create TextEncoding values.

6.1.2 Unicode.h

This header file defines most of the other public data types, constants, and prototypes related to the Unicode Converter.

6.1.3 Types.h

This includes a definition of the UniChar type, as well as other standard types used by UnicodeConverter and TextCommon APIs. For the SDK, these may be in TextCommon.h.

6.1.4 Errors.h

This incudes definitions for the TextCommon and Unicode Converter public error codes. For the SDK, these may be in TextCommon.h.

6.X [Others deleted]

7 Public types and constants

7.1 UniChar and related types

The type UniChar is a 16-bit Unicode character. All the Unicode Converter functions assume that the Unicode character has the normal byte order for an unsigned 16-bit integer on the current platform and that any initial byte-order prefix character has been removed. They further assume that each Unicode character is word aligned.

```
typedef UInt16 UniChar;
typedef UniChar * UniCharArrayPtr;
typedef const UniChar * ConstUniCharArrayPtr;
```

7.2 Error codes

Unicode Converter functions can return general error codes such as noErr (function completed successfully) and paramErr (one or more of the input parameters has an invalid value), as well as memory and resource errors. In addition, they can return a number of error and status codes which are specific to text handling and text conversion (some of these are also returned by the high-level Text Encoding Converter):

```
enum {
     // general text errors
     kTextUnsupportedEncodingErr           = -8738,
     kTextMalformedInputErr                = -8739,
     kTextUndefinedElementErr              = -8740,
     // text conversion errors
     kTECMissingTableErr                   = -8745,
     kTECTableChecksumErr                  = -8746,
     kTECTableFormatErr                    = -8747,
     kTECBufferBelowMinimumSizeErr         = -8750,
     kTECArrayFullErr                      = -8751,
    kTECPartialCharErr                       = -8753,
    kTECUnmappableElementErr                 = -8754,
    kTECIncompleteElementErr                 = -8755,
    kTECDirectionErr                         = -8756,
    // text conversion status codes
    kTECUsedFallbacksStatus                  = -8783,
    kTECOutputBufferFullStatus               = -8785,
};
```

Here are general descriptions of the error and status codes. In some cases the individual function descriptions provide more detail about the meaning of a particular error code for that function.

kTextUnsupportedEncodingErr

The encoding or mapping is not supported with the current set of tables; no corresponding 'thdr' or 'uhdr' resource is present.
(CreateTextToUnicodeInfo/CreateTextToUnicodeInfoByEncoding, CreateUnicodeToTextInfo/CreateUnicodeToTextInfoByEncoding, CreateUnicodeToTextRunInfo/CreateUnicodeToTextRunInfoByEncoding, ChangeTextToUnicodeInfo, ChangeUnicodeToTextInfo, GetTextEncodingBaseName)

kTECMissingTableErr

The encoding is partially supported but a specific table necessary for this function is not present.
(CreateTextToUnicodeInfo/CreateTextToUnicodeInfoByEncoding, CreateUnicodeToTextInfo/CreateUnicodeToTextInfoByEncoding, CreateUnicodeToTextRunInfo/CreateUnicodeToTextRunInfoByEncoding, ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun, ChangeTextToUnicodeInfo, ChangeUnicodeToTextInfo, GetTextEncodingBaseName)

kTECTableChecksumErr

A specific table necessary for this function had a checksum error.
(CreateTextToUnicodeInfo/CreateTextToUnicodeInfoByEncoding, CreateUnicodeToTextInfo/CreateUnicodeToTextInfoByEncoding, CreateUnicodeToTextRunInfo/CreateUnicodeToTextRunInfoByEncoding, ChangeTextToUnicodeInfo, ChangeUnicodeToTextInfo)

kTECTableFormatErr

The table format cannot be handled by this code.
(TruncateForTextToUnicode, ConvertFromTextToUnicode/ConvertFromPStringToUnicode, ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun, UpgradeScriptInfoToLocaleIdentifier, RevertLocaleIdentifierToScriptInfo, UpgradeScriptInfoToTextEncoding, RevertTextEncodingToScriptInfo)

kTextMalformedInputErr

The text input contained a sequence that is not legal in the specified encoding, such as a DBCS high byte followed by an invalid low byte (e.g. 0x8120 in Shift-JIS).
(ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTextUndefinedElementErr

The text input contained a code point which is undefined in the specified

encoding.
(ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTECBufferBelowMinimumSizeErr

The output text buffer is too small to allow processing of the first input text element.
(should be returned by ConvertFromTextToUnicode/ConvertFromPStringToUnicode, ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTECArrayFullErr

Tje supplied name buffer or TextRun, TextEncoding, or UnicodeMapping array is too small.
(ConvertFromUnicodeToTextRun, QueryUnicodeMappings, GetTextEncodingBaseName)

kTECPartialCharErr

The input text ends in the middle of a multibyte character, conversion stopped.
(ConvertFromTextToUnicode/ConvertFromPStringToUnicode)

kTECUnmappableElementErr

An input text element cannot be mapped to the specified output encoding(s) using the specified options. This error can only occur if kUnicodeUseFallbacksBit is not set.
(ConvertFromTextToUnicode/ConvertFromPStringToUnicode, ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTECIncompleteElementErr

The input text ends with a text element that may be incomplete, or contains a text element that too long for the internal buffers.
(ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTECDirectionErr

Error in directionality processing: direction stack overflow, etc.
(ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTECUsedFallbacksStatus

The function has completely converted the input string to the specified target using one or more fallbacks. This error can only occur if kUnicodeUseFallbacksBit is set.
(ConvertFromTextToUnicode/ConvertFromPStringToUnicode, ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

kTECOutputBufferFullStatus

Part of the input text has been converted, but the output buffer does not have room to contain the results of converting all of the input text. The caller may resume converting from the point at which it stopped.
(ConvertFromTextToUnicode/ConvertFromPStringToUnicode, ConvertFromUnicodeToText/ConvertFromUnicodeToPString, ConvertFromUnicodeToTextRun)

7.3 Unicode Conversion Contexts

These types are opaque to the application. They hold all the state information used for conversion of a single stream of text; they are also used by the truncation functions.

Applications cannot perform any operation on these data types except through the Unicode Converter functions which use it explicitly.

```
typedef struct OpaqueTextToUnicodeInfo * TextToUnicodeInfo;
typedef const TextToUnicodeInfo ConstTextToUnicodeInfo;
typedef struct OpaqueUnicodeToTextInfo * UnicodeToTextInfo;
typedef const UnicodeToTextInfo ConstUnicodeToTextInfo;
typedef struct OpaqueUnicodeToTextRunInfo * UnicodeToTextRunInfo;
```

A TextToUnicodeInfo is used by ConvertFromTextToUnicode, TruncateForTextToUnicode, and ConvertFromPStringToUnicode.

A UnicodeToTextInfo is used by ConvertFromUnicodeToText, TruncateForUnicodeToText, and ConvertFromUnicodeToPString.

A UnicodeToTextRunInfo is used by ConvertFromUnicodeToTextRun.

7.4 TextEncoding

The type TextEncoding is a 32-bit value containing four bit-fields. To the application programmer, the TextEncoding should be considered opaque; it is typedef’ed as a UInt32 and can be passed by value. Since TextEncodings are persistent objects, and may eventually exist on other platforms, however, the format must be known (it is described in section 10.1). Nevertheless, application programmers should only use the APIs provided for TextEncoding creation and field extraction.

The four bit fields contain the following:

- A packed version (16 bits) of the TextEncodingBase value.
- A packed version (6 bits) of the TextEncodingVariant value.
- A packed version (8 bits) of the TextEncodingFormat value.
- 2 bits to designate the packing version used for TextEncoding. For the packing described here, these bits should be set to 0.

The TextEncodingBase value is the primary specification of the source or target encoding. The values 0 through 32 correspond to Macintosh script codes. Some values for TextEncodingBase are actually meta-values, such as kTextEncodingUnicodeDefault, that in different situations may refer to different actual TextEncodingBase values.

The TextEncodingVariant value specifies the minor variant of the base encoding. For a given TextEncodingBase, the enumeration of variants always begins with 0. The value kTextEncodingDefaultVariant specifies the default variant of the base encoding.

Determining whether two encodings should be treated as different base encodings or different variants is a gray area. Generally, if two different encodings could both be used for body text in the same language on the same version of a particular localized platform, the encodings should be considered variants of the same base encoding; otherwise, they should be different base encodings. On the Mac, different base encodings can be distinguished by script code, language code, region code, or special font names that designate a completely different encoding (such as Symbol). Variants of the same base encoding typically coexist in the same system as font variants.

For example, the MacIcelandic and MacTurkish encodings would typically not coexist on the same Mac today. In any case, they can be distinguished as language or region variants, even though they both use script code smRoman. However, the Sai Mincho and Hon Mincho fonts do coexist on a Mac Japanese system and they cannot be distinguished by language or region, although they implement slightly different character sets (though not different enough for a user to think of them as something completely different, as with Symbol or Zapf Dingbats). So MacIcelandic and MacTurkish should be different base encodings, while Sai Mincho and Hon Mincho should be treated as implementing different variants of MacJapanese.

The TextEncodingFormat value designates a particular way of algorithmically transforming a particular encoding, say for transmission through communication channels that may only handle 7-bit values. Examples include UTF-7 and UTF-8 for Unicode. These transformations are not viewed as different encodings, merely as different formats for representing the same encoding. Also, as long as ISO 10646 uses only the BMP, then UCS-2 (16-bit) and UCS-4 (32-bit) can be viewed as different formats for the same encoding. The value kTextEncodingDefaultFormat specifies the default format of the base encoding.

```
typedef UInt32 TextEncodingBase;
typedef UInt32 TextEncodingVariant;
typedef UInt32 TextEncodingFormat;

// values for TextEncodingBase

enum {
    // Mac encodings
    kTextEncodingMacRoman = 0L,
    kTextEncodingMacJapanese = 1,
    kTextEncodingMacChineseTrad = 2,
    kTextEncodingMacKorean = 3,
    kTextEncodingMacArabic = 4,
    kTextEncodingMacHebrew = 5,
    kTextEncodingMacGreek = 6,
    kTextEncodingMacCyrillic = 7,
    kTextEncodingMacDevanagari = 9,
    kTextEncodingMacGurmukhi = 10,
    kTextEncodingMacGujarati = 11,
    kTextEncodingMacOriya = 12,
    kTextEncodingMacBengali = 13,
    kTextEncodingMacTamil = 14,
    kTextEncodingMacTelugu = 15,
    kTextEncodingMacKannada = 16,
    kTextEncodingMacMalayalam = 17,
    kTextEncodingMacSinhalese = 18,
    kTextEncodingMacBurmese = 19,
    kTextEncodingMacKhmer = 20,
    kTextEncodingMacThai = 21,
    kTextEncodingMacLaotian = 22,
    kTextEncodingMacGeorgian = 23,
    kTextEncodingMacArmenian = 24,
    kTextEncodingMacChineseSimp = 25,
    kTextEncodingMacTibetan = 26,
    kTextEncodingMacMongolian = 27,
    kTextEncodingMacEthiopic = 28,
    kTextEncodingMacCentralEurRoman = 29,
    kTextEncodingMacVietnamese = 30,
    kTextEncodingMacExtArabic = 31,
    kTextEncodingMacSymbol = 33,
    kTextEncodingMacDingbats = 34,
    kTextEncodingMacTurkish = 35,
    kTextEncodingMacCroatian = 36,
    kTextEncodingMacIcelandic = 37,
    kTextEncodingMacRomanian = 38,
    kTextEncodingMacUkrainian = 0x98,
    kTextEncodingMacHFS = 0xFF,                   // Meta-value, don't use in a table.
    // Unicode & ISO UCS encodings begin at 0x100
    kTextEncodingUnicodeDefault = 0x100,          // Meta-value, don't use in a table.
    kTextEncodingUnicodeV1_1 = 0x101,
    kTextEncodingISO10646_1993 = 0x102,
    kTextEncodingUnicodeV2_0 = 0x103,             // new location for Korean Hangul
    // ISO 8-bit and 7-bit encodings begin at 0x200
    kTextEncodingISOLatin1 = 0x201,               // ISO 8859-1
    kTextEncodingISOLatin2 = 0x202,               // ISO 8859-2
    kTextEncodingISOLatinCyrillic = 0x205,        // ISO 8859-5
    kTextEncodingISOLatinArabic = 0x206,          // ISO 8859-6, = ASMO 708, =DOS CP 708
    kTextEncodingISOLatinGreek = 0x207,           // ISO 8859-7
    kTextEncodingISOLatinHebrew = 0x208,          // ISO 8859-8
    kTextEncodingISOLatin5 = 0x209,               // ISO 8859-9
    // MS-DOS & Windows encodings begin at 0x400
    kTextEncodingDOSLatinUS = 0x400,              // code page 437
    kTextEncodingDOSGreek = 0x405,                // code page 737 (formerly page 437G)
    kTextEncodingDOSBalticRim = 0x406,            // code page 775
    kTextEncodingDOSLatin1 = 0x410,               // code page 850, "Multilingual"
    kTextEncodingDOSGreek1 = 0x411,               // code page 851
    kTextEncodingDOSLatin2 = 0x412,               // code page 852, Slavic
    kTextEncodingDOSCyrillic = 0x413,             // code page 855, IBM Cyrillic
    kTextEncodingDOSTurkish = 0x414,              // code page 857, IBM Turkish
    kTextEncodingDOSPortuguese = 0x415,           // code page 860
    kTextEncodingDOSIcelandic = 0x416,            // code page 861
    kTextEncodingDOSHebrew = 0x417,               // code page 862
    kTextEncodingDOSCanadianFrench = 0x418,       // code page 863
    kTextEncodingDOSArabic = 0x419,               // code page 864
    kTextEncodingDOSNordic = 0x41A,               // code page 865
    kTextEncodingDOSRussian = 0x41B,              // code page 866
    kTextEncodingDOSGreek2 = 0x41C,               // code page 869, IBM Modern Greek
    kTextEncodingDOSThai = 0x41D,                 // code page 874, also for Windows
    kTextEncodingDOSJapanese = 0x420,             // code page 932, also for Windows
    kTextEncodingDOSChineseSimplif = 0x421,       // code page 936, also for Windows
```

```
      kTextEncodingDOSKorean = 0x422,               // code page 949, also for Windows;
                                                    //          Unified Hangul Code
      kTextEncodingDOSChineseTrad = 0x423,          // code page 950, also for Windows
      kTextEncodingWindowsLatin1 = 0x500,           // code page 1252
      kTextEncodingWindowsANSI = 0x500,             // code page 1252 (alternate name)
      kTextEncodingWindowsLatin2 = 0x501,           // code page 1250, Central Europe
      kTextEncodingWindowsCyrillic = 0x502,         // code page 1251, Slavic Cyrillic
      kTextEncodingWindowsGreek = 0x503,            // code page 1253
      kTextEncodingWindowsLatin5 = 0x504,           // code page 1254, Turkish
      kTextEncodingWindowsHebrew = 0x505,           // code page 1255
      kTextEncodingWindowsArabic = 0x506,           // code page 1256
      kTextEncodingWindowsBalticRim = 0x507,        // code page 1257
      kTextEncodingWindowsKoreanJohab =0x510,       // code page 1361, for Windows NT
      // Various national standards begin at 0x600
      kTextEncodingUS_ASCII = 0x600,
      kTextEncodingJIS_X0201_76 = 0x620,
      kTextEncodingJIS_X0208_83 = 0x621,
      kTextEncodingJIS_X0208_90 = 0x622,
      kTextEncodingJIS_X0212_90 = 0x623,
      kTextEncodingJIS_C6226_78 = 0x624,
      kTextEncodingGB_2312_80 = 0x630,
      kTextEncodingGBK_95 = 0x631,                  // annex to GB 13000-93; for Windows 95
      kTextEncodingKSC_5601_87 = 0x640,             // KSC 5601-92 without Johab annex
      kTextEncodingKSC_5601_92_Johab = 0x641,       // KSC 5601-92 Johab annex
      kTextEncodingCNS_11643_92_P1 = 0x651,         // CNS 11643-1992 plane 1
      kTextEncodingCNS_11643_92_P2 = 0x652,         // CNS 11643-1992 plane 2
      kTextEncodingCNS_11643_92_P3 = 0x653,         // CNS 11643-1992 plane 3 (was plane 14
                                                    //    in 1986 version)
      // ISO 2022 collections begin at 0x800
      kTextEncodingISO_2022_JP = 0x820,
      kTextEncodingISO_2022_JP_2 = 0x821,
      kTextEncodingISO_2022_CN = 0x830,
      kTextEncodingISO_2022_CN_EXT = 0x831,
      kTextEncodingISO_2022_KR = 0x840,
      // EUC collections begin at 0x900
      kTextEncodingEUC_JP = 0x920,                  // ISO 646,1-byte katakana,JIS208,JIS212
      kTextEncodingEUC_CN = 0x930,                  // ISO 646, GB 2312-80
      kTextEncodingEUC_TW = 0x931,                  // ISO 646, CNS 11643-1992 Planes 1-16
      kTextEncodingEUC_KR = 0x940,                  // ISO 646, KS C 5601-1987
      // Other defacto standards begin at 0xA00
      kTextEncodingShiftJIS = 0xA01,                // plain Shift-JIS
      kTextEncodingKOI8_R = 0xA02,                  // Russian Internet standard
      kTextEncodingBig5 = 0xA03,                    // Big-5 (has variants)
      kTextEncodingMacRomanLatin1 = 0xA04,          // MacRoman permuted to align with 8859-1
      kTextEncodingHZ_GB_2312 = 0xA05               // HZ (RFC 1842, for Chinese mail & news)
};

/* values for TextEncodingVariant */
```

```
enum {
      /* Default TextEncodingVariant, for any TextEncodingBase */
      kTextEncodingDefaultVariant = 0 ,

      /* Variants of kTextEncodingMacJapanese */
      kJapaneseStandardVariant = 0,
      kJapaneseStdNoVerticalsVariant = 1,
      kJapaneseBasicVariant = 2,
      kJapanesePostScriptScrnVariant = 3,
      kJapanesePostScriptPrintVariant = 4,
      kJapaneseVertAtKuPlusTenVariant = 5,
      kJapaneseStdNoOneByteKanaVariant = 6,
      kJapaneseBasicNoOneByteKanaVariant = 7,

      /* Variants of kTextEncodingMacHebrew */
      kHebrewStandardVariant = 0,
      kHebrewFigureSpaceVariant = 1,

      /* Variants of kTextEncodingUnicodeV1_1 */
      kUnicodeNoSubset = 0,
      kUnicodeNoCompatibilityVariant = 1,
      kUnicodeMaxDecomposedVariant = 2,
      kUnicodeNoComposedVariant = 3,
      kUnicodeNoCorporateVariant = 4,

      // Variants of Big-5 encoding
      kBig5_BasicVariant = 0,
      kBig5_StandardVariant = 1,                // 0xC6A1-0xC7FC: kana, Cyrillic,
                                                //   enclosed numerics
      kBig5_ETenVariant = 2                     // adds kana, Cyrillic, radicals,
                                                //   etc with hi bytes C6-C8,F9
};

/* TextEncodingFormat type & values */

typedef UInt32 TextEncodingFormat;
enum {
      // Default TextEncodingFormat for Any TextEncodingBase
      kTextEncodingDefaultFormat = 0,

      // Formats for Unicode & ISO 10646
      kUnicode16BitFormat = 0,
      kUnicodeUTF7Format = 1,
      kUnicodeUTF8Format = 2,
      kUnicode32BitFormat = 3
};
```

7.5 UnicodeMapping

The type UnicodeMapping defines a combination of a particular base encoding variant and a Unicode encoding.

```
typedef SInt32 UnicodeMapVersion;

/* values for UnicodeMapVersion */

enum {
     kUnicodeUseLatestMapping = -1
};


typedef struct UnicodeMapping UnicodeMapping;
typedef UnicodeMapping * UnicodeMappingPtr;
typedef UnicodeMapping * UnicodeMappingArrayPtr;
typedef const UnicodeMapping * ConstUnicodeMappingPtr;
typedef const UnicodeMapping * ConstUnicodeMappingArrayPtr;
struct UnicodeMapping {
     TextEncoding unicodeEncoding;
     TextEncoding otherEncoding;
     UnicodeMapVersion mappingVersion;
};
```

If mappingVersion is `kUnicodeUseLatestMapping`, then it specifies the latest mapping. Otherwise it should specify an explicit version.

7.6 TextEncodingRun

The type TextEncodingRun specifies the starting character position of a run of text and the TextEncoding of that text. The ConvertFromUnicodeToTextRun function uses this data type to return the result of a multiple encoding conversion.

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

7.7 ScriptCodeRun

The type ScriptCodeRun specifies the starting character position of a run of text and the ScriptCode of that text. The ConvertFromUnicodeToScriptCodeRun function uses this data type to return the result of a multiple encoding conversion.

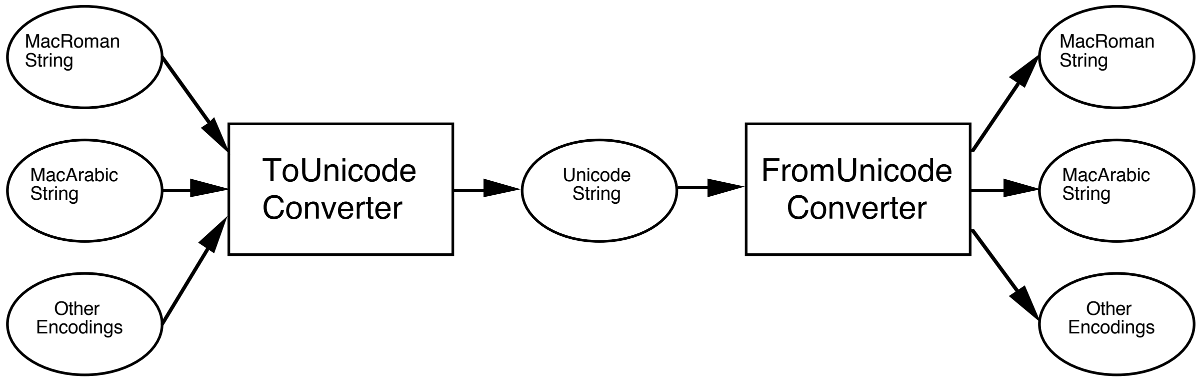
```
typedef struct ScriptCodeRun ScriptCodeRun ;
struct ScriptCodeRun {
     ByteOffset   offset;
     ScriptCode   scriptCode;
};
```

8 Application Programmer Interfaces—The Converter

The Unicode converter has two main parts. The ToUnicode Converter converts string in other encodings to Unicode while the FromUnicode Converter converts Unicode strings to other encodings.

As shown in Figure 2, an the Text Encoding Converter uses the two parts together to convert strings between any two encodings. Of course, conversion is only useful when there is a substantial overlap in the character repertoire of the two encodings, for example between MacRoman and ISO Latin/1.

Figure 2 Text Encoding Converter using the Unicode Converter as a Hub



A possible future use of the converter is to transliterate between non-Roman text and a Roman-based writing system, for example from Arabic to a Romanized form or possibly from Hanzi to Pinyin. However, we have not designed the converter to handle transliteration and such use may require enhancements.

An application must create and keep one instance of TextToUnicodeInfo for each separate text stream that is being converted to Unicode, and one instance of UnicodeToTextInfo for each text stream that is being converted from Unicode to something else. For example, if an application is using the Unicode Converter as a hub to convert a stream of DOS Arabic to Macintosh Arabic on the fly (see section 5.4) while simultaneously converting a separate stream of ISO 8859-6 to Macintosh Arabic on the fly, then it must keep two sets of TextToUnicodeInfo and UnicodeToTextInfo: one set for the DOS Arabic to Unicode to MacArabic conversion and another set for the ISO 8859-6 to Unicode to Macintosh Arabic conversion.

This section describes the public APIs. See the individual design documents for details on the implementation behind these interfaces.

8.0 InitializeUnicode

This API is used to Initialize the UnicodeConverter when using the static library version of the Converter. This API is not present in the CFM versions of the library. The UnicodeConverter is initialized automatically by CFM.

```
OSErr InitializeUnicode(void);
```

If the function returns noErr, it has successfully initialized the UnicodeConverter. Otherwise, there are no error condition besides the usual memory and O/S errors. You may call InitializeUnicode more than once. The function will check if the Converter has already been initialized and exit if so.

8.1 TextEncoding functions

The CreateTextEncoding API takes three parameters and returns a TextEncoding (Internally, it uses the SPI macro CreateTextEncodingPriv). The GetTextEncodingBase API extracts only the TextEncodingBase value from the TextEncoding. The GetTextEncodingVariant API extracts only the TextEncodingVariant value from the TextEncoding. The GetTextEncodingFormat API extracts only the TextEncodingFormat value from the TextEncoding. There are no error returns from any of these functions.

```
pascal TextEncoding CreateTextEncoding(
     TextEncodingBase base,
     TextEncodingVariant variant,
     TextEncodingFormat format);

pascal TextEncodingBase GetTextEncodingBase ( TextEncoding encoding );

pascal TextEncodingVariant GetTextEncodingVariant ( TextEncoding encoding );

pascal TextEncodingFormat GetTextEncodingFormat ( TextEncoding encoding );
```

The ResolveDefaultTextEncoding returns a TextEncoding value in which any meta-values in its encoding parameter have been resolved to real values. Please note that all APIs in the Unicode Converter perform this translation automatically. This API is useful for developers that take TextEncodings as parameters in their APIs. An example is the high-level Text Encoding converter.

```
pascal TextEncoding ResolveDefaultTextEncoding( TextEncoding encoding );
```

8.2 Creating Unicode Conversion Contexts

This section describes the functions which create each of the three types of Unicode conversion contexts (these contexts hold the state information used for conversion of a single stream of text). These functions also find and load the necessary table resources. They may move memory.

```
pascal OSStatus CreateTextToUnicodeInfo(
     ConstUnicodeMappingPtr iUnicodeMapping,     // Desired mapping
     TextToUnicodeInfo *oTextToUnicodeInfo);     // Returned TextToUnicodeInfo

pascal OSStatus CreateUnicodeToTextInfo(
     ConstUnicodeMappingPtr iUnicodeMapping,     // Desired mapping
     UnicodeToTextInfo *oUnicodeToTextInfo);     // Returned UnicodeToTextInfo


pascal OSStatus CreateUnicodeToTextRunInfo(
     ItemCount iNumberOfMappings,                // Number of desired mappings.
     const UnicodeMapping iUnicodeMappings[],    // Array of desired mappings
     UnicodeToTextRunInfo *oUnicodeToTextInfo);  // Returned context
 );

//The following APIs use kTextEncodingUnicodeDefault for the Unicode encoding

pascal OSStatus CreateTextToUnicodeInfoByEncoding(
     TextEncoding iEncoding,                    // Encoding for text source
     TextToUnicodeInfo *oTextToUnicodeInfo);    // Returned TextToUnicodeInfo

pascal OSStatus CreateUnicodeToTextInfoByEncoding(
     TextEncoding iEncoding,                    // Destination's text encoding
     UnicodeToTextInfo *oUnicodeToTextInfo);    // Returned UnicodeToTextInfo

pascal OSStatus CreateUnicodeToTextRunInfoByEncoding(
     ItemCount iNumberOfEncodings,              // Number of desired encodings.
     const TextEncoding iEncodings[],           // Array of desired encodings
     UnicodeToTextRunInfo *oUnicodeToTextInfo); // Returned context
 );

pascal OSStatus CreateUnicodeToTextRunInfoByScriptCode(
     ItemCount iNumberOfScriptCodes,            // Number of desired scripts.
     const ScriptCode iScripts[],               // Array of desired scripts.
     UnicodeToTextRunInfo *oUnicodeToTextInfo); // Returned context
 );
```

About instances of TextToUnicodeInfo:
- used by ConvertFromTextToUnicode, TruncateForTextToUnicode, and ConvertFromPStringToUnicode.
- to free one, you must call DisposeTextToUnicodeInfo.

About instances of UnicodeToTextInfo :
- used by ConvertFromUnicodeToText, TruncateForUnicodeToText, and ConvertFromUnicodeToPString.
- to free one, you must call DisposeUnicodeToTextInfo.

About instances of UnicodeToTextRunInfo :
- used by ConvertFromUnicodeToTextRun, ConvertUnicodeToScriptRun.
- to free one, you must call DisposeUnicodeToTextRunInfo.

In CreateUnicodeToTextRunInfo, the order of the mappings referenced by unicodeMapping determines the priority of the target encodings in the call to ConvertFromUnicodeToTextRun. In CreateUnicodeToTextRunInfoByEncoding, the order of encodings passed also determines the priority. Likewise, CreateUnicodeToTextRunInfoByScript, uses the order of scripts to determine the priority. One feature that CreateUnicodeToTextRunInfoByScriptCode provides is that if you pass NULL for the script array, it will use all the scripts installed in the system. The primary script will be the one with highest priority and ScriptOrder(itlm resource) will determine the priority of the rest.

If the function returns noErr, it has successfully created the conversion context. Otherwise, there are several error conditions besides the usual memory and resource errors (if an error occurs, then the returned context is invalid):
- kTextUnsupportedEncodingErr
- kTECMissingTableErr
- kTECTableChecksumErr

8.3 Disposing Unicode Conversion Contexts

These functions release a Unicode conversion context. The context (the parameter to the dispose function) becomes invalid.

```
pascal OSStatus DisposeTextToUnicodeInfo(
     TextToUnicodeInfo *ioTextToUnicodeInfo );

pascal OSStatus DisposeUnicodeToTextInfo(
     UnicodeToTextInfo *ioUnicodeToTextInfo );

pascal OSStatus DisposeUnicodeToTextRunInfo(
     UnicodeToTextRunInfo *ioUnicodeToTextInfo );
```

An application should only free a conversion context that it has obtained through a call to the corresponding context creation function. For example, DisposeTextToUnicodeInfo should only be called for instances of TextToUnicodeInfo created with CreateTextToUnicodeInfo. An application should never dispose of the same conversion context more than once.

If the function returns noErr, it has successfully released the conversion context. Otherwise, there is one error condition:

- paramErr (The conversion context is invalid: for example, NULL)

8.4 Control flags for conversion functions

The following control flags and corresponding masks are defined for the main conversion functions ConvertFromTextToUnicode, ConvertFromUnicodeToText, and ConvertFromUnicodeToTextRun (some control flags cannot be used with all of these functions).

```
enum {
     kUnicodeUseFallbacksBit = 0,
     kUnicodeKeepInfoBit = 1,
     kUnicodeDirectionalityBits = 2,
     kUnicodeVerticalFormBit = 4,
     kUnicodeLooseMappingsBit = 5,
     kUnicodeStringUnterminatedBit = 6,
     kUnicodeTextRunBit = 7,
     kUnicodeKeepSameEncodingBit = 8
};

enum {
     kUnicodeUseFallbacksMask = 1L << kUnicodeUseFallbacksBit,
     kUnicodeKeepInfoMask = 1L << kUnicodeKeepInfoBit,
     kUnicodeDirectionalityMask = 3L << kUnicodeDirectionalityBits,
     kUnicodeVerticalFormMask = 1L << kUnicodeVerticalFormBit,
     kUnicodeLooseMappingsMask = 1L << kUnicodeLooseMappingsBit,
     kUnicodeStringUnterminatedMask = 1L << kUnicodeStringUnterminatedBit,
     kUnicodeTextRunMask = 1L << kUnicodeTextRunBit,
     kUnicodeKeepSameEncodingMask = 1L << kUnicodeKeepSameEncodingBit
};
```

The remaining (currently undefined) bits are reserved and should be 0.

<u>kUnicodeUseFallbacksBit</u>

If clear, then the converter will return upon reading a source text element that has no equivalent in the target encoding. If the bit is set, then the converter will substitute one or more fallback characters and continue converting the string. When converting to Unicode, the fallback character is the Unicode replacement character (U+FFFD). When converting from Unicode, the fallback character(s) may be defined by the mapping table for the target encoding or by an application-defined fallback handler. (used with ConvertFromTextToUnicode, ConvertFromUnicodeToText, ConvertFromUnicodeToTextRun)

<u>kUnicodeKeepInfoBit</u>

If clear, then the converter will initialize the conversion context before converting the string. If the bit is set, then the converter will use the current state. This is useful if the application calling the converter must convert a stream in pieces that are not block delimited. (used with ConvertFromUnicodeToText, ConvertFromUnicodeToTextRun)

<u>kUnicodeDirectionalityBits</u>

Bits 2 and 3 are used for a two-bit enumerated value that affects how the converter resolves the direction of bi-directional characters.

```
enum {
      kUnicodeDefaultDirection,
      kUnicodeLeftToRight,
      kUnicodeRightToLeft
};
```

If the value is kUnicodeDefaultDirection, then the converter uses the value of the first strong direction character in the string. If the value is kUnicodeLeftToRight, then the converter assumes the base paragraph direction is left-to-right. If the value is kUnicodeRightToLeft, then the converter assumes the base paragraph direction is right-to-left. (used with ConvertFromUnicodeToText, ConvertFromUnicodeToTextRun)

<u>kUnicodeVerticalFormBit</u>

If clear, then the converter will map text elements which have both abstract and vertical presentation forms in the target encoding to the abstract form. If the bit is set, the converter will map such characters to their vertical forms. (used with ConvertFromUnicodeToText, ConvertFromUnicodeToTextRun)

<u>kUnicodeLooseMappingsBit</u>

If clear, then the converter will only use the strict equivalence portion of the mapping table. If the bit is set and the text element is not found in the strict equivalence portion of the table, then the converter will try the loose mapping section. (used with ConvertFromUnicodeToText, ConvertFromUnicodeToTextRun)

<u>kUnicodeStringUnterminatedBit</u>

This affects how the ConvertFromUnicodeToText scanner handles direction resolution and text element boundaries at the end of an input buffer. If the end of buffer should be treated as the end of text, then the bit should be clear. Otherwise, the converter assumes that the next call with the current context will supply another buffer of text that should be treated as a continuation of the current text.
When the ConvertFromUnicodeToText scanner reaches the end of the current input buffer and the current text element could extend beyond the buffer boundary (depending on what comes next): If kUnicodeStringUnterminatedBit = 0 the scanner will treat the end of the buffer as the end of the current text element, otherwise it will return unicodeElementErr.
When the ConvertFromUnicodeToText directionality analyzer reaches the end of the current input buffer and the direction of the current text element is still unresolved: If kUnicodeStringUnterminatedBit = 0, it will treat the end of the buffer as a block separator for direction resolution, otherwise it will set the direction as undetermined (both R and L).

<u>kUnicodeTextRunBit</u>

If clear, ConvertFromUnicodeToTextRun tries to convert Unicode text to the single encoding from the list of encodings in the UnicodeToTextRunInfo that produces the best result, i.e. that is able to convert the greatest amount of input text.
If set, ConvertFromUnicodeToTextRun may generate a target string that combines text in any of the encodings in the UnicodeToTextRunInfo.
(used with ConvertFromUnicodeToTextRun only).

<u>kUnicodeKeepSameEncodingBit</u>

Ignored unless kUnicodeTextRunBit is set. If kUnicodeKeepSameEncodingBit is clear, ConvertFromUnicodeToTextRun tries to keep most of the target string in one encoding, switching to other encodings only when necessary. If kUnicodeKeepSameEncodingBit is set, ConvertFromUnicodeToTextRun tries to minimize encoding changes in the target string—that is, once it is forced to make an encoding change, it tries to stay in the new encoding as long as possible. (used with ConvertFromUnicodeToTextRun only)

8.5 ConvertFromTextToUnicode

This function is the main function for converting strings from another encoding to Unicode.

```
pascal OSStatus ConvertFromTextToUnicode(
    TextToUnicodeInfo iTextToUnicodeInfo,  // ToUnicode context
    ByteCount iSourceLen,                  // Source string length in bytes
    ConstLogicalAddress iSourceStr,        // Source string pointer
    OptionBits iControlFlags,              // Conversion control flags
    ItemCount iOffsetCount,                // Number of input offsets
    ByteOffset iOffsetArray[],             // Input array of offsets
    ItemCount *oOffsetCount,               // Number of offset converted
    ByteOffset oOffsetArray[],             // Output array of offsets
    ByteCount iBufLen,                  // Output buffer length in bytes
    ByteCount *oSourceRead,             // Number of bytes converted
    ByteCount *oUnicodeLen,             // Output stream length in bytes
    UniCharArrayPtr oUnicodeStr );      // Output buffer pointer
```

The parameter textToUnicodeInfo is the TextToUnicodeInfo obtained from CreateTextToUnicodeInfo. The function modifies the contents of textToUnicodeInfo.

The input parameters sourceLen and sourceStr specify an array of bytes containing the source string.

The following bit flags are valid for the input parameter controlFlags:

- kUnicodeUseFallbacksBit

The parameters offsetCount and offsetArray specify an ordered list of significant byte offsets, for example at font or style changes, in the source string. The converter reads the application supplied offsets and writes the corresponding new offsets for the Unicode string. This allows style information to be mapped from the input text to the output text. All of the entries in offsetArray must be less than sourceLen.

The input parameter bufLen is the length in bytes reserved by the caller for the output string. The relationship between the size of the source string and the Unicode string is complex and depends on the source encoding and the contents of the string. For one byte encodings, for example MacRoman, the Unicode string will be at least double the size. For most two byte encodings, for example Shift-JIS, the Unicode string will be less than double the size. The worst case is Korean, where it is not unusual for the Unicode string to be 3 times larger. For international robustness the application should allocate a buffer at least 3 times larger than the source string.

If the function returns noErr, it has completely converted the source string to the specified Unicode variant without using fallbacks. Otherwise, there are several error conditions:

- paramErr
- kTECTableFormatErr
- kTECPartialCharErr
- kTECUnmappableElementErr
- kTECBufferBelowMinimumSizeErr
- kTECOutputBufferFullStatus
- kTECUsedFallbacksStatus

Except for paramErr, when an error occurs the function returns as much of the converted string as it can. The output parameter sourceRead contains the number of bytes converted before the error occurred.

8.6 ConvertFromUnicodeToText

This is the main function for converting strings from Unicode to another encoding.

```
pascal OSStatus ConvertFromUnicodeToText(
    UnicodeToTextInfo iUnicodeToTextInfo,  // FromUnicode context
    ByteCount iUnicodeLen,                 // Unicode string length in bytes
    ConstUniCharArrayPtr iUnicodeStr,      // Unicode input string
    OptionBits iControlFlags,              // Control flags
    ItemCount iOffsetCount,                // Number of input offsets
    ByteOffset iOffsetArray[],             // Input array of offsets
    ItemCount *oOffsetCount,               // Number of offset converted
    ByteOffset oOffsetArray[],             // Output array of offsets
    ByteCount iBufLen,                     // Output buffer length in bytes
    ByteCount *oInputRead,                 // Number of bytes converted
    ByteCount *oOutputLen,                 // Output string length in bytes
    LogicalAddress oOutputStr );           // Converted character array
```

The parameter unicodeToTextInfo is the UnicodeToTextInfo obtained from CreateUnicodeToTextInfo. The function modifies the contents of unicodeToTextInfo.

The following bit flags are valid for the input parameter controlFlags:
- kUnicodeUseFallbacksBit
- kUnicodeKeepInfoBit
- kUnicodeDirectionalityMask
- kUnicodeVerticalFormBit
- kUnicodeLooseMappingsBit
- kUnicodeStringUnterminatedBit

The input parameter bufLen is the length in bytes reserved by the caller for the output string. The relationship between the size of the Unicode string and the output string is complex and depends on the target encoding and the contents of the string. For many encodings, for example MacRoman and Shift-JIS, the output string will be between half and the same size as the Unicode string. For some non-Macintosh encodings, the output can be even larger.

The parameters offsetCount and offsetArray specify an ordered list of significant byte offsets, for example at font or style changes, in the Unicode string. The converter reads the application supplied offsets and writes the corresponding new offsets in the converted string. This allows style information to be mapped from the input text to the output text.

If the function returns noErr, it has completely converted the Unicode string to the target encoding without using fallbacks. Otherwise, there are several error conditions :
- paramErr
- kTECMissingTableErr
- kTECTableFormatErr
- kTextMalformedInputErr
- kTextUndefinedElementErr
- kTECIncompleteElementErr
- kTECDirectionErr
- kTECUnmappableElementErr
- kTECBufferBelowMinimumSizeErr
- kTECOutputBufferFullStatus
- kTECUsedFallbacksStatus
- error codes from caller-defined fallback handler

When an error occurs the function returns the output up to the character that caused the error. The output parameter inputRead contains the number of bytes converted before the error occurred.

8.7 ConvertFromUnicodeToTextRun & ConvertFromUnicodeToScriptCodeRun

ConvertFromUnicodeToTextRun is the main function for converting strings from Unicode to some other encodings. ConvertUnicodeToScriptRun is similar in functionality as ConvertFromUnicodeToTextRun but the output is returned in the form of a ScriptCodeRun as opposed to a TextEncodingRun.

```
pascal OSStatus ConvertFromUnicodeToTextRun(
    UnicodeToTextRunInfo iUnicodeToTextInfo,    // UnicodeToTextRun context
    ByteCount iUnicodeLen,                     // Unicode string length in bytes
    ConstUniCharArrayPtr iUnicodeStr,          // Unicode input string
    OptionBits iControlFlags,                  // Control flags
    ItemCount iOffsetCount,                    // Number of input offsets
    ByteOffset iOffsetArray[],                 // Input array of offsets
    ItemCount *oOffsetCount,                   // Number of offset converted
    ByteOffset oOffsetArray[],                 // Output array of offsets
    ByteCount iBufLen,                         // Output buffer length in bytes
    ByteCount *oInputRead,                     // Number of bytes converted
    ByteCount *oOutputLen,                     // Output string length in bytes
    LogicalAddress oOutputStr,                 // Converted character array
    ItemCount iEncodingRunBufLen,              // Output encodingRun buffer length
    ItemCount *oEncodingRunOutLen,             // Output encodingRun length
    TextEncodingRun oEncodingRuns[]            // Output encodingRun
 );

pascal OSStatus ConvertFromUnicodeToScriptCodeRun(
    UnicodeToTextRunInfo iUnicodeToTextInfo,    // UnicodeToTextRun context
    ByteCount iUnicodeLen,                     // Unicode string length in bytes
    ConstUniCharArrayPtr iUnicodeStr,          // Unicode input string
    OptionBits iControlFlags,                  // Control flags
    ItemCount iOffsetCount,                    // Number of input offsets
    ByteOffset iOffsetArray[],                 // Input array of offsets
    ItemCount *oOffsetCount,                   // Number of offset converted
    ByteOffset oOffsetArray[],                 // Output array of offsets
    ByteCount iBufLen,                         // Output buffer length in bytes
    ByteCount *oInputRead,                     // Number of bytes converted
    ByteCount *oOutputLen,                     // Output string length in bytes
    LogicalAddress oOutputStr,                 // Converted character array
    ItemCount iScriptRunBufLen,                // Output scriptCodeRunbuffer length
    ItemCount *oScriptRunOutLen,               // Output scriptCodeRunlength
    ScriptCodeRun oScriptCodeRuns[]            // Output scriptCodeRun
 );
```

The parameter unicodeToTextInfo is the UnicodeToTextRunInfo obtained from CreateUnicodeToTextRunInfo. The function modifies the contents of unicodeToTextInfo.

The following bit flags are valid for the input parameter controlFlags:

- kUnicodeUseFallbacksBit.
- kUnicodeKeepInfoBit
- kUnicodeDirectionalityMask
- kUnicodeVerticalFormBit
- kUnicodeLooseMappingsBit
- kUnicodeStringUnterminatedBit
- kUnicodeTextRunBit
- kUnicodeKeepSameEncodingBit

If kUnicodeTextRunBit is clear, then the converter tries to convert the Unicode text to the best single encoding from the list of encodings in the UnicodeToTextRunInfo—i.e., the encoding that results in conversion of the greatest amount of input text. If the complete text can be converted into more than one of these encodings, then the converter chooses among them based on the order of encodings in the unicodeMapping parameter for CreateUnicodeToTextRunInfo. If the converter can convert the complete Unicode text into some encoding, it returns with noErr. If kUnicodeTextRunBit is clear, the encodingRuns parameter will always point to a value equal to 1 since ConvertFromUnicodeToTextRun only converts to a single encoding run.

If kUnicodeTextRunBit is set and the converter cannot convert the complete Unicode text into the first encoding, the converter then tries to convert the first text element that failed to the remaining encodings (in order). What it does after this text element depends on the setting of kUnicodeKeepSameEncodingBit:

- If kUnicodeKeepSameEncodingBit is clear, the converter returns to the original encoding and attempts to continue conversion with that encoding (this is equivalent to converting each text element to the first encoding that works, in the order specified). If kUnicodeUseFallbacksBit is set and a text element cannot be converted using any of the encodings in the list, the converter uses the fallbacks for the first encoding in the list.
- If kUnicodeKeepSameEncodingBit is set, the converter continues with the new target encoding until it encounters a text element that cannot be converted using the new encoding. If kUnicodeUseFallbacksBit is set and the text element cannot be converted using any of the encodings in the list, the converter uses the fallbacks for the current encoding. *(Actually it searches for valid fallback mappings in the same way, by going through all of the encodings in the list until it finds one that has fallback mappings. If it finds one, then that encoding becomes the new default encoding. If it doesn't find one, it uses the default fallback for the current encoding).*

Here is a summary of the use of these two control bits:-

- If the desired result is to keep the converted text in a single encoding run, keep kUnicodeTextRunBit clear.
- If the desired result is to keep as much as possible of the converted text in one encoding, set kUnicodeTextRunBit and keep kUnicodeKeepSameEncodingBit clear.

- If the desired result is to minimize the number of encoding runs, or to minimize the changes of target encoding, set both kUnicodeTextRunBit and kUnicodeKeepSameEncodingBit.

The parameters unicodeLen, unicodeStr, offsetCount, offsetArray, bufLen, inputRead, outputLen, and outputStr are described in the ConvertFromUnicodeToText section. Please refer to that section for detailed information of these parameters.

For ConvertFromUnicodeToTextRun, the input parameter encodingRunBufLen specifies size (number of entries) allocated for the encodingRuns array. The converter returns the number of valid encoding runs in the place pointed to by encodingRunOutLen. Each entry in the encodingRun specifies the starting offset in the converted text and the associated TextEncoding.

For ConvertFromUnicodeToScriptCodeRun, the input parameter scriptRunBufLen specifies size (number of entries) allocated for the ScriptCodeRuns array. The converter returns the number of valid encoding runs in the place pointed to by encodingRunOutLen. Each entry in the scriptCodeRun specifies the starting offset in the converted text and the associated ScriptCode.

If the function returns noErr, it has completely converted the Unicode string to the one (or more) of the specified target encodings. Otherwise, there are several error conditions:

- paramErr
- kTECMissingTableErr
- kTECTableFormatErr
- kTextMalformedInputErr
- kTextUndefinedElementErr
- kTECIncompleteElementErr
- kTECDirectionErr (In this case the output parameters are invalid)
- kTECUnmappableElementErr
- kTECBufferBelowMinimumSizeErr
- kTECOutputBufferFullStatus
- kTECUsedFallbacksStatus
- kTECArrayFullErr
- error codes from caller-defined fallback handler

When an error occurs the function returns the output up to the character that caused the error. The output parameter inputRead contains the number of bytes converted before the error occurred.

8.8 TruncateForTextToUnicode

This function truncates a multibyte string. This function is provided so that truncation will occur appropriately, and not in the middle of a two-byte character. This can be useful to do before calling ConvertFromTextToUnicode so that the string it is handed terminates with complete characters. If truncation is attempted by the programmer, it can be dangerous; it can destroy or corrupt information if done improperly, for example if truncation breaks a string between the first and second bytes of a two byte character.

Note that this only relies upon the type of character that is found in that "range"; i.e., the size of the character is only dependent upon the scanner table information not the "validity" or existence of the particular character within the table.

```
pascal OSStatus TruncateForTextToUnicode(
     ConstTextToUnicodeInfo iTextToUnicodeInfo,  // ToUnicode context
     ByteCount iSourceLen,                  // Source string length in bytes
     ConstLogicalAddress iSourceStr,        // Source string
     ByteCount iMaxLen,                     // Maximum allowable length
     ByteCount *oTruncatedLen );            // Truncated length in bytes
```

The parameter textToUnicodeInfo is the TextToUnicodeInfo obtained from CreateTextToUnicodeInfo. The function does not modify the contents of textToUnicodeInfo. Therefore it is safe to call this function between calls to ConvertFromTextToUnicode.

The input parameters sourceLen, and sourceStr specify the source string.

The output parameter truncatedLen is the length of the longest portion of sourceStr that is less than or equal to maxLen.

If the function returns noErr, it has successfully truncated the string. Otherwise, there is one error condition :

- paramErr (in this case the output parameter truncatedLen is invalid).
- kTECTableFormatErr

8.9 TruncateForUnicodeToText

This function truncates a Unicode string. Truncation is a dangerous operation that can destroy or corrupt information if done improperly, for example if truncation breaks a string between a spacing character and a non-spacing character. This function allows the caller to specify qualities of the truncated string.

```
pascal OSStatus TruncateForUnicodeToText(
     ConstUnicodeToTextInfo iUnicodeToTextInfo, // FromUnicode context
     ByteCount iSourceLen,                // Unicode string length in bytes
     ConstUniCharArrayPtr iSourceStr,     // Unicode string pointer
     OptionBits iControlFlags,            // Truncation control flags
     ByteCount iMaxLen,                   // Maximum allowable length in bytes
     ByteCount *oTruncatedLen );          // Truncated length in bytes
```

The parameter unicodeToTextInfo is the UnicodeToTextInfo obtained from CreateUnicodeToTextInfo. The function does not modify the contents of unicodeToTextInfo. Therefore it is safe to call this function between calls to ConvertFromUnicodeToText.

The input parameters unicodeLen, and unicodeStr specify the original Unicode string.

The input parameter controlFlags contains two bit flags that specify qualities of the truncated string.

```
enum {
     kUnicodeTextElementSafeBit,             // Text elements are complete
     kUnicodeRestartSafeBit                  // Safe for converter restart
};
enum {
     kUnicodeTextElementSafeMask = 1L << kUnicodeTextElementSafeBit,
     kUnicodeRestartSafeMask = 1L << kUnicodeRestartSafeBit
};
```

If kUnicodeTextElementSafeBit is set, then the truncated string will contain complete text elements. Applications should normally set this bit, even if other bits are set.

If kUnicodeRestartSafeMask is set, then the truncated string will be safe for processing by ConvertFromUnicodeToText even if the string is not block delimited. See section 5.4 for a typical use of this flag.

The output parameter truncatedLen is the length of the longest portion of sourceStr that satisfies the qualities specified by controlFlags and is less than or equal to maxLen.

If the function returns noErr, it has successfully truncated the string. Otherwise, there is one error condition:

- paramErr (in this case the output parameter truncatedLen is invalid).

8.10 ConvertFromPStringToUnicode

This function converts a Pascal string in a Macintosh script encoding to a Unicode string. It provides a convenient way to convert short Macintosh strings without the full generality and complexity of ConvertFromTextToUnicode.

```
pascal OSStatus ConvertFromPStringToUnicode(
     TextToUnicodeInfo iTextToUnicodeInfo,  // TextToUnicodeInfo instance
     ConstStr255Param iPascalStr,           // Pascal string to convert
     ByteCount iBufLen,                     // Length of output buffer
     ByteCount *oUnicodeLen,                // Unicode string length (bytes)
     UniCharArrayPtr oUnicodeStr );         // Unicode output string
```

The parameter textToUnicodeInfo is the instance of TextToUnicodeInfo obtained from CreateTextToUnicodeInfo. The function modifies the contents of textToUnicodeInfo.

The input parameter bufLen is the length of the output buffer reserved by the caller.

This function uses the fallbacks necessary to map the string.

The error conditions are identical to the errors for ConvertFromTextToUnicode.

8.11 ConvertFromUnicodeToPString

This function converts a Unicode string to a Pascal string in a Macintosh script encoding. It provides a convenient way to convert short Macintosh strings without the full generality and complexity of ConvertFromUnicodeToText.

```
pascal OSStatus ConvertFromUnicodeToPString(
     UnicodeToTextInfo iUnicodeToTextInfo,  // UnicodeToTextInfo instance
     ByteCount iUnicodeLen,                 // Unicode string length in bytes
```

```
    ConstUniCharArrayPtr iUnicodeStr,      // Unicode input string
    Str255 oPascalStr );                   // Converted PString
```

The parameter unicodeToTextInfo is the instance of UnicodeToTextInfo obtained from CreateUnicodeToTextInfo. The function modifies the contents of unicodeToTextInfo.

The input parameters unicodeLen and unicodeStr together specify the Unicode string.

This function uses the loose mappings and fallbacks necessary to map the string.

The error conditions are identical to the errors for ConvertFromUnicodeToText.

8.12 QueryUnicodeMappings and CountUnicodeMappings

QueryUnicodeMappings returns information about the available conversion mappings on a particular system. It returns a list of all available UnicodeMappings in the current system whose fields match the fields of findMapping which are specified by the bits in filter.

```
pascal OSStatus QueryUnicodeMappings(
    OptionBits iFilter                     // What to match of...
    ConstUnicodeMappingPtr iFindMapping    // ...this UnicodeMapping
    ItemCount iMaxCount,                   // Maximum number of mappings
    ItemCount *oActualCount,               // Actual number of mappings
    UnicodeMapping oReturnedMappings[] );  // Available mappings
```

```
/* filter bits for QueryUnicodeMappings filter parameter */
enum {
    kUnicodeMatchUnicodeBaseBit = 0,
    kUnicodeMatchUnicodeVariantBit = 1,
    kUnicodeMatchUnicodeFormatBit = 2,
    kUnicodeMatchOtherBaseBit = 3,
    kUnicodeMatchOtherVariantBit = 4,
    kUnicodeMatchOtherFormatBit = 5
};
```

```
enum {
    kUnicodeMatchUnicodeBaseMask = 1L << kUnicodeMatchUnicodeBaseBit,
    kUnicodeMatchUnicodeVariantMask = 1L << kUnicodeMatchUnicodeVariantBit,
    kUnicodeMatchUnicodeFormatMask = 1L << kUnicodeMatchUnicodeFormatBit,
    kUnicodeMatchOtherBaseMask = 1L << kUnicodeMatchOtherBaseBit,
    kUnicodeMatchOtherVariantMask = 1L << kUnicodeMatchOtherVariantBit,
    kUnicodeMatchOtherFormatMask = 1L << kUnicodeMatchOtherFormatBit
};
```

The filtering of UnicodeMappings can be done on any of the three TextEncoding subfields of unicodeEncoding and on any of the three TextEncoding subfields of otherEncoding. The filter parameter is a set of six bit flags that correspond to these six subfields. The list in returnedMappings will only contain mappings that match the designated fields in findMapping—i.e., the fields for which the corresponding bit in the filter parameter is 1. No filtering is performed on fields for which the corresponding bit in the filter parameter is 0 (these are don't care fields).

For example, to obtain a list of all UnicodeMappings between Unicode 1.1 (default variant & format) and the default variant & format for any other encoding—i.e. a list in which each non-Unicode base encoding shows up once—we would set up filter and findMapping as follows:

```
findMapping.unicodeMapping = CreateTextEncoding(
     kTextEncodingUnicodeV1_1,
     kTextEncodingDefaultVariant,
     kTextEncodingDefaultFormat);

findMapping.otherEncoding = CreateTextEncoding(
     kTextEncodingMacRoman,                /* anything, will be ignored */
     kTextEncodingDefaultVariant,
     kTextEncodingDefaultFormat);

filter = kUnicodeMatchUnicodeBaseMask |
     kUnicodeMatchUnicodeVariantMask | kUnicodeMatchUnicodeFormatMask |
     kUnicodeMatchOtherVariantMask | kUnicodeMatchOtherFormatMask;
```

The input parameter maxCount is the number of elements in the array allocated by the caller for the results. If there are more mappings than the array can hold, the function returns as many mappings as will fit and returns unicodeBufErr. The output parameter actualCount is the number of mappings found, which may be greater than maxCount .

The output parameter returnedMappings is an array that holds the appropriate values for each mapping found. The order of the entries is not guaranteed.

If the function returns noErr, then actualCount is less than or equal to maxCount and returnedMappings is filled in with appropriate values. Otherwise, the followinf error conditions are possible

- paramErr
- kTECArrayFullErr(There were too many tables to fit in returnedMappings)

CountUnicodeMappings counts available mappings using the same filtering procedure as QueryUnicodeMappings.

```
pascal OSStatus CountUnicodeMappings(
     OptionBits iFilter                    // What to match of...
     ConstUnicodeMappingPtr iFindMapping   // ...this UnicodeMapping
     ItemCount *oActualCount,              // Actual number of mappings
```

It can return noErr or paramErr.

8.13 Getting encoding names

GetTextEncodingName returns the name of the specified TextEncoding for a particular region/language and encoding.

```
typedef UInt32 TextEncodingNameSelector;
enum {
    kTextEncodingFullName = 0,
    kTextEncodingBaseName = 0,
    kTextEncodingVariantName = 0,
    kTextEncodingFormatName = 0,
};
```

```
pascal OSStatus GetTextEncodingName(
    TextEncoding iEncoding,                // Encoding whose name is desired…
    TextEncodingNameSelector iNamePartSelector,
    RegionCode iPreferredRegion,           // …for this region/laguage…
    TextEncoding iPreferredEncoding,       // …and this preferred encoding
    ByteCount iBufLen,                     // Buffer length (bytes)
    ByteCount *oNameLength,                // Name length (bytes)
    RegionCode oActualRegion,              // name region code
    TextEncoding oActualEncoding,          // name encoding
    BytePtr oMappingName );                // Name
```

Here, iEncoding is the encoding whose name is desired. The iNamePartSelector parameter specifies which part of the name is desired. The iPreferredRegion and iEncodingForName specify the preferred region/language and encoding to use for the name. The iPreferredRegion parameter accepts a System 7 region code, which implies a language as well. If the name cannot be returned using the preferred region/language, it will return the name with another region code for the same language, or in a default language (probably English). The oActualRegionparameter specifies which region/language was actually used. If the name cannot be returned in the preferred encoding, then it will be returned in another encoding, probably Unicode or US-ASCII; oActualEncoding specifies which encoding was actually used.

If iEncoding specifies something other than a default value for variant and format, the name will reflect the specified variant and/or format if possible.

If the function returns noErr, it has found and loaded the name of the mapping. Otherwise, there are three error conditions besides the usual resource errors:

- kTextUnsupportedEncodingErr
- kTECMissingTableErr
- kTECArrayFullErr

8.14 Changing Unicode Conversion Contexts

```
pascal OSStatus ChangeTextToUnicodeInfo(
    TextToUnicodeInfo ioTextToUnicodeInfo,      // ToUnicode context
    ConstUnicodeMappingPtr  iUnicodeMapping );  // Desired mapping
```

```
pascal OSStatus ChangeUnicodeToTextInfo(
    UnicodeToTextInfo ioUnicodeToTextInfo,      // FromUnicode context
    ConstUnicodeMappingPtr  iUnicodeMapping );  // Desired mapping
```

ChangeTextToUnicodeInfo changes the mapping associated with a particular instance of TextToUnicodeInfo.

ChangeUnicodeToTextInfo changes the mapping associated with a particular UnicodeToTextInfo. The main use for this function is when converting from Unicode and the Unicode string contains characters that require multiple target encodings.

These functions do not alter the current conversion state in any other way. In the future we will supply functions ResetUnicodeToTextInfo & ResetUnicodeToTextRunInfo for resetting the other aspects of the conversion state.

If these functions return noErr, they have successfully changed the mapping associated with the conversion context. Otherwise, there are several error conditions besides the usual memory and resource errors (if an error occurs, then the returned context is invalid):

- paramErr (the conversion context passed in was invalid)
- kTextUnsupportedEncodingErr
- kTECMissingTableErr
- kTECTableChecksumErr

8.15 Installing Fallback Handlers

SetFallbackUnicodeToText sets a caller-defined fallback handler for a specific UnicodeToTextInfo used with ConvertFromUnicodeToText and ConvertFromUnicodeToPString. It is intended for use in System 7 and by Blue clients in MacOS 8; it is not preemptive safe. Non-Blue clients in MacOS 8 should use SetFallbackUnicodeToTextPreemptive.

```
pascal OSStatus SetFallbackUnicodeToText(
     UnicodeToTextInfo iUnicodeToTextInfo,       //
     UnicodeToTextFallbackUPP iFallback,         // caller-defined handler
     OptionBits iControlFlags,                   //
     LogicalAddress iInfoPtr);                   // context pointer for handler

pascal OSStatus SetFallbackUnicodeToTextRun(
     UnicodeToTextRunInfo iUnicodeToTextRunInfo,
     UnicodeToTextFallbackUPP iFallback,
     OptionBits iControlFlags,
     LogicalAddress iInfoPtr);


pascal OSStatus SetFallbackUnicodeToTextPreemptive(
     UnicodeToTextInfo iUnicodeToTextInfo,
     UnicodeToTextFallbackPreemptiveProcPtr iFallback,
     OptionBits iControlFlags,
     LogicalAddress iInfoPtr);

pascal OSStatus SetFallbackUnicodeToTextRunPreemptive(
     UnicodeToTextRunInfo iUnicodeToTextRunInfo,
     UnicodeToTextFallbackPreemptiveProcPtr iFallback,
     OptionBits iControlFlags,
     LogicalAddress iInfoPtr);
```

The parameter unicodeToTextInfo is the UnicodeToTextInfo obtained from CreateUnicodeToTextInfo. The function modifies the contents of unicodeToTextInfo.

The input parameter controlFlags contains a single two-bit field:

```
enum {
    kUnicodeFallbackSequencingBits = 0
};
enum {
    kUnicodeFallbackSequencingMask = 3L << kUnicodeFallbackSequencingBits
};
```

The kUnicodeFallbackSequencingMask contains a two-bit enumerated value that affects the order of calling the converter default fallback handler and caller-defined fallback handler.

```
/* Fallback Handler Usage Values: */
enum {
    kUnicodeFallbackDefaultOnly = 0L,       // default one only
    kUnicodeFallbackCustomOnly,             // caller-defined one only
    kUnicodeFallbackDefaultFirst,           // default one then caller-defined
    kUnicodeFallbackCustomFirst             // caller-defined then default one
};
```

Here is how the converter actions depend on this value:

<u>kUnicodeFallbackDefaultOnly</u>

The converter will just call the converter default fallback handler. The SetFallbackUnicodeToText parameters fallback and contextPtr will be ignored since no caller-defined fallback handler will be called.

<u>kUnicodeFallbackCustomOnly</u>

The converter will only call the caller-defined fallback handler defined in the parameter fallback.

<u>kUnicodeFallbackDefaultFirst</u>

The converter will first try the default fallback handler, continuing if successful; otherwise (if the default fallback handler failed) it will try the caller-defined one.

<u>kUnicodeFallbackCustomFirst</u>

The converter will first try the caller-defined fallback handler, continuing if successful; otherwise (if the caller-defined fallback handler failed) it will try the default fallback handler.

The parameter contextPtr specifies a context pointer for the caller-defined fallback handler. When the converter invokes the caller-defined fallback handler, the parameter contextPtr will be passed as the last parameter to the handler.

The parameter fallback specifies the caller-defined fallback handler. For SetFallbackUnicodeToTextPreemptive this parameter is a pointer to the caller-defined routine, as indicated below.

```
// For use with SetFallbackUnicodeToTextPreemptive:

typedef pascal OSStatus (*UnicodeToTextFallbackPreemptiveProcPtr )(
     UniChar *srcUniStr,                  //
     ByteCount srcUniStrLen,              //
     ByteCount *srcConvLen,               //
     BytePtr destStr,                     //
     ByteCount destStrLen,                //
     ByteCount *destConvLen,              //
     LogicalAddress contextPtr            //
);

// Example of use

pascal OSStatus myUnicodeToTextFallbackProc(UniChar *srcUniStr,
                  ByteCount srcUniStrLen, ByteCount *srcConvLen,
                  BytePtr destStr, ByteCount destStrLen,
                  ByteCount *destConvLen, LogicalAddress contextPtr) {
     // actual fallback handler implementation here
}
…
main () {
     UnicodeToTextFallbackPreemptiveProcPtr fallback;
     …
     fallback =
          (UnicodeToTextFallbackPreemptiveProcPtr) myUnicodeToTextFallbackProc;
     status = SetFallbackUnicodeToTextPreemptive(unicodeToTextInfo, fallback,
          controlFlags, infoPtr);
     …
}
```

For SetFallbackUnicodeToText, you must pass a UniversalProcPtr instead. This is derived from a pointer to the caller-defined routine by using the predefined macro NewUnicodeToTextFallbackProc, as shown below. The parameters and return value for the fallback handler itself are identical to those for SetFallbackUnicodeToTextPreempt, but the ProcPtr typedef has a different name (to make the Interfacer work correctly!).

```
// For use with SetFallbackUnicodeToText:

typedef pascal OSStatus (*UnicodeToTextFallbackProcPtr)(
     UniChar *srcUniStr,                  //
     ByteCount srcUniStrLen,              //
     ByteCount *srcConvLen,               //
     BytePtr destStr,                     //
     ByteCount destStrLen,                //
     ByteCount *destConvLen,              //
     LogicalAddress contextPtr            //
);
```

```
#if GENERATINGCFM
typedef UniversalProcPtr UnicodeToTextFallbackUPP;
#define NewUnicodeToTextFallbackProc(userRoutine)                \
            (UnicodeToTextFallbackUPP) NewRoutineDescriptor(     \
            (ProcPtr)(userRoutine),                              \
            uppUnicodeToTextFallbackProcInfo,                    \
            GetCurrentArchitecture())
#else
typedef UnicodeToTextFallbackProcPtr UnicodeToTextFallbackUPP;
#define NewUnicodeToTextFallbackProc(userRoutine)                \
            ((UnicodeToTextFallbackUPP) (userRoutine))
#endif

// Example of use

pascal OSStatus myUnicodeToTextFallbackProc(UniChar *srcUniStr,
                  ByteCount srcUniStrLen, ByteCount *srcConvLen,
                  BytePtr destStr, ByteCount destStrLen,
                  ByteCount *destConvLen, LogicalAddress contextPtr) {
     // actual fallback handler implementation here
}
…
main () {
     UnicodeToTextFallbackUPP fallback;
     …
     fallback = NewUnicodeToTextFallbackProc(myUnicodeToTextFallbackProc);
     status = SetFallbackUnicodeToTextPreempt(unicodeToTextInfo, fallback,
                  controlFlags, infoPtr);
     …
}
```

The parameters of the caller-supplied fallback handler are as follows:

The input parameter srcUniStr specifies the Unicode text element that the fallback handler should handle. The parameter srcUniStrLen specified the length of the text element string in bytes. In most cases the length is two, indicating a single Unicode character. The length may be more than two if the text element consists of a base character followed by combining characters, or if the text element consists of conjoining Korean jamos, etc.

The output parameter srcConvLen indicates how much of the text element was actually handled by the fallback handler (it specifies the length in bytes of the portion that was handled).

The input parameter destStr points to the output buffer. The maximum size of the buffer (in bytes) is specified by the input parameter destStrLen. The output parameter destConvLen indicates the actual length in bytes of the generated fallback sequence.

The parameter contextPtr is used to pass handler-specific data. It simply passes the contextPtr specified in SetFallbackUnicodeToText to the caller-defined fallback handler. The usage of the contextPtr is up to the caller-defined fallback handler.

The caller-defined fallback handler should not move memory, or call any toolbox routine that would move memory. If it needs memory, the memory should be allocated before the call to SetFallbackUnicodeToText function, and a memory reference should be passed either directly as contextPtr, or in the data referenced by contextPtr.

The caller-defined fallback handler should return noErr if it can handle the fallback, or kTECUnmappableElementErr if it cannot.

If SetFallbackUnicodeToText or SetFallbackUnicodeToTextPreemptive returns noErr, it has successfully installed the caller-defined fallback handler. Otherwise, there is one error condition:

- paramErr

8.16 UpgradeScriptInfoToTextEncoding

This function is written to enable programmers to translate encodings from the world of the Script Manager (script codes, language codes, region codes, and font names) to the world of Unicode and Text Objects (TextEncodings). For many of the ScriptIDs, there is a one-to-one correspondence with a particular TextEncodingBase value. But TextEncodingBase values are a super-set of ScriptIDs; that is, some combinations of ScriptID, language, region, and font name may result in a different TextEncodingBase value than would be given based on the ScriptID alone.

```
pascal OSStatus UpgradeScriptInfoToTextEncoding(
     ScriptCode iTextScriptID,              // input, req. if others are absent
     LangCode iTextLanguageID,              // input, req. if others are absent
     RegionCode iRegionID,                  // input, req. if others are absent
     ConstStr255Param iTextFontname,        // input, req. if others are absent
     TextEncoding *oEncoding);              // output

enum {
     kTextScriptDontCare = -128,
     kTextLanguageDontCare = -128,
     kTextRegionDontCare = -128
};
```

Of the four input parameters, a caller must specify at least one, but may specify two, three, or all four. Unspecified parameters must be set to the appropriate don't-care value as described below. The function will use as much information as is supplied in order to determine the most appropriate TextEncoding value. If multiple input parameters are specified, they will be cross-checked for validity.

textScriptID - Any explicit script code (i.e. in the range 0-32), as defined by the Script Manager. In addition, the following meta-values are supported:

- smSystemScript (-1): use system script

- smCurrentScript (-2): determine script from font in the grafPort
- smInputScript (-4): use script for current keyboard layout
- kUnicodeScriptDontCare (-128): this parameter is not specified

textLanguageID - Any valid Script Manager language code (langXxx), or kUnicodeLanguageDontCare if this parameter is unspecified.

regionID - Any valid Script Manager region code (verXxx), or kUnicodeRegionDontCare if this parameter is unspecified.

textFontname - must either be a font name that has special meaning for UpgradeScriptInfoToTextEncoding (i.e. Symbol or Zapf Dingbats), or the name of a font that is current installed. A value of NULL indicates this parameter is unspecified.

If the function returns noErr, it has successfully translated the values provided by the user to the appropriate TextEncoding value. Otherwise, it can return the following error codess:

- paramErr (This can be caused by two values which conflict, for instance, if the LanguageID is not understood by the Script Manager to belong to the ScriptID that was provided)
- kTECTableFormatErr

8.16.1 Operation

1. Resolve region and language, as illustrated by the following pseudo-code:

```
if (region is specified) {
     languageFromRegion = mapRegionToLanguage(region);
     if (language is unspecified)
          language = languageFromRegion;
     else if (language != languageFromRegion)
          return error;
}
```

At this point, we no longer need the region information. Region and language may both have been unspecified, in which case language is still unspecified at this point.

2. Resolve language and script. First check for special languages, as indicated in the following table:

If language is specified and is…	If script is specified, error unless it is…	If script is not specified, set it to…	Set TextEncodingBase to…
langCroatian or langSlovenian	smRoman	smRoman	kTextEncodingMacCroatian
langIcelandic	smRoman	smRoman	kTextEncodingMacIcelandic
langRomanian	smRoman	smRoman	kTextEncodingMacRomanian
langTurkish	smRoman	smRoman	kTextEncodingMacTurkish
langGreek	smRoman or smGreek	smRoman	kTextEncodingMacGreek
langUkrainian	smCyrillic	smCyrillic	kTextEncodingMacUkrainian

Otherwise (if language is unspecified or is not in the table above), proceed as with region and language (more pseudo-code):

```
if (language is specified) {
     scriptFromLanguage = mapLanguageToScript(language);
     if (script is unspecified)
          script = scriptFromLanguage;
     else if (script != scriptFromLanguage)
          return error;
}
```

Again, note that script may still be unspecified at this point (if region and language were also unspecified).

3. Resolve script and font name, as illustrated by the following pseudo-code:

```
if (fontname is specified) {
     if (fontname == "Symbol")
          TextEncodingBase = kTextEncodingMacSymbol;
     else if (fontname == "Zapf Dingbats")
          TextEncodingBase = kTextEncodingMacSymbol;
     else {
          if (font not installed)
               return error;
          scriptFromFontname = mapFontnameToScript(fontname);
          if (script is unspecified)
               script = scriptFromFontname;
          else if (script != scriptFromLanguage)
               return error;
          TextEncodingBase = script;
     }
else {
     if (script is unspecified)
          return error;
     TextEncodingBase = script;
}
```

8.17 RevertTextEncodingToScriptInfo

Since applications may require the use of Script and Font Manager functions, parameters must be in the format which are understood by the Script and Font Managers. Thus, RevertTextEncodingToScriptInfo is provided to enable translation from TextEncodings into a minimum of the ScriptID, with corresponding LanguageID or font name if they are unambiguous.

```
pascal OSStatus RevertTextEncodingToScriptInfo(
     TextEncoding iEncoding,                  // required input parameter
     ScriptCode *oTextScriptID,               // required output parameter
```

```
    LangCode *oTextLanguageID,                // optional output parameter
    Str255 oTextFontname);                    // optional output parameter
```

encoding - a valid TextEncoding. This is the primary input data parameter.

textScriptID - if this is a NULL pointer, the function returns an error. Otherwise it sets the indicated location to the appropriate script code.

textLanguageID - if this is not a NULL pointer, then the indicated location will be set to the appropriate language code if the language is unambiguous (e.g. for Japanese), or to kTextEncodingLanguageDontCare if the language is ambiguous.

textFontname - if this parameter is not NULL, then it will be set to the appropriate font name if the font name is unambiguous (e.g. for Symbol), or to a zero-length string if the font name is ambiguous.

If the function returns noErr, it has successfully translated the TextEncoding into textScriptID and optionally the other values. Otherwise, it can return the following error codes:

- paramErr
- kTECTableFormatErr

9 [Deleted]

10 [Deleted]

11 [Deleted]

12 [Deleted]

13 Localization

In addition to file and folder names and balloon help, the Unicode Converter permits specification of localized names for encodings, using 'tnam' resources in the files in the Text Encodings folder. Other than these items, it requires no localization. We will enlist the help of the countries to verify the quality of the conversion. For the tables that must be complete by Beta, we will enlist the help of localizers for verification.

Appendix A
Unicode Plain Text on the Macintosh

A.1 Introduction

This document describes the conventions for a plain text document on the Macintosh.

A.2 Requirements

Since we expect most documents created on a Macintosh will use a richer text model than pure Unicode, the emphasis here is on easy interchange with other platforms. In particular:

- An application should be able to import and export Unicode plain text files from other platforms with no data loss.
- An application should be able to import a Unicode plain text file into a rich text environment easily.

A.3 Content

A plain text Unicode document can contain any valid Unicode V1.1 character. In particular, it can contain control characters in the range U+0000 through U+001F and U+0080 through 009F. It may also contain codes in the Corporate and Private Use Zones although these may not interchange properly.

The individual Unicode characters must be 16-bit integers appropriate for the CPU. The first character of every Unicode plain text file on the Macintosh must be a ZERO-WIDTH NO BREAK SPACE (U+FEFF).

A.4 Display

Interpretation of control characters and escape sequences depends on the application. For SimpleText-like display, we suggest the following conventions:

Code Point	**Mnemonic**	**Interpretation & effect**
U+0009	HT	Move to the next tab position
U+000D	CR	Paragraph separator; move to the next line and to the beginning margin

Other control characters and escape sequences have no visible effect. There are normally no line separators, line breaking is the responsibility of the application.

A.5 File Type

A file which conforms to the content rules of A.3 should have the type ‘utxt’.

Appendix B
Encoding Tables

B.1 Introduction

This appendix describes the tables that the Unicode Converter uses.

For each set of related mappings, there is one 'uhdr' resource, one or more 'utom' resources, one or more 'ufrm' resources, possibly other resources, and possibly a plug-in. For each set of related encodings, there is on 'thdr' resource, one or more 'tnam' resources, and possibly one or more 'tprp' resources. All the resources and plug-ins for a set of related mappings and encodings are generally in one file (although sometimes certain encodings and mappings, especially newly added ones, will have their own files). If this file contains a plug-in, the file type is 'ecpg'; otherwise the file type is 'utbl'.

The TextEncodingBase values that correspond to Macintosh scripts, for example kTextEncodingMacArabic and kTextEncodingMacHebrew, are the same as the script codes used by the Script Manager. Inside Macintosh Text, pages 6-52 and 6-53 list these codes.

All planned tables will support both the Unicode V1.1 specification (and ISO 10646-1983) and the Unicode 2.0 specification. The converter design and code must be able to support subsets of these, as well as the various implementation levels of 10646 (although we may not supply tables that support all of these).

B.2 General requirements

The Unicode Converter must work with any encoding that has the following characteristics:

- The character repertoire has a substantial number of characters in common with ISO 10646.
- All code points are less than or equal to 4 bytes.
- The code point mapping does not depend on any global state, for example as set by a locking shift or embedded escape sequence.
- The code point mapping does not require more than 4 contextual forms (initial, medial, final, and alone).
- One code point must map to not more than 32 Unicode characters.
- If the character set supports user or corporate defined characters, the converter supports only the portion that fits in the Unicode private use zone.

B.3 Encodings with perfect round trip conversion

The Unicode Converter must allow perfect round-trip conversion, with use of the Corporate Use Zone, for at least the following encodings:

- All Mac OS encodings
- ISO 8859 parts 1, 2, 5, 6, 7, 8, 9
- JIS X0208
- GB 2312-80 (Simplified Chinese)
- KSC 5601-87 (Korean)
- TIS 620-2533 (Thai)

B.4 Unicode Converter Tables

The following table lists the standard encodings and variants that are supported as part of the Unicode Converter Project. This set is designed to (1) meet the needs of Cyberdog and Mac runtime for Java, and (2) test most of the critical code paths.

Category	Encoding base	Encoding variant, if more than one for this base
Western-language encodings:		
	Mac OS Roman	
	Mac OS Icelandic	
	ISO 8859-1 (Latin-1)	
	Windows Latin-1	
	DOS Latin-US	
	Mac Latin-1 (permutation of Mac OS Roman)	

Symbol encodings:
- Mac OS Symbol
- Mac OS Dingbats

Turkish encodings:
- Mac OS Turkish
- ISO 8859-9 (Latin-5)
- Windows Latin-5

Central European encodings:
- Mac OS Central European
- Mac OS Croatian
- Mac OS Romanian
- ISO 8859-2 (Latin-2)
- Windows Latin-2

Cyrillic encodings:
- Mac OS Cyrillic
- Mac OS Ukrainian
- ISO 8859-5 (Latin/Cyrillic)
- Windows Cyrillic
- KOI8-R

Greek encodings:
- Mac OS Greek
- ISO 8859-l (Latin/Greek)
- Windows Greek

Chinese encodings:
    Mac OS Chinese Simplified (EUC-CN)
    Mac OS Chinese Traditional (Big 5)
    EUC-TW
Japanese encodings:
    Mac OS Japanese (Shift-JIS)   StandardVariant
    "   BasicVariant
    "   PostScriptScrnVariant
    "   VertAtKuPlusTenVariant
    "   StdNoOneByteKanaVariant
    "   BasicNoOneByteKanaVariant
Korean encodings:
    Mac OS Korean (EUC-KR)
Arabic encodings:
    Mac OS Arabic
    ISO 8859-6 (Latin/Arabic)
    Windows Arabic
    DOS Arabic
Hebrew encodings:
    Mac OS Hebrew   StandardVariant
    ISO 8859-8 (Latin/Hebrew)
    Windows Hebrew
Thai encodings:
    Mac OS Thai
Indic encodings:
    Mac OS Devanagari
[tests 2-byte scanner]

Other encodings we plan to provide but are not part of minimum requirements

    GBK (Windows 95 Chinese)
    EUC-JP (needed for JIS 212 support) [tests 3-byte encodings]
    Mac OS Japanese (Shift-JIS)   StdNoVerticalsVariant
    "   PostScriptPrintVariant
    Mac OS Hebrew   FigureSpaceVariant [test variants for 1-byte scripts]