Text Encoding Reference

Preliminary Release

March, 1997

C H A P T E R 1

Text Encoding Conversions Reference

Contents

Text Encoding Conversions Constants and Data Types

Mac OS 8 provides data types, constants, and functions for creating text encoding specifications and obtaining the values that comprise an existing text encoding specification.

All text to be converted or truncated is expressed in a particular coded character set identified by a text encoding specification. Many of the Mac OS 8 system components include functions that take a text encoding specification as a parameter. Here are a few ways in which text encoding specifications are used:

- Text objects contain the text encoding specifications for the text they encapsulate.
- The Locale Object Manager locale objects contain text encoding specifications for user-displayable text strings they include.
- Encoding conversions performed by the Unicode Converter or the High-Level Encoding Converter require at least two text encoding specifications, the source and target text encodings.

You use text encoding specifications with many of the Text Object Manager functions. For example, when you use `InstallTextIntoTextObject` to place text in a text object or `AppendTextToTextObject` to append it to one, you give the text encoding specification for the text. When you call `GetTextObjectTextRuns` to obtain information about the text runs in a text object, the Text Object Manager returns the text encoding specification for each text run.

When you want the Locale Object Manager to return a particular locale object name string to you in a text object, you call `GetLocaleObjectName`, passing as one of its parameters the text encoding specification for the name. One of the standard attributes that a locale object can have associated with it is a text encoding specification.

You provide text encoding specifications to identify the source and target encodings for use with the Unicode Converter functions, such as `ConvertTextToUnicode`. You also provide a text encoding specification to determine where to safely truncate text when you call `TruncateForTextToUnicode`. When you use text encoding specifications with the Unicode Converter, you don't specify text encodings directly as parameters to these conversion and truncation functions. Rather, you give the text encoding specifications through a Unicode mapping data structure and you provide a Unicode mapping data structure from within a conversion information structure. See "Unicode Converter Reference" for more information.

Text Encoding Specification

A text encoding specification is an opaque scalar value that specifies the text encoding base, the text encoding variant, the text encoding format, and the packing version used for the text encoding or text encoding scheme. You use the following data types to specify three of these values when you create a text encoding specification: `TextEncodingBase` (page 1-4), `TextEncodingVariant` (page 1-8), and `TextEncodingFormat` (page 1-11). You don't specify the packing version. The values you specify are packed into an unsigned 32-bit value, which you can then pass by value either directly or from within other data structures to the functions that use text encodings or text encoding schemes.

A text encoding specification is defined by the `TextEncoding` data type.

```
typedef UInt32 TextEncoding; /* text encoding specification */
```

Text Encoding Base

You use a text encoding base data type to give the primary specification of the text encoding, or coded character set, used to express the text to which it pertains. A text encoding base is defined by the `TextEncodingBase` data type.

```
typedef UInt32 TextEncodingBase;
```

The `GetTextEncodingBase` function (page 1-15) returns the base encoding from the text encoding specification you pass it. If the encoded character set is a variant of the text encoding base, the text encoding specification can contain the variant name (page 1-8) that identifies the encoded character set. Obtaining the base encoding name does not also return the variant name. (To obtain the variant of the base encoding, you use `GetTextEncodingVariant` (page 1-16)).

The first 33 values (0 through 32) defined for text encoding bases correspond to the Macintosh script codes defined for System 7. For example, if the text encoding base is 4, then the base encoding is MacOS Arabic (`kTextEncodingMacArabic`), corresponding to System 7's Arabic, which was represented in System 7 by the constant `smArabic`.

Some values you can specify as text encoding bases are metavalues. A metavalue can refer to any of various real values, depending on the circumstances in which it is used. For example, the constant `kTextEncodingUnicodeDefault` represents the default base encoding for Unicode. You can use these enumerated constants to specify text encoding bases:

```
enum {
    kTextEncodingMacRoman       = 0,        /* Roman base encoding */
    kTextEncodingMacJapanese    = 1L,       /* Japanese base encoding */
    kTextEncodingMacTradChinese = 2L,       /* Traditional Chinese base encoding */
    kTextEncodingMacKorean      = 3L,       /* Korean base encoding */
    kTextEncodingMacArabic      = 4L,       /* Arabic base encoding */
    kTextEncodingMacHebrew      = 5L,       /* Hebrew base encoding */
    kTextEncodingMacGreek       = 6L,       /* Greek base encoding */
    kTextEncodingMacCyrillic    = 7L,       /* Cyrillic base encoding */
    kTextEncodingMacRSymbol     = 8L,       /* base encoding for right-to-left
                                                    symbols */
    kTextEncodingMacDevanagari  = 9L,       /* Devanagari base encoding */
    kTextEncodingMacGurmukhi    = 10L,      /* Gurmukhi base encoding */
    kTextEncodingMacGujarati    = 11L,      /* Gujarati base encoding */
    kTextEncodingMacOriya       = 12L,      /* Oriya base encoding */
    kTextEncodingMacBengali     = 13L,      /* Bengali base encoding */
    kTextEncodingMacTamil       = 14L ,     /* Tamil base encoding */
    kTextEncodingMacTelugu      = 15L,      /* Teluga base encoding */
    kTextEncodingMacKannada     = 16L,      /* Kannada/Kanarese base encoding */
    kTextEncodingMacMalayalam   = 17L,      /* Malayalam base encoding */
    kTextEncodingMacSinhalese   = 18L,      /* Sinhalese base encoding */
    kTextEncodingMacBurmese     = 19L,      /* Burmese base encoding */
    kTextEncodingMacKhmer       = 20L,      /* Khmer base encoding */
    kTextEncodingMacThai        = 21L,      /* Thai base encoding */
    kTextEncodingMacLaotian     = 22L,      /* Laotian base encoding */
    kTextEncodingMacGeorgian    = 23L,      /* Georgian base encoding */
    kTextEncodingMacArmenian    = 24L,      /* Armenian base encoding */
    kTextEncodingMacSimpChinese = 25L,      /* Simplified Chinese base encoding */
    kTextEncodingMacTibetan     = 26L,      /* Tibetan base encoding */
    kTextEncodingMacMongolian   = 27L,      /* Mongolian base encoding */
    kTextEncodingMacGeez        = 28L,      /* Geez/Ethiopic base encoding */
    kTextEncodingMacEastEurRoman    = 29,   /* extended Roman base encoding for
                                                Slavic and Baltic languages */
    kTextEncodingMacCentralEurRoman = 29,   /* extended Roman base encoding for
                                                Slavic and Baltic languages */
```

```
kTextEncodingMacVietnamese  = 30,       /* Roman base encoding for Vietnamese */
kTextEncodingMacExtArabic   = 31,       /* Extended Arabic base encoding for
                                            Sindhi */
kTextEncodingMacUninterp    = 32,       /* base encoding for uninterpreted
                                            symbols */
kTextEncodingMacSymbol      = 33,       /* base encoding for symbols font */
kTextEncodingMacDingbats    = 34.       /* base encoding for Zapf Dingbats font */
kTextEncodingMacTurkish     = 35,       /* Turkish base encoding */
kTextEncodingMacCroatian    = 36,       /* Croatian base encoding */
kTextEncodingMacIcelandic   = 37,       /* Icelandic base encoding */
kTextEncodingMacRomanian    = 38,       /* Romanian base encoding */
kTextEncodingMacUkrainian   = 152,      /* Ukrainian base encoding */
kTextEncodingMacBulgarian   = 153,      /* Bulgarian base encoding */
kTextEncodingMacHFS         = 0xFF,
/* Unicode and ISO Universal Multiple-Octet Coded Character Set (UCS) encodings */
    begin at 0x100 */
kTextEncodingUnicodeDefault = 0x100,    /* Unicode default base encoding,
                                            metavalue*/
kTextEncodingUnicodeV1_1    = 0x101,    /* Unicode version 1.1 base encoding,
                                            UCS4 format is not supported */
kTextEncodingISO10646_1993  = 0x102,    /* ISO10646, 1993 Unicode base encoding,
                                            UCS4 format is not supported */
/* ISO 8-bit and 7-bit encodings begin at 0x200 */
kTextkTextEncodingISOLatin1     = 0x201,    /* ISO 8859-1 */
kTextEncodingISOLatin2          = 0x202,    /* ISO 8859-2 */
kTextEncodingISOLatinCyrillic   = 0x205,    /* ISO 8859-5 */
kTextEncodingISOLatinArabic     = 0x206,    /* ISO 8859-6,=ASMO 708,=DOS cp708 */
kTextEncodingISOLatinGreek      = 0x207,    /* ISO 8859-7 */
kTextEncodingISOLatinHebrew     = 0x208,    /* ISO 8859-8 */
kTextEncodingISOLatin5          = 0x209,    /* ISO 8859-9 */
/* MS-DOS and Windows encodings begin at 0x400 */
kTextEncodingDOSLatinUS         = 0x400,    /* code page 437 */
kTextEncodingDOSGreek           = 0x405,    /* code page 737 (was cp 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 */
    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 */
    /* Various national standards begin at 0x600 */
    kTextEncodingUS_ASCII           = 0x600,    /* U.S. ASCII */
    kTextEncodingJIS_X0201_76       = 0x620,    /* JIS X0201, 1976 */
    kTextEncodingJIS_X0208_83       = 0x621,    /* JIS X0208, 1983 */
    kTextEncodingJIS_X0208_90       = 0x622,    /* JIS X0208, 1990 */
    kTextEncodingJIS_X0212_90       = 0x623,    /* JIS X0212, 1990 */
    kTextEncodingGB_2312_80         = 0x630,    /* GB 2312, 1980 */
    kTextEncodingKSC_5601_87        = 0x640,    /* Korean standard (KSC) 5601, 1987 */
    /* ISO 2022 collections begin at 0x800 */
    kTextEncodingISO_2022_JP        = 0x820,    /* ISO 2022, JP */
    kTextEncodingISO_2022_JP_2      = 0x821,    /* ISO 2022, JP 2 */
    kTextEncodingISO_2022_KR        = 0x840,    /* ISO 2022, KR */
    /* EUC (Extended Unix Code) collections begin at 0x900 */
    kTextEncodingEUC_JP             = 0x920,
    kTextEncodingEUC_KR             = 0x940,
    /* Other defacto standards begin at 0xA00 */
    kTextEncodingShiftJIS           = 0xA01,    /* plain Shift-JIS */
    kTextEncodingKOI8_R             = 0xA02     /* Russian internet standard */
};
```

Text Encoding Variant Data Type and Variants

A text encoding variant identifies a text encoding, or coded character set, some of whose less commonly used characters vary from those specified by the base encoding scheme with which the variant is related. In this case, variations in mapping usually exist only for relatively insignificant characters.

A variant can also specify a subset of a base encoding. For example, a variant of a base encoding that in its full specification includes vertical forms might exist for a subset of that encoding that does not include them. One variant of Unicode might exist that does not contain combining characters, another might exist that does not include coded characters in the Compatibility Zone, and yet another might exist that does not include coded characters in the Corporate Use Zone. An application might want to use the variant that does not include Corporate Use Zone coded characters to ensure round-trip fidelity for encoding text that is sent out over the Internet. Coded characters defined by Apple, for example, in the Corporate Use Zone are local to Apple and cannot be translated with certainty of fidelity.

Variants of the same base encoding can coexist in the same system as font variants. Two different text encodings that can both be used for body text in the same language on the same version of a localized platform are usually considered variants of the same base encoding. For example, the MacOS Icelandic and MacOS Turkish text encodings are considered different base encodings even though they belong to the same script; they normally do not coexist on the same Macintosh® system, and they each have their own language and region codes.

The enumeration of variants for a given text encoding base always begins with 0, and the constant specifying the default for the variant of a base encoding always implies the first variant in the set of variants defined by the enumeration.

A text encoding variant is defined by the `TextEncodingVariant` data type.

```
typedef UInt32 TextEncodingVariant;
```

You can explicitly specify a variant of a base encoding or you can specify the default variant of that base when your application calls the `CreateTextEncoding` function (page 1-14) to create a new text encoding. For example, if you specify the default constant `kTextEncodingDefaultVariant` as the value of the `CreateTextEncoding` function's `encodingVariant` parameter and the base encoding is MacOS Japanese (`kTextEncodingMacJapanese`), the resulting text encoding specification will identify the `kJapaneseStandardVariant` variant (page 1-8).

To obtain the text encoding variant of a text encoding specification, you use the `GetTextEncodingVariant` function (page 1-16). To create a new text encoding specification for a variant, you call the `CreateTextEncoding` function (page 1-14).

The following Unicode Converter enumeration defines constants for the default variant of any base text encoding and for variants of the MacOS Japanese, MacOS Hebrew, and Unicode Version 1.1 base encodings.

```
enum {
    /* Default TextEncodingVariant for any TextEncodingBase *
    kTextEncodingDefaultVariant     = 0,        /* default text encoding variant for
                                                    any text encoding base*/
    /* Variants of kTextEncodingMacJapanese */
    kJapaneseStandardVariant        = 0,        /* standard Japanese variant */
    kJapaneseStdNoVerticalsVariant  = 1,        /* without vertical presentation
                                                    forms */
    kJapaneseBasicVariant           = 2,        /* variant for basic interchange */
    kJapanesePostScriptScrnVariant  = 3,        /* PostScript-for-screens variant */
    kJapanesePostScriptPrintVariant = 4,        /* PostScript-for-printers variant */
    kJapaneseVertAtKuPlusTenVariant = 5,        /* variant in which the vertical forms
                                                    are located in the space beginning
                                                    at ku +10. */
    /* Variants of kTextEncodingMacHebrew */
    kHebrewStandardVariant          = 0,        /* standard Hebrew variant */
    kHebrewFigureSpaceVariant       = 1,        /* Hebrew figure space variant */

    /* Variants of kTextEncodingUnicodeV1_1 */
    kUnicodeNoSubset                = 0,        /* variant with full set of Unicode
                                                    10646 character encodings */
    kUnicodeNoCompatibilityVariant  = 1,        /* Unicode 10646 variant that does not
                                                    include character encodings in the
                                                    Compatibility zone */
    kUnicodeNoComposedVariant       = 3         /* Unicode 10646 variant that does not
                                                    include encodings for composed
                                                    characters */
};
```

Enumerator descriptions

`kJapaneseStandardVariant`

The standard Japanese variant.

`kJapaneseStdNoVerticalsVariant`
: The standard Japanese no-verticals variant. This variant is for users who don't want separately-encoded vertical forms, for example GX users.

`kJapaneseBasicVariant`
: The Japanese variant for basic interchange; all Shift-JIS platforms support the subset of characters in this variant.

`kJapanesePostScriptScrnVariant`
: The Japanese variant for screen bitmap version of the Sai Mincho and Chu Gothic fonts.

`kJapanesePostScriptPrintVariant`
: The Japanese variant for PostScript printing versions of the Sai Mincho and Chu Gothic PostScript fonts. This version includes two-byte halfwidth characters in addition to one-byte halfwidth characters.

`kJapaneseVertAtKuPlusTenVariant`
: The Japanese variant that includes encodings for vertical forms beginning at ku +10, that is, beginning with cells at row 11 in the JIS X0208-90 chart. In this chart, which is organized into rows and columns, a row is referred to as “ku” (meaning “ward”).

  Beginning at ku +10, which is equivalent to row 11, are rows of cells not used for the standard JIS X0208-90. These cells can be used for vertical presentation forms. Vertical forms can also be stored in another unused portion at the bottom of the chart beginning at ku +84. The `kJapaneseVertAtKuPlusTenVariant` constant indicates that vertical forms are stored beginning at ku +10. This variant is the same as the JIS X0208-90 standard in other aspects. This variant does not include Apple extensions.

`kHebrewStandardVariant`
: The standard Hebrew variant.

`kHebrewFigureSpaceVariant`
: The Hebrew variant in which xD4 represents figure space, not left single quote as in the standard variant.

`kUnicodeNoSubset`
: The standard 10646 Unicode encoded character set in which the full set of Unicode characters are supported.

`kUnicodeNoCompatibilityVariant`
: The Unicode 10646 variant that does not include encoded characters belonging to the Compatibility Zone.

`kUnicodeNoComposedVariant`
: The Unicode 10646 variant that does not include encoded characters that are composed characters.

Text Encoding Format

A text encoding format identifies the packing format of the encoded characters comprising the character set. For example, a character set such as Unicode might use 16-bit representations, as is the case for the UCS-2 format, or it might use 32-bit representations, as is the case for the UCS-4 format.

A text encoding format is defined by the `TextEncodingFormat` data type.

```
typedef UInt32 TextEncodingFormat;
```

The `GetTextEncodingFormat` function (page 1-17) returns the text encoding format of a text encoding specification.

You can explicitly specify a format of a text encoding or you can specify the default text encoding format when your application calls the `CreateTextEncoding` function (page 1-14) to create a new text encoding.

The following enumeration defines constants for specifying text encoding formats:

```
enum {
    /* default text encoding format for any text encoding base */
    kTextEncodingDefaultFormat = 0
    /* format for kTextEncodingUnicodeV1_1 */
    kUnicode16BitFormat = 0,
    /* format for Unicode Transformation Format 7 */
    kUnicodeUTF7Format = 1,
    /* format for Unicode Transformation Format 8 */
    kUnicodeUTF8Format = 2,
    /* formats for kTextEncodingISO10646_1993 */
    kISO10646UCS2Format = 0
};
```

Enumerator descriptions

`kTextEncodingDefaultFormat`
: The standard format for any base encoding.

`kUnicode16BitFormat`
: The 16-bit character encoding format specified by the Unicode 1.1 Standard.

`kUnicodeUTF7Format`
: The Unicode transformation format in which character encodings are represented by 7 bits.

`kUnicodeUTF8Format`
: The Unicode transformation format in which character encodings are represented by 8 bits.

`kISO10646UCS2Format`
: The 16-bit (double octet) format specified by the ISO/IEC 10646-1 UCS-2 (Universal Character Set containing 2 bytes) standard.

Unicode Character and String Pointer Data Types

The Unicode Converter functions that use a Unicode character data type 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. These functions also assume that each Unicode character is aligned on a 2-byte boundary. A 16-bit Unicode character is defined by the `UnicodeToTextRunInfo` data type.

```
typedef UInt16 UniChar;/* 16-bit Unicode character */
```

You specify a Unicode character array pointer to indicate an array used to hold a Unicode string. A Unicode character array pointer is defined by the `UniCharArrayPtr` data type.

```
typedef UniChar *UniCharArrayPtr; /* Unicode string pointer */
```

You specify a constant Unicode character array pointer for Unicode strings used within the scope of a function whose contents are not modified by that function. A constant Unicode character array pointer is defined by the `ConstUniCharArrayPtr` data type.

```
typedef const UniChar *ConstUniCharArrayPtr; /* Unicode string pointer */
```

Text Encoding Conversions Functions

Mac OS 8 provides functions you use to create or modify a text encoding specification and functions you use to obtain the contents of an existing text encoding specification. (These functions are provided because you must not directly modify a text encoding specification.)

You use the `CreateTextEncoding` function (page 1-14) to create a text encoding specification.

To obtain the values comprising an existing text encoding specification, you use these functions:

- `GetTextEncodingBase` to obtain the text encoding base name
- `GetTextEncodingVariant` to obtain the encoding variant
- `GetTextEncodingFormat` to obtain the encoding format

To obtain the name of the text encoding base of a text encoding specification in a language you specify, you use the `GetTextEncodingBaseName` function.

Creating a New Text Encoding Specification and Obtaining Values From an Existing One

A text encoding specification is an opaque scalar value that specifies the text encoding base name, the text encoding variant, the text encoding format, and the packing version used for the text encoding. The field values identify a particular coded character set used to represent the text. You use the `CreateTextEncoding` function to create a text encoding specification.

When you create a text encoding specification, the three values that you specify are packed into an unsigned integer, which you can then pass by value from within other data structures to the functions that use text encodings.

You use Text Encoding Conversions functions to obtain the contents of a text encoding specification; you must not access this data structure directly. You use the `GetTextEncodingBase`, `GetTextEncodingVariant`, and `GetTextEncodingFormat` functions to obtain the text encoding base, variant, and format from the specification.

CreateTextEncoding

Creates a text encoding specification from values you supply to the function and returns the text encoding specification.

```
TextEncoding CreateTextEncoding (
                    TextEncodingBase encodingBase,
                    TextEncodingVariant encodingVariant,
                    TextEncodingFormat encodingFormat);
```

`encodingBase` A text encoding base of type `TextEncodingBase` (page 1-4).

`encodingVariant`
A variant of the text encoding base (page 1-8). To specify the default variant for the base encoding given in the `encodingBase` parameter, you can use the `kTextEncodingDefaultVariant` (page 1-8) constant.

`encodingFormat`
A text encoding format (page 1-11).

function result
The text encoding specification that the function creates from the values you pass it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See "Creating a New Text Encoding Specification and Obtaining Values From an Existing One" (page 1-13) and the introduction following "Text Encoding Conversions Reference" (page 1-3) for more information.

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of text encodings.

GetTextEncodingBase

Obtains the text encoding base value from the specified text encoding.

```
TextEncodingBase GetTextEncodingBase (TextEncoding encoding);
```

`encoding`	A text encoding specification (page 1-3) containing the text encoding base you want to obtain.
function result	The text encoding base (page 1-4) belonging to the text encoding you specified.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See "Creating a New Text Encoding Specification and Obtaining Values From an Existing One" (page 1-13) and the introduction following "Text Encoding Conversions Reference" (page 1-3) for more information.

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of text encodings.

GetTextEncodingVariant

Obtains the text encoding variant value from the specified text encoding.

```
TextEncodingVariant GetTextEncodingVariant (TextEncoding encoding);
```

`encoding`	A text encoding specification (page 1-3) containing the text encoding variant you want to obtain.
function result	The text encoding variant (page 1-8) belonging to the text encoding you specified.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See "Creating a New Text Encoding Specification and Obtaining Values From an Existing One" (page 1-13) and the introduction following "Text Encoding Conversions Reference" (page 1-3) for more information.

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of text encodings.

GetTextEncodingFormat

Obtains the text encoding format value from the specified text encoding.

```
TextEncodingFormat GetTextEncodingFormat (TextEncoding encoding);
```

`encoding`	A text encoding specification (page 1-3) containing the text encoding format you want to obtain.
function result	The text encoding format (page 1-11) belonging to the text encoding you specified.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

See “Creating a New Text Encoding Specification and Obtaining Values From an Existing One” (page 1-13) and the introduction following “Text Encoding Conversions Reference” (page 1-3) for more information.

See the chapter “Introduction to Text Handling and Internationalization on Mac OS 8” in *Inside Macintosh: Text Handling and Internationalization* for an overview of text encodings.

GetTextEncodingBaseName

Obtains and returns the name of the text encoding base, whose specification you provide in the Unicode mapping, in the language you request.

```
OSStatus GetTextEncodingBaseName (
                        ConstUnicodeMappingPtr unicodeMapping,
                        LangCode languageID,
                        ByteCount bufLen,
                        ByteCount *nameLength,
                        UniCharArrayPtr mappingName);
```

`unicodeMapping`	A Unicode mapping data structure containing the encoding specification (page 1-4) whose text encoding base name you want to obtain. This data structure, which you provide, contains text encoding specifications for a Unicode encoding and any other text encoding. The function returns the name of the base encoding for the other text encoding. For information about the Unicode mapping data structure, see the "Unicode Converter Reference."
`languageID`	A System 7 Script Manager language code. A language code is used to indicate a particular written version of a language on the Macintosh. Note that this is not an ISO language code. See the System 7 *Inside Macintosh: Text* book for information on Script Manager language codes.
`bufLen`	The length in bytes of the buffer supplied by your application and pointed to by the `mappingName` parameter.
`nameLength`	A pointer to a value of type `ByteCount`. On output, the length in bytes of the name of the text encoding base returned in the `mappingName` parameter. The name can be up to 64 bytes long.
`mappingName`	A pointer to a Unicode character array (page 1-12) for the text encoding base name string. On output, the name as a Unicode string in the language specified by the `languageID` parameter. If the name does not exist in the specified language, the name is returned in the default language, which is English.

function result A result code. The function returns a `noErr` result code if it has found and returned the text encoding base name. If one or more of the input parameter values is invalid, the function returns a `paramErr` result code. If the converter could not find one of the mapping tables specified by the Unicode mapping structure you supply or one of the resources associated with it, the function returns a `unicodeNoTableErr` result code.

DISCUSSION

`GetTextEncodingBaseName` returns the base encoding name as a Unicode string. If `GetTextEncodingBaseName` cannot return the base encoding name in the language you specify, it returns the name in the default language, which is usually English.

Note
In the next developer release of the Unicode Converter, the type definition for the `unicodeMapping` parameter, which is now `ConstUnicodeMappingPtr`, will be changed to `TextEncoding;` and the type definition for the `languageID` parameter, which is now `LangCode`, will be changed to `LocaleIdentifier`. ◆

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

C H A P T E R 2

High-Level Text Encoding Converter Reference

Contents

High-Level Text Encoding Converter Constants and Data Types

Conversion Object Reference

You use a conversion object reference to refer to a conversion object. A conversion object is an opaque data type that specifies a conversion path between source and destination text encodings and specifies other context needed for performing a conversion. When you want to perform a conversion, you must pass a conversion object reference to the conversion function.

You create a conversion object and obtain a reference to it by calling the `TECCreateConverter` (page 2-22) or `TECCreateConverterFromPath` (page 2-23) functions. Once you obtain a conversion object reference, you can pass it to any of the functions that perform conversions. If you will be performing the same conversion multiple times, you should create the conversion object once and then reuse it.

Although you need to specify only the source and destination encodings to create and obtain a conversion object reference, in some cases the converter must perform intermediate conversions to convert text from a source encoding to a destination encoding. If you use `TECCreateConverter` to create a conversion object reference, the converter will choose the best intermediate conversion path; this process occurs internally and is transparent to your application. However, you may want to explicitly define the conversion path the converter should follow instead of relying on system behavior. If you know the most expedient path involving intermediate conversions, specifying it as an explicit path can improve performance in two areas.

Firstly, it can decrease the amount of time the converter requires to build the conversion object. When the converter builds the conversion object in response to your call to `TECCreateConverter`, it determines if it needs to perform intermediate conversions to achieve the encoding conversion you request. If so, the converter must poll all conversion plug-ins to determine the conversions they enable, then it must determine the best path using available conversions. Specifying your own conversion path eliminates this internal query and assessment process, allowing the converter to more quickly create the conversion object.

Secondly, it can assure you of the best conversion-process performance. If the converter needs to assess the best intermediate conversion path to use, it will choose, from among all possible paths, the shortest path to the destination encoding from the source one—that is, the path entailing the fewest number of conversions; this might not be the most expedient path (although it usually is). For example, the converter might elect to convert the source text to Unicode and then from Unicode to the destination encoding because this process entails only one intermediate conversion. However, a path involving two intermediate conversions might actually be faster.

To create a conversion object reference that contains an explicit conversion path, you use `TECCreateConverterFromPath`.

Both of these functions—`TECCreateConverter` and `TECCreateConverterFromPath` return the same kind of conversion object reference, `TECObjectRef`, which you can pass to any of the High-Level Text Encoding Converter functions that take a parameter of this type.

For conversions that entail use of a text encoding scheme, the converter stores and maintains state information identifying the current encoding in the conversion object. It also stores escape sequences, special control characters, and other information pertaining to encoding state changes in the conversion object. All of this is transparent to your application.

A conversion object has the ability to maintain state information. Therefore, your application can use the same conversion object reference to convert multiple segments of a single text stream. You might want to do this when the text is arriving in packets or when you do not know how large an output buffer to allocate for the converted string.

When you are finished using a conversion object to convert a particular stream of text, you can use the `TECClearConverterContextInfo` function (page 2-26) to clear its context. You can then use the same reference to the same conversion object—now cleared of its former context pertaining to the last text stream you used it for—to convert another text stream. The High-Level Text Encoding Converter provides this feature to encourage you to keep conversion objects around for multiple uses instead of creating them anew for converting different streams of text because the conversion object creation process incurs a large amount of overhead.

When you know you will no longer have need of a particular conversion object, you should release the memory allocated for it by calling the `TECDisposeConverter` function (page 2-25). A reference persists until you dispose of it. The `TECObjectRef` data type defines a conversion object reference.

```
typedef struct OpaqueTECObjectRef* TECObjectRef;
```

Text Encoding Conversion Information

When you call the `TECGetAvailableTextEncodingConversions` function (page 2-13), you pass an array of text encoding conversion information structures. The function fills the text encoding conversion information structures of the array with information about each type of supported conversion. A text encoding conversion information structure is defined by the `TECConversionInfo` data structure.

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

Field descriptions

`sourceEncoding`	A text encoding data structure giving the source encoding in which the text to be converted to the destination encoding is expressed. For information on text encoding specifications, see the “Text Encoding Conversions Reference.”
`destinationEncoding`	A text encoding data structure giving the destination encoding to which the text to be converted from the source encoding is expressed.
`reserved1`	Reserved.
`reserved2`	Reserved.

High-Level Text Encoding Converter Functions

Obtaining Information About Available Text Encodings

The number and kind of text encodings that the High-Level Text Encoding Converter supports depends on the conversion plug-ins currently installed in the system. Encoding conversion plug-ins, which provide conversion services between two encodings, are installed in the Text Encodings folder within the System folder.

You can use the `TECGetAvailableTextEncodings` function to obtain a list of text encoding specifications for all available text encodings that can be used for converting text. To know how large an array to allocate to hold this list, you can first call the `TECCountAvailableTextEncodings` function.

For any of the text encoding specifications returned in the list, you can use the `TECGetTextEncodingLocalizedName` function to obtain the text encoding name. You can display these encoding names in dialog boxes or menus, for example, to advertise available encodings to the user.

TECCountAvailableTextEncodings

Identifies the number of text encodings the encoding converter supports, which depends on its current configuration.

```
pascal OSStatus TECCountAvailableTextEncodings (ItemCount
                        *numberEncodings);
```

`numberEncodings`
: A pointer to a value of type `ItemCount`. On output, this pointer refers to the number of currently supported text encodings.

function result A result code. If `TECCountAvailableTextEncodings` returns a result code other than `noErr`, then one of the conversion plug-ins providing conversion services for various text encodings encountered an error condition while the High-Level

Encoding Converter polled it for the number of encodings it supports. For possible result codes, see “High-Level Text Encoding Converter Result Codes” on page 2-30.

DISCUSSION

`TECCountAvailableTextEncodings` counts and returns the number of text encodings that you can use to perform conversions based on the current configuration of the High-Level Text Encoding Converter. You need this number in order to determine how large of an array to allocate as the buffer that you pass to `TECGetAvailableTextEncodings` (page 2-8) in the `availableEncodings` parameter. Therefore, you should call this function before you call `TECGetAvailableTextEncodings` in order to allocate an array with enough elements to accommodate the specifications for all of these text encodings.

`TECCountAvailableTextEncodings` counts each instance of the same encoding. That is, if different conversion plug-ins support the same text encoding for any of the conversion processes they provide, `TECCountAvailableTextEncodings` includes each instance of the text encoding in its sum. Consequently, the same text encoding may be counted more than one time. However, the `TEGetAvailableTextEncodings` function does not return duplicate text encoding specifications. This means `TECCountAvailableTextEncodings` may return a number that is greater than the number of array elements required to hold the text encoding specifications that `TEGetAvailableTextEncodings` returns. However, to provide sufficient space, you should base the number of elements you allocate for the array on this count.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For background information, see “Obtaining Information About Available Text Encodings” on page 2-6.

TECGetAvailableTextEncodings

Returns the text encoding specifications identifying the encodings the converter is currently configured to handle.

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

`availableEncodings[]`
: An array composed of text encoding specification data structures. On output, the `TECGetAvailableTextEncodings` function fills the array with the specifications for the text encodings the encoding converter currently supports. To determine how large of an array to allocate, use the `TECCountAvailableTextEncodings` function (page 2-6). For information on text encoding specifications, see the “Text Encoding Conversions Reference.”

`maxAvailableEncodings`
: The quantity of text encoding specification data structures that the `availableEncodings` array can contain.

`actualAvailableEncodings`
: A pointer to a value of type `ItemCount`. On output, this pointer refers to the number of text encodings the function returned in the `availableEncodings` array.

function result A result code. If `TECGetAvailableTextEncodings` returns a result code other than `noErr`, then one of the conversion plug-ins providing conversion services for various text encodings encountered an error condition while the High-Level Encoding Converter polled it for the encodings it supports. For possible result codes, see “High-Level Text Encoding Converter Result Codes” on page 2-30.

DISCUSSION

The `TECGetAvailableTextEncodings` function returns the text encoding specifications in the array you pass to the function as the `availableEncodings` parameter, eliminating any duplicate information in the process. For example, if the converter is configured to support a number of conversion plug-ins some of which support the same text encoding, then `TECGetAvailableTextEncodings` returns only one instance of the specification for that encoding. Consequently, the number of encodings `TECGetAvailableTextEncodings` returns in the available encodings array may be fewer than the number of elements you allocated for the array based on your call to `TECCountAvailableTextEncodings` (page 2-15). `TECGetAvailableTextEncodings` tells you the number of specifications it returns in the `actualAvailableEncodings` parameter.

You can pass any of the specifications returned in the list to the `TECGetTextEncodingLocalizedName` function (page 2-10) to obtain the name of its text encoding.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For brief, background information, see "Obtaining Information About Available Text Encodings" on page 2-6.

TECGetTextEncodingLocalizedName

Returns the text encoding name in the requested language, given a specification for the text encoding.

```
pascal OSStatus TECGetTextEncodingLocalizedName(
                     TextEncoding textEncoding,
                     RegionCode region,
                     TextEncoding preferredEncoding,
                     ByteCount bufLen,
                     TextEncoding *nameEncoding,
                     ByteCount *nameLength,
                     Byte encodingName[]);
```

`textEncoding`	The text encoding specification for the encoding whose name you want to obtain. For information on text encoding specifications, see the "Text Encoding Conversions Reference."
`region`	A region code.
`preferredEncoding`	A text encoding specification.
`bufLen`	The length in bytes of the buffer given in the `encodingName` parameter.
`nameEncoding`	A pointer to a text encoding specification.
`nameLength`	A pointer to a value of type `ByteCount`. On output, the value gives the length of the name string returned in the `encodingName` parameter.
`encodingName`	An array of `Byte` types.
function result	A result code. If `TECGetTextEncodingLocalizedName` returns a result code other than `noErr`, then the conversion plug-in providing conversion services for the text encoding whose name you requested encountered an error condition. In this case, the High-Level Encoding Converter returns the error code passed through from the plug in. For possible result codes, see "High-Level Text Encoding Converter Result Codes" on page 2-30.

DISCUSSION

You use `TECGetTextEncodingLocalizedName` to obtain the name of an encoding in the language you specify so that you can display the encoding name to your user. You might want to tell your user about the encoding conversions that are available, giving a list of pairs showing source and destination encodings. In this case, you could call this function for each text encoding of a conversion after you determine the conversions made possible by the current configuration.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For background information, see "Obtaining Information About Available Text Encodings" on page 2-6.

To obtain a list of specifications for all text encodings available on the system, use the `TECGetAvailableTextEncodings` function (page 2-8).

To determine the destination encodings supported for a particular source encoding in order to identify available encoding conversions, you can call the `TECGetDestinationTextEncodings` function (page 2-17).

Identifying Direct Encoding Conversions

The High-Level Text Encoding Converter performs conversions between any two encodings if those encodings are supported by the current configuration of the converter, which depends on the current constellation of conversion plug-ins.

Conversion plug-ins provide services supporting direct conversion between two encodings. To obtain information identifying all direct conversions that the High-Level Text Encoding Converter can perform, you use the `TECGetDirectTextEncodingConversions` function. To determine the size of the array to pass to this function to obtain the returned information, you can first call the `TECCountDirectTextEncodingConversions` function.

The converter may not always be able to perform a direct conversion between two specific encodings supported by a single conversion plug-in. However, it may be able to carry out the requested conversion by including intermediate conversions. This entails using the services of more than one conversion plug-in to achieve the conversion from the specified source encoding to the destination one.

The `TECGetDirectTextEncodingConversions` and `TECCountDirectTextEncodingConversions` functions do not take into account conversions from a source to destination encoding entailing intermediate conversions. However, you can call `TECCreateConverter` (page 2-22) to find out if a specific conversion is possible when the `TECGetDirectTextEncodingConversions` function does not report it as a supported direct conversion. If the conversion you are interested in is not possible, `TECCreateConverter` cannot create the conversion object and the function will return an error result code.

TECCountDirectTextEncodingConversions

Gives the number of direct conversions that the encoding converter supports in its current configuration.

```
pascal OSStatus TECCountDirectTextEncodingConversions (
                        ItemCount *numberOfEncodings);
```

`numberOfEncodings`
A pointer to a value of type `ItemCount`. On output, this pointer refers to the number of direct conversions that the converter is currently configured to support.

function result A result code. If `TECCountDirectTextEncodingConversions` returns a result code other than `noErr`, then one of the conversion plug-ins polled by the converter encountered an error condition. In this case, the High-Level Encoding Converter returns the error code passed through from the plug in. For possible result codes, see "High-Level Text Encoding Converter Result Codes" on page 2-30.

DISCUSSION

You use the number that `TECCountDirectTextEncodingConversions` returns to determine how large to make the array you pass to `TECGetDirectTextEncodingConversions` (page 2-13).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For background information, see "Identifying Direct Encoding Conversions" on page 2-11.

TECGetDirectTextEncodingConversions

Returns the types of direct conversions the converter handles in its current configuration.

```
pascal OSStatus TECGetDirectTextEncodingConversions(
                        TECConversionInfo directConversions[],
                        ItemCount maxDirectConversions,
                        ItemCount *actualDirectConversions);
```

`directContersions[]`
: An array composed of text encoding conversion information structures (page 2-5). On output, the `TECGetDirectTextEncodingConversions` function fills the array with the text encoding conversion information structures identifying the types of conversions the encoding converter currently supports. To determine how large of an array to allocate, use the `TECCountDirectTextEncodingConversions` function (page 2-12).

`maxDirectConversions`
: The quantity of text encoding conversion information structures that the `directConversions` array can contain.

`actualDirectConversions`
: A pointer to a value of type `ItemCount`. On output, this pointer refers to the number of text encoding conversion information structures returned in the direct conversions array.

function result A result code. If `TECGetDirectTextEncodingConversions` returns a result code other than `noErr`, then one of the conversion plug-ins accessed by the converter encountered an error condition. In this case, the High-Level Encoding Converter returns the error code passed through from the plug in. For possible result codes, see "High-Level Text Encoding Converter Result Codes" on page 2-30.

DISCUSSION

The `TECGetDirectTextEncodingConversions` function returns the text encoding conversion information structures in the array you pass to the function as the `directConversions` parameter. Each element consists of a text encoding conversion information structure giving the source and destination encodings.

If you want to display the available direct encoding conversions to your user, you can obtain the source and destination text encoding names in the language you specify. To do this, for each encoding you pass the text encoding specification returned in a text encoding conversion information structure to the `TECGetTextEncodingLocalizedName` function (page 2-10).

You can use the conversion speed information this function returns to assess the fastest path when you want to specify intermediate conversions explicitly in creating a conversion object using the `TECCreateConverterFromPath` function (page 2-23).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For background information, see “Identifying Direct Encoding Conversions” on page 2-11.

Identifying Possible Destination Encodings

You can identify all possible destination encodings to which the encoding converter can convert a specific source encoding by calling the `TECGetDestinationTextEncodings` function. To determine how large of an array to allocate to hold the information `TECGetDestinationTextEncodings` returns, you can first call `TECCountDestinationTextEncodings`.

TECCountDestinationTextEncodings

Returns the number of encodings to which the High-Level Text Encoding Converter can convert the specified source encoding using a single, direct conversion process from the source encoding to another one.

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

`inputEncoding`
: A text encoding specification giving the source text encoding. For information on text encoding specifications, see the "Text Encoding Conversions Reference."

`numberOfEncodings`
: A pointer to a value of type `ItemCount`. On output, this pointer refers to the number of text encodings that the source encoding given in the `inputEncoding` parameter can be converted to directly.

function result A result code. If `TECCountDestinationTextEncodings` returns a result code other than `noErr`, then one of the conversion plug-ins polled by the converter encountered an error condition. In this case, the High-Level Encoding Converter returns the error code passed through from the plug-in. For possible result codes, see "High-Level Text Encoding Converter Result Codes" on page 2-30.

DISCUSSION

`TECCountDestinationTextEncodings` returns the number of direct encoding conversions possible from the given source encoding to any supported destination encodings. A direct encoding conversion consists of a conversion from the source encoding to a destination encoding that does not require one or more intermediate conversions. For example, suppose MacJapanese is the source encoding; conversion from MacJapanese to EUC_JP is considered direct whereas conversion from MacJapanese to EUC_JP and from EUC_JP to ISO2022-JP with ISO2022-JP as the destination encoding is considered indirect.

You can use the number that `TECCountDestinationTextEncodings` returns to determine how many text encoding specification elements to allocate for the array you pass to `TECGetDestinationTextEncodings` (page 2-17), which you call to obtain the text encoding specifications for the supported destination encodings.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

TECGetDestinationTextEncodings

Returns the encoding specifications for the destination text encodings to which the High-Level Text Encoding Converter can directly convert the specified source encoding.

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

`inputEncoding`
: A text encoding specification identifying the source text encoding. For information on text encoding specifications, see the “Text Encoding Conversions Reference.”

`destinationEncodings[]`
: An array of text encoding specification data structures. On output, the `TECGetDestinationTextEncodings` function fills the array elements with specifications for the destination encodings to which the High-Level Text Encoding Converter can directly convert the source encoding given in the `inputEncoding` parameter. Your application allocates memory for this array to accommodate the encodings that `TECGetDestinationTextEncodings` returns. To determine how large of an array to allocate, use the

`TECCountDestinationTextEncodings` function (page 2-15). For information on text encoding specifications, see the "Text Encoding Conversions Reference."

`maxDestinationEncodings`
: The maximum number of destination text encodings. This is the number of elements composing the array you provide in the `destinationEncodings` parameter.

`actualDestinationEncodings`
: A pointer to a value of type `ItemCount`. On output, this pointer refers to the number of text encoding specifications the function returned in the destination encodings array.

function result A result code. If `TECGetDestinationTextEncodings` returns a result code other than `noErr`, then one of the conversion plug-ins accessed by the converter encountered an error condition. In this case, the High-Level Encoding Converter returns the error code passed through from the plug-in. For possible result codes, see "High-Level Text Encoding Converter Result Codes" on page 2-30.

DISCUSSION

`TECGetDestinationTextEncodings` returns specifications for all possible destination text encodings to which the source encoding can be directly converted. You can display the names of these target encodings to your user in a specific language. You can call the `TECGetTextEncodingLocalizedName` function (page 2-10) to obtain a text encoding name based on its specification.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Identifying Text Encodings from Internet Names and Vice Versa

The Internet has its own set of names that identify text encodings in which text is expressed.

Given an Internet name, you can determine its equivalent text encoding for the Mac OS 8 platform using the `TECGetTextEncodingFromInternetName` function. To determine the equivalent Internet name for a text encoding supported on the Mac OS 8 platform, you can use the `TECGetTextEncodingInternetName` function.

TECGetTextEncodingFromInternetName

Given an Internet name, returns the corresponding Mac OS 8 text encoding specification.

```
pascal OSStatus TECGetTextEncodingFromInternetName(
                    TextEncoding *textEncoding,
                    ConstStr255Param encodingName);
```

`textEncoding`	A pointer to a text encoding data structure. On output, the structure contains the Mac OS 8 text encoding specification that corresponds to the Internet name specified by the `encodingName` parameter. For information about text encodings, see the “Text Encoding Conversions Reference.”
`encodingName`	The Internet encoding name for which you want the corresponding Mac OS 8 equivalent.
function result	A result code.

DISCUSSION

Names of text encodings defined on Mac OS 8 differ from names defined for the Internet to represent the same encodings. You can use `TECGetTextEncodingFromInternetName` to obtain the Mac OS 8 name for a text encoding given its Internet name.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

TECGetTextEncodingInternetName

Returns the Internet name for the specified text encoding.

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

`textEncoding`	The text encoding specification for the encoding whose Internet name you want to obtain. For information on text encoding specifications, see the “Text Encoding Conversions Reference.”
`encodingName`	The Internet name, returned by the function, that represents the text encoding specified by the `textEncoding` parameter.
function result	A result code.

DISCUSSION

The `TECGetTextEncodingInternetName` function returns the Internet name for the text encoding whose specification you provide.

If there are multiple Internet names for the same text encoding, the function returns the one that is designated the preferred name.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Creating and Deleting Conversion Objects

The High-Level Encoding Converter functions that perform conversions rely on a conversion object for information describing the encodings to and from which the text is to be converted. You pass a conversion object reference to a text conversion function when you call the function. You create a conversion object that specifies the source and destination encodings using the `TECCreateConverter` function and the function returns a reference to the object.

You can also create a conversion object that explicitly specifies the conversion path the converter should follow when it must perform intermediate conversions to achieve conversion to the final destination encoding. For this purpose, you use the `TECCreateConverterFromPath` function, which also returns a reference to the object. You can use a conversion object reference multiple times to convert portions of a single text stream.

After you use a conversion object for one conversion process, you can use it again for another. You can use the `TECClearConverterContextInfo` function to clear the context of a conversion object before you use it for the next conversion process. When you are entirely finished with a conversion object reference, you must delete the conversion object and reference using the `TECDisposeConverter` function.

TECCreateConverter

Creates a text encoding conversion object based on the specified source and destination encodings and returns a reference to it.

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

`newEncodingConverter`
: A pointer to a conversion object reference (page 2-3). On output, the reference pertains to the newly created text encoding conversion object.

`inputEncoding`
: The specification for the source text encoding. A source encoding identifies the encoding in which a byte stream to be converted using this reference is currently expressed. For information on the text encoding specification data structure, see the "Text Encoding Conversions Reference."

`outputEncoding`
: The specification for the destination text encoding. A destination encoding identifies the encoding to which a byte stream expressed in a source encoding is to be converted.

function result A result code. If `TECCreateConverter` returns a result code other than `noErr`, then it did not successfully create the conversion object reference. If the current configuration of the converter does not support either the source or destination encoding, the function returns a `kUnknownEncodingErr` result code. See "High-Level Text Encoding Converter Result Codes" (page 2-30) for additional result codes.

DISCUSSION

Given a source encoding and a destination encoding, `TECCreateConverter` determines a conversion path, creates a text encoding conversion object, and returns a reference to it. You use a conversion object reference with conversion functions such as `TECConvertText` (page 2-28). A conversion function, such as `TECConvertText`, relies on a conversion object for information identifying the type of conversion it is to perform—that is, for the source and destination encodings and other characteristics of the conversion including state information and references to required plug-ins.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To create a conversion object reference that contains a specific conversion path that you define—that is, a path from the source encoding to the destination encoding that entails specific intermediate conversions—use the `TECCreateConverterFromPath` function (page 2-23).

For brief, background information, see "Creating and Deleting Conversion Objects" on page 2-21.

TECCreateConverterFromPath

Creates a conversion object that includes a specific conversion path from a source encoding through intermediate encodings to a destination encoding and returns a reference to it.

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

`newEncodingConverter`	A pointer to a conversion object reference (page 2-3). On output, the reference pertains to the newly created text encoding conversion object.
`inPath[]`	An ordered array of text encoding specifications. The elements of the array begin with the source encoding specification followed by a series of specifications for the encodings through which the text is successively converted until it is converted to the destination encoding. For information on the text encoding specification data structure, see the "Text Encoding Conversions Reference."
`inEncodings`	The number of text encoding specifications composing the conversion path array given in the `inPath` parameter.
function result	A result code. If `TECCreateConverterFromPath` returns a result code other than `noErr`, then it did not successfully create the conversion object reference. If the current configuration of the converter does not support all of the encodings composing the array, the function returns a `kUnknownEncodingErr` result code. See "High-Level Text Encoding Converter Result Codes" (page 2-30) for additional result codes.

DISCUSSION

You use `TECCreateConverterFromPath` to create and obtain a reference to a conversion object that specifies a conversion path you define. You give the conversion sequence from the source encoding through intermediate encodings to the destination encoding instead of allowing the High-Level Text Encoding Converter to determine the path internally, as it does when you use `TECCreateConverter` (page 2-22) to create the conversion object. To specify the conversion sequence, you create an array of text encoding specification data types that identify the encoding conversions through which the text to be converted should pass from its source to destination encoding.

Each adjacent pair of text encodings you specify in the array must represent a conversion that is supported by the current configuration of the encoding converter. Otherwise, the function will return an error result code, in which case it will not create the conversion object. Therefore, you should use the `TECGetDestinationTextEncodings` function (page 2-17) to determine each subsequent step in the sequence from the source to the destination encoding.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To create a conversion object reference that allows the High-Level Text Encoding Converter to determine the conversion path from the source to destination encodings you specify, use the `TECCreateConverter` function (page 2-22).

For brief, background information, see "Creating and Deleting Conversion Objects" on page 2-21.

TECDisposeConverter

Disposes of the specified conversion object reference and the object it refers to.

```
pascal OSStatus TECDisposeConverter (TECObjectRef newEncodingConverter);
```

`newEncodingConverter`
The text encoding conversion object reference to be disposed of. This can be a reference returned by `TECCreateConverter` (page 2-22) or `TECCreateConverterFromPath` (page 2-23).

function result A result code. See "High-Level Text Encoding Converter Result Codes" (page 2-30) for result codes.

DISCUSSION

You use `TECDisposeConverter` to dispose of an encoding converter object and its reference when you have finished using the reference. Be sure not to specify a conversion object reference as a parameter to another function after you use `TECDisposeConverter` to dispose of it.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For brief, background information, see "Creating and Deleting Conversion Objects" on page 2-21.

TECClearConverterContextInfo

Clears the context pertaining to the last conversion process for which the conversion object was used.

```
pascal OSStatus TECClearConverterContextInfo (TECObjectRef
                        encodingConverter);
```

`encodingConverter`

The reference to the text encoding conversion object whose context is to be cleared.This can be a reference returned by `TECCreateConverter` (page 2-22) or `TECCreateConverterFromPath` (page 2-23).

function result	A result code. If `TECClearConverterContextInfo` was unable to clear the context, it returns a result code passed through from one of the conversion plug-ins. For possible result codes, see “High-Level Text Encoding Converter Result Codes” on page 2-30.

DISCUSSION

Creating a conversion object and obtaining a reference to it entails some overhead and expense. It is more economical for the system to allow you to reuse an existing conversion object than it is for it to create a new one containing the same conversion information. To make it possible for you to reuse conversion objects that you’ve already created, the High-Level Text Encoding Converter provides the `TECClearConverterContextInfo` function, which you call to clear a conversion object of any state and context information it contains that was used for the last conversion. Calling this function returns the conversion object to its original state.

When you convert multiple segments of a text string, you call the conversion function for each string using the same conversion object. In this case, you want the conversion object to maintain state. It is only after you completely convert all segments of the text string that you clear the conversion object used for the process.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Converting Text Between Encodings

TECConvertText

Converts a stream of text from a source encoding to a destination one, relying on the referenced conversion object for the encoding identifications.

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

`encodingConverter`
: The reference to the text encoding conversion object to be used for the conversion.This can be a reference returned by `TECCreateConverter` (page 2-22) or `TECCreateConverterFromPath` (page 2-23).

`inputBuffer`
: The source text stream. This is the text to be converted.

`inputBufferLength`
: The length in bytes of the stream of text given in the `inputBuffer` parameter.

`actualInputLength`
: A pointer to a value of type `ByteCount`. On output, this value returns the number of source text bytes, passed to the function in the `inputBuffer` parameter, that `TECConvertText` converted and returned in the `outputBuffer` parameter. If the buffer you specified in the `outputBuffer` parameter is not large enough to hold the entire converted text stream, you can use the value returned by this parameter to distinguish the remainder of the text to be converted. You can then call `TECConvertText` to convert the rest of the text stream.

`outputBuffer` A pointer to a buffer for a byte stream. On output, this buffer holds the converted text. For information on how to assess the size of the output buffer to allocate, see the function discussion. If the buffer that you allocate is not large enough to hold the entire converted text stream, `TECConvertText` returns the number of bytes of source text that were converted in the `actualInputLength` parameter. You can then call the `TECConvertText` again to convert the remaining text.

`outputBufferLength`
The length in bytes of the buffer provided by the `outputBuffer` parameter.

`actualOutputLength`
The length in bytes of the converted text returned in the buffer specified by the `outputBuffer` parameter.

function result
A result code. If there is not enough memory available for `TECConvertText` to convert the text, the function returns the appropriate Memory Manager result code.

DISCUSSION

`TECConvertText` converts the stream of text you pass it to the destination encoding specified in the text encoding conversion object whose reference you give. The function relies on the specified conversion object for information identifying the source encoding in which the text is expressed, the train of encodings forming the intermediate conversion path, if one is explicate specified, and any conversion options implicitly specified by default in the conversion object or explicitly set by your application. The function returns the text in the output buffer that you provide.

In allocating an output buffer, a good rule of thumb is larger is better, basing your estimate on the byte requirements of the destination encoding. You should always allocate a buffer that is at least 32 bytes long. For the function to complete successfully, the output buffer you allocate must be large enough to accommodate at least part of the converted text. In this case, if the function cannot convert all of the text, it will execute successfully and return the portion of the text it converted and information about the remaining text, so that you can call the function again to convert that text.

If the output buffer you allocate is too small to accommodate any converted text, the function will fail. For example, if the destination encoding requires additional bytes to identify the text encoding, such as an escape sequence preceding the converted text, your buffer must be large enough to accommodate the escape sequences and the text. If the destination encoding happens to be a text encoding scheme, such as ISO 2022, which begins in ASCII and switches to other coded character sets through limited combinations of escape sequences, then you need to allocate enough space to accommodate escape sequences signaling switches. ISO 2022 requires 3 bytes for an escape sequence preceding the 2-byte character it introduces. If you allocate a buffer that is less than 5 bytes, `TECConvertText` will fail.

If you are not sure of the required size for the output buffer and the one you allocate is too small to hold all of the converted text, you can call `TECConvertText` multiple times. `TECConvertText` returns the number of source bytes it removed from input in the `actualInputLength` parameter. You can use this number to identify the next byte to be taken and how many bytes remain.

During the conversion process, the High-Level Text Encoding Converter may need to create temporary buffers to contain intermediate conversions. The converter disposes of these buffers when it no longer needs them.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

High-Level Text Encoding Converter Result Codes

Many of the High-Level Text Encoding Converter functions return result codes. The various result codes specific to the converter are listed here.

Unicode Converter Reference

Contents

Unicode Converter Reference

Unicode Converter Constants and Data Types

The Unicode Converter performs table lookup-based conversions. Its primary use is to convert text from any text encoding to Unicode or to convert Unicode text to any text encoding. The Unicode Converter also allows you to convert text encoded in the coded character set of one text encoding to another using Unicode as a hub. (Table lookup-based conversion converts text encoded in a single text encoding to another; it does not deal with text encoding schemes.) Applications that want to convert text between any two text encodings typically use the High-Level Encoding Converter. However, when you want extensive error reporting and control over the conversion mapping process, you can use the Unicode Converter to convert between any two text encodings using Unicode as the hub.

For data types, constants and functions pertaining to text encodings, see the "Text Encoding Conversions Reference" chapter. This chapter describes the text encoding specification data type, data types you use to specify the base, variant, and format of a text encoding specification, and the functions you use to create a text encoding specification and obtain the values comprising one.

Conversion Information Reference for Converting to Unicode

The Unicode Converter `ConvertFromTextToUnicode` (page 3-32) and `ConvertPStringToUnicode` (page 3-62) functions that convert a text stream in another encoding to Unicode use a data structure that contains mapping and state information. When you call one of these functions, you pass a conversion information reference that points to the private conversion information data structure created for the conversion process. A conversion information reference is defined by the `TextToUnicodeInfo` data type.

```
typedef struct OpaqueTextToUnicodeInfo *TextToUnicodeInfo;
```

Your application cannot directly create or modify the contents of the private data structure pointed to by a conversion information reference. Instead, your application must first call the `CreateTextToUnicodeInfo` function (page 3-29) to provide the mapping information required for the conversion. The function creates and returns a conversion information reference. You can then pass this reference to `ConvertFromTextToUnicode` (page 3-32) or `ConvertPStringToUnicode` (page 3-62) to identify the information to be used in performing the actual conversion.

Your application can use the same conversion information reference to convert multiple segments of a single text stream to Unicode. After you have finished using a conversion information reference, you should release the memory allocated for it by calling the `DisposeTextToUnicodeInfo` function (page 3-36). Although a conversion information reference persists until you dispose of it, you should use the same conversion information reference only to convert segments of text belonging to the text stream for which you created the reference. For each new conversion process, you should create a new conversion information reference to convert another string of text even if you intend to use the same mapping information. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to that particular reference and the single text stream for which it is used.

Another function, the `TruncateForTextToUnicode` function (page 3-58), also requires a conversion information reference as a parameter. This function does not modify the contents of the private data structure to which the conversion information reference refers.

For functions that do not modify the conversion information contents, the Unicode Converter defines a data type that restricts the way in which the conversion information can be used. The declaration of the pointer is prefixed with the `const` keyword making the object pointed to a constant, but not the pointer—that is, not the reference—to it a constant. Functions that do not modify the conversion information contents declare a parameter of this type, indicating that the conversion information contents will remain unchanged within the scope of the function.The `TruncateForTextToUnicode` function takes a parameter of this type. A conversion information reference for use with functions that do not modify the conversion information contents is defined by the `ConstTextToUnicodeInfo` data type.

```
typedef const TextToUnicodeInfo ConstTextToUnicodeInfo;
```

Conversion Information References for Converting From Unicode to a Single Encoding

The Unicode Converter `ConvertFromUnicodeToText` (page 3-41) and `ConvertUnicodeToPString` (page 3-64) functions that convert Unicode to a single encoding use a data structure that contains mapping and state information. A conversion information reference is a pointer to a private data structure that the Unicode Converter uses to store private state and mapping information when converting a Unicode string to another encoding. A conversion information reference for this purpose is defined by the `UnicodeToTextInfo` data type.

```
typedef struct OpaqueUnicodeToTextInfo *UnicodeToTextInfo;
```

Your application cannot directly create or modify the contents of the private data structure pointed to by a conversion information reference. Instead, your application must first call `CreateUnicodeToTextInfo` (page 3-38) to provide the mapping information required for the conversion. The Unicode Converter creates and returns a conversion information reference. You can then pass this reference to `ConvertFromUnicodeToText` or `ConvertUnicodeToPString` to identify the information used to perform the actual conversion.

Your application can use the same conversion information reference to convert multiple segments of a single text stream. After you have finished using a conversion information reference, you should release the memory allocated for it by calling the `DisposeUnicodeToTextInfo` function (page 3-36). Although a conversion information reference persists until you dispose of it, you should use the same conversion information reference only to convert segments of text belonging to the text stream for which you created the reference. For each new conversion process, you should create a new conversion information reference to convert another stream of text even if you intend to use the same mapping information stored in an existing conversion information reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to that particular reference and the single text stream for which it is used.

Another function, `TruncateForUnicodeToText` (page 3-60) also requires a conversion information reference as a parameter. This function does not modify the contents of the private data structure the conversion information reference refers to. For functions that do not modify the conversion information contents, the Unicode Converter defines a data type that adds the `const` keyword to the declaration of the pointer to restrict the way in which the conversion information can be used by these functions. Prefixing the declaration of the pointer with `const` makes the object pointed to a constant, but not the pointer—that is, not the reference, to it—a constant. Functions that do not modify the conversion information contents declare a parameter of this type, indicating that the conversion information contents will remain unchanged within the scope of the function.The Unicode Converter `TruncateForUnicodeToText` function takes a parameter of this type. A constant information reference for determining where to truncate a Unicode string to be converted is defined by the `TruncateForUnicodeToText` data type.

```
typedef const UnicodeToTextInfo ConstUnicodeToTextInfo;
```

Conversion Information Reference for Converting From Unicode to One or More Encodings

The Unicode Converter functions that convert a Unicode string to one or more encodings require that you pass a conversion information reference. A conversion information reference is a pointer to an opaque data structure that the Unicode Converter uses to store private state and mapping information when converting a Unicode string to other encodings.

Your application cannot directly modify the contents of a private data structure containing conversion information. Instead, your application must call `CreateUnicodeToTextRunInfo` (page 3-47) to obtain a reference to the data structure. When you call `CreateUnicodeToTextRunInfo`, you provide the mapping information required for the conversion and the Unicode Converter creates the private data structure. You pass the reference returned from `CreateUnicodeToTextRunInfo` to `ConvertFromUnicodeToTextRun` (page 3-50) to identify the information to be used for the conversion. This function modifies the contents of the private data structure pointed to by the conversion information reference. A conversion information reference for this purpose is defined by the `UnicodeToTextRunInfo` data type.

```
typedef struct OpaqueUnicodeToTextRunInfo *UnicodeToTextRunInfo;
```

Your application can use the same conversion information reference to convert multiple segments of a single text stream from Unicode. After you have finished using a conversion information reference, you should release the memory allocated for it by calling the `DisposeUnicodeToTextRunInfo` function (page 3-36).

Although a conversion information reference persists until you dispose of it, you should use the same conversion information reference only to convert segments of text belonging to the text stream for which you created the reference. For each new conversion process, you should create a new conversion information reference to convert another stream of text even if you intend to use the same mapping information. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to the text stream for which it is used.

Text Encoding Run Structure

A stream of text may contain many text segments each of which may be expressed in a different encoding from that of the preceding or following text segment. This is referred to as a text encoding run. When you use `ConvertFromUnicodeToTextRun` (page 3-50) to convert a range of text containing text runs, the Unicode Converter returns information for each text run consisting of the starting offset of the text run segment and its text encoding specification. To obtain this information, you pass a pointer to an array of text run structures to the `ConvertFromUnicodeToTextRun` function when you call it to perform the conversion. Be sure to allocate enough array elements to accommodate information for all of the converted text runs. A text encoding run structure is defined by the `TextEncodingRun` data type.

```
struct TextEncodingRun {
    ByteOffset          offset;         /* beginning of the text run */
    TextEncoding        textEncoding;   /* text encoding for the run */
};
typedef struct TextEncodingRun TextEncodingRun;
```

Field descriptions

`offset`	The beginning character position of a run of text in the converted text string.
`textEncoding`	The encoding in which the text string beginning at the offset identified by the `offset` field is specified.

Conversion Control Flags

Your application uses control flags to determine the manner in which the conversion of text from one encoding to another is performed. The conversion functions—`ConvertFromTextToUnicode` (page 3-32), `ConvertFromUnicodeToText` (page 3-41), and `ConvertFromUnicodeToTextRun` (page 3-50)—allow you to set control flags specifying the conversion process behavior.

These functions take a `controlFlags` parameter whose value you can set using the bitmask constants defined for the flags. A different subset of control flags applies to each of these functions. Using the bitmask constants, you can perform the bitwise `OR` operation to set the pertinent flags for a particular function's parameters. For example, when you call a function, you might pass the following `controlFlags` parameter setting:

```
controlflags=kUnicodeUseFallbacksMask | kUnicodeLooseMappingsMask;
```

An exception to this is the directionality field, which is part of the control flags parameter. The directionality field allows you to specify the base line direction. To set the directionality field, which consists of multiple bits, you use the `kUnicodeDirectionalityBits` constant, not its equivalent mask, because you must shift the bits. For example, to include directionality among the control flags settings, when you call the function, you would pass the following `controlFlags` parameter assignment; this setting turns on loose and fallback mapping and it sets a left-to-right base line direction:

```
controlFlags=kUnicodeLooseMappingsMask | kUnicodeUseFallbacksMask |
(kUnicodeLeftToRight << kUnicodeDirectionalityBits);
```

The following enumerations define constants for the control flag masks.

```
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
};
```

Enumerator descriptions

`kUnicodeUseFallbacksMask`

A mask for setting the Unicode-use-fallbacks conversion control flag. The Unicode-use-fallbacks control flag determines how the Unicode Converter responds in regard to use of fallback mappings when it encounters a source text element for which there is no target encoding equivalent.

You can use this mask for the `controlFlags` parameter of `ConvertFromTextToUnicode` (page 3-32), `ConvertFromUnicodeToText` (page 3-41), and `ConvertFromUnicodeToTextRun` (page 3-50).

Set the flag for `ConvertFromTextToUnicode` to direct the Unicode Converter to use a fallback handler to perform fallback mapping. To do this, it substitutes one or more characters for the text element it cannot translate to the target encoding and continues converting the text string. Fallback mappings are mappings that do not preserve the meaning or identity of the source character but represent a useful approximation of it.

If Unicode-use-fallbacks conversion control flag is clear, `ConvertFromUnicodeToText` will complete execution and return to your application when the converter encounters a source text element for which there is no equivalent target encoding, after checking for a loose mapping, if that control flag is set.

If you set this flag and you did not call `SetFallbackUnicodeToText` (page 3-78) to associate an application-supplied fallback handler with the conversion information reference you pass to `ConvertFromUnicodeToText`, the Unicode Converter uses the system-supplied default fallback handler to perform fallback mapping.

If you set the flag for `ConvertFromUnicodeToText` and you called `SetFallbackUnicodeToText`, the Unicode Converter performs fallback mapping according to the control flags that you set when you called `SetFallbackUnicodeToText`. For `SetFallbackUnicodeToText`, you stipulate which fallback handler the Unicode Converter should call—the application-defined fallback handler or the default handler—if a fallback handler is required, and the sequence in which the Unicode Converter should call the fallback handlers if either can be used, for example, when the other fails or is unavailable. Depending on the `SetFallbackUnicodeToText` setting, the Unicode Converter uses the fallback character mapping defined by the application-supplied fallback handler or the fallback character defined by the mapping table for the target encoding.

If you also set the `kUnicodeLooseMappingsBit` control flag for `ConvertFromUnicodeToText`, the Unicode Converter will attempt to map a source text element for which there is no strict mapping target encoding equivalent to a loose mapping before it attempts to perform fallback mapping. If no loose mapping exists in the mapping table, the Unicode Converter will perform fallback mapping.

Fallback mapping from Unicode is a process in which a Unicode character is mapped to one coded character or a sequence of coded characters in another coded character set that may not have the same meaning or use, but that may provide an approximate graphic representation or even textual representation of the corresponding Unicode character. In general, fallback characters are not reversible, and therefore, do not lend themselves to round-trip fidelity conversions.

`kUnicodeKeepInfoMask`

A mask for setting the keep-information control flag. The keep-information control flag governs whether the Unicode Converter should keep the current state or initialize the conversion information data structure, whose reference you pass it, before converting the text string.

If keep-information control flag is clear, the converter will initialize the conversion information data structure before converting the text string.

If you set this flag, the converter will use the current state. This is useful if your application must convert a stream of text in pieces that are block delimited. You can set or clear this flag for the `controlFlags` parameter of the

ConvertFromTextToUnicode, ConvertFromUnicodeToText, and the ConvertFromUnicodeToTextRun functions.

kUnicodeDirectionalityMask

A mask for the directionality control flag. This flag is a field composed of multiple bits.

To set the directionality field, you use the kUnicodeDirectionalityBits constant, not this mask, because you must shift the bits.

The directionality control flags allow you to specify the global, or base, line direction for the text being converted. This determines which direction the converter should use for resolution of neutral coded characters, such as spaces that occur between sets of coded characters having different directions—for example, between Latin and Arabic characters—rendering ambiguous the direction of the space character. The following enumeration defines the possible settings:

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

The constant kUnicodeDefaultDirection tells the converter to use the value of the first strong direction character in the string. The constant kUnicodeLeftToRight tells the converter that the base paragraph direction is left-to-right. The constant kUnicodeRightToLeft tells the converter that the base paragraph direction is right-to-left.

If the directionality control flag is clear, the converter maps these text elements to their abstract forms.

This flag is valid for the controlFlags parameter of the ConvertFromUnicodeToText and ConvertFromUnicodeToTextRun functions.

kUnicodeVerticalFormBitMask

A mask for setting the vertical form control flag. The vertical form control flag tells the Unicode Converter how to map text elements for which there are both abstract and vertical presentation forms in the target encoding.

If you set this flag, the converter maps these text elements to their vertical forms. You can set or clear this flag for the `controlFlags` parameter of the `ConvertFromUnicodeToText` and `ConvertFromUnicodeToTextRun` functions.

A presentation form is a form of a graphic symbol representing a character that depends on the position of the character relative to other characters. Presentation forms can also be graphic symbols that represent multiple characters, including 1) for Arabic contextual forms, different forms that represent each Arabic character depending on its position relative to other characters 2) for Arabic ligatures, single forms that represent a sequence of Arabic characters 3) certain Latin ligatures, 4) different forms for some CJK punctuation and Japanese kana, depending on whether they are intended for horizontal or vertical display. Presentation variants include full-width and halfwidth characters. Some character sets encode presentation forms instead of or in addition to encoding abstract characters. For example, a number of text elements can be represented in Unicode either as single characters or as character sequences. Similarly, the presentation forms encoded in Unicode can also be represented using characters for the abstract forms.

`kUnicodeLooseMappingsBit`

A mask for setting the Unicode-loose-mapping control flag. The Unicode-loose-mapping control flag determines whether the Unicode Converter should use the loose mapping portion of a mapping table for character mapping if the strict mapping portion of the table does not include a target encoding equivalent for the source text element.

If this flag is clear, the converter will use only the strict equivalence portion; if it cannot convert using strict equivalence, the converter returns a `unicodeNotFoundErr` result code, and returns control to your application.

If you set this flag and a conversion for the source text element does not exist in the strict equivalence portion of the mapping table, then the converter will use the loose mapping section.

You can set or clear this flag for the `controlFlags` fields of the `ConvertFromUnicodeToText` and `ConvertFromUnicodeToTextRun` functions. The bitmask for this flag is defined by the constant `kUnicodeLooseMappingsMask`.

Strict and loose mappings occur within the context of multiple semantics and multiple representations. A mapping table has both strict equivalence and loose mapping sections. Strict mapping occurs when the mapping of a coded character from Unicode to Character Set X, for example, is the exact reverse of the mapping of that coded character from Character Set X to Unicode. In the case of multiple semantics—that is, when a single coded character in Character Set X represents two distinct but similar text elements, such as “double vertical line” and “parallel”, and two separate coded characters exist for these text elements in Unicode—a strict mapping exists between the single coded character in Character Set X and only one of the two coded characters in Unicode. Mapping to the other coded character in Character Set X would constitute a loose mapping. Loose mappings from Unicode to Character Set X are considered additional mappings that match the semantics established for the characters in Character Set X. For more information on strict mapping, see “Introduction to Text Handling and Internationalization on Mac OS 8” in *Inside Macintosh: Text Handling and Internationalization*.

`kUnicodeStringUnterminatedMask`

A mask for setting the string-unterminated control flag. This control flag determines how the Unicode Converter handles direction resolution and text element boundaries at the end of an input buffer.

If this bit is clear, the converter will assume that the next call you make using the current context will supply another buffer of text that should be treated as a continuation of the current text.

If you set this bit, the converter will treat the end of the buffer as the end of text.

You can set or clear this flag for the `controlFlags` parameter of the `ConvertFromUnicodeToText` and `ConvertFromUnicodeToTextRun` functions.

If this flag is set, when the Unicode Converter converts text from Unicode and it reaches the end of the current input buffer, the Unicode Converter will treat the end of the buffer as the end of the current text element. This treatment is valid for end-of-buffer characters that would never be part of a longer text element beyond the buffer—characters such as zero (0) with a non-joiner or a control character. However, if the last character in the input buffer were A, for example, the next buffer could begin with a diacritical mark, which would be part of the same text element; to handle cases such as this, you should clear this flag. If you clear this flag and the current text element extends beyond the buffer boundary because of the last character, the Unicode Converter will return the `unicodeElementErr` result code.

In attempting to analyze the text direction, when the Unicode Converter reaches the end of the current input buffer and the direction of the current text element is still unresolved, if you set this flag, the Unicode Converter will treat the end of the buffer as a block separator for direction resolution. If you clear this flag, it will set the direction as undetermined.

`kUnicodeTextRunMask`

A mask for setting the text-run control flag. This control flag determines how the Unicode Converter converts Unicode text to another encoding when more than one possible target encoding exists.

If this flag is clear, `ConvertFromUnicodeToTextRun` attempts to convert the Unicode text to the single encoding from the list of encodings in the Unicode-to-text-run conversion information reference that produces the best result, that is, that provides for the greatest amount of source text conversion.

If you set this flag, `ConvertFromUnicodeToTextRun` (page 3-50), which is the only function to which it applies, may generate a target string that combines text in any of the encodings specified by the Unicode-to-text-run conversion information reference.

`kUnicodeKeepSameEncodingMask`

A mask for setting the keep-same-encoding control flag. This control flag determines how the Unicode Converter treats the conversion of Unicode text following a text element that could not be converted to the first target encoding when multiple target encodings exist. This control flag applies only if the `kUnicodeTextRunMask` control is set.

If you set this flag, `ConvertFromUnicodeToTextRun` attempts to minimize encoding changes in the conversion of the source text string; that is, once it is forced to make an encoding change, it attempts to use that encoding as the conversion target for as long as possible.

If you clear this flag, `ConvertFromUnicodeToTextRun` attempts to keep most of the converted string in one encoding, switching to other encodings only when necessary.

The following enumeration defines constants for these control flags. Except for the `kUnicodeDirectionalityBits` constant, which you use to set the directionality field to specify the base line direction, these constants are provided for your information only. You do not need to use them in setting a function's `controlFlags` parameter. Rather, you use the masks for the control flags, described previously, to set the flags.

```
enum {
    kUnicodeUseFallbacksBit         = 0,    /* use fallback handler */
    kUnicodeKeepInfoBit             = 1,    /* use current conversion information data
                                                structure state */
    kUnicodeDirectionalityBits      = 2,    /* offset for bidirectionality
                                                specification field (2 bits) */
    kUnicodeVerticalFormBit         = 4,    /* use vertical presentation for text */
    kUnicodeLooseMappingsBit        = 5,    /* use loose mapping */
    kUnicodeStringUnterminatedBit   = 6,    /* unterminated string */
    kUnicodeTextRunBit              = 7,    /* use multiple target encodings */
    kUnicodeKeepSameEncodingBit     = 8     /* use same encoding for next text run */

};
```

Enumerator descriptions

`kUnicodeUseFallbacksBit`
: The Unicode-use-fallbacks conversion control flag.

`kUnicodeKeepInfoBit`
: The keep-information control flag.

`kUnicodeDirectionalityBits`
: The offset for the directionality specification field (2 bits).

`kUnicodeVerticalFormBit`
: The vertical form control flag.

`kUnicodeLooseMappingsBit`
: The Unicode-loose-mapping control flag.

`kUnicodeStringUnterminatedBit`
: The string-unterminated control flag.

`kUnicodeTextRunBit`
: The text-run control flag.

`kUnicodeKeepSameEncodingBit`
: The keep-same-encoding control flag.

Control Flags for Truncating a Unicode String

Your application can use two control flags to specify aspects of how a string is truncated when you call `TruncateForUnicodeToText` (page 3-60) to identify where to properly truncate a Unicode string before converting the text to any encoding. The following enumeration defines constants for setting these two control flags:

```
enum {
    kUnicodeTextElementSafeMask = 1L << kUnicodeTextElementSafeBit,
    kUnicodeRestartSafeMask = 1L << kUnicodeRestartSafeBit
};
```

Enumerator descriptions

`kUnicodeTextElementSafeMask`
: A mask for setting the text-element-safe control flag. This control flag determines whether a string should be truncated to include complete text elements. If the text-element-safe control is set, then the truncated string, as identified by `TruncateForUnicodeToText`, will contain

complete text elements. You should usually set this control flag, even if you set other flags.

`kUnicodeRestartSafeMask`
: A mask for setting the Unicode-restart-safe control flag. This control flag specifies whether the string should be truncated so that it is safe for processing by the `ConvertFromUnicodeToText` function (page 3-41). If the Unicode-restart-safe control flag is set, then the truncated string will be safe for processing by the `ConvertFromUnicodeToText` function even if the string is not block delimited, that is, it does not begin on a paragraph boundary.

The following enumerations define constants for these control flags. These constants are provided for your information only. You do not need to use them in setting the `TruncateForUnicodeToText` function's `controlFlags` parameter. Rather, you use the masks for the control flags, described previously, to set the flags comprising the `controlFlags` parameter.

```
enum {
    kUnicodeTextElementSafeBit      = 0,    /* text contains complete elements */
    kUnicodeRestartSafeBit          = 1     /* text is safe for use with
                                               ConvertFromUnicodeToText function */
};
```

Enumerator descriptions

`kUnicodeTextElementSafeBit`
: The text-element-safe control flag.

`kUnicodeRestartSafeBit`
: The Unicode-restart-safe control flag.

Control Flags for Specifying the Fallback Handlers and Their Calling Order

You can set control flags to identify which fallback handler is to be used—or if both are to be used in the case of the first one failing to convert the character, the order in which they are called—when you assign a fallback handler for use with a specific conversion information reference. A fallback handler is a function that is used when the Unicode Converter cannot convert a Unicode character to the target encoding using the strict equivalence, or, if kUnicodeLooseMappingsBit is set, loose mapping portions of the specified mapping table.

A fallback mapping is a sequence of one or more characters in the target encoding that are not exactly equivalent to the source characters but which preserve some of the information of the original. For example, (C) is a possible fallback mapping for ©.

When your application calls ConvertFromUnicodeToText (page 3-41) or ConvertUnicodeToPString (page 3-64) and the specified mapping table does not contain a target encoding equivalent for a Unicode source text element, the Unicode Converter uses the Unicode-to-text fallback handler associated with the conversion information reference you pass to the function. The Unicode Converter supplies a default fallback handler that you can associate with a conversion information reference, but you can also supply your own fallback handler and use it instead of or in addition to the default handler if it is unable to perform the fallback mapping.

You set the controlFlags parameter of the SetFallbackUnicodeToText function (page 3-78) to specify whether the Unicode Converter should call the default or application-supplied fallback handler or whether it should call both handlers—and if so, the order in which they are called.

The following enumerations define constants for the setting the controlFlags parameter of the SetFallbackUnicodeToText function.

```
enum {
    kUnicodeFallbackDefaultOnly     = 0L,   /* use default fallback handler only */
    kUnicodeFallbackCustomOnly      = 1L,   /* use custom fallback handler only */
    kUnicodeFallbackDefaultFirst    = 2L,   /* use default first, then custom */
    kUnicodeFallbackCustomFirst     = 3L    /* use custom first, then default */
};
```

Enumerator descriptions

kUnicodeFallbackDefaultOnly
: Sets the controlFlag parameter to direct the Unicode Converter to call its own default fallback handler to find a fallback element and not call the application-defined one.

kUnicodeFallbackCustomOnly
: Sets the controlFlag parameter to direct the Unicode Converter to call only the specified application-defined

fallback handler. The Unicode Converter will not call the default fallback handler.

`kUnicodeFallbackDefaultFirst`
: Sets the `controlFlag` parameter to direct the Unicode Converter to call its own default fallback handler first to find a fallback element, and if that fails to find an appropriate target element, to call the application-defined fallback handler.

`kUnicodeFallbackCustomFirst`
: Sets the `controlFlag` parameter to direct the Unicode Converter to call the application-defined fallback handler first, and if that fails, to call its own default fallback handler.

The following enumerations define constants for the control flag and its mask. These constants are provided for your information only.

```
enum {
    kUnicodeFallbackSequencingBits = 0
};
```

Enumerator descriptions

`kUnicodeFallbackSequencingBits`
: The offset for the control flag field that indicates which fallback handler is to be used first.

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

Enumerator descriptions

`kUnicodeFallbackSequencingMask`
: The mask for the control flag field that indicates which fallback handler is to be used first.

Constants for Script Manager Value Conversions To and From Text Encodings

For backward-compatibility, the Unicode Converter provides the `UpgradeScriptInfoToTextEncoding` (page 3-83) function that you can use to upgrade Script Manager language code, region code, script code, and font values to a text encoding and the `RevertTextEncodingToScriptInfo` function (page 3-87) to revert a text encoding to its corresponding Script Manager values. The Unicode Converter defines three constants for use with these functions.

If the one of these values can be correctly derived from another value, you can use the constant defined by the Unicode Converter for that value when you call `UpgradeScriptInfoToTextEncoding`. A constant is not defined for the font value; you can specify `NULL` instead of a constant for the font.

If the `RevertTextEncodingToScriptInfo` function cannot return the language code, the function returns a value of `kTextLanguageDontCare`.

When you call `UpgradeScriptInfoToTextEncoding`, you can specify values for one or more parameters, but you must specify at least one parameter value. If you do not specify a value for a parameter of this function, you must use one of the constants defined by the following enumerations to indicate this:

```
enum {
    kTextLanguageDontCare = –128
};

enum {
    kTextScriptDontCare = –128
};

enum {
    kTextRegionDontCare = –128
};
```

Enumerator descriptions

`kTextLanguageDontCare`
: Specifies that the language code is not provided for the translation.

`kTextScriptDontCare`
: Specifies that the script code is not provided for the translation.

`kTextRegionDontCare`
: Specifies that the region code is not provided for the translation.

Filter Indicators for Querying for Matching Unicode Mappings

The Unicode Converter allows you to query it for a list of all conversion mappings available on the system whose field values match field values you use as filters. For example, you can check for all available mappings between Unicode Version 1.1 and the default variant encoding of other encodings. To identify the fields of the conversion mappings to be checked for matches, you specify the fields of the two encoding specifications belonging to a Unicode mapping data structure.

You call the `QueryUnicodeMappings` function to obtain conversion mappings. When you call this function, you specify a pointer to a Unicode mapping data structure (page 3-24). A Unicode mapping data structure contains fields that specify text encodings. For information about text encodings and the constants you use to specify them, see the “Text Encoding Conversions Reference.” To identify the fields of the Unicode mapping data structure whose values are used as matching criteria, you set a `filter` parameter that you pass to the `QueryUnicodeMappings` function (page 3-67). The filter parameter value identifies which of the six fields of the Unicode mapping data structure are to be used as matching criteria—the three fields of the Unicode encoding specification and the three fields of the other text encoding specification. The function will return only those mappings whose corresponding field values match the ones you specify.

You can use the constants defined by the following enumeration to set the filter indicators of the `QueryUnicodeMappings` function’s `filter` field.

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

Enumerator descriptions

`kUnicodeMatchUnicodeBaseMask`
A mask for setting the match-Unicode-base filter. This filter indicator directs `QueryUnicodeMappings` to match against the text encoding base value belonging to the text encoding specified by the `unicodeEncoding` field of the

Unicode mapping data structure. `QueryUnicodeMappings` returns only those mappings whose corresponding field value matches this one. If this filter is not set, the function ignores the field it represents.

`kUnicodeMatchUnicodeVariantMask`

A mask for setting the match-Unicode-variant filter. This filter indicator directs `QueryUnicodeMappings` to match against the text encoding variant value belonging to the text encoding specified by the `unicodeEncoding` field of the Unicode mapping data structure. If this filter is not set, the function ignores the field it represents.

`kUnicodeMatchUnicodeFormatMask`

A mask for setting the match-Unicode-format filter. This filter indicator directs `QueryUnicodeMappings` to match against the text encoding format value belonging to the text encoding specified by the `unicodeEncoding` field of the Unicode mapping data structure.

`kUnicodeMatchOtherBaseMask`

A mask for setting the match-other-base filter. This filter directs `QueryUnicodeMappings` to match against the text encoding base value belonging to the text encoding specified by the `otherEncoding` field of the Unicode mapping data structure. If this filter is not set, the function will ignore the field it represents.

`kUnicodeMatchOtherVariantMask`

A mask for setting the match-other-variant filter. This filter directs `QueryUnicodeMappings` to match against the text encoding variant value belonging to the text encoding specified by the `otherEncoding` field of the Unicode mapping data structure. If this filter is not set, the function will ignore the field it represents.

`kUnicodeMatchOtherFormatBit`

A mask for setting the match-other-format filter. This filter directs `QueryUnicodeMappings` to match against the text encoding format value belonging to the text encoding specified by the `otherEncoding` field of the Unicode mapping data structure. If this filter is not set, the function will ignore the field it represents.

The following enumerations define constants for these filter indicators. These constants are provided for your information only. You do not need to use them in setting the `filter` parameter. Rather, you use the masks for the filter indicators, described previously, to set the `filter` parameter.

```
enum {
    kUnicodeMatchUnicodeBaseBit = 0,    /* match on Unicode base encoding name */
    kUnicodeMatchUnicodeVariantBit = 1, /* match on Unicode variant */
    kUnicodeMatchUnicodeFormatBit = 2,  /* match on Unicode format */
    kUnicodeMatchOtherBaseBit = 3,      /* match on base encoding name of other
                                            encoding */
    kUnicodeMatchOtherVariantBit = 4    /* match on variant of other encoding */
    kUnicodeMatchOtherFormatBit = 5,    /* match on format of other encoding */
};
```

Enumerator descriptions

`kUnicodeMatchUnicodeBaseBit`
: The match-Unicode-base filter indicator.

`kUnicodeMatchUnicodeVariantBit`
: The match-Unicode-variant filter indicator.

`kUnicodeMatchUnicodeFormatBit`
: The match-Unicode-format filter indicator.

`kUnicodeMatchOtherBaseBit`
: The match-other-base filter indicator.

`kUnicodeMatchOtherVariantBit`
: The match-other-variant filter indicator.

`kUnicodeMatchOtherFormatBit`
: The match-Unicode-format filter indicator.

Unicode Character and String Pointer Data Types

The Unicode Converter functions that use a Unicode character data type 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. These functions also assume that each Unicode character is aligned on a 2-byte boundary. A 16-bit Unicode character is defined by the `UniChar` data type.

```
typedef UInt16 UniChar;/* 16-bit Unicode character */
```

You specify a Unicode character array pointer to indicate an array used to hold a Unicode string. A Unicode character array pointer is defined by the `UniCharArrayPtr` data type.

```
typedef UniChar *UniCharArrayPtr; /* Unicode string pointer */
```

You specify a constant Unicode character array pointer for Unicode strings used within the scope of a function whose contents are not modified by that function. A constant Unicode character array pointer is defined by the `ConstUniCharArrayPtr` data type.

```
typedef const UniChar *ConstUniCharArrayPtr; /* Unicode string pointer */
```

Region Code Data Type

A region code data type is a 16-bit value used to hold a region code that specifies a particular region as defined by System 7. The `UpgradeScriptInfoToTextEncoding` function (page 3-83) takes a region code parameter. A region code is defined by the `RegionCode` data type.

```
typedef short RegionCode; /* region code */
```

For a description of a region code of this type, see the System 7 *Inside Macintosh: Text* book.

Unicode Mapping Structure

A Unicode mapping data structure identifies a Unicode encoding specification and a particular base encoding specification to be used for conversion of a text string. You use a data structure of this type to specify the text encodings to and from which the text string is to be converted. Because the Unicode Converter always converts to and from Unicode, at least one of these is always a Unicode encoding specification for any conversion. A Unicode mapping data structure is defined by the `UnicodeMapping` data type.

```
struct UnicodeMapping {
    TextEncoding        unicodeEncoding;
    TextEncoding        otherEncoding;
    UnicodeMapVersion   mappingVersion;
};
typedef struct UnicodeMapping UnicodeMapping
typedef UnicodeMapping *UnicodeMappingPtr;
```

Field descriptions

`unicodeEncoding`	A Unicode text encoding specification of type `TextEncoding`. You use one of these base encodings, described in see the “Text Encoding Conversions Reference” chapter to identify the Unicode format: `kTextEncodingUnicodeDefault`, `kTextEncodingUnicodeV1_1`, `kTextEncodingUnicodeV2_0`, `kTextEncodingISO10646_1993`.
`otherEncoding`	Any text encoding specification. You can give a Unicode text encoding specification or a text encoding specification containing any other base encoding variant. For information about text encoding specifications, see the “Text Encoding Conversions Reference” chapter.
`mappingVersion`	The version of the Unicode mapping table to be used. (To specify that the latest version is to be used, set this field to the `kUnicodeUseLatestMapping` constant.) A later developer release will address how to obtain a value from a Unicode mapping table to pass as this parameter when you want to specify a version of a Unicode mapping table other than the latest one.

Many Unicode Converter functions take a pointer to a Unicode mapping data structure as a parameter. For functions that do not modify the Unicode mapping contents, the Unicode Converter defines a data type that adds the `const` keyword to the declaration of the pointer to restrict the way in which the Unicode mapping can be used by these functions. Prefixing the declaration of the pointer with `const` makes the object pointed to a constant, but not the pointer itself. The `CreateTextToUnicodeInfo` (page 3-29), `CreateUnicodeToTextInfo` (page 3-38), `QueryUnicodeMappings` (page 3-67), `CreateUnicodeToTextRunInfo` (page 3-47), `GetTextEncodingBase`, and `GetTextEncodingBaseName` functions do not modify the Unicode mapping contents and therefore take a parameter of this type, indicating that the Unicode mapping contents will remain unchanged within the scope of the function. (For information on the `GetTextEncodingBase` and `GetTextEncodingBaseName` functions, see the “Text Encoding Conversions Reference” chapter.) A constant pointer to a Unicode mapping data structure is defined by the `ConstUnicodeMappingPtr` data type.

```
typedef const UnicodeMapping *ConstUnicodeMappingPtr;
```

The `ChangeTextToUnicodeInfo` (page 3-70) and `ChangeUnicodeToTextInfo` (page 3-72) functions modify the contents of the Unicode mapping data structure and take a pointer parameter of type `UnicodeMappingPtr`.

Unicode Mapping Version

When performing conversions, you specify the version of the Unicode mapping table to be used for the conversion. You provide the version number in the mapping version field of the Unicode mapping data structure (page 3-24) that is passed to a function. A Unicode mapping version is defined by the `UnicodeMapVersion` data type.

```
typedef SInt32  UnicodeMapVersion;
```

Note
For this release, you can direct the converter to use the latest version by setting the mapping version field to `kUnicodeUseLatestMapping` (page 3-26). A future developer release will address how to obtain a value from a Unicode mapping table to pass as this parameter when you want to specify a version of a Unicode mapping table other than the latest one.

Latest Unicode Mapping Version

Instead of explicitly specifying the mapping version of the Unicode mapping table to be used for conversion of a text string, you can specify that the latest version be used. When you create a Unicode mapping structure (page 3-24), you can assign the use-latest-mapping constant to the mapping version field and the Unicode Converter will use the latest Unicode mapping table for the conversion function to which you pass the Unicode mapping structure. The following enumeration defines the use-latest-mapping constant:

```
enum {
    kUnicodeUseLatestMapping   = -1
};
```

Fallback Handler Function

To convert a text string, you specify a conversion information reference when you call a conversion function. When the Unicode Converter encounters a source text element for which there is no target encoding equivalent, it may use loose mappings and fallback characters to perform the conversion. A fallback mapping is a sequence of one or more characters in the target encoding that provide a semblance for another text element. A fallback character attempts to provide character identity for the source text element; the fallback character or character sequence is not exactly equivalent to the source characters but it preserves some of the information of the original. The fallback character or character sequence used depends on the target encoding; it is an entry in the mapping table for the encoding. If fallback mappings are used and the specified fallback handler fails—or both handlers fail when both the application-supplied one and the system default one are specified—then the converter uses the default fallback sequence to represent the source character in the converted string,

A fallback handler is a function that the Unicode Converter uses to perform fallback mapping. The Unicode Converter supplies a default fallback handler. You can provide your own fallback handler and use it alone, or you can use both. You can specify the order in which both handlers are called if one fails. Depending on how control flags governing this process are set, the converter may use its own fallback handler for this purpose or one supplied by your application. You supply your own fallback handler as a function that adheres to a function prototype defined by the Unicode Converter.

To assign your fallback handler to a conversion information reference from within your application's main task only, you use the `SetFallbackUnicodeToText` (page 3-83) or `SetFallbackUnicodeToTextRun` (page 3-76) function. In this case, you pass a universal procedure pointer (`UniversalProcPtr`) as the function's `fallback` parameter.

You supply your own fallback handler as a function that adheres to the following prototype defined by the Unicode Converter. This function is defined by the Unicode Converter as follows:

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

To assign your fallback handler to a conversion information reference from any task other than the main task of your application, you use the `SetFallbackUnicodeToTextPreemptive` (page 3-78) or `SetFallbackUnicodeToTextRunPreemptive` (page 3-81) function. In this case, you pass a pointer to your application-defined function as the function's `fallback` parameter. The prototype for this function is defined by the Unicode Converter as follows:

```
typedef pascal OSStatus (*UnicodeToTextFallbackPreemptiveProcPtr)(
                            UniChar         *srcUniStr,
                            ByteCount       srcUniStrLen,
                            ByteCount       *srcConvLen,
                            TextPtr         destStr,
                            ByteCount       destStrLen,
                            ByteCount       *destConvLen,
                            LogicalAddress  contextPtr,
                            ConstUnicodeMappingPtr unicodeMappingPtr);
```

For information about creating a fallback handler function, see the description of the `MyUnicodeToTextFallbackProc` function (page 3-90).

Unicode Converter Functions

You use the Unicode Converter functions to convert text to or from Unicode. The Unicode Converter consists of two main symmetrical parts, one of which you use to convert text to Unicode from any other encoding and the other of which you use to convert text from Unicode to any other encoding. Each of these parts contains its own set of functions. Using the Unicode Converter with Unicode as an intermediary encoding, you can also convert text from any source encoding to any target encoding.

Converting to Unicode

You can convert text in any encoding to Unicode, but to do so you must first create and obtain a reference to a private data structure containing the mapping and state information the Unicode Converter uses to perform the conversion. You use the `CreateTextToUnicodeInfo` function to provide information for a conversion information data structure and obtain a reference to it. You then pass the reference to the `ConvertFromTextToUnicode` function to perform the conversion. When your application is finished using the reference, you should dispose of it by calling the `DisposeTextToUnicodeInfo` function.

You can also use the Unicode Converter functions to convert text from one encoding to another using Unicode as an intermediary encoding; this is a two-part process. You use these functions to convert the text to Unicode, then you use the functions described in "Converting From Unicode" (page 3-38) to convert the text from Unicode to the final target encoding.

CreateTextToUnicodeInfo

Creates a conversion information data structure required for converting a single stream of text in any encoding to Unicode, and returns a reference to the conversion information structure.

```
pascal OSStatus CreateTextToUnicodeInfo (
                              ConstUnicodeMappingPtr unicodeMapping,
                              TextToUnicodeInfo *textToUnicodeInfo);
```

`unicodeMapping`
: A pointer to a Unicode mapping structure (page 3-24). Your application provides this data structure to identify the mapping to be performed.

`textToUnicodeInfo`
: A pointer to a conversion information data structure (page 3-3) for converting text to Unicode. On output, the parameter returns a conversion information reference to the private data structure that holds mapping table information you supply as the `UnicodeMapping` parameter and state information related to the

conversion. The information contained in the conversion information reference is required for conversion of a text stream in any encoding to Unicode.

function result

A result code. If the function returns a result code other than `noErr`, then the reference returned in the `textToUnicodeInfo` parameter refers is invalid. If one of the mapping tables specified by the Unicode mapping structure you supply or one of the resources associated with it was not found, the function returns a `unicodeNoTableErr` result code. If one of the table resources you specified has a checksum error, the function returns a `unicodeChecksumErr` result code.

DISCUSSION

For each stream of text in a single encoding that you want to convert to Unicode, your application must first call the `CreateTextToUnicodeInfo` function to create a conversion information data structure and obtain a reference to it. When you call `CreateTextToUnicodeInfo`, it locates and loads the mapping table resources, based on the mapping table information you provide, that are required for the conversion.

You can use the same conversion information reference to convert multiple text segments of a single text stream. A conversion information reference persists until you dispose of it. After you finish converting a text stream using a conversion information reference, you should release the memory allocated for the reference by calling the `DisposeTextToUnicodeInfo` function (page 3-36).

You should use the same conversion information reference only to convert segments of text belonging to the text stream for which you created the reference.You should create a new conversion information reference to convert another stream of text even if you intend to use the same mapping information stored in an existing conversion information reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to the single text stream for which it is used.

For example, to convert a document in a single encoding to Unicode, your application can use a single conversion information reference. Each time you call the `ConvertFromTextToUnicode` function to convert a segment of text belonging to the text stream, you pass the function the same reference, repeating the process until the entire text stream is converted. If you use the same conversion information reference to convert multiple segments of the same text stream, you should set the Unicode-keep-information control flag (page 3-7) when you call the conversion function. This is because how the conversion is performed might depend on the next character. The Unicode Converter might need to refer to the next character in the following text segment, for example, to determine the text direction for Hebrew or Arabic text.

You pass a reference returned from `CreateTextToUnicodeInfo` to `ConvertFromTextToUnicode` (page 3-32) or `ConvertPStringToUnicode` (page 3-62) to identify the information to be used for the conversion. These two functions modify the contents of the conversion information reference.

You pass a reference returned from `CreateTextToUnicodeInfo` to `TruncateForTextToUnicode` (page 3-58) to identify the information to be used to truncate the string. This function does not modify the contents of the conversion information reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

If you are converting the text stream to Unicode as an intermediary encoding, and then from Unicode to the final target encoding, you must use `CreateUnicodeToTextInfo` (page 3-38) to create a conversion information reference for the second part of the process.

For general information about Unicode and the Unicode Converter, see “Unicode Converter Constants and Data Types” (page 3-3).

See the chapter “Introduction to Text Handling and Internationalization on Mac OS 8” in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

ConvertFromTextToUnicode

Converts a string from any encoding to Unicode.

```
pascal OSStatus ConvertFromTextToUnicode (
                    TextToUnicodeInfo iTextToUnicodeInfo,
                    ByteCount iSourceLen,
                    ConstLogicalAddress iSourceStr,
                    OptionsBits iControlFlags,
                    ItemCount iOffsetCount,
                    ByteOffset iOffsetArray[],
                    ItemCount *oOffsetCount,
                    ByteOffset oOffsetArray[],
                    ByteCount iBufLen,
                    ByteCount *oSourceRead,
                    ByteCount *oUnicodeLen,
                    UniCharArrayPtr oUnicodeStr);
```

`iTextToUnicodeInfo`
A conversion information reference of type `TextToUnicodeInfo` containing mapping and state information used for the conversion. Your application obtains a reference when it uses `CreateTextToUnicodeInfo` (page 3-29).

`iSourceLen` The length in bytes of the source string to be converted.

`iSourceStr` The logical address of the source string to be converted.

`iControlFlags` Conversion control flags. Bitmasks control flags that apply to this function are `kUnicodeUseFallbacksMask` and `kUnicodeKeepInfoMask`. For a description of the conversion control flags, see “Conversion Control Flags” (page 3-7).

`iOffsetCount` The number of offsets in the `iOffsetArray` parameter. Your application supplies this value. The number of entries in `iOffsetArray` must be fewer than the number of bytes specified in `iSourceLen`. If you don’t want offsets returned to you, specify `0` (zero) for this parameter.

`iOffsetArray` An array of type `ByteOffset`. On input, you specify the array that contains an ordered list of significant byte offsets pertaining to the source string. These offsets may identify font or style changes, for example, in the source string. All array

	entries must be less than the length in bytes specified by the `iSourceLen` parameter. If you don't want offsets returned to your application, specify `NULL` for this parameter and `0` (zero) for `iOffsetCount`.
`oOffsetCount`	A pointer to an `ItemCount`. On output, this value contains the number of offsets that were mapped in the output stream.
`oOffsetArray`	An array of type `ByteOffset`.On output, this array contains the corresponding new offsets for the Unicode string produced by the converter.
`iBufLen`	The length in bytes of the output buffer pointed to by the `oUnicodeStr` parameter. Your application supplies this buffer to hold the returned converted string. The `oUnicodeLen` parameter may return a byte count that is less than this value if the converted byte string is smaller than the buffer size you allocated.
`oSourceRead`	A pointer to a value of type `ByteCount`. On output, this value contains the number of bytes of the source string that were converted. If the function returns a `unicodePartConvertErr`, this parameter returns the number of bytes that were converted before the error occurred.
`oUnicodeLen`	A pointer to a value of type `ByteCount`. On output, this value contains the length in bytes of the converted stream.
`oUnicodeStr`	A pointer to an array used to hold a Unicode string. On input, this value points to the beginning of the array for the converted string. On output, this buffer holds the converted Unicode string. (For guidelines on estimating the size of the buffer needed, see the following discussion.) For a description of the `UniCharArrayPtr` data type, see "Unicode Character and String Pointer Data Types" (page 3-23).
function result	A result code. If the `ConvertFromTextToUnicode` function returns a `noErr` result code, it has completely converted the source string to the Unicode variant you specified without using fallback characters. If the function returns the `paramErr`, the string was not converted. If the `ConvertFromTextToUnicode` returns the `unicodeFallbacksErr` result code, the function has

completely converted the input string to the specified target using one or more fallbacks because you set the `kUnicodeUseFallbacksBit` control flag.

If the function returns the `unicodePartConvertErr` result code, the function was not able to convert the entire source string to the specified Unicode variant and it did not attempt to use fallbacks because you did not set the `kUnicodeUseFallbacksBit` control flag. When this error occurs, the function may have been able to convert part of the string, but not the entire string because, for example, a subtable required for conversion of some of the text was missing from the mapping table used for the conversion.

If the function returns a `unicodeBufErr`, the output buffer specified by `iBufLen` is too short to hold the converted string. For result codes of `unicodePartConvertErr` and `unicodeBufErr`, the `oSourceRead` parameter contains the number of bytes converted before the error occurred. To convert the remaining part of the string, you can call this function again, passing the function the rest of the string.

DISCUSSION

The `ConvertFromTextToUnicode` function converts a text string in any encoding to Unicode. You specify the source string's encoding in the Unicode mapping data structure that you pass to `CreateTextToUnicodeInfo` (page 3-29) to obtain a conversion information reference for the conversion. You pass the reference returned by `CreateTextToUnicodeInfo` to `ConvertFromTextToUnicode` as the `iTextToUnicodeInfo` parameter. The source encoding can be any encoding, including a Unicode variant or another text encoding.

In addition to converting a text string in any encoding to Unicode, the `ConvertFromTextToUnicode` function can map style or font information from the source text string to the returned converted string. The converter reads the application-supplied offsets, which apply to the source string, and returns the corresponding new offsets in the converted string. If you do not want the offsets at which font or style information occurs mapped to the resulting string, you should pass `NULL` for `iOffsetArray` and 0 (zero) for `iOffsetCount`.

Your application must allocate a buffer to hold the resulting converted string and pass a pointer to the buffer in the `oUnicodeStr` parameter. To determine the size of the output buffer to allocate, you should consider the size of the source string, its encoding type, and its content in relation to the resulting Unicode string.

For example, for 1-byte encodings, such as Mac OS Roman, the Unicode string will be at least double the size of the source string. For most 2-byte encodings, such as Shift-JIS, the Unicode string will be less than double the size of the source string. In some cases, such as Mac OS Korean, it is not unusual for the Unicode string to be three times as large as the source string.

However, for Mac OS Arabic and Mac OS Hebrew, some 1-byte characters, such as punctuation characters and digits, are converted to three Unicode characters. Depending on how many of these the source string contains, the resulting Unicode string could be more than three times the size of the original.

To convert a single text stream, your application can use a conversion information reference. You pass the same reference to the `ConvertFromTextToUnicode` function each time you call the function to convert a segment of text belonging to the single text stream, repeating the process until the entire stream of text is converted.

You should use the same conversion information reference only to convert segments of text belonging to the text stream for which you created the reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to the single text stream for which it is used. When you finish converting all of the text reliant on the conversion information reference, release the memory allocated for the reference by calling the `DisposeUnicodeToTextInfo` function (page 3-45).

If you use the same conversion information reference to convert multiple segments of the same text stream, you should set the Unicode-keep-information control flag (page 3-7) when you call the conversion function. This is because how the conversion is performed might depend on the next character in the text stream. The Unicode Converter might need to refer to the next character in the following text segment, for example, to determine the text direction for Hebrew or Arabic text.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SPECIAL CONSIDERATIONS

This function modifies the contents of the conversion information data structure specified by the reference you passed in the `iTextToUnicodeInfo` parameter.

SEE ALSO

If you are converting the text stream to Unicode as an intermediary encoding, and then from Unicode to the final target encoding, you use `ConvertFromUnicodeToText` (page 3-41) for the second part of the process.

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

DisposeTextToUnicodeInfo

Releases the memory allocated for the specified conversion information reference.

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

`textToUnicodeInfo`
: A pointer to a conversion information data structure (page 3-3) for converting text to Unicode. On input, you specify a reference that points to the conversion information to be disposed of, which your application created using `CreateTextToUnicodeInfo` (page 3-29).

function result
: A result code. The function returns a `noErr` result code if it successfully disposes of the conversion information reference. If your application specifies an invalid conversion information reference, such as `NULL`, the function returns a `paramErr` result code.

DISCUSSION

The `DisposeTextToUnicodeInfo` function disposes of the conversion information reference and releases the memory allocated for it. Your application should not attempt to dispose of the same data structure more than once.

You must use this function only to release the memory for a reference of type `OpaqueTextToUnicodeInfo` that your application created through the `CreateTextToUnicodeInfo` function (page 3-29). You must not use it for any other type of conversion information reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see “Unicode Converter Constants and Data Types” (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Converting From Unicode

You can convert text from Unicode to another encoding, but to do so you must first obtain a reference to a private data structure containing the mapping and state information the Unicode Converter uses to perform the conversion. You use the `CreateUnicodeToTextInfo` function to obtain this reference. You then pass the reference to the `ConvertFromUnicodeToText` function to perform the conversion. When your application is finished using the reference, you must dispose of it and the memory allocated for it by calling the `DisposeUnicodeToTextInfo` function.

Converting text from one encoding to another using Unicode as an intermediary encoding is a two-part process. You use the functions described in "Converting to Unicode" (page 3-29) to convert the text to Unicode, then you use these functions to convert the text from Unicode to the final, target encoding.

CreateUnicodeToTextInfo

Creates a data structure containing the information required for converting strings from Unicode to another encoding, and returns a reference to the private structure.

```
pascal OSStatus CreateUnicodeToTextInfo (
                        ConstUnicodeMappingPtr unicodeMapping,
                        UnicodeToTextInfo *unicodeToTextInfo);
```

`unicodeMapping`

A Unicode mapping structure (page 3-24). Your application provides this data structure to identify the mapping to be performed.

`unicodeToTextInfo`
: A pointer to a conversion information data structure (page 3-3) for converting Unicode strings to text. On output, the parameter returns a conversion information reference to the private data structure that holds the mapping table information you supply as the `UnicodeMapping` parameter and the state information related to the conversion. The `CreateUnicodeToTextInfo` function creates the private data structure and returns the reference to it as this parameter if the function completes successfully. The information contained in the conversion information reference is required for the conversion of a Unicode string to any other encoding, including a Unicode variant.

function result
: A result code. If the function returns a result code other than `noErr`, then the reference returned in the `unicodeToTextInfo` parameter refers to an invalid conversion information data structure. If one of the mapping tables specified by the Unicode mapping structure you supply or one of the resources associated with it was not found, the function returns a `unicodeNoTableErr` result code. If one of the table resources you specified has a checksum error, the function returns a `unicodeChecksumErr` result code.

DISCUSSION

For each Unicode string that you want to convert to another encoding, your application must call the `CreateUnicodeToTextInfo` function to create a conversion information data structure and obtain a reference to it. The Unicode Converter uses the mapping table information you provide when creating the reference.

You can use the same conversion information reference to convert multiple Unicode strings belonging to the same text stream to the encoding specified in the mapping table. You should use the same conversion information reference only to convert segments of text belonging to the single text stream for which you created the reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to that particular reference and the single text stream for which it is used. When you are finished converting all of the text reliant on the conversion information reference, you must release the memory allocated for the reference by calling the `DisposeUnicodeToTextInfo` function (page 3-45).

If you use the same conversion information reference to convert multiple segments of the same text stream, you should set the Unicode-keep-information control flag (page 3-7) when you call the conversion function. This is because how the conversion is performed might depend on the next character. The Unicode Converter might need to refer to the next character in the following text segment, for example, to determine the text direction for Hebrew or Arabic text.

The `CreateUnicodeToTextInfo` function locates and loads the mapping table resources required for the conversion.

You pass the reference returned from the `CreateUnicodeToTextInfo` function to the `ConvertFromUnicodeToText` function (page 3-41) or the `ConvertUnicodeToPString` function (page 3-64), to identify the information to be used for the conversion. These two functions modify the contents of the conversion information reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

ConvertFromUnicodeToText

Converts a string from Unicode to the specified encoding.

```
pascal OSStatus ConvertFromUnicodeToText (
                     UnicodeToTextInfo iUnicodeToTextInfo,
                     ByteCount iUnicodeLen,
                     ConstUniCharArrayPtr iUnicodeStr,
                     OptionBits iControlFlags,
                     ItemCount iOffsetCount,
                     ByteOffset iOffsetArray[],
                     ItemCount *oOffsetCount,
                     ByteOffset oOffsetArray[],
                     ByteCount iBufLen,
                     ByteCount *oInputRead,
                     ByteCount *oOutputLen,
                     LogicalAddress oOutputStr);
```

`iUnicodeToTextInfo`
: A conversion information reference of type `UnicodeToTextInfo` for converting text from Unicode. You use the `CreateUnicodeToTextInfo` function (page 3-38) to obtain a reference to specify for this parameter.

`iUnicodeLen` The length in bytes of the Unicode string to be converted.

`iUnicodeStr` The logical address of the Unicode string to be converted.

`iControlFlags` Conversion control flags. You can use these bitmasks to set the control flags that apply to this function:

```
kUnicodeUseFallbacksMask
kUnicodeKeepInfoMask
kUnicodeVerticalFormMask
kUnicodeLooseMappingsMask
kUnicodeStringUnterminatedMask
```

For a description of these control flags, see “Conversion Control Flags” (page 3-7).

To set the directionality field, which is also a valid control flag for this parameter, you use the `kUnicodeDirectionalityBits` constant, not the mask, because you must shift the bits.

`iOffsetCount`	The number of offsets contained in the array provided by the `iOffsetArray` parameter. Your application supplies this value.The number of entries in `iOffsetArray` must be fewer than the number of bytes specified in `iUnicodeLen`. If you don't want offsets returned to you, specify `0` (zero) for this parameter.
`iOffsetArray`	An array of type `ByteOffset`. On input, you specify the array that gives an ordered list of significant byte offsets pertaining to the Unicode source string to be converted. These offsets may identify font or style changes, for example, in the source string. If you don't want offsets returned to your application, specify `NULL` for this parameter and `0` (zero) for `iOffsetCount`.
`oOffsetCount`	A pointer to an `ItemCount`.
`oOffsetArray`	An array of type `ByteOffset`.On output, this array contains the corresponding new offsets for the converted string in the new encoding.
`iBufLen`	The length in bytes of the output buffer pointed to by the `oOutputStr` parameter. Your application supplies this buffer to hold the returned converted string. The `oOutputLen` parameter may return a byte count that is less than this value if the converted byte string is smaller than the buffer size you allocated.
`oInputRead`	A pointer to a value of type `ByteCount`. On output, this value gives the number of bytes of the Unicode string that were converted. If the function returns a `unicodePartConvertErr`, this parameter returns the number of bytes that were converted before the error occurred.
`oOutputLen`	A pointer to a value of type `ByteCount`. On output, this value give the length in bytes of the converted text stream.
`oOutputStr`	A value of type `LogicalAddress`. On input, this value points to a buffer for the converted string. On output, the buffer holds the converted text string. (For guidelines on estimating the size of the buffer needed, see the following discussion.)
function result	A result code. The function returns a `noErr` result code if it has completely converted the Unicode string to the target encoding without using fallback character sequences. If the function returns the `paramErr`, the function does not convert the string.

If the `ConvertFromUnicodeToText` returns the `unicodeFallbacksErr` result code, the function has completely converted the input string to the specified target using one or more fallbacks because you set the Unicode-use-fallbacks control flag.

If the function returns a `unicodeTableFormatErr` because the mapping table specified an unknown table format or index format or a `unicodeVariantErr` because the specified target encoding was not found in the mapping table, the function does not convert the string.

If the function returns a `unicodeCharErr` because the source text contained an invalid Unicode character, a `unicodeElementErr` because the source string contained a text element too long to process, a `unicodeNotFoundErr` because a Unicode text element in the source string is not in the mapping table, or a `unicodeBufErr` because the output buffer specified by `iBufLen` is too short to hold the converted string, then the function did not completely convert the string. If it converted part of the string, the buffer pointed to by the `oOutputStr` parameter contains the converted portion. To convert the remaining part of the string, you can call this function again, passing the function the rest of the string.

DISCUSSION

The `ConvertFromUnicodeToText` function converts a Unicode text string to the target encoding you specify in the Unicode mapping data structure that you pass to `CreateUnicodeToTextInfo` (page 3-38) when you call it to obtain a conversion information reference for the conversion process. You pass the reference returned by `CreateUnicodeToTextInfo` to `ConvertFromUnicodeToText` as the `iUnicodeToTextInfo` parameter. The target encoding can be any encoding, including a Unicode variant or another text encoding.

In addition to converting the Unicode string, `ConvertFromUnicodeToText` can map style or font information from the source text string to the returned converted string. The converter reads the application-supplied offsets and returns the corresponding new offsets in the converted string. If you do not want font or style information offsets mapped to the resulting string, you should pass `NULL` for `iOffsetArray` and `0` (zero) for `iOffsetCount`.

Your application must allocate a buffer to hold the resulting converted string and pass a pointer to the buffer in the `oOutputStr` parameter. To determine the size of the output buffer to allocate, you should consider the size and content of the Unicode source string in relation to the type of encoding to which it will be converted. For example, for many encodings, such as Mac OS Roman and Shift-JIS, the size of the returned string will be between half the size and the same size as the source Unicode string. However, for some encodings that are not Mac OS ones, such as EUC-JP which has some 3-byte characters for Kanji, the returned string could be larger than the source Unicode string. For Mac OS Arabic and Mac OS Hebrew, the result will usually be less than half the size of the Unicode string.

To convert a single Unicode text stream, your application can use the same conversion information reference. You pass this reference to the `ConvertFromTextToUnicode` function each time you call the function to convert a segment of text belonging to the text stream, repeating the process until the entire text stream is converted.

You should use the same conversion information reference only to convert segments of text belonging to the text stream for which you created the reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to the single text stream for which it is used. When you are finished converting all of the text reliant on the conversion information reference, release the memory allocated for the reference by calling `DisposeUnicodeToTextInfo` (page 3-45).

If you use the same conversion information reference to convert multiple segments of the same text stream, you should set the Unicode-keep-information control flag (page 3-7) when you call the conversion function. This is because how the conversion is performed might depend on the next character in the text stream. The Unicode Converter might need to refer to the next character in the following text segment, for example, to determine the text direction for Hebrew or Arabic text.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SPECIAL CONSIDERATIONS

This function modifies the contents of the conversion information data structure pointed to by the reference you passed as the `iUnicodeToTextInfo` parameter.

SEE ALSO

If you are converting the text stream to Unicode as an intermediary encoding, and then from Unicode to the final target encoding, you use `ConvertFromTextToUnicode` (page 3-32) for the first part of the process and `ConvertFromUnicodeToText` for the second part of the process.

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

DisposeUnicodeToTextInfo

Releases the memory allocated for the specified conversion information reference.

```
pascal OSStatus DisposeUnicodeToTextInfo (
                    UnicodeToTextInfo *unicodeToTextInfo);
```

`unicodeToTextInfo`
: A pointer to a conversion information data structure (page 3-5) for converting Unicode text to another encoding. On input, you specify a reference that points to the conversion information to be disposed of, which your application created using `CreateUnicodeToTextInfo` (page 3-38).

function result
: A result code. The function returns `noErr` if it disposes of the conversion information reference successfully. If your application specifies an invalid conversion information reference, such as `NULL`, the function returns a `paramErr` result code.

DISCUSSION

The `DisposeUnicodeToTextInfo` function disposes of the conversion information reference and releases the memory allocated for it. Your application should not attempt to dispose of the same conversion information reference more than once.

You must use this function only to release the memory for a reference of type `OpaqueUnicodeToTextInfo` that your application created through the `CreateUnicodeToTextInfo` function (page 3-38). You must not use it for any other type of conversion information reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Converting From Unicode to Multiple Encodings

It may not be possible to convert a Unicode string to a single target encoding. To handle these cases, the Unicode Converter provides the `ConvertFromUnicodeToTextRun` function that allows you to specify a number of possible target encodings and how the function should use these target encodings, if necessary, when converting the Unicode string. Before you use the `ConvertFromUnicodeToTextRun` function, you must first create and obtain a reference to a private data structure containing the mapping and state information the Unicode Converter uses to perform the conversion. You use the `CreateUnicodeToTextRunInfo` function to provide information for the private data structure and obtain a reference to it. You then pass the reference to the `ConvertFromUnicodeToTextRun` function to perform the conversion. When your application is finished using the reference, you must dispose of it and the memory allocated for it by calling the `DisposeUnicodeToTextRunInfo` function.

CreateUnicodeToTextRunInfo

Creates a data structure containing the information required for converting a Unicode text string to any one or more encodings, and returns a reference to the structure.

```
pascal OSStatus CreateUnicodeToTextRunInfo (
                    ItemCount numberOfMappings,
                    ConstUnicodeMappingPtr unicodeMapping,
                    UnicodeToTextRunInfo *unicodeToTextInfo);
```

`numberOfMappings`
: The number of mappings specified by your application for converting from Unicode to any other encoding types, including other forms of Unicode.

`unicodeMapping`
: An array of Unicode mapping structures (page 3-24). Your application provides this data structure to identify the mappings to be used for the conversion. The order in which the mappings are specified in this data structure affects the conversion process. When the `ConvertFromUnicodeToTextRun` function (page 3-50) requires more than one target encoding to convert the text string, the Unicode Converter uses the order of mappings as they appear in this array to determine the priority of target encodings.

`unicodeToTextInfo`
: A pointer to a conversion information data structure (page 3-6) for converting Unicode text strings to any one or more encodings. On output, a conversion information reference to the private data structure that holds the mapping table information you supply as the `unicodeMapping` parameter and the state information related to the conversion. The `CreateUnicodeToTextRunInfo` function creates the private data structure and returns a reference to the structure if the function completes successfully.

function result
: A result code. If the converter could not find one of the mapping tables specified by the Unicode mapping structure you supply or one of the resources associated with it, the function returns a `unicodeNoTableErr` result code. If one of the table resources you specified has a checksum error, the function returns a `unicodeChecksumErr` result code. If the function returns a result code other than `noErr`, then the reference returned in the `unicodeToTextInfo` parameter is invalid.

DISCUSSION

For each Unicode string to be converted to one or more encodings belonging to a set of mappings, your application must call the `CreateUnicodeToTextRunInfo` function to create a conversion information data structure and obtain a reference to it. You pass the reference returned from the `CreateTextToTextRunInfo` function to the `ConvertFromUnicodeToTextRun` function (page 3-50) to provide the mapping and state information to be used for the conversion. The `CreateUnicodeToTextRunInfo` function locates and loads the mapping table resources required for the conversion.

You can use the same conversion information reference to convert multiple Unicode strings belonging to the same text stream to the encodings specified in the mapping table. You should use the same conversion information reference only to convert the text stream for which you created the reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to the single text stream for which it is used. When you are finished converting all of the text reliant on the conversion information reference, release the memory allocated for the reference by calling `DisposeUnicodeToTextRunInfo` (page 3-56).

If you use the same conversion information reference to convert multiple Unicode strings, you should set the Unicode-keep-information control flag when you call the conversion function. This is because how the conversion is performed might depend on the next character. The Unicode Converter might need to refer to the next character in the following text segment, for example, to determine the text direction for Hebrew or Arabic text.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see “Unicode Converter Constants and Data Types” (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

ConvertFromUnicodeToTextRun

Converts a string from Unicode to one or more encodings.

```
pascal OSStatus ConvertFromUnicodeToTextRun (
                    UnicodeToTextRunInfo iUnicodeToTextInfo,
                    ByteCount iUnicodeLen,
                    ConstUniCharArrayPtr iUnicodeStr,
                    OptionBits iControlFlags,
                    ItemCount iOffsetCount,
                    ByteOffset iOffsetArray[],
                    ItemCount *oOffsetCount,
                    ByteOffset oOffsetArray[],
                    ByteCount iBufLen,
                    ByteCount *oInputRead,
                    ByteCount *oOutputLen,
                    LogicalAddress oOutputStr,
                    ItemCount iEncodingRunBufLen,
                    ItemCount *oEncodingRunOutLen,
                    TextEncodingRun oEncodingRuns[]);
```

`iUnicodeToTextInfo`
A conversion information reference for converting Unicode text to one or more encodings. You use the `CreateUnicodeToTextRunInfo` function (page 3-47) to obtain a reference to specify for this parameter.

`iUnicodeLen` The length in bytes of the Unicode string to be converted.

`iUnicodeStr` A pointer to the Unicode string to be converted.

`iControlFlags` Conversion control flags. The following constants define the masks for control flags valid for this parameter. You can use these masks to set the `iControlFlags` parameter:
`kUnicodeUseFallbacksBit`
`kUnicodeKeepInfoBit`

`kUnicodeDirectionalityMask`
`kUnicodeVerticalFormBit`
`kUnicodeLooseMappingsBit`
`kUnicodeStringUnterminatedBit`
`kUnicodeTextRunBit`
`kUnicodeKeepSameEncodingBit`

For a description of these control flags, see "Conversion Control Flags" (page 3-7).

To set the directionality field, which is also a valid control flag for this parameter, you use the `kUnicodeDirectionalityBits` constant, not the mask, because you must shift the bits.

If the Unicode-text-run control flag is clear, `ConvertFromUnicodeToTextRun` attempts to convert the Unicode text to the single encoding from the list of encodings in the Unicode-to-text-run conversion information reference that produces the best result, that is, that provides for the greatest amount of source text conversion. If the complete source text can be converted into more than one of the encodings specified in the Unicode mapping structures array, then the converter chooses among them based on their order in the array. If this flag is clear, the `oEncodingRuns` parameter will always point to a value equal to 1. The Unicode-use-fallbacks control flag is not applicable if the Unicode-text-run control flag is clear.

If you set the Unicode-use-fallbacks control flag, the converter will use the default fallback characters for the current encoding. If the converter cannot handle a character using the current encoding, even using fallbacks, the converter attempts to convert the character using the other encodings, beginning with the first encoding specified in the list and skipping the encoding where it failed.

If you set the `kUnicodeTextRunBit` control flag, the converter attempts to convert the complete Unicode text string into the first encoding specified in the Unicode mapping structures array you passed to `CreateUnicodeToTextRunInfo` (page 3-47) to create the conversion information reference used for this conversion. If it cannot do this, the converter then attempts to convert the first text element that failed to the remaining encodings, in their specified order in the array. How the converter does this depends on the setting of the Unicode-keep-same-encoding control flag:

- If the Unicode-keep-same-encoding control flag 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 the Unicode-keep-same-encoding control flag is set, the converter continues with the new target encoding until it encounters a text element that cannot be converted using the new encoding. When the converter cannot convert a text element using any of the encodings in the list and the Unicode-keep-same-encoding control flag is set, the converter uses the fallbacks default characters for the current encoding.

For a description of the complete set of conversion control flags, see "Conversion Control Flags" (page 3-7).

`iOffsetCount`	The number of offsets in the array pointed to by the `iOffsetArray` parameter. Your application supplies this value. The number of entries in `iOffsetArray` must be fewer than half the number of bytes specified in `iUnicodeLen`. If you don't want offsets returned to you, specify `0` (zero) for this parameter.
`iOffsetArray`	An array of type `ByteOffset`. On input, you specify the array that contains an ordered list of significant byte offsets pertaining to the source Unicode string. These offsets may identify font or style changes, for example, in the Unicode string. If you don't want offsets returned to your application, specify `NULL` for this parameter and `0` (zero) for `iOffsetCount`.
`oOffsetCount`	A pointer to an `ItemCount`.
`oOffsetArray`	An array of type `ByteOffset`.On output, this array contains the corresponding new offsets for the resulting converted string.
`iBufLen`	The length in bytes of the output buffer pointed to by the `oOutputStr` parameter. Your application supplies this buffer to hold the returned converted string. The `oOutputLen` parameter may return a byte count that is less than this value if the converted byte string is smaller than the buffer size you allocated.

`oInputRead`	A pointer to a value of type `ByteCount`. On output, this value contains the number of bytes of the Unicode source string that were converted. If the function returns a result code other than `noErr`, then this parameter returns the number of bytes that were converted before the error occurred.
`oOutputLen`	A pointer to a value of type `ByteCount`. On output, this value contains the length in bytes of the converted string.
`oOutputStr`	A value of type `LogicalAddress`. On input, this value points to the start of the buffer for the converted string. On output, this buffer contains the converted string in one or more encodings. When an error occurs, the `ConvertFromUnicodeToTextRun` function returns the converted string up to the character that caused the error. (For guidelines on estimating the size of the buffer needed, see the following discussion.)
`iEncodingRunBufLen`	The number of text encoding run elements you allocated for the encoding run array pointed to by the `oEncodingRuns` parameter.
`oEncodingRunOutLen`	A pointer to a value of type `ItemCount`. On output, this value contains the number of valid encoding runs returned in the `oEncodingRuns` parameter.
`oEncodingRuns`	An array of elements of type `TextEncodingRun`. On input, this refers to the array of text encoding run data structures. Your application should allocate an array with the number of elements you specify in the `iEncodingRunBufLen` parameter. On output, this array contains the encoding runs for the converted text string. Each entry in the encoding run array specifies the starting offset in the converted text string and the associated encoding specification.
function result	A result code. The function returns a `noErr` result code if it has completely converted the Unicode string to the target encoding without using fallback character sequences. If the function returns the `paramErr` because one or more of the input parameter values is invalid, the function does not convert the string.

If `ConvertFromUnicodeToTextRun` returns the `unicodeFallbacksErr` result code, the function has completely converted the input string to the specified target using one or more fallbacks because you set Unicode-use-fallbacks control flag.

If the function returns a `unicodeVariantErr` because the specified target encoding was not found in the mapping table, the function does not convert the string.

If the function returns a `unicodeTableFormatErr` because the mapping table specified an unknown table format or index format, the function might have partially converted the string.

If the function returns a `unicodeCharErr` because the source text contained an invalid Unicode character, a `unicodeElementErr` because the source string contained a text element too long to process, a `unicodeNotFoundErr` because a Unicode text element in the source string is not in the mapping table, or a `unicodeBufErr` because the output buffer specified by `iBufLen` is too short to hold the converted string, then the function did not completely convert the string. If it converted part of the string, the array pointed to by the `oOutputStr` parameter contains the converted portion. To convert the remaining part of the string, you can call this function again, passing the function the rest of the string.

DISCUSSION

To use the `ConvertFromUnicodeToTextRun` function, you must first set up an array of Unicode mapping structures (page 3-24) containing in order of precedence the mapping information for the conversion. To create a conversion information reference, you call the `CreateUnicodeToTextRunInfo` function passing it the Unicode mapping array pointer. You pass the returned reference as the `iUnicodeToTextInfo` parameter when you call the `ConvertFromUnicodeToTextRun` function.

Two of the control flags that you can set for the `iControlFlags` parameter allow you to control how the Unicode Converter uses the multiple encodings in converting the text string. These flags are explained in the description of the `iControlFlags` parameter. Here is a summary of how to use these two control flags:

- If you want to keep the converted text in a single encoding run, leave the Unicode-text-run control flag clear.
- If you want to keep as much as possible of the converted text in one encoding, set the Unicode-multiple-run control flag and leave clear the Unicode-keep-same-encoding control flag.
- If you want to minimize the number of encoding runs or to minimize the changes of target encoding, set both the Unicode-text-run and Unicode-keep-same-encoding control flags.

The `ConvertFromUnicodeToTextRun` function returns the converted string in the array pointed to by the `oOutputStr` parameter. Beginning with the first text element in the `oOutputStr` array, the elements of the array pointed to by the `oEncodingRuns` parameter identify the encodings of the converted string. The number of elements in the `oEncodingRuns` array may not correspond to the number of elements in the `oOutputStr` array. This is because the `oEncodingRuns` array only includes elements for each change of encoding in the converted string.

You can use the same conversion information reference to convert multiple Unicode strings belonging to a same text stream to the encodings specified in the mapping table. You should use the same conversion information reference only to convert the single text stream for which you created the reference. This is because the Unicode Converter stores private state information in a conversion information reference that is relevant only to the text stream for which it is used. When you are finished converting all of the text reliant on the conversion information reference, release the memory allocated for the reference by calling the `DisposeUnicodeToTextRunInfo` function (page 3-47).

If you use the same conversion information reference to convert multiple Unicode strings of a single text stream, you should set the Unicode-keep-info control flag (page 3-7) when you call the conversion function. This is because how the conversion is performed might depend on the next character. The Unicode Converter might need to refer to the next character in the following text segment, for example, to determine the text direction for Hebrew or Arabic text.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SPECIAL CONSIDERATIONS

This function modifies the contents of the conversion information reference specified by the `iUnicodeToTextInfo` parameter.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

DisposeUnicodeToTextRunInfo

Releases the memory allocated for the specified conversion information reference.

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

`unicodeToTextRunInfo`

A pointer to a conversion information data structure (page 3-6) for converting Unicode text to another encoding. On input, you specify a reference that points to the conversion information to be disposed of, which your application created using `CreateUnicodeToTextRunInfo` (page 3-47).

function result

A result code. The function returns a `noErr` result code if it disposes of the conversion information reference successfully. If your application specifies an invalid reference, such as `NULL`, the function returns `paramErr`.

DISCUSSION

The `DisposeUnicodeToTextRunInfo` function disposes of the conversion information reference specified by the `unicodeToTextRunInfo` parameter and releases the memory allocated for it. Your application should not attempt to dispose of the same conversion information reference more than once.

You must use this function only to release the memory for a conversion information reference that your application created through the `CreateUnicodeToTextRunInfo` function (page 3-47). You must not use it for any other type of conversion information reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see “Unicode Converter Constants and Data Types” (page 3-3).

See the chapter “Introduction to Text Handling and Internationalization on Mac OS 8” in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Truncating Strings Before Converting Them

The Unicode Converter provides two functions that you can use to identify where to properly truncate text before converting it: you use the `TruncateForTextToUnicode` function to locate where to truncate text to be converted from another encoding to Unicode and the `TruncateforUnicodeToText` function to locate where to truncate text to be converted from Unicode to another encoding.

TruncateForTextToUnicode

Identifies where your application should truncate a multibyte string to be converted to Unicode so that the string is not broken in the middle of a two-byte character.

```
pascal OSStatus TruncateForTextToUnicode(
                    ConstTextToUnicodeInfo textToUnicodeInfo,
                    ByteCount sourceLen,
                    ConstLogicalAddress sourceStr,
                    ByteCount maxLen,
                    ByteCount *truncatedLen);
```

`textToUnicodeInfo`
The conversion information reference (page 3-3) pertaining to the text string to be truncated. The `TruncateForTextToUnicode` function does not modify the contents of this private data structure.

`sourceLen`	The length in bytes of the multibyte string to be truncated.
`sourceStr`	The logical address of the multibyte string to be truncated.
`maxLen`	The maximum allowable length of the truncated string.
`truncatedLen`	A pointer to a value of type `ByteCount`. On output, this value contains the length of the longest portion of the multibyte string, pointed to by the `sourceStr` parameter, that is less than or equal to the length specified by the `maxLen` parameter. This identifies the byte after which you can truncate the string.

function result

A result code. The function returns a `noErr` result code if it successfully truncates the string. If the function returns `paramErr`, the content of the returned `truncatedLen` parameter is invalid.

DISCUSSION

Your application can use this function to truncate a string properly before you call the `ConvertFromTextToUnicode` function so that the string you pass to `ConvertFromTextToUnicode` is terminated with complete characters. To avoid the possibility of corrupting the contents of the string or breaking a string between the first and second bytes of a two-byte character, it is best to use this function instead of truncating the string yourself. You can call this function repeatedly to properly truncate a text string, each time identifying the new beginning of the string, until the last portion of the text is less than or equal to the maximum allowable length.

Because the `TruncateForTextToUnicode` function does not modify the contents of the conversion information reference, you can call this function safely between calls to the `ConvertFromTextToUnicode` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see “Unicode Converter Constants and Data Types” (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

TruncateForUnicodeToText

Identifies where your application should truncate a Unicode string to be converted to any encoding so that the string is broken in a way that preserves the text element integrity.

```
pascal OSStatus TruncateForUnicodeToText (
                    ConstUnicodeToTextInfo unicodeToTextInfo,
                    ByteCount sourceLen, ConstUniCharArrayPtr sourceStr,
                    OptionBits controlFlags,
                    ByteCount maxLen,
                    ByteCount *truncatedLen);
```

`unicodeToTextInfo`
: A conversion information reference (page 3-5) pertaining to the Unicode string to be truncated. The `TruncateForUnicodeToText` function does not modify the contents of this private data structure.

`sourceLen`
: The length in bytes of the Unicode string to be truncated.

`sourceStr`
: The Unicode string to be truncated.

`controlFlags`
: Truncation control flags. You can use the `kUnicodeTextElementSafeMask` and the `kUnicodeRestartSafeMask` masks (page 3-16) to set the control flags that apply to this function.

If you set the Unicode-text-element-safe control flag, the truncated string will contain complete text elements. You should normally set this bit.

If you set the Unicode-restart-safe control flag, you can safely process the string using the `ConvertFromUnicodeToText` function even if the string is not block delimited.

`maxLen`
: The maximum allowable length of the truncated string.

`truncatedLen` A pointer to a value of type `ByteCount`. On output, this value contains the length of the longest portion of the Unicode source string, pointed to by the `sourceStr` parameter, that satisfies the conditions specified by the control flags and that is less than or equal to the value of the `maxLen` parameter. This returned parameter identifies the byte after which you should truncate the string.

function result

A result code. The function returns a `noErr` result code if it successfully truncates the string. If the function returns `paramErr`, the content of the returned `truncatedLen` parameter is invalid.

DISCUSSION

Your application can use this function to truncate a Unicode string properly before you call the `ConvertFromUnicodeToText` function (page 3-41) to convert the string. `TruncateForUnicodeToText` identifies where to truncate the Unicode string so that your application does not break it in the middle of a text element, such as between a letter and a combining diacritic. To avoid the possibility of corrupting the contents of the string, it is best to use this function instead of truncating the string yourself. Using the `TruncateForUnicodeToText` function to identify where to truncate the string ensures that the string you pass to the `ConvertFromUnicodeToText` function is terminated with complete characters.

You can call this function repeatedly to properly truncate a text segment, each time identifying the new beginning of the string, until the last portion of the text is less than or equal to the maximum allowable length.

Because this function does not modify the contents of the conversion information reference, you can call this function between calls to the `ConvertFromUnicodeToText` function.

If the string to be truncated is not block delimited—for example, if you are truncating text contained in incoming packets before converting it—you should set the `kUnicodeRestartSafeBit` control flag before you call the `TruncateForUnicodeToText` function for each packet's text. This allows the Unicode Converter to resolve the character direction of the text without benefit of the full block-delimited context. For this scenario, you would prefix any truncated portion of the text to the text of the next packet before calling the `TruncateForUnicodeToText` function for that packet.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Converting Unicode From and To Pascal Strings

The Unicode Converter provides functions that limit effort and the overhead involved in converting Pascal strings to and from Unicode. You can use the `ConvertPStringToUnicode` function to convert a Pascal string to a Unicode string and the `ConvertUnicodeToPString` function to convert a Unicode string to Pascal.

ConvertPStringToUnicode

Converts a Pascal string in a Mac OS text encoding to a Unicode string.

```
pascal OSStatus ConvertPStringToUnicode (
                    TextToUnicodeInfo textToUnicodeInfo,
                    ConstStr255Param pascalStr,
                    ByteCount bufLen,
                    ByteCount *unicodeLen,
                    UniCharArrayPtr unicodeStr);
```

`textToUnicodeInfo`	A conversion information reference (page 3-3) pertaining to the Pascal string to be converted. You use the `CreateTextToUnicodeInfo` function (page 3-29) to obtain the reference.
`pascalStr`	The Pascal string to be converted to Unicode.
`bufLen`	The length in bytes of the output buffer pointed to by the `unicodeStr` parameter. Your application supplies this buffer to hold the returned converted string. The `unicodeLen` parameter may return a byte count that is less than this value if the converted string is smaller than the buffer size you allocated.
`unicodeLen`	A pointer to a value of type `ByteCount`. On output, the length in bytes of the converted Unicode string returned in the `unicodeStr` parameter.
`unicodeStr`	A pointer to a Unicode character array (page 3-23). On output, this buffer holds the converted Unicode string.
function result	A result code. If the `ConvertPStringToUnicode` function returns a `noErr` result code, it has completely converted the Pascal string to the Unicode variant you specified without using fallback characters. If the function returns the `paramErr`, the string was not converted. If the `ConvertPStringToUnicode` returns the `unicodeFallbacksErr` result code, the function has completely converted the input string to the specified target using one or more fallbacks. If the function returns a `unicodeBufErr`, the output buffer specified by `bufLen` is too short to hold the converted string. For a result code of `unicodeBufErr`, the `sourceRead` parameter contains the number of bytes converted before the error occurred. To convert the remaining part of the string, you can call this function again, passing the function the rest of the string.

DISCUSSION

The `ConvertPStringToUnicode` function provides an easy and efficient way to convert a short Pascal string to a Unicode string without incurring the overhead associated with the `ConvertFromTextToUnicode` function (page 3-32).

If necessary, this function automatically uses fallback characters to map the text elements of the string.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

ConvertUnicodeToPString

Converts a Unicode string to Pascal in a Mac OS 8 text encoding.

```
pascal OSStatus ConvertUnicodeToPString (
                    UnicodeToTextInfo unicodeToTextInfo,
                    ByteCount unicodeLen,
                    ConstUniCharArrayPtr unicodeStr,
                    Str255 pascalStr);
```

`unicodeToTextInfo`	A conversion information reference (page 3-5) pertaining to the Unicode string to be converted. You use the `CreateUnicodeToTextInfo` function (page 3-38) to obtain the reference for the conversion.
`unicodeLen`	The length in bytes of the Unicode string to be converted. This is the string your application provides in the `unicodeStr` parameter.
`unicodeStr`	An array (page 3-23) containing the Unicode string to be converted.
`pascalStr`	A Pascal string pointer. On output, the converted Pascal string returned by the function.
function result	A result code. The function returns a `noErr` result code if it has completely converted the Unicode string to Pascal without using fallback character sequences. If the function returns the `paramErr`, the function does not convert the string. If the `ConvertFromUnicodeToText` returns the `unicodeFallbacksErr` result code, the function has completely converted the input string to the specified target using one or more fallbacks. If the function returns a `unicodeTableFormatErr` because the mapping table specified an unknown table format or index format or a `unicodeVariantErr` because the specified target encoding was not found in the mapping table, the function does not convert the string. If the function returns a `unicodeCharErr` because the source text contained an invalid Unicode character, a `unicodeElementErr` because the source string contained a text element too long to process, a `unicodeNotFoundErr` because a Unicode text element in the source string is not in the mapping table, or a `unicodeBufErr` because the output buffer specified by `pascalStr` is too short to hold the converted string, then the function did not completely convert the string. If it converted part of the string, the array pointed to by the `unicodeStr` parameter contains the converted portion. To convert the remaining part of the string, you can call this function again, passing the function the rest of the string.

DISCUSSION

The `ConvertUnicodeToPString` function provides an easy and efficient way to convert a Unicode string to Pascal in a Mac OS 8 text encoding without incurring the overhead associated with the `ConvertFromUnicodeToText` function (page 3-41).

If necessary, this function uses the loose mapping and fallback characters to map the text elements of the string. For fallback mappings, it uses the handler associated with the conversion information reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Obtaining Unicode Mapping and Text Encoding Base Name Information

The Unicode Converter provides functions you can use to obtain information. You can obtain a list of the mapping tables available on the system that match specified criteria using the `QueryUnicodeMappings` function.

QueryUnicodeMappings

Returns a list of the conversion mappings available on the system that meet specified matching criteria and returns the number of mappings found.

```
pascal OSStatus QueryUnicodeMappings (
                    OptionBits filter,
                    ConstUnicodeMappingPtr findMapping,
                    ItemCount maxCount,
                    ItemCount *actualCount,
                    UnicodeMappingPtr returnedMappings);
```

`filter` Filter control flags representing the six fields of the Unicode mapping data structure, pointed to by the `findMapping` parameter, that this function uses to match against in determining which mappings on the system to return to your application. The filter indicator enumerations (page 3-21) define the constants for the field's flags and their masks. You can include in the search criteria any of the three text encoding fields for both the Unicode encoding and the other specified encoding. For any flag not turned on, the field value is ignored and the function does not check the corresponding field of the mappings on the system.

`findMapping`

A Unicode mapping data structure (page 3-24) containing the text encodings whose field values are to be matched.

`maxCount`

The maximum number of mappings that can be returned. You provide this value to identify the number of elements in the array pointed to by the `returnedMappings` parameter that your application allocated. If the function identifies more matching mappings than the array can hold, it returns as many of them as fit. The function also returns a `unicodeBufErr` in this case.

`actualCount` A pointer to a value of type `ItemCount`. On output, the number of matching mappings found. This number may be greater than the number of mappings specified by `maxCount` if more matching mappings are found than can fit in the `returnedMappings` array.

`returnedMappings`

A pointer to an array of Unicode mapping data structures (page 3-24). On input, this pointer refers to an array for the matching mappings returned by the function. You should estimate the number of matching mappings you expect will be found and allocate the array with enough elements to hold them. On output, this array holds the matching mappings. If there are more matches than the array can hold, the function returns as many of them as will fit and a `unicodeBufErr` error result. The `actualCount` parameter identifies the number of matching mappings actually found, which may be greater than the number returned.

function result A result code. If the function returns a `noErr` result code, the value retuned in the `actualLen` parameter is less than or equal to the value returned in the `maxCount` parameter and the `returnedMappings` parameter contains all of the matching mappings found. If the function returns a `unicodeBufErr`, the function found more mappings than your `returnedMappings` array could accommodate.

DISCUSSION

You can use the `QueryUnicodeMappings` function to obtain all mappings on the system up to the number allowed by your `returnedMappings` array by specifying a value of zero for the `filter` field.

You can use the function to obtain very specific mappings by setting individual filter indicator flags. You can filter on any of the three text encoding subfields of the Unicode mapping data structure's `unicodeEncoding` specification and on any of the three text encoding subfields of the mapping's `otherEncoding` specification.The `filter` parameter is a set of six indicator flags that correspond to these six subfields. The list provided in the `returnedMappings` parameter will contain only mappings that match the fields of the Unicode mapping data structure whose text encodings fields you identify by setting their corresponding filter indicator flags. No filtering is performed on fields for which you do not set the corresponding filter indicator.

For example, to obtain a list of all mappings in which one of the encodings is the default variant and default format of the Unicode 1.1 base encoding and the other encoding is the default variant and default format of a base encoding other than Unicode 1.1, you would set up the `filter` and `findMappings` parameter as follows. To set up these parameters, you use the constants defined for the text encoding bases, the text encoding default variants, the text encoding default formats, and the filter indicator bitmasks (page 3-21). (For information on text encoding bases, text encoding default variants, and text encoding default formats and their constants, see the "Text Encoding Conversions Reference" chapter.) In this example, the text encoding base field of the Unicode mapping data structure's `otherEncoding` field is ignored, so you can specify any value for it. When you call `QueryUnicodeMappings`, passing it these parameters, the function will return a list of mappings between the Unicode encoding you specified and every other available encoding in which each non-Unicode base encoding shows up once because you specified its default variant and default format.

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

findMapping.otherEncoding = CreateTextEncoding(
                                kTextEncodingMacRoman,
                                kTextEncodingDefaultVariant,
                                kTextEncodingDefaultFormat);

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

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Changing the Conversion Information Structure's Mapping Information

You can use two Unicode Converter functions to change the mapping associated with a conversion information reference. To change the mapping for a reference used to convert text to Unicode, use the `ChangeTextToUnicodeInfo` function. To change the mapping for a reference used to convert Unicode text to another encoding, use the `ChangeUnicodeToText` function.

ChangeTextToUnicodeInfo

Changes the mapping information for the specified conversion information reference used to convert text to Unicode to the new mapping you provide.

```
pascal OSStatus ChangeTextToUnicodeInfo (
                    TextToUnicodeInfo textToUnicodeInfo,
                    ConstUnicodeMappingPtr unicodeMapping);
```

`textToUnicodeInfo`
: The conversion information reference (page 3-3) containing the mapping to be modified. You use the `CreateTextToUnicodeInfo` function (page 3-29) to obtain a text-to-Unicode conversion information reference.

`unicodeMapping`
: A Unicode mapping data structure (page 3-24) identifying the new mapping to be used. This is the mapping that replaces the existing mapping in the conversion information reference.

function result A result code. If `ChangeTextToUnicodeInfo` returns a result code of `noErr`, then the function has successfully changed the mapping associated with the conversion context. If it returns another result code, the function has not changed the mapping. The function returns the `paramErr` result code if one or more of the input parameter values is invalid, the `unicodeNoTableErr` result code if one of the mapping tables specified by the Unicode mapping structure you supply or one of the resources associated with it was not found, and the `unicodeChecksumErr` result code if one of the table resources needed for the mapping has a checksum error.

DISCUSSION

The `ChangeTextToUnicodeInfo` function allows you to provide new mapping information for text to be converted to Unicode. The function replaces the mapping table information that currently exists in the conversion information reference pointed to by the `textToUnicodeInfo` parameter with the information contained in the `UnicodeMapping` data structure you supply as the `UnicodeMapping` parameter.

`ChangeTextToUnicodeInfo` resets the conversion information reference's fields as necessary.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

ChangeUnicodeToTextInfo

Changes the mapping information contained in the specified conversion information reference used to convert Unicode text to another encoding.

```
pascal OSStatus ChangeUnicodeToTextInfo (
                        UnicodeToTextInfo unicodeToTextInfo,
                        ConstUnicodeMappingPtr UnicodeMapping);
```

`unicodeToTextInfo`
: The conversion information reference (page 3-5) to be modified. You use the `CreateUnicodeToTextInfo` function (page 3-38) to obtain a reference of this type.

`UnicodeMapping`
: The Unicode mapping data structure (page 3-24) to be used. This is the new mapping that replaces the existing mapping in the conversion information data reference.

function result A result code. If `ChangeUnicodeToTextInfo` returns a result code other than `noErr`, then the function has successfully changed the mapping associated with the conversion context. If it returns another result code, the function has not changed the mapping. The function returns the `paramErr` result code if one or more of the input parameter values is invalid, the `unicodeNoTableErr` result code if one of the mapping tables specified by the Unicode mapping structure you supply or one of the resources associated with it was not found, the `unicodeChecksumErr` result code if one of the table resources needed for the conversion has a checksum error.

DISCUSSION

The `ChangeUnicodeToTextInfo` function allows you to provide new mapping information for converting text from Unicode to another encoding. The function replaces the mapping table information that currently exists in the specified conversion information reference with the information contained in the new Unicode mapping data structure you provide.

`ChangeUnicodeToTextInfo` resets the conversion information reference's fields as necessary.

This function is especially useful for converting a string from Unicode if the Unicode string contains characters that require multiple target encodings and you know the next target encoding.

For example, you can change the other (target) encoding of the Unicode mapping data structure pointed to by the `UnicodeMapping` parameter before you call the `ConvertFromUnicodeToText` function (page 3-41) to convert the next character or sequence of characters that require a different target encoding.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To convert a Unicode string to multiple target encodings when you do not know the required target encodings, you must call the `ConvertFromUnicodeToTextRun` function (page 3-50).

For general information about Unicode and the Unicode Converter, see “Unicode Converter Constants and Data Types” (page 3-3).

See the chapter “Introduction to Text Handling and Internationalization on Mac OS 8” in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Setting the Fallback Handler

The Unicode Converter provides functions for setting the fallback handler for a particular conversion information reference to be used for converting from Unicode to any encoding.

You assign a fallback handler to a conversion information reference to be used for fallback mapping. A fallback mapping is a sequence of one or more characters in the target encoding for a text element that are not exactly equivalent to the source encoding characters but which preserve some of the information of the original. For example, (C) is a possible fallback mapping for ©. In general, the Unicode Converter uses fallback characters as a last resort in converting text between encodings because they are not reversible and therefore do not lend themselves to round-trip fidelity conversions.

You use the `SetFallbackUnicodeToText` function from within your application's main task to associate your fallback handler with a conversion information reference to be used for converting a single text run using the `ConvertFromUnicodeToText` function or the `ConvertUnicodeToPString` function.

You use the `SetFallbackUnicodeToTextRun` function from within your application's main task to associate your fallback handler with a conversion information reference for multiple text runs to be used with the `ConvertFromUnicodeToTextRun` function.

You use the `SetFallbackUnicodeToTextPreemptive` function from within any other task but your application's main one to associate your fallback handler with a conversion information reference to be used for converting a single text run using the `ConvertFromUnicodeToText` function or the `ConvertUnicodeToPString` function.

You use the `SetFallbackUnicodeToTextRunPreemptive` function from within your application's main task to associate your fallback handler with a conversion information reference for multiple text runs to be used with the `ConvertFromUnicodeToTextRun` function.

SetFallbackUnicodeToText

Associates an application-defined fallback handler with a specific `UnicodeToTextInfo` conversion information reference for a single text run to be used with either the `ConvertFromUnicodeToText` function (page 3-41) or the `ConvertUnicodeToPString` function (page 3-64). You can call `SetFallbackUnicodeToText` from within your application's main task only.

```
pascal OSStatus SetFallbackUnicodeToText (
                     UnicodeToTextInfo unicodeToTextInfo,
                     UnicodeToTextFallbackUPP fallback,
                     OptionBits controlFlags,
                     LogicalAddress infoPtr);
```

`unicodeToTextInfo`	The conversion information reference with which the fallback handler is to be associated. You use the `CreateUnicodeToTextInfo` function (page 3-38) to obtain a reference of this type.
`fallback`	A universal procedure pointer to the application-defined fallback routine. For a description of the function prototype that your fallback handler must adhere to, see "Fallback Handler Function" (page 3-27). For a description of how to create your own fallback handler, see "MyUnicodeToTextFallbackProc" (page 3-90).
`controlFlags`	Control flags (page 3-17) that stipulate which fallback handler the Unicode Converter should call—the application-defined fallback handler or the default handler—if a fallback handler is required, and the sequence in which the Unicode Converter should call the fallback handlers if either can be used when the other fails or is unavailable.
`infoPtr`	The logical address of a context containing data to be passed to the application-defined fallback handler. The Unicode Converter passes this pointer to the application-defined fallback handler as the last parameter when it calls the fallback handler. Your application can use this context to store data required by your fallback handler whenever it is called. A context is similar in use to a System 7 reference constant (refcon).
function result	A result code. The function returns a `noErr` result code if it has successfully installed the application-defined fallback handler. If one or more of the input parameter values is invalid, the function can return an Unicode Converter `paramErr` result code.

DISCUSSION

You use this function to specify a fallback handler to be used for converting a Unicode text segment to another encoding when the Unicode Converter cannot convert the text using the mapping table specified by the conversion information reference passed to the `ConvertFromUnicodeToText` function (page 3-41). You can define multiple fallback handlers and associate them with different conversion information references, depending on your requirements.

If you don't want the Unicode Converter to use a fallback handler defined by your application, you can set the `controlFlags` parameter to direct it to use its default fallback handler only. You can also tell the Unicode Converter to use both fallback handlers and the order in which to call them. For example, you can tell the Unicode Converter to try its fallback handler first, and then use yours, if its handler is unavailable or unable to perform the conversion.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
No	No	No

CALLING RESTRICTIONS

This function can be called only by an application's main task.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

SetFallbackUnicodeToTextRun

Associates an application-defined fallback handler with a specific conversion information reference for multiple text runs to be used with the

`ConvertFromUnicodeToTextRun` function (page 3-50). You can call `SetFallbackUnicodeToTextRun` from within your application's main task only.

```
pascal OSStatus SetFallbackUnicodeToTextRun (
                    UnicodeToTextRunInfo unicodeToTextRunInfo,
                    UnicodeToTextFallbackUPP fallback,
                    OptionBits controlFlags,
                    LogicalAddress infoPtr);
```

`unicodeToTextInfo`	The conversion information reference with which the fallback handler is to be associated. You use the `CreateUnicodeToTextRunInfo` function (page 3-47) to obtain a reference of this type.
`fallback`	A universal procedure pointer to the application-defined fallback routine. For a description of the function prototype that your fallback handler must adhere to, see "Fallback Handler Function" (page 3-27). For a description of how to create your own fallback handler, see "MyUnicodeToTextFallbackProc" (page 3-90).
`controlFlags`	Control flags (page 3-17) that stipulate which fallback handler the Unicode Converter should call—the application-defined fallback handler or the default handler—if a fallback handler is required, and the sequence in which the Unicode Converter should call the fallback handlers if either can be used when the other fails or is unavailable.
`infoPtr`	The logical address of a context containing data to be passed to the application-defined fallback handler. The Unicode Converter passes this pointer to the application-defined fallback handler as the last parameter when it calls the fallback handler. Your application can use this context to store data required by your fallback handler whenever it is called. A context is similar in use to a System 7 reference constant (refcon).
function result	A result code. The function returns a `noErr` result code if it has successfully installed the application-defined fallback handler. If one or more of the input parameter values is invalid, the function can return an Unicode Converter `paramErr` result code.

DISCUSSION

You use this function to specify a fallback handler to be used for converting a Unicode text segment to another encoding when the Unicode Converter cannot convert the text using the mapping table specified by the conversion information reference passed to the `ConvertFromUnicodeToText` function (page 3-41). You can define multiple fallback handlers and associate them with different conversion information references, depending on your requirements.

If you don't want the Unicode Converter to use a fallback handler defined by your application, you can set the `controlFlags` parameter to direct it to use its default fallback handler only. You can also tell the Unicode Converter to use both fallback handlers and the order in which to call them. For example, you can tell the Unicode Converter to try its fallback handler first, and then use yours, if its handler is unavailable or unable to perform the conversion.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
No	No	No

CALLING RESTRICTIONS

This function can be called only by an application's main task.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

SetFallbackUnicodeToTextPreemptive

Associates an application-defined fallback handler with a specific conversion information reference for a single text run to be used with either the

`ConvertFromUnicodeToText` function (page 3-41) or the `ConvertUnicodeToPString` function (page 3-64). You can call `SetFallbackUnicodeToTextPreemptive` from within any task other than your application's main task.

```
pascal OSStatus SetFallbackUnicodeToTextPreemptive(
                    UnicodeToTextInfo unicodeToTextInfo,
                    UnicodeToTextFallbackPreemptiveProcPtr fallback,
                    OptionBits controlFlags,
                    LogicalAddress infoPtr);
```

`unicodeToTextInfo`	The conversion information reference with which the fallback handler is to be associated. You use the `CreateUnicodeToTextInfo` function (page 3-38) to obtain a reference of this type. `SetFallbackUnicodeToTextPreemptive` modifies the conversion information reference contents.
`fallback`	A pointer to the application-defined fallback handler routine. For a description of the function prototype that your fallback handler must adhere to, see the "Fallback Handler Function" (page 3-27). For a description of how to create your own fallback handler, see "MyUnicodeToTextFallbackProc" (page 3-90).
`controlFlags`	Control flags (page 3-17) that stipulate which fallback handler the Unicode Converter should call—the application-defined fallback handler or the default handler—if a fallback handler is required, and the sequence in which the Unicode Converter should call the fallback handlers if either can be used when the other fails or is unavailable.
`infoPtr`	The logical address of a context containing data to be passed to the application-defined fallback handler. The Unicode Converter passes this pointer to the application-defined fallback handler as the last parameter when it calls the fallback handler. Your application can use this context to store data required by your fallback handler whenever it is called. A context is similar in use to a System 7 reference constant (refcon).

function result A result code. The function returns a `noErr` result code if it has successfully installed the application-defined fallback handler. If one or more of the input parameter values is invalid, the function can return a Unicode Converter `paramErr` result code.

DISCUSSION

You use this function to specify a fallback handler to be used for converting a Unicode text segment to another encoding when the Unicode Converter cannot convert the text using the mapping table specified by the conversion information reference passed to the `ConvertFromUnicodeToText` function (page 3-41) or the `ConvertUnicodeToPString` function (page 3-64). You can define multiple fallback handlers and associate them with different conversion information references, depending on your requirements.

If you don't want the Unicode Converter to use a fallback handler defined by your application, you can set the `controlFlags` parameter to direct it to use its default fallback handler only. You can also tell the Unicode Converter to use both fallback handlers and the order in which to call them. For example, you can tell the Unicode Converter to try its fallback handler first, and then use yours if its handler is unavailable or unable to perform the conversion.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

SetFallbackUnicodeToTextRunPreemptive

Associates an application-defined fallback handler with a specific conversion information reference for multiple text runs to be used with the `ConvertFromUnicodeToTextRun` function (page 3-50). You can call `SetFallbackUnicodeToTextRunPreemptive` from within any task other than your application's main task.

```
pascal OSStatus SetFallbackUnicodeToTextRunPreemptive(
                    UnicodeToTextRunInfo unicodeToTextRunInfo,
                    UnicodeToTextFallbackPreemptiveProcPtr fallback,
                    OptionBits controlFlags,
                    LogicalAddress infoPtr);
```

`unicodeToTextInfo`	The conversion information reference with which the fallback handler is to be associated. You use the `CreateUnicodeToTextRunInfo` function (page 3-38) to obtain a reference of this type. `SetFallbackUnicodeToTextRunPreemptive` modifies the conversion information reference contents.
`fallback`	A pointer to the application-defined fallback handler routine. For a description of the function prototype that your fallback handler must adhere to, see the "Fallback Handler Function" (page 3-27). For a description of how to create your own fallback handler, see "MyUnicodeToTextFallbackProc" (page 3-90).
`controlFlags`	Control flags (page 3-17) that stipulate which fallback handler the Unicode Converter should call—the application-defined fallback handler or the default handler—if a fallback handler is required, and the sequence in which the Unicode Converter should call the fallback handlers if either can be used when the other fails or is unavailable.

`infoPtr`	The logical address of a context containing data to be passed to the application-defined fallback handler. The Unicode Converter passes this pointer to the application-defined fallback handler as the last parameter when it calls the fallback handler. Your application can use this context to store data required by your fallback handler whenever it is called. A context is similar in use to a System 7 reference constant (refcon).
function result	A result code. The function returns a `noErr` result code if it has successfully installed the application-defined fallback handler. If one or more of the input parameter values is invalid, the function can return an Unicode Converter `paramErr` result code.

DISCUSSION

You use this function to specify a fallback handler to be used for converting a Unicode text segment to another encoding when the Unicode Converter cannot convert the text using the mapping table specified by the conversion information reference passed to the `ConvertFromUnicodeToText` function (page 3-41). You can define multiple fallback handlers and associate them with different conversion information references, depending on your requirements.

If you don't want the Unicode Converter to use a fallback handler defined by your application, you can set the `controlFlags` parameter to direct it to use its default fallback handler only. You can also tell the Unicode Converter to use both fallback handlers and the order in which to call them. For example, you can tell the Unicode Converter to try its fallback handler first, and then use yours, if its handler is unavailable or unable to perform the conversion.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Converting Between Script Manager Values and Text Encoding Specifications

You can convert Script Manager values to a text encoding specification using the `UpgradeScriptInfoToTextEncoding` function and a text encoding specification to Script Manager values using the `RevertTextEncodingToScriptInfo` function.

UpgradeScriptInfoToTextEncoding

Converts a System 7 script code, a language code, a region code, and a font name to a text encoding.

```
pascal OSStatus UpgradeScriptInfoToTextEncoding (
                        ScriptCode textScriptID,
                        LangCode textLanguageID,
                        RegionCode regionID,
                        ConstStr255Param textFontname,
                        TextEncoding *encoding);
```

`textScriptID` A valid Script Manager script code. The System 7 Script Manager defines constants for script codes using this format: `sm`*`Xxx`*. To designate the system script, specify the meta-value of `smSystemScript`. To designate the current script based on the font specified in the `grafPort`, specify the meta-value of `smCurrentScript`. To designate the script for the current keyboard layout, specify `smInputScript`. To indicate that you do not want to provide a script code for this parameter, specify the constant `kTextScriptDontCare` (page 3-19). See the System 7 volume *Inside Macintosh: Text* for more information on the Script Manager's script codes, language codes, and region codes.

`textLanguageID`	A valid Script Manager language code. The System 7 Script Manager defines constants for language codes using this format: `lang`*`Xxx`*. To indicate that you do not want to provide a language code for this parameter, specify the constant `kTextLanguageDontCare` (page 3-19).
`regionID`	A valid Script Manager region code. The System 7 Script Manager defines constants for region codes using this format: `ver`*`Xxx`*. To indicate that you do not want to provide a region code for this parameter, specify the constant `kTextRegionDontCare` (page 3-19).
`textFontname`	The name of a font, such as Symbol or Zapf Dingbats, each of which has their own text encoding base, or the name of a font that is currently installed on the system. To indicate that you do not want to provide a font name, specify a value of `NULL`.
`encoding`	A pointer to a value of type `TextEncoding`. On output, this value holds the text encoding specification that the function created from the other values you provided. For information on text encoding specifications and their constants, see the "Text Encoding Conversions Reference" chapter.
function result	A result code. `UpgradeScriptInfoToTextEncoding` returns a `noErr` result code if it has successfully translated the values you specified to the corresponding text encoding specification. If it returns other result codes, the function has not upgraded the System 7 values to a text encoding specification. The function returns `paramErr` if two or more of the input parameter values conflict in some way—for example, the System 7 language code does not belong to the script whose script code you specified, or if the input parameter values are invalid. The function returns a `unicodeTextEncodingDataErr` result code if the internal data tables used for translation are invalid.

DISCUSSION

The `UpgradeScriptInfoToTextEncoding` function allows you to translate the specification for an encoding from the world of the Script Manager, which uses script codes, language codes, region codes, and font names, to the Mac OS 8 world of the Unicode and High-Level Encoding Converter and text objects, which uses text encoding specifications. A one-to-one correspondence exists between many of the Script Manager's script codes and a particular Mac OS 8 text encoding base value. However, because text encodings are a super set of script codes, some combinations of script code, language code, region code, and font name might result in a different text encoding base value than would be the case if the translation were based on the script code alone.

When you call the `UpgradeScriptInfoToTextEncoding` function, you can specify any combination of its parameters, but you must specify at least one.

If you don't specify an explicit value for a parameter, you must pass the don't-care constant appropriate to that parameter. `UpgradeScriptInfoToTextEncoding` will use as much information as you supply to determine the equivalent text encoding or the closest approximation. If you provide more than one parameter, all parameters will be checked against one another to ensure that they are valid in combination.

`UpgradeScriptInfoToTextEncoding` first attempts to resolve the language and region codes, if you specify them. If you specified the region but not the language, it maps the region to a language. If you specified both, it ensures that they are valid in combination.

After the language is resolved—if you specified the language code, region code, or both—`UpgradeScriptInfoToTextEncoding` resolves the language in relation to the script. First, the function checks for special languages, using the information presented in Table 3-1.

Table 3-1 Resolving Language and Script Codes to Text Encoding Bases

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 `langSlovenain`	`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`

If the function finds the language in Table 3-1, it assigns the appropriate text encoding base name.

If the resolved language is not listed in Table 3-1 and you did not specify a script,`UpgradeScriptInfoToTextEncoding` derives the appropriate script from the language. If you specified a script, it checks the language against the script to ensure that they are valid in combination.

Note
If you did not specify a language, region, or script, the script remains unresolved at this point in the process. ◆

If the script is resolved or you specified only a script and font name, `UpgradeScriptInfoToTextEncoding` next attempts to resolve the script and font name. If you specified a font name of either Symbol or Zapf Dingbats—each of which has its own base encoding—the function assigns the proper text encoding base name. If the font you specified is not either of these and it is not currently installed on the system, the function returns an error.

If the script is resolved and you specified a font currently installed, it checks the font against the script to ensure that they are valid in combination. If so, the function assigns the proper text encoding base to the script, completing the translation. If not, it returns an error.

If the script is yet unresolved and you specified the name of an installed font, the function derives the script from the font.

If you did not specify a font and the script was resolved earlier in the process, the function assigns the proper text encoding base to the script. Finally, if the script is still unresolved, the function returns an error.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
No	No	No

CALLING RESTRICTIONS

This function can be called only by an application's main task.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

RevertTextEncodingToScriptInfo

Converts the given Mac OS 8 text encoding specification to the corresponding Script Manager's script code and, if possible, language code, region code, and font name.

```
pascal OSStatus RevertTextEncodingToScriptInfo (
                      TextEncoding encoding,
                      ScriptCode *textScriptID,
                      LangCode *textLanguageID
                      Str255 textFontname);
```

`encoding`	The text encoding specification to be converted. For information on text encoding specifications and their constants, see the “Text Encoding Conversions Reference” chapter.
`textScriptID`	A pointer to a value of type `ScriptCode`. On output, a System 7 Script Manager script code that corresponds to the text encoding specification you identified in the `encoding` parameter. If you do not pass a pointer for this parameter on input, the function will return a `paramErr` result code.
`textLanguageID`	A pointer to a value of type `LangCode`. On input, to indicate that you do not want the function to return the language code, specify `NULL` as the value of this parameter. On output, the appropriate language code, if the language can be unambiguously derived from the text encoding specification, for example, Japanese, and you did not set the parameter to `NULL`. If you do not specify `NULL` on input and the language is ambiguous—that is, the function cannot accurately derive it from the text encoding specification—the function returns a value of `kTextLanguageDontCare` (page 3-19).
`textFontname`	A Pascal string. On input, to indicate that you do not want the function to return the font name, specify `NULL` as the value of this parameter. On output, the name of the appropriate font if the font can be unambiguously derived from the text encoding specification, for example, Symbol, and you did not set the parameter to `NULL`. If you do not specify `NULL` on input and the font is ambiguous—that is, the function cannot accurately derive it from the text encoding specification—the function returns a zero-length string.
function result	A result code. The function returns a `noErr` result code if it has successfully translated the encoding specification into the script code, and, optionally, the language code and font name. The function returns `paramErr` if the text encoding specification input parameter value is invalid. The function returns a `unicodeTextEncodingDataErr` result code if the internal data tables used for translation are invalid.

DISCUSSION

The Unicode Converter provides the `RevertTextEncodingToScriptInfo` function for applications that use the System 7 Script Manager and Font Manager functions, which require that encoding specifications be expressed in the format used by these managers. The Unicode Converter provides the `RevertTextEncodingToScriptInfo` function to allow you to convert information in a Mac OS 8 text encoding specification into at least the script code used for System 7 and the appropriate language code and font name, if they can be unambiguously derived, as is the case for Japanese. Your application can then use this information to display text to a user on the screen.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
No	No	No

CALLING RESTRICTIONS

This function can be called only by an application's main task.

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Application-Defined Function

MyUnicodeToTextFallbackProc

Converts a Unicode text element for which there is no target encoding equivalent in the appropriate mapping table to the fallback character sequence defined by your fallback handler, and returns the converted character sequence to the Unicode Converter.

```
pascal OSStatus MyUnicodeToTextFallbackProc(
                        UniChar *srcUniStr,
                        ByteCount srcUniStrLen,
                        ByteCount *srcConvLen,
                        TextPtr *destStr,
                        ByteCount destStrLen,
                        ByteCount *destConvLen,
                        LogicalAddress *contextPtr
                        ConstUnicodeMappingPtr unicodeMappingPtr);
```

`srcUniStr`	A pointer to a value consisting of one or more entities of type `UniChar`. (This value could be an array.) On input, this value specifies the Unicode text element to be treated by the fallback handler. The Unicode Converter passes this parameter to your fallback handler when it calls it. This is the text element that the Unicode Converter was unable to convert.
`srcUniStrLen`	The length in bytes of the text element string passed as the `srcUniStr` parameter. For most text strings, the length is 2 bytes, indicating a single Unicode character. However, the length may exceed 2 bytes, for example, if the text element contains a base character followed by combining characters or if the text element consists of conjoining Korean jamos.

`srcConvLen`	A pointer to a value of type `ByteCount`. On output, the length in bytes of the portion of the text element that was actually processed by your fallback handler. Your fallback handler returns this value.
`destStr`	A pointer to a value of type `ByteCount`. The Unicode Converter passes this pointer to your handler when the Unicode Converter calls it. On output, the converted string returned by your handler.
`destStrLen`	The maximum size in bytes of the buffer provided by the `destStr` parameter.
`destConvLen`	A pointer to a value of type `ByteCount`. On output, the length in bytes of the fallback character sequence generated by your fallback handler. Your handler should return this length.
`contextPtr`	A pointer to a context. On input, the pointer to the context containing data for your fallback handler. The Unicode Converter passes this context to your handler whenever it calls it. This is the context pointer that you specified as the `infoPtr` parameter of the `SetFallbackUnicodeToTextPreemptive` function (page 3-78) or the `SetFallbackUnicodeToTextRunPreemptive` function (page 3-81). How you use the data passed to you in this context is particular to your handler. A context is similar in use to a System 7 reference constant (refcon).
`unicodeMappingPtr`	A constant pointer to a Unicode mapping data structure (page 3-24). A Unicode mapping data structure identifies a Unicode encoding specification and a particular base encoding specification.
function result	A result code. Your function should return a `noErr` result code if it has successfully handled the conversion to the fallback.

DISCUSSION

The Unicode Converter calls your fallback handler when it cannot convert a text string using the mapping table specified by the conversion information reference passed to either the `ConvertFromUnicodeToText` function or the `ConvertUnicodeToPString` function. The control flags you set for the `controlFlags` field of the `SetFallbackUnicodeToTextPreemptive` function or the `SetFallbackUnicodeToTextRunPreemptive` function stipulate which fallback handler the Unicode Converter should call and which one to try first if both can be used.

When the Unicode Converter calls your handler, it passes to it the Unicode text string to be converted and its length, a buffer for the converted string you return and the buffer length, and a pointer to the context containing the data your application supplied to be passed on to your fallback handler. For a description of the function prototype your handler should adhere to, see "Fallback Handler Function" (page 3-27).

After you convert the Unicode text segment to fallback characters, you return the fallback character sequence of the converted text in the buffer provided to you and the length in bytes of this fallback character sequence. You also return the length in bytes of the portion of the source Unicode text element that your handler actually processed.

You provide a fallback-handler function for use with the `ConvertFromUnicodeToText` function (page 3-41), the `ConvertUnicodeToPString` function (page 3-64) function, or the `ConvertFromUnicodeToTextRun` function (page 3-50). You associate an application-defined fallback handler with a particular conversion information reference (page 3-5) you intend to pass to the conversion function when you call it.

You use different functions to associate a fallback handler with a conversion information reference depending on whether you call the function from within your application's main task or if you call the function from any task other than your application's main one.

To associate a fallback-handler function with a conversion information reference from within a task other than your application's main one, you use the `SetFallbackUnicodeToTextPreemptive` and`SetFallbackUnicodeToTextRunPreemptive` functions. For these functions, you pass a pointer to your fallback-handler function as the `fallback` parameter.

To associate a fallback-handler function with a conversion information reference from within your application's main task, you use the

`SetFallbackUnicodeToText` (page 3-83) and `SetFallbackUnicodeToTextRun` (page 3-76) functions. For these functions you must pass a universal procedure pointer (`UniversalProcPtr`). This is derived from a pointer to your function by using the predefined macro `NewUnicodeToTextFallbackProc`.

For a complete description of how to use this universal procedure pointers, refer to the book *Inside Macintosh: PowerPC System Software.*

SEE ALSO

For general information about Unicode and the Unicode Converter, see "Unicode Converter Constants and Data Types" (page 3-3).

See the chapter "Introduction to Text Handling and Internationalization on Mac OS 8" in *Inside Macintosh: Text Handling and Internationalization* for an overview of the Unicode Converter and the conversion process.

Result Codes

The Unicode Converter functions can return result codes specific to the Unicode Converter and also general error codes such as `noErr` (meaning the function completed successfully), `paramErr` (meaning one or more of the input parameters has an invalid value), and memory, operating system, and resource errors. The result codes specific to the Unicode Converter functions are listed here.