Programming With the Text Encoding Conversion Manager

For Version 1.2.1 of the Text Encoding Conversion Manager

10/16/97
Technical Publications

Apple Computer, Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for typographical errors.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Tables, and Listings

CHAPTER 1

About Text Encodings and Conversions

Contents

About Text Encodings and Conversions

This chapter introduces the Text Encoding Conversion Manager. As a prelude, it explains why text encoding conversion is necessary. Then it describes the Text Encoding Conversion Manager's two main components—the Text Encoding Converter and the Unicode Converter—suggesting why you should choose one over the other for your conversion processes. The remainder of the chapter explores some of the terms and concepts that pervade text encoding and the process of converting from one encoding to another, including

- Characters, codes, coded character sets, and character encoding schemes
- Text representation and text elements
- Text encoding specifications
- Unicode, in the context of its emergence as a solution to text encoding complexities
- Round-trip fidelity, strict and loose mapping, Corporate Use Zone mappings, and fallback mappings

Finally, the chapter highlights the Text Encoding Conversion Manager package contents and gives a terse history of its past releases.

You should read this chapter if you are developing

- Internet-savvy applications, such as web browsers or e-mail applications.
- Applications that transfer text across platforms.
- Applications based in Unicode, such as a word processor or file system that operates in Unicode.

After reading this chapter, you should read the reference chapters that describe basic text types for specifying text encodings and other aspects of conversion, the Text Encoding Converter, and the Unicode Converter. The reference chapters are meant to be used as you develop your applications. Although this book doesn't include tutorial chapters, you can consult its descriptions of data structures and functions to gain a high-level understanding of how to use the converters.

For general information about how the Mac OS handles text, consult *Inside Macintosh: Text.*

The Text Encoding Conversion Manager software can run on Mac OS System 7.1 or later. The converter libraries and associated files are installed by default as part of Mac OS 8 and as part of Mac OS Runtime for Java.

Why You Need to Convert Text From One Encoding to Another

This section explains in broad terms why you need to convert text from one encoding used to represent the text to another. Like the following sections that introduce the two Text Encoding Conversion Manager converters, this section uses terminology fundamental to the text encoding conversion process. These terms and the concepts they represent are explored in depth later in “Character Encoding and Other Concepts Fundamental to Text Encoding Conversion” (page 17) and in Appendix B.

Central to any discussion of text encoding and text encoding conversion is the concept of a character, which is an abstract unit of text context. Characters are often identified with or confused with one or more of the following concepts, but it is important to keep the notion of an abstract character separate from these concepts:

- A graphic representation corresponding to a character (this graphic representation is what most people think of as the character)
- A key or key sequence used to input a character
- A number or number sequence used in a computer system to represent a character

In this book we are concerned primarily with abstract characters and with their numeric representation in a computer system. In order to represent textual characters in a file or in a computer’s memory, some sort of mapping must be used to assign numeric values to the textual characters. The mapping can vary depending on the character set, which may depend on the language being used and other factors.

For example, in the ASCII character set, the character A is represented by the value 65, B is represented by 66, and so on. Because ASCII has 128 characters, 7 bits is enough to represent any member of the set (7-bit ASCII characters are usually stored in 8-bit bytes). Each integer value represented by a bit combination is called a code point. (The terms bit combination and code point are further explained in “Character Encoding and Other Concepts Fundamental to Text Encoding Conversion” (page 17).) Larger character sets, such as the Japanese Kanji set, must use more bytes to represent each of their members.

Interpretive problems can occur if a computer attempts to read data that was encoded using a mapping different from what it expects. The other mapping might contain similar characters mapped in a different order, different characters altogether, or the characters may be specially encoded for data transmission. To handle text correctly in these and other similar cases, some method of identifying the various mappings and converting between them is necessary. Text encoding conversion addresses these problems and requirements.

Here are two examples of the many cases for which text conversion is necessary:

- A Mac OS computer receives text in asynchronous packets over the Internet from a remote server. The Mac OS expects text to use the Mac OS Arabic character set, while the server uses the ISO 8859-6 standard.
- A Mac OS application attempts to read a text file created on a Windows 95 computer. The Mac OS application expects text to use the Mac OS Roman character set, while the Windows 95 file uses the Windows Latin-1 character set.

Deciding Which Encoding Converter to Use

The Text Encoding Conversion Manager provides two converters—the Text Encoding Converter and the Unicode Converter—that you can use to handle text encoding conversion on the Mac OS.

The Text Encoding Converter is the primary converter for converting between different text encodings. It was designed to address most of your conversion requirements, and you should use it for most cases. You can use it to convert from one supported encoding to another. When you use the Text Encoding Converter, neither the source encoding nor the destination one must be Unicode, although they can be.

When you use the Unicode Converter, you always convert to or from Unicode; that is, either the source or the destination encoding must be Unicode. You should use the Unicode Converter if you are writing applications based in Unicode, such as a word processor or file system that operates in Unicode.

Even when your application is not Unicode based, you might want to use the Unicode Converter for special cases where you want to control the conversion behavior more closely. The Unicode Converter is also the better choice if you want to map offsets for style run boundaries for styled text; the Text Encoding Converter does not offer this service.

The Text Encoding Converter

The Text Encoding Converter uses plug-ins, which are code fragments containing the information required to perform a conversion. A plug-in can handle one or more types of conversions. Plug-ins are the true conversion engines. The Text Encoding Converter provides a uniform conversion protocol, but includes no implementation for any specific kind of conversion. In other words, it supplies a generic framework for conversion but does none of the conversion work itself; rather, the plug-ins perform the actual conversions.

This section looks briefly at plug-ins, Appendix A describes them in greater detail, and *Mac OS Runtime Architectures* gives general information about CFM-based plug-ins.

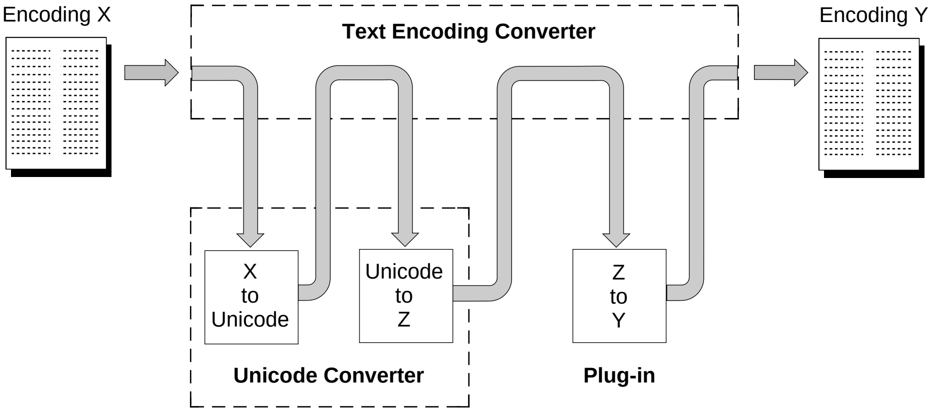
When you launch your application, the Text Encoding Converter scans the Text Encodings folder in the System Folder in search of available plug-ins. The Text Encoding Converter includes many predefined plug-ins—the Unicode converter is one of them—but you can also write and provide your own.

The Text Encoding Converter examines available plug-ins to determine which one or more to use to establish the most direct conversion path. Plug-ins can handle algorithmic conversions such as conversion from JIS to Shift-JIS. (Algorithmic conversions are different from conversion processes that use mapping tables. Mapping tables, which the Unicode Converter uses exclusively, are explained later.) Plug-ins can also handle code-switching schemes such as ISO 2022.

If a plug-in exists for the exact conversion required, then the Text Encoding Converter calls that plug-in's conversion function to convert the text. Such a one-step conversion is called a **direct conversion.** Otherwise, the Text Encoding Converter attempts an **indirect conversion** by finding two or more plug-ins that can be used in succession to perform the required translation. In such cases, the Unicode Converter might be treated as a plug-in.

For example, Figure 1-1 shows a conversion path from encoding X to encoding Y that uses both the Unicode Converter and another plug-in. The Unicode Converter converts encoding X to Unicode, then it converts the Unicode text to text in encoding Z. The other plug-in converts the text from encoding Z to encoding Y.

Figure 1-1 A possible conversion path used by the Text Encoding Converter



In general, you do not need to be concerned about the conversion path taken by the Text Encoding Converter; it is resolved automatically. However, if you want to explicitly specify the conversion path, there are functions you can call to do so.

When you use the Text Encoding Converter, you specify the source and destination encodings for the text. To convert text, you must create a **converter object**. This object describes the conversion path required to perform the text conversion. You can also create a converter object to handle multiple encoding runs. If the requisite plug-ins are available, the Text Encoding Converter can convert text from any encoding to runs of any other encodings.

When handling code-switching schemes, the Text Encoding Converter automatically maintains state information that identifies the current encoding in the converter object. Any escape sequences, control characters, and other information pertaining to state changes in the converter object are also detected and generated as necessary.

Because each converter object can maintain state information, you can use the same converter object to convert multiple segments of a single text stream. For example, suppose you receive text containing 2-byte characters in packets over a network. If the end of a packet transmission splits a character—that is, only 1 of the 2 bytes is received—the converter object does not attempt to convert the character until it receives the second byte.

In some cases, you may not be able to determine the encoding used to express text you receive from an unknown source, such as text delivered over the Internet. To minimize the amount of guesswork required to successfully convert such text, the Text Encoding Converter allows the use of **sniffers**. Sniffers are to text encodings what protocol analyzers are to networking protocols. They analyze the text and provide a list of the most probable encodings used to express it. Several sniffers are provided; you can also write your own sniffers when creating text conversion plug-ins.

The Unicode Converter

This section describes the Unicode Converter, which you can use to convert between any available non-Unicode text encoding and the various, supported implementations of Unicode. For background information on Unicode, the problems it addresses, and the standards bodies responsible for its emergence, see “About Unicode” (page 21) and Appendix B. For definition of some of the terms used in this section, see “Character Encoding and Other Concepts Fundamental to Text Encoding Conversion” (page 17).

The Unicode Converter does not itself incorporate any knowledge of the specifics of any text encoding. Instead, it uses loadable, replaceable mapping tables that provide the information about any text encoding required to perform the conversion.

All information about a particular coded character set used in a text encoding is incorporated in a mapping table. A mapping table associates coded representations of characters belonging to one coded character set with their equivalent representations in another and accounts for the various conditions that arise when coded representations of characters cannot be directly mapped to each other.

The Unicode Converter can also handle conversions between Unicode and text encodings that use a packing scheme.

To convert text using the Unicode Converter, you must create a **Unicode converter object**, which references the necessary mapping tables and maintains state information. Because each Unicode converter object is discrete, you can retain several objects concurrently within your application, one for each type of conversion you need to make.

The Unicode Converter supports multiple encoding runs. An **encoding run** is a continuous sequence of text all of which is expressed in the same text encoding; a given string might contain multiple encoding runs, such as a sequence of text in Mac OS Roman encoding followed by a sequence in Mac OS Arabic. The Unicode Converter allows you to convert a single block of Unicode text to multiple runs in other text encodings. For example, you could convert a Unicode string into one that contains both Mac OS Arabic and Mac OS Roman encodings. You might find this useful when preparing text to display using the Script Manager.

Character Encoding and Other Concepts Fundamental to Text Encoding Conversion

In considering how text is converted from one encoding to another, it is useful to understand what constitutes coded character sets and character encoding schemes. To do so, it is helpful to have a set of terms that describe the discrete entities comprising a coded character set, a character encoding scheme, and their underlying concepts.

This section explores

- characters and character repertoires
- coded character sets and code points
- presentation forms
- character encoding schemes

For a more complete treatment of these and other concepts such as packing schemes, multiple character sets, and code-switching schemes for multiple character sets, see Appendix B.

Characters

A person using a writing system thinks of a character in terms of its visual form, its written structure and its meaning in conjunction with other characters. A computer, on the other hand, deals with characters primarily in terms of their numeric encodings.

A **character** is a unit of information used for the organization, control, or representation of text data. Letters, ideographs, digits, and symbols in a writing system are all examples of characters. A character is associated with a name, and optionally, but commonly, with a representative image or rendering called a glyph. **Glyph images** are the visual elements used to represent characters. Aspects of text presentation such as font and style apply to glyph images, not to characters.

A **character repertoire** is a collection of *distinct* characters. Two characters are distinct if and only if they have distinct names in the context of an identified character repertoire. Two characters that are distinct in name may have identical images or renderings (for example, LATIN CAPITAL LETTER A and GREEK CAPITAL LETTER ALPHA). Characters constituting a character repertoire can belong to different scripts.

Coded Character Sets

A **coded character set** comprises a mapping from a set of abstract characters (that is, the character repertoire) to a set of integers. The integers in the set are within a range that can be expressed by a bit pattern of a particular size: 7 bits, 8 bits, 16 bits, and so on. Each of the integers in the set is called a **code point.** The set of integers may be larger than the character repertoire; that is, there may be “unassigned” code points that do not correspond to any character in the repertoire. Examples of coded character sets include

- ASCII, a fixed-width 7-bit encoding
- ISO 8859-1 (Latin-1), a fixed-width 8-bit encoding

JIS X0208, a Japanese standard whose code points are fixed–width 14-bit values (normally represented as a pair of 7-bit values). Many other standards for East Asian languages follow a similar pattern, using code points represented as two or three 7-bit values. These standards are typically not used directly, but are used in one of the character encoding schemes discussed in “Character Encoding Schemes” (page 19).

Presentation Forms

The term **presentation form** is generally used to mean a kind of abstract shape that represents a standard way to display a character or group of characters in a particular context as specified by a particular writing system. The term glyph by itself may refer to either presentation forms or to glyph images. Examples of characters with multiple presentation forms include

- Arabic characters that vary in appearance depending on the characters surrounding them
- Latin or Arabic ligatures, which are single forms that represent a sequence of characters
- Japanese kana and CJK punctuation characters, which vary in appearance depending on whether they are to be displayed horizontally or vertically
- Katakana full-width and half-width variants

A coded character set may encode presentation forms instead of or in addition to its basic characters.

Character Encoding Schemes

A **character encoding scheme** is a mapping from a sequence of elements in one or more coded character sets to a sequence of bytes. A character encoding scheme can include coded character sets, but it can also include more complex mapping schemes that combine multiple coded character sets, typically in one of the following ways:

- **Packing schemes** use a sequence of 8-bit values to encode text. Because of this, they are generally not suitable for electronic mail. In these schemes, certain characters function as a local shift, which controls the interpretation of the next 1 to 3 bytes. The most well known example is Shift-JIS, which includes characters from JIS X0201, JIS X0208, and space for 2444 user-defined characters. The EUC (Extended UNIX Coding) packing schemes were originally developed for UNIX systems; they use units of 1 to 4 bytes. (Appendix B describes Shift-JIS, EUC, and other packing schemes, in detail.) Packing schemes are often used for the World Wide Web, which can handle 8-bit values. Both the Text Encoding Converter and the Unicode Converter support packing schemes.
- **Code-switching schemes** typically use a sequence of 7-bit values to encode text, so they are suitable for electronic mail. Escape sequences or other special sequences are used to signal a shift among the included character sets. Examples include the ISO 2022 family of encodings (such as ISO 2022-JP), and the HZ encoding used for Chinese. Code switching schemes are often used for Internet mail and news, which cannot handle 8-bit values.

The Text Encoding Converter can handle code-switching schemes, but the Unicode Converter cannot.

A character encoding scheme may also be used to convert a single coded character set into a form that is easier for certain systems to handle. For example, the Unicode standard defines two **universal transformation formats** that permit the use of Unicode on systems that make assumptions about certain byte values in text data. The two universal transformation formats are UTF-7 and UTF-8. The Text Encoding Converter can handle both formats, but the Unicode Converter can only handle the UTF-8 format.

Many Internet protocols allow you to specify a "charset" parameter, which is designed to indicate the character encoding scheme for text.

A **transfer encoding syntax** (also called "content transfer encoding") is a transformation applied to text encoded using a character encoding scheme to allow it to be transmitted by a specific protocol or set of protocols. Examples include "quoted-printable" and "base64". Such a transformation is typically needed to allow 8-bit values to be sent through a channel that can handle only 7-bit values, and may even handle some 7-bit values in special ways. The Text Encoding Conversion Manager does not currently handle transfer encoding syntax.

Text Encoding Specifications

One of the primary data types used by both the Text Encoding Converter and the Unicode Converter is a text encoding specification. This section highlights the text encoding specification. The chapter "Basic Text Types Reference" describes it fully, including its three components, and the values you specify for them.

A **text encoding specification** is a set of numeric codes used to identify a text encoding, which may be simple coded character set or a character encoding scheme. It contains these three parts that specify the text encoding: the text encoding base, the text encoding variant, and the text encoding format. You use two text encoding specifications—one for the source encoding of the text and one for its the destination encoding—when you call the Text Encoding Converter or the Unicode Converter to convert text.

The **text encoding base** value is the primary specification of the source or target encoding. The **text encoding variant** specifies one among possibly several minor variants of a particular base encoding or group of base encodings. A **text encoding format** specifies a way of formatting or algorithmically transforming a particular base encoding. (UTF-7 format is the Unicode standard formatted for transmission through channels that can handle only 7-bit values.)

Note
Text encoding specifications are similar to the Mac OS script codes in that they identify an encoding. However, they are more precise; they do not imply anything about language or region; and they are not necessarily identified with a range of font family IDs. ◆

About Unicode and the Complexities of Conversion

This section looks briefly at Unicode, its emergence in response to the problems it addresses, and the standards bodies who sponsor it. Then it discusses some of the complexities involved in converting text between various encodings when conversion exceeds the simplicity of a one-to-one mapping. The section discusses these concepts in the context of how the Unicode Converter handles them.

About Unicode

Most character sets and character encoding schemes developed in the past are limited in their coverage, usually supporting just one language or a small set of languages. In addition, character encoding schemes are often complex, usually involving byte values whose interpretation depends on preceding byte values. Multilingual software has traditionally had to implement methods for supporting and identifying multiple character encodings.

A simpler solution is to combine the characters for all commonly used languages and symbols into a single universal coded character set. Unicode is such a universal coded character set, and offers the simplest solution to the problem of text representation in multilingual systems. Because Unicode also contains a wide assortment of technical, typographic, and other symbols, it offers advantages even to developers of applications that only handle a single language. Unicode provides more representational power than any other single character set or encoding scheme. However, because Unicode is a single coded character set, it doesn't require the use of escape sequences or other complexities to identify transitions between coded character sets.

Because Unicode includes the character repertoires of most common character encodings, it facilitates data interchange with other platforms. Using Unicode, text manipulated by your application and shared across applications and platforms can be encoded in a single coded character set; this text can also be easily localized.

Unicode provides some special features, such as combining or nonspacing marks and conjoining jamos. These features are a function of the variety of languages that Unicode handles. If you have coded applications that handle text for the languages these features support, they should be familiar to you. If you have used a single coded character set such as ASCII almost exclusively, these features will be new to you.

The following two bodies, involved in the effort to standardize the world's languages for use in computing, define Unicode standards:

- The Unicode Consortium, a technical committee composed of representatives from many different companies, publishes the Unicode standard. Version 2.0 of the Unicode Standard was published in July 1996. However, the standard is evolving constantly, and updates are posted at the Unicode Consortium Web site `<http://www.unicode.org/>`.
- ISO (the International Organization for Standardization) and the IEC (the International Electrotechnical Commission), two of the international bodies active in character encoding standards, publish ISO/IEC 10646. This standard specifies the Universal Multiple-Octet Coded Character Set (UCS), a standard whose code point assignments are identical with Unicode.

ISO/IEC 10646

The ISO/IEC 10646 standard defines two alternative forms of encoding:

- a 32-bit encoding, which is the canonical form. The 32-bit form is referred to as UCS-4 (Universal Character Set containing 4 bytes)
- a 16-bit form that is referred to as UCS-2

The ISO/IEC 10646 nomenclature refers to coded characters as multiples of octets, while the Unicode nomenclature refers to coded characters as indivisible 16-bit entities. The Unicode standard does not include the UCS-4 format.

Round-Trip Fidelity

When the Unicode Converter is able to convert a text string expressed in one text encoding to Unicode and back again to the original text encoding, with the final text string matching exactly the source text string—that is, without incurring any changes to the original—round-trip fidelity has been achieved.

For certain national and international standards that the Unicode Consortium used as sources for the Unicode coded character set, Unicode provides round-trip fidelity. Because the repertoires of those coded character sets have been effectively incorporated into the Unicode coded character set, conversion involving them will always produce round-trip fidelity. Text in one of those coded character sets can be mapped to Unicode and back again with no loss of information. Coded characters that were distinct in the source encoding will be distinct in Unicode.

However, perfect round-trip conversion is not always possible. Many character encodings include characters that do not have distinct representations in Unicode, or which may have no representation at all. For example, a source text string from a vendor coded character set might contain a ligature that is not represented in Unicode. In this case, that information may be lost during the round trip.

The Unicode Converter uses a variety of conventional methods to attempt to find some way to map the source coded representation of a character onto a sequence of Unicode coded representations in such a way as to preserve its identity and interchangeability.

Here are some of the methods used to map code representations of characters when high fidelity achieved through an exact or strict mapping is not possible:

- loose mapping
- fallback mapping
- mapping of characters to the Corporate Use Zone

Multiple Semantics and Multiple Representations

In many character encodings, certain characters may have multiple semantics, either by explicit definition, ambiguous definition, or established usage.

For example, the JIS X0208 standard specifies the JIS X0208 character 0x2142 as having two meanings: double vertical line and parallel to. Each meaning corresponds to a distinct Unicode code representation. The meaning “double vertical line" corresponds to the Unicode coded representation U+2016 "DOUBLE VERTICAL LINE". The meaning "parallel to" corresponds to the Unicode coded representation U+2225 "PARALLEL TO". Either one is a valid match for the JIS character.

Multiple representation exists when an encoding provides more than one way of representing a particular element of text. For example, in Unicode the text element consisting of an 'a' with acute accent can be represented using either the single character LATIN SMALL LETTER A WITH ACUTE or the sequence LATIN SMALL LETTER A plus COMBINING ACUTE ACCENT. The presentation forms encoded in Unicode can also be represented using coded representations for the abstract forms, and this also constitutes a condition of multiple representation.

Strict and Loose Mapping

A **strict mapping** preserves the information content of text *and* permits round-trip fidelity. A **loose mapping** preserves the information content of text but does not permit round-trip fidelity. A mapping table has both strict equivalence and loose mapping sections that identify how a mapping is to occur. Loose and strict mappings occur within the context of multiple semantics and multiple representations.

First, an example that illustrates the difference in the case of multiple semantics. The ASCII character at 0x2D is called HYPHEN-MINUS. Unicode includes a HYPHEN-MINUS character at U+002D for ASCII compatibility. However, Unicode also has separate characters HYPHEN (U+2010) and MINUS SIGN (U+2212); each of these characters represents one aspect of the meaning of HYPHEN-MINUS.

The ASCII character HYPHEN-MINUS is typically mapped to Unicode HYPHEN-MINUS. All three of the Unicode characters—HYPHEN-MINUS, HYPHEN, and MINUS SIGN—should typically be mapped to ASCII HYPHEN-MINUS, since it includes all of their meanings. The mapping from Unicode HYPHEN-MINUS to ASCII is strict, since mapping from ASCII back to Unicode produces the original Unicode character. However, the mappings from Unicode HYPHEN and MINUS SIGN to ASCII are loose, since they do not provide round-trip fidelity. The mapping from ASCII HYPHEN-MINUS to Unicode is, of course, strict.

Second, an example that illustrates the difference in the case of multiple representation. The Latin-1 character LATIN SMALL LETTER A WITH ACUTE (0xE1) is typically mapped to Unicode LATIN SMALL LETTER A WITH

ACUTE (U+00E1), so the reverse is a strict mapping. However, the Unicode sequence LATIN SMALL LETTER A plus COMBINING ACUTE ACCENT can also be mapped to the Latin-1 character as a loose mapping.

There are two important things to note here. First, calling a mapping from one character set to another strict or loose depends on how the second character set is mapped back to the first; strictness or looseness depends on the mappings in both directions. Second, neither strict nor loose mappings necessarily preserve the number of characters; either can map a sequence of one or more characters in the source encoding to one or more characters in the destination encoding.

Fallback Mappings

A fallback mapping is a sequence of one or more coded characters in the destination encoding that is not exactly equivalent to a character in the source encoding but which preserves some of the information of the original. For example, (C) is a possible fallback mapping for ©. In general, fallback mappings are used 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.

Corporate Use Zone

Code space in the Unicode standard is divided into areas and zones. One area, called the Private Use Area, includes a zone called the Corporate Use Zone.

Some characters which are in Mac OS encodings but not in Unicode are mapped to code points in the Unicode Corporate Use Zone. This permits round-trip fidelity for these characters. The Apple logo is an example.

Apple provides a registry of its assignments in the Unicode Corporate Use Zone that you can check to ensure that you don't use the same code representations. The URL is
`<ftp://ftp.unicode.org/Public/MAPPINGS/VENDORS/APPLE/CORPCHR.TXT>`.

Although they allow the Unicode Converter to guarantee perfect round trips for certain code representations, characters in the Unicode Corporate Use Zone are not portable to other systems.

About the Text Encoding Manager Package

The Text Encoding Conversion Manager comprises the Text Encoding Converter, the Unicode Converter, Basic Text Types, and the Text Encodings folder that includes files containing mapping tables and text plug-ins. The first three of these components are delivered as shared libraries called `UnicodeConverter` (the Unicode Converter), `TextEncodingConverter` (the Text Encoding Converter), and `TextCommon` (Basic Text Types).

About Earlier Releases

Text Encoding Converter (TEC) Manager 1.0.x was released for use with Cyberdog 1.0 and 1.2 and with Macintosh Runtime for Java 1.0. TEC Manager 1.1 was released for use with Cyberdog 2.0, and TEC Manager 1.2 was released with Mac OS 8.

Header files for TEC Manager 1.2 were distributed with the Universal Interfaces 3.0. In previous versions of the Text Encoding Conversion Manager, the Unicode Converter was called the Low-Level Encoding Converter and the Text Encoding Converter was called the High-Level Encoding Converter.

Checking the Version

Versions 1.2.1 and later of the Text Encoding Conversion Manager include the `TECGetInfo` function, which returns the product version number and other information. This function does not exist in previous releases; absence of this function identifies the version in use as 1.2 or earlier.

You can determine if an earlier release of the Text Encoding Conversion Manager is in use by soft-linking to the `TECGetInfo` function.

Unicode Converter 68K Static Libraries

The Text Encoding Conversion Manager 1.3 includes a 68K static library version of the Unicode Converter and Basic Text functions for those of you who do not want to use the Code Fragment Manager (CFM) 68K version of the Unicode Converter.

The static libraries are provided for you to link directly into your applications rather than relying on operating system support; the shared libraries that require CFM are distributed with Mac OS, beginning with Mac OS 8.

These static libraries use resources from the Text Encoding Converter extension and from the files in the Text Encodings folder, so both of these must be present whether you use the CFM 68K version of the Unicode Converter or the 68K shared libraries.

To use the static libraries, install the full shared library version comprised of the table files and extension. If you use the 68K static libraries, explicitly initialize and terminate the Unicode Converter using its functions provided for this purpose.

Basic Text Types Reference

Contents

This chapter describes the Mac OS basic text data types, constants, and functions, which you can use to create text encoding specifications, to obtain values from existing specifications, to obtain localized names corresponding to text encoding specifications, and to obtain information about the Text Encoding Conversion Manager itself. It also includes result codes returned for both the Text Encoding Converter functions and the Unicode Converter functions.

For a description of types, constants, and functions pertaining to the Text Encoding Converter, see Chapter 3, "Text Encoding Converter Reference." For a description of types, constants, and functions pertaining to the Unicode Converter, see Chapter 4, "Unicode Converter Reference."

Basic Text Constants

Text Encoding Base

You use a base text encoding data type to specify which text encoding or text encoding scheme you have used to express a given text. The text encoding base value is the primary specification of the source or target encoding. Values 0 through 32 correspond directly to Mac OS script codes. Values 33 through 254 are for other Mac OS encodings that do not have their own script codes, such as the Symbol encoding implemented by the Symbol font. You can also specify a meta-value as a base text encoding, such as `kTextEncodingMacHFS` and `kTextEncodingUnicodeDefault`. A meta-value is mapped to a real value.

The function `GetTextEncodingBase` (page 51) returns the text encoding base of a text encoding specification.

A base text encoding is defined by the `TextEncodingBase` data type.

```
typedef UInt32 TextEncodingBase;
```

You can use these enumerated constants to specify base text encodings:

```
enum {
    /* Mac OS encodings */
    kTextEncodingMacRoman = 0L,
    kTextEncodingMacJapanese = 1,
kTextEncodingMacChineseTrad = 2,
kTextEncodingMacKorean = 3,
kTextEncodingMacArabic = 4,
kTextEncodingMacHebrew = 5,
kTextEncodingMacGreek = 6,
kTextEncodingMacCyrillic = 7,
kTextEncodingMacDevanagari = 9,
kTextEncodingMacGurmukhi = 10,
kTextEncodingMacGujarati = 11,
kTextEncodingMacOriya = 12,
kTextEncodingMacBengali = 13,
kTextEncodingMacTamil = 14,
kTextEncodingMacTelugu = 15,
kTextEncodingMacKannada = 16,
kTextEncodingMacMalayalam = 17,
kTextEncodingMacSinhalese = 18,
kTextEncodingMacBurmese = 19,
kTextEncodingMacKhmer = 20,
kTextEncodingMacThai = 21,
kTextEncodingMacLaotian = 22,
kTextEncodingMacGeorgian = 23,
kTextEncodingMacArmenian = 24,
kTextEncodingMacChineseSimp = 25,
kTextEncodingMacTibetan = 26,
kTextEncodingMacMongolian = 27,
kTextEncodingMacEthiopic = 28,
kTextEncodingMacCentralEurRoman = 29,
kTextEncodingMacVietnamese = 30,
kTextEncodingMacExtArabic = 31,

/* The following use script code 0, smRoman */
kTextEncodingMacSymbol = 33,
kTextEncodingMacDingbats = 34,
kTextEncodingMacTurkish = 35,
kTextEncodingMacCroatian = 36,
kTextEncodingMacIcelandic = 37,
kTextEncodingMacRomanian = 38,

/* The following use script code 4, smArabic */
kTextEncodingMacFarsi = 0x8C, /* Like MacArabic but uses Farsi digits */
```

```
/* The following use script code 7, smCyrillic */
kTextEncodingMacUkrainian = 0x98,

/* The following use script code 32, smUninterp */
kTextEncodingMacVT100 = 0xFC, /* VT100/102 font: Latin-1 chars, box dwg… */

/* Special Mac OS encodings */
kTextEncodingMacHFS = 0xFF, /* metavalue. */

/* Unicode & ISO UCS encodings begin at 0x100
kTextEncodingUnicodeDefault = 0x100, /* Meta-value. */
kTextEncodingUnicodeV1_1 = 0x101,
kTextEncodingISO10646_1993 = 0x101, /* code points identical to Unicode 1.1 */
kTextEncodingUnicodeV2_0 = 0x103, /* new location for Korean Hangul */

/* ISO 8-bit and 7-bit encodings begin at 0x200 */
kTextEncodingISOLatin1 = 0x201, /* ISO 8859-1 */
kTextEncodingISOLatin2 = 0x202, /* ISO 8859-2 */
kTextEncodingISOLatinCyrillic = 0x205, /* ISO 8859-5 */
kTextEncodingISOLatinArabic = 0x206, /* ISO 8859-6, = ASMO 708, =DOS CP 708 */
kTextEncodingISOLatinGreek = 0x207, /* ISO 8859-7 */
kTextEncodingISOLatinHebrew = 0x208, /* ISO 8859-8 */
kTextEncodingISOLatin5 = 0x209, /* ISO 8859-9 */

/* MS-DOS & Windows encodings begin at 0x400 */
kTextEncodingDOSLatinUS = 0x400, /* code page 437 */
kTextEncodingDOSGreek = 0x405, /* code page 737 (formerly 437G) */
kTextEncodingDOSBalticRim = 0x406, /* code page 775 */
kTextEncodingDOSLatin1 = 0x410, /* code page 850, "Multilingual" */
kTextEncodingDOSGreek1 = 0x411, /* code page 851 */
kTextEncodingDOSLatin2 = 0x412, /* code page 852, Slavic */
kTextEncodingDOSCyrillic = 0x413, /* code page 855, IBM Cyrillic */
kTextEncodingDOSTurkish = 0x414, /* code page 857, IBM Turkish */
kTextEncodingDOSPortuguese = 0x415, /* code page 860 */
kTextEncodingDOSIcelandic = 0x416, /* code page 861 */
kTextEncodingDOSHebrew = 0x417, /* code page 862 */
kTextEncodingDOSCanadianFrench = 0x418, /* code page 863 */
kTextEncodingDOSArabic = 0x419, /* code page 864 */
kTextEncodingDOSNordic = 0x41A, /* code page 865 */
kTextEncodingDOSRussian = 0x41B, /* code page 866 */
kTextEncodingDOSGreek2 = 0x41C, /* code page 869, IBM Modern Greek */
```

```
kTextEncodingDOSThai = 0x41D, /* code page 874, also for Windows */
kTextEncodingDOSJapanese = 0x420, /* code page 932, also for Windows */
kTextEncodingDOSChineseSimplif = 0x421, /* code page 936, also for Windows */
kTextEncodingDOSKorean = 0x422, /* code page 949, also for Windows;Unified Hangul */
kTextEncodingDOSChineseTrad = 0x423, /* code page 950, also for Windows */
kTextEncodingWindowsLatin1 = 0x500, /*code page 1252 */
kTextEncodingWindowsANSI = 0x500, /* code page 1252 (alternate name) */
kTextEncodingWindowsLatin2 = 0x501, /* code page 1250, Central Europe */
kTextEncodingWindowsCyrillic = 0x502, /* code page 1251, Slavic Cyrillic */
kTextEncodingWindowsGreek = 0x503, /* code page 1253 */
kTextEncodingWindowsLatin5 = 0x504, /* code page 1254, Turkish */
kTextEncodingWindowsHebrew = 0x505, /* code page 1255 */
kTextEncodingWindowsArabic = 0x506, /* code page 1256 */
kTextEncodingWindowsBalticRim = 0x507, /* code page 1257 */
kTextEncodingWindowsKoreanJohab =0x510, /* code page 1361, for Windows NT */

/* Various national standards begin at 0x600 */
kTextEncodingUS_ASCII = 0x600,
kTextEncodingJIS_X0201_76 = 0x620,
kTextEncodingJIS_X0208_83 = 0x621,
kTextEncodingJIS_X0208_90 = 0x622,
kTextEncodingJIS_X0212_90 = 0x623,
kTextEncodingJIS_C6226_78 = 0x624,
kTextEncodingGB_2312_80 = 0x630,
kTextEncodingGBK_95 = 0x631, /* annex to GB 13000-93; for Windows 95 */
kTextEncodingKSC_5601_87 = 0x640, /* same as KSC 5601-92 without Johab annex */
kTextEncodingKSC_5601_92_Johab = 0x641, /* KSC 5601-92 Johab annex */
kTextEncodingCNS_11643_92_P1 = 0x651, /* CNS 11643-1992 plane 1 */
kTextEncodingCNS_11643_92_P2 = 0x652, /* CNS 11643-1992 plane 2 */
kTextEncodingCNS_11643_92_P3 = 0x653, /* CNS 11643-1992 plane 3
                                     (11643-1986 plane 14) */

/* ISO 2022 collections begin at 0x800 */
kTextEncodingISO_2022_JP = 0x820,
kTextEncodingISO_2022_JP_2 = 0x821,
kTextEncodingISO_2022_CN = 0x830,
kTextEncodingISO_2022_CN_EXT = 0x831,
kTextEncodingISO_2022_KR = 0x840,

/* EUC collections begin at 0x900 */
kTextEncodingEUC_JP = 0x920, /* ISO 646,1-byte Katakana,JIS 208,JIS 212 */
```

```
    kTextEncodingEUC_CN = 0x930, /* ISO 646, GB 2312-80 */
    kTextEncodingEUC_TW = 0x931, /* ISO 646, CNS 11643-1992 Planes 1-16 */
    kTextEncodingEUC_KR = 0x940, /* ISO 646, KS C 5601-1987 */

    /* Miscellaneous standards begin at 0xA00 */
    kTextEncodingShiftJIS = 0xA01, /* plain Shift-JIS */
    kTextEncodingKOI8_R = 0xA02, /* Russian Internet standard */
    kTextEncodingBig5 = 0xA03, /* Big-5 */
    kTextEncodingMacRomanLatin1 = 0xA04, /* Mac OS Roman permuted to align
                                            with 8859-1 */
    kTextEncodingHZ_GB_2312 = 0xA05 /* HZ (RFC 1842, for Chinese mail & news) */

    /* Other platform encodings */
    kTextEncodingNextStepLatin = 0xB01, /* NextStep encoding */

    /* EBCDIC & IBM host encodings begin at 0xC00 */
    kTextEncodingEBCDIC_US = 0xC01, /* basic EBCDIC-US */
    kTextEncodingEBCDIC_CP037 = 0xC02 /* code page 037, extended EBCDIC-US Latin1 */

    /* Special value */
    kTextEncodingMultiRun = 0xFFF, /* Multiple encoded text, external run info */
};
```

Text Encoding Variant

A text encoding variant specifies one among possibly several minor variants of a particular base encoding or group of base encodings. Text encoding variants are often used to support special cases such as the following:

- Differences among fonts that are all intended to support the same encoding. For example, different fonts associated with the MacJapanese and MacArabic encodings support slightly different encoding variants. These fonts would typically coexist on the same system without the user being aware of any differences.
- Artificial variants created by excluding some of the characters in an encoding. For example, the MacJapanese encoding includes separately-encoded vertical forms for some characters. In some contexts (such as with QuickDraw GX), it may be desirable to exclude these.

- Different mappings of a particular character or group of characters for different usages.

For a given text encoding base or small set of related text encoding base values, there may be an enumeration of `TextEncodingVariant` values, which always begins with 0, the default variant. In addition, for a possibly larger set of related text encoding base values, there may be bit masks that can be used independently to designate additional artificial variants. For example, there is an enumeration of six variants for the Mac OS Japanese encoding. In addition, there are bit masks that can also be used as part of the variant for any Japanese encoding to exclude 1-byte kana or to control the mapping of the reverse solidus (backslash) character.

Languages that are dissimilar but use similar character sets are generally not designated as variants of the same base encoding (for example, MacIcelandic and MacTurkish both use a slight modification of the MacRoman character set, but they are considered separate base encodings).

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

```
typedef UInt32 TextEncodingVariant;
```

When you create a new text encoding, you can specify an explicit variant of a base encoding or you can specify the default variant of that base.

The function `GetTextEncodingVariant` (page 52) returns the text encoding variant of a text encoding specification.

The following enumeration defines constants for the default variant of any base text encoding and for variants of the Mac OS Japanese, Mac OS Arabic, Mac OS Farsi, Mac OS Hebrew, and Unicode base encodings.

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

    /* Variants of kTextEncodingMacJapanese */
    kMacJapaneseStandardVariant = 0,
    kMacJapaneseStdNoVerticalsVariant = 1,
    kMacJapaneseBasicVariant = 2,
    kMacJapanesePostScriptScrnVariant = 3,
    kMacJapanesePostScriptPrintVariant = 4,
    kMacJapaneseVertAtKuPlusTenVariant = 5,
    /* Variant options for most Japanese encodings (including MacJapanese, Shift-JIS,
        EUC-JP, ISO 2022-JP). These can be OR-ed into the variant value in any
        combination. */
    kJapaneseNoOneByteKanaOption = 0x20,
    kJapaneseUseAsciiBackslashOption = 0x40,

    /* Variants of kTextEncodingMacArabic */
    kMacArabicStandardVariant = 0, /* Cairo font & WorldScript tables */
    kMacArabicTrueTypeVariant = 1, /* Baghdad, Geeza, Kufi, Nadeem fonts */
    kMacArabicThuluthVariant = 2, /* Thuluth font */
    kMacArabicAlBayanVariant = 3, /* Al Bayan font */

    /* Variants of kTextEncodingMacFarsi */
    kMacFarsiStandardVariant = 0, /* Tehran font & WorldScript tables */
    kMacFarsiTrueTypeVariant = 1, /* TrueType fonts */

    /* Variants of kTextEncodingMacHebrew */
    kMacHebrewStandardVariant = 0,
    kMacHebrewFigureSpaceVariant = 1,

    /* Variants of kTextEncodingMacIcelandic */
    kMacIcelandicStandardVariant = 0,
    kMacIcelandicTrueTypeVariant = 1

    /* Variants of Unicode & ISO 10646 encodings */
    kUnicodeNoSubset = 0,
    kUnicodeNoCompatibilityVariant = 1,
    kUnicodeMaxDecomposedVariant = 2,
    kUnicodeNoComposedVariant = 3,
    kUnicodeNoCorporateVariant = 4,
};
```

Constant descriptions

`kTextEncodingDefaultVariant`

The standard default variant for any base encoding.

Mac OS Japanese variants

`kMacJapaneseStandardVariant`

The standard Japanese variant. Shift-JIS with JIS Roman modifications, extra 1-byte characters, 2-byte Apple extensions, and some vertical presentation forms in the range 0xEB40—0xEDFE ("ku plus 84").

`kMacJapaneseStdNoVerticalsVariant`
An artificial variant for callers who don't want to use separately encoded vertical forms (for example, developers using QuickDraw GX).

`kMacJapaneseBasicVariant`
An artificial variant without Apple 2-byte extensions.

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

`kMacJapanesePostScriptPrintVariant`
The Japanese variant for PostScript printing versions of the Sai Mincho and Chu Gothic PostScript fonts. This version includes 2-byte half-width characters in addition to 1-byte half-width characters.

`kMacJapaneseVertAtKuPlusTenVariant`
The Japanese variant for the Hon Mincho and Maru Gothic fonts used in the Japanese localized version of System 7.1. It does not include the standard Apple extensions, and encodes vertical forms at a different location.

Japanese options

`kJapaneseNoOneByteKanaOption`
This variant indicates that the JIS X0201 Kana values should be excluded from the mapping. These characters are 1-byte characters in JIS X0201 and in Shift-JIS (although not in EUC-JP); they are often referred to as half-width kana (although some Shift-JIS versions, including the Mac OS PostScript variants, have both 1-byte and 2-byte half-width Kana). These characters are often excluded when interchanging Japanese text on the Internet. This variant is not currently supported.

`kJapaneseUseAsciiBackslashOption`
This governs the interpretation of the 1-byte code point 0x5C. This is REVERSE SOLIDUS in ASCII, but YEN SIGN in canonical JIS Roman, JIS X0201, Shift-JIS, and so on. However, when these Japanese encodings are used for file names on systems that treat REVERSE SOLIDUS as a path separator, the 1-byte code 0x5C is usually interpreted as

REVERSE SOLIDUS and should be mapped as such. This variant is not currently supported.

Mac OS Arabic variants

`kMacArabicStandardVariant`
This variant is supported by the Cairo font (the system font for Arabic) and is the encoding supported by the text processing utilities.

`kMacArabicTrueTypeVariant`
This variant is used for most of the Arabic TrueType fonts: Baghdad, Geeza, Kufi, Nadeem.

`kMacArabicThuluthVariant`
This variant is used for the Arabic PostScript-only fonts: Thuluth and Thuluth bold.

`kMacArabicAlBayanVariant`
This variant is used for the Arabic TrueType font Al Bayan.

Mac OS Farsi variants

`kMacFarsiStandardVariant`
This variant is supported by the Tehran font (the system font for Farsi) and is the encoding supported by the text processing utilities.

`kMacFarsiTrueTypeVariant`
This variant is used for most of the Farsi TrueType fonts: Ashfahan, Amir, Kamran, Mashad, NadeemFarsi.

Mac OS Hebrew variants

`kMacHebrewStandardVariant`
The standard Hebrew variant.

`kMacHebrewFigureSpaceVariant`
The Hebrew variant in which 0xD4 represents figure space, not left single quotation mark as in the standard variant.

Mac OS Icelandic variants

`kMacIcelandicStandardVariant`
The Standard Icelandic encoding supported by the bitmap versions of Chicago, Geneva, Monaco, and New York in the Icelandic system. This is also the variant supported by the text processing utilities.

`kMacIcelandicTrueTypeVariant`
: The variant used for the bitmap versions of Courier, Helvetica, Palatino, and Times in the Icelandic system, and for the TrueType versions of Chicago, Geneva, Monaco, New York, Courier, Helvetica, Palatino, and Times.

Unicode variants

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

`kUnicodeNoCompatibilityVariant`
: An artificial Unicode variant that does not include encoded characters belonging to the Compatibility Zone. This variant is not currently supported.

`kUnicodeMaxDecomposedVariant`
: This variant is not currently supported.

`kUnicodeNoComposedVariant`
: An artificial Unicode variant that does not include encoded characters that are composed characters. This variant is not currently supported.

`kUnicodeNoCorporateVariant`
: An artificial Unicode variant that does not allow use of characters in the Corporate Use Zone. This variant is not currently supported.

Text Encoding Format

A text encoding format specifies a way of formatting or algorithmically transforming a particular base encoding. For example, the UTF-7 format is the Unicode standard formatted for transmission through channels that can handle only 7-bit values. Other text encoding formats for Unicode include UTF-8 and 16-bit or 32-bit formats. These transformations are not viewed as different base encodings. Rather, they are different formats for representing the same base encoding.

Similar to text encoding variant values, text encoding format values are specific to a particular text encoding base value or to a small set of text encoding base values. A text encoding format is defined by the `TextEncodingFormat` data type.

```
typedef UInt32 TextEncodingFormat;
```

The function `GetTextEncodingFormat` (page 52) returns the text encoding format of a text encoding specification.

The following enumeration defines constants for specifying text encoding formats:

```
enum {
    /* Default TextEncodingFormat for any TextEncodingBase */
    kTextEncodingDefaultFormat  = 0,

    /* Formats for Unicode encodings */
    Unicode16BitFormat          = 0,
    kUnicodeUTF7Format          = 1,
    kUnicodeUTF8Format          = 2,
    kUnicode32BitFormat         = 3
};
```

constant descriptions

`kTextEncodingDefaultFormat`

The standard default format for any base encoding.

For Unicode and ISO10646

`kUnicode16BitFormat`

The 16-bit character encoding format specified by the Unicode standard, equivalent to the UCS-2 format for ISO 10646. This includes support for the UTF-16 method of including non-BMP characters in a stream of 16-bit values.

`kUnicodeUTF7Format`

The Unicode transformation format in which characters encodings are represented by a sequence of 7-bit values. This format cannot be handled by the Unicode Converter, only by the Text Encoding Converter.

`kUnicodeUTF8Format`

The Unicode transformation format in which characters are represented by a sequence of 8-bit values.

`kUnicode32BitFormat`

The UCS-4 32-bit format defined for ISO 10646. This format is not currently supported.

Text Encoding Name Selector

You use a selector for the `GetTextEncodingName` function to indicate which part of an encoding name you want to determine. The text encoding name selector is defined by the `TextEncodingNameSelector` data type:

```
typedef UInt32 TextEncodingNameSelector;
```

The following enumeration defines the allowable constants for selecting parts of encoding names:

```
enum {
    kTextEncodingFullName       = 0,
    kTextEncodingBaseName       = 1,
    kTextEncodingVariantName    = 2,
    kTextEncodingFormatName     = 3
};
```

Constant descriptions

`kTextEncodingFullName`
Selector that requests the full name of the text encoding.

`kTextEncodingBaseName`
Requests the name of the base encoding.

`kTextEncodingVariantName`
Requests the name of the encoding variant, if available.

`kTextEncodingFormatName`
Requests the name of the encoding format, if available.

Script Manager Derivation Specifiers

For backward compatibility with earlier releases of the Mac OS, the Text Encoding Conversion Manager provides the functions `UpgradeScriptInfoToTextEncoding` (page 56) and `RevertTextEncodingToScriptInfo` (page 58) that you can use to derive Script Manager values from a text encoding or vice versa.

When using these functions, you can specify a Script Manager language code, script code, and/or font values to derive a text encoding. These three constants are defined to allow you to identify any part of the derivation you don’t care about. When reverting from a text encoding to Script Manager values, the Unicode Converter returns these constants for a corresponding value it does not derive: `kTextLanguageDontCare`, `kTextScriptDontCare`, and `kTextRegionDontCare`.

```
enum {
    kTextLanguageDontCare   = -128
    kTextScriptDontCare     = -128
    kTextRegionDontCare     = -128
    );
```

Constant descriptions

`kTextLanguageDontCare`
Indicates that language code is not provided for the derivation.

`kTextScriptDontCare`
Indicates that the code is not provided for the derivation.

`kTextRegionDontCare`
The region code is not provided for the derivation.

Text Encoding Conversion Manager Result Codes

Many of the Text Encoding Conversion Manager functions return result codes. This section includes result codes that are common to both converters and those that are specific to one or the other. This section explains briefly the conditions under which result codes are returned. Some functions that return a result code include descriptions that give result code information specific to the function. Chapter 3, “Text Encoding Converter Reference,” and Chapter 4, “Unicode Converter Reference,” describe these functions.

Text Encoding Conversion Manager functions can return result codes specific to text encoding conversions 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.

`kTextUnsupportedEncodingErr`	-8738	The encoding or mapping is not supported for this function by the current set of tables or plug-ins.
`kTextMalformedInputErr`	-8739	The text input contains a sequence that is not legal in the specified encoding, such as a DBCS high byte followed by an invalid low byte (0x8120 in Shift-JIS).
`kTextUndefinedElementErr`	-8740	The text input contains a code point that is undefined in the specified encoding.
`kTECMissingTableErr`	-8745	The specified encoding is partially supported, but a specific table required for this function is missing.
`kTECTableChecksumErr`	-8746	A specific table required for this function has a checksum error.
`kTECTableFormatErr`	-8747	The table format is either invalid or it cannot be handled by the current version of the code.
`kTECCorruptConverterErr`	-8748	The converter object is invalid. Returned by the Text Encoding Converter functions only.
`kTECNoConversionPathErr`	-8749	The converter supports both the source and target encodings, but cannot convert between them either directly or indirectly. Returned by the Text Encoding Converter functions only.
`kTECBufferBelowMinimumSizeErr`	-8750	The output text buffer is too small to accommodate the result of processing of the first input text element.
`kTECArrayFullErr`	-8751	The supplied `TextEncodingRun`, `ScriptCodeRun`, or `UnicodeMapping` array is too small.
`kTECPartialCharErr`	-8753	The input text ends in the middle of a multibyte character and conversion stopped.
`kTECUnmappableElementErr`	-8754	An input text element cannot be mapped to the specified output encoding(s) using the specified options. For the Unicode Converter, this error can occur only if `kUnicodeUseFallbacksBit` is not set.

kTECIncompleteElementErr	-8755	The input text ends with a text element that might be incomplete, or contains a text element that is too long for the internal buffers.
kTECDirectionErr	-8756	An error, such as a direction stack overflow, occurred in directionality processing.
kTECGlobalsUnavailableErr	-8770	Global variables have already been deallocated, premature termination.
kTECItemUnavailableErr	-8771	An item (for example, a name) is not available for the specified region (and encoding, if relevant).
kTECUsedFallbacksStatus	-8783	The function has completely converted the input string to the specified target using one or more fallbacks. For the Unicode Converter, this status code can only occur if kUnicodeUseFallbacksBit is set.
kTECNeedFlushStatus	-8784	The application disposed of a converter object by calling TECDisposeConverter, but there is still text contained in internal buffers. Returned by the Text Encoding Converter functions only.
kTECOutputBufferFullStatus	-8785	The converter successfully converted part of the input text, but the output buffer was not large enough to accommodate the entire input text after conversion. Convert the remaining text beginning from the position where the conversion stopped.

Basic Text Structures and Other Types

TextEncoding

A **text encoding specification** is a set of numeric codes used to identify text encodings. It specifies the base text encoding, the text encoding variant, and the text encoding format. It is used, for example, to identify the encoding of text passed to a text converter. Two such specifications are needed—for source and destination encoding—when calling the Text Encoding Converter or the Unicode Converter to convert text.

You can use these data types when you create a text encoding specification:

- `TextEncodingBase`, described in "Text Encoding Base" (page 31)
- `TextEncodingVariant`, described in "Text Encoding Variant" (page 35)
- `TextEncodingFormat`, described in "Text Encoding Format" (page 40)

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

```
typedef UInt32 TextEncoding;
```

TextEncodingRun

It is not always possible to convert text expressed in Unicode to another single encoding because no other single encoding encompasses the Unicode character encoding range. To adjust for this, you can create a Unicode mapping structure array that specifies the target encodings the Unicode text should be converted to when multiple encodings must be used.

If the `kUnicodeTextRunMask` flag is set, `ConvertFromUnicodeToTextRun` and `ConvertFromUnicodeToScriptCodeRun` may convert Unicode text to a string of text containing multiple text encoding runs. Each run contains text expressed in a different encoding from that of the preceding or following text segment. For each text encoding run in the string, a `TextEncodingRun` structure indicates the beginning offset and the text encoding for that run.

Functions that convert text from Unicode to a text run return the converted text in an array of text encoding run structures. A text encoding run structure is defined by the `TextEncodingRun` data type.

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

Field descriptions

`offset` The beginning character position of a run of text in the converted text string.

`textEncoding` The encoding of the text run that begins at the position specified.

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

Field descriptions

`offset` The byte offset at which the given text encoding begins. The offset is from the beginning of the text buffer.

`textEncoding`

The text encoding that begins at the byte offset.

TECInfo

The converter information structure is used by the function `TECGetInfo` (page 55) to hold returned information about the Unicode Converter, the Text Encoding Converter, and Basic Text Types.

```
struct TECInfo {
    UInt16  format;
    UInt16  tecVersion;
    UInt32  tecTextConverterFeatures;
    UInt32  tecUnicodeConverterFeatures;
    UInt32  tecTextCommonFeatures;
    Str31   tecTextEncodingsFolderName;
    Str31   tecExtensionFileName;
};
typedef struct TECInfo TECInfo;
typedef TECInfo * TECInfoPtr;
typedef TECInfoPtr * TECInfoHandle;
```

```
enum {
    kTECInfoCurrentFormat = 1
};

enum {
    kTECKeepInfoFixBit = 0,
    kTECFallbackTextLengthFixBit = 1
};

enum {
    kTECKeepInfoFixMask = 1L << kTECKeepInfoFixBit,
    kTECFallbackTextLengthFixMask = 1L << kTECFallbackTextLengthFixBit
};
```

Field descriptions

`format`
: The current format of the returned structure. The format of the structure is indicated by the `kTECInfoCurrentFormat` constant. Any future changes to the format will always be backward compatible; any new fields will be added to the end of the structure.

`tecVersion`
: The current version of the Text Encoding Conversion Manager extension in BCD (binary coded decimal), with the first byte indicating the major version; for example, 0x0121 for 1.2.1.

`tecTextConverterFeatures`
: New features or bug fixes in the Text Encoding Converter. No bits are currently defined.

`tecUnicodeConverterFeatures`
: Bit flags indicating new features or bug fixes in the Unicode Converter. The bits and their masks currently defined are `kTECKeepInfoFixBit`, `kTECFallbackTextLengthFixBit`, `kTECKeepInfoFixMask`, `kTECFallbackTextLengthFixMask`.

  The `kTECKeepInfoFixBit` is set if the Unicode Converter has a bug fix to stop ignoring certain control flags if the `kUnicodeKeepInfoBit` flag is set.

  The `kTECFallbackTextLengthFixBit` is set if the Unicode Converter has a bug fix to use the source length (`srcConvLen`) and destination length (`destConvLen`) returned by a caller-supplied fall-back handler for any status it returns except `kTECUnmappableElementErr`. Previously it honored only these values if `noErr` was returned.

`tecTextCommonFeatures`
Bit flags indicating new features or bug fixes in Basic Text Types (the Text Common static library). No bits are currently defined.

`tecTextEncodingsFolderName`
A Pascal string with the (possibly localized) name of the Text Encodings folder.

`tecExtensionFileName`
A Pascal string with the (possibly localized) name of the Text Encoding Conversion Manager extension file.

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;
```

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

```
typedef UniChar *UniCharArrayPtr;
```

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;
```

Basic Text Functions

You must not directly modify text encoding specifications. Instead, the Mac OS provides you with functions for creating them, modifying them, and obtaining their contents.

Creating a Text Encoding Specification

CreateTextEncoding

Creates and returns a text encoding specification.

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

`encodingBase` A base text encoding of type `TextEncodingRun` (page 46).

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

`encodingFormat`
A format for the base text encoding. To specify the default format for the base encoding, you can use the `kTextEncodingDefaultFormat` constant.

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

DISCUSSION

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 to the functions that use text encodings.

SEE ALSO

The data type `TextEncodingRun` (page 46)

"Text Encoding Variant" (page 35)

"Text Encoding Format" (page 40)

Obtaining Information From a Text Encoding Specification

You use the functions described in this section to obtain the contents of a text encoding specification; you must not access the structure directly.

GetTextEncodingBase

Returns the base encoding of the specified text encoding.

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

`encoding` A text encoding specification whose base encoding you want to obtain.

function result The base encoding portion of the specified text encoding.

SEE ALSO

"Text Encoding Base" (page 31)

The data type `TextEncodingRun` (page 46)

GetTextEncodingVariant

Returns the variant from the specified text encoding.

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

`encoding` A text encoding specification containing the variant you want to obtain.

function result The text encoding variant portion of the specified text encoding.

SEE ALSO

"Text Encoding Base" (page 31)

"Text Encoding Variant" (page 35)

GetTextEncodingFormat

Returns the format value of the specified text encoding.

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

`encoding` A text encoding specification containing the text encoding format you want to obtain.

function result The text encoding format value contained in the text encoding you specified.

SEE ALSO

"Text Encoding Base" (page 31)

"Text Encoding Format" (page 40)

ResolveDefaultTextEncoding

Returns a text encoding specification in which any meta-values have been resolved to real values. Currently, this affects only the base encoding values packed into the text encoding specification.

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

`encoding` A text encoding specification possibly containing meta-values that you want to resolve into a text encoding specification containing only real values.

function result A text encoding specification containing only real base encoding values.

DISCUSSION

This function is useful for application developers who are providing APIs that take text encoding specifications as parameters. All APIs in the Unicode Converter and Text Encoding Converter perform this translation automatically.

SEE ALSO

"Text Encoding Base" (page 31)

GetTextEncodingName

Returns the localized name for a specified text encoding.

```
OSStatus GetTextEncodingName (TextEncoding iEncoding,
                    TextEncodingNameSelector iNamePartSelector,
                    RegionCode iPreferredRegion,
                    TextEncoding iPreferredEncoding,
                    ByteCount iOutputBufLen,
                    ByteCount *oNameLength,
                    RegionCode *oActualRegion,
                    TextEncoding *oActualEncoding,
                    TextPtr oEncodingName);
```

`iEncoding` A text encoding specification whose name you want to obtain.

`iNamePartSelector`
The portion of the encoding name you want to obtain. See “Text Encoding Name Selector” (page 42) for a list of possible values.

`iPreferredRegion`
The preferred region to use for the name. You can specify a Mac OS region code (which also implies a language) for this parameter. If the function cannot return the name for the preferred region, it returns the name using a region code with the same language or in a default language (for example, English).

`iPreferredEncoding`
The preferred encoding to use for the name. For example, you might want the name returned encoded in ASCII, Mac OS Roman, or Shift-JIS. If the function cannot return the name using the preferred encoding, it returns the name using another encoding, such as Unicode or ASCII.

`iOutputBufLen` The length in bytes of the output buffer that your application provides for the returned encoding name.

`oNameLength` A pointer to a value of type `ByteCount`. On output, this parameter holds the actual length, in bytes, of the text encoding name. The value represents the full length of the name, which might be greater than the size of the output buffer, specified by the `iOutputBufLen` parameter. The length of the portion of the name actually contained in the output buffer is thus the smaller of `oNameLength` and `iOutputBufLen`.

`oActualRegion` A pointer to a value of type `RegionCode`. On output, this parameter holds the actual region associated with the returned encoding name.

`oActualEncoding`
A pointer to a value of type `TextEncoding`. On output, this parameter holds the actual encoding associated with the returned encoding name.

`oEncodingName` A pointer to a buffer you provide. On output, this parameter holds the text encoding name.

function result A result code. For a list of possible result codes, see "Text Encoding Conversion Manager Result Codes" (page 43).

DISCUSSION

In addition to various resource and memory errors, this function can return the following result codes:

- `kTextUnsupportedEncodingErr`, which indicates that the encoding whose name you want to obtain is not supported.
- `kTECMissingTableErr`, which indicates the name resource associated with the encoding is missing.
- `kTECTableFormatErr` or `kTECTableCheckSumErr`, which indicates that the name resource associated with that encoding is invalid.

SEE ALSO

"Text Encoding Base" (page 31)

Obtaining Converter Information

TECGetInfo

Allocates a converter information structure of type `TECInfo` in the application heap using `NewHandle`, fills it out, and returns a handle.

```
pascal OSStatus TECGetInfo (TECInfoHandle *tecInfo);
```

`tecInfo` A handle to a structure of type `TECInfo` (page 47) containing information about the converter.

function result A result code. This function can return memory errors. For a list of other possible result codes, see "Text Encoding Conversion Manager Result Codes" (page 43).

DISCUSSION

When you are finished with the handle, your application must dispose of it using `DisposeHandle`. You must also perform any required preflighting or memory rearrangement before calling `TECGetInfo`.

Converting Between Script Manager Values and Text Encodings

UpgradeScriptInfoToTextEncoding

Converts any combination of a Mac OS script code, a language code, a region code, and a font name to a text encoding.

```
pascal OSStatus UpgradeScriptInfoToTextEncoding (
      ScriptCode iTextScriptID,
      LangCode iTextLanguageID,
      RegionCode iRegionID,
      ConstStr255Param iTextFontname,
      TextEncoding *oEncoding);
```

`iTextScriptID` A valid Script Manager script code. The Mac OS 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 graphics port (`grafPort`), specify the metavalue of `smCurrentScript`. To indicate that you do not want to provide a script code for this parameter, specify the constant `kTextScriptDontCare`. See *Inside Macintosh: Text* for more information on the Script Manager's script codes, language codes, and region codes.

`iTextLanguageID`

A valid Script Manager language code. The Mac OS 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`.

`iRegionID` A valid Script Manager region code. The Mac OS 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`.

`iTextFontname` The name of a font associated with a particular text encoding specification, such as Symbol or Zapf Dingbats, or the name of any 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`.

`oEncoding` 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.

function result A result code. This function returns `paramErr` if two or more of the input parameter values conflict in some way—for example, the Mac OS 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 `kTECTableFormatErr` result code if the internal mapping tables used for translation are invalid. For a list of other possible result codes, see "Text Encoding Conversion Manager Result Codes" (page 43).

DISCUSSION

The `UpgradeScriptInfoToTextEncoding` function allows you to derive a text encoding specification from script codes, language codes, region codes, and font names. A one-to-one correspondence exists between many of the Script Manager's script codes and a particular Mac OS text encoding base value. However, because text encodings are a superset 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 script, language, or region code parameter, you must pass the don't-care constant appropriate to that parameter. If you don't specify an explicit value for `iTextFontName`, you must pass `NULL`. `UpgradeScriptInfoToTextEncoding` uses 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 are checked against one another to ensure that they are valid in combination.

A font name, such as 'Symbol' or 'Zapf Dingbats', can indicate a particular text encoding base. Other font names can indicate particular variants associated with a particular text encoding base. Otherwise, the font name is used to obtain a script code, and this script code will be checked against any script code you supply (in this case, the font must be installed; if it is not, the function returns a paramErr result code). If you do not supply either a language code or a region code and the script code you supply or the one that is derived matches the system script, then the system's localization is used to determine the appropriate region and language code. This is used for deriving text encoding base values that depend on region and language, such as kTextEncodingMacTurkish.

SEE ALSO

The function RevertTextEncodingToScriptInfo (page 58)

"Text Encoding Base" (page 31)

RevertTextEncodingToScriptInfo

Converts the given Mac OS text encoding specification to the corresponding script code and, if possible, language code and font name.

```
pascal OSStatus RevertTextEncodingToScriptInfo (
      TextEncoding iEncoding,
      ScriptCode *oTextScriptID,
      LangCode *oTextLanguageID
      Str255 oTextFontname);
```

iEncoding The text encoding specification to be converted.

oTextScriptID A pointer to a value of type ScriptCode. On output, a Mac OS script code that corresponds to the text encoding specification you identified in the iEncoding parameter. If you do not pass a pointer for this parameter on input, the function returns a paramErr result code.

`oTextLanguageID`
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`.

`oTextFontname` 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 `paramErr` if the text encoding specification input parameter value is invalid. The function returns a `kTECTableFormatErr` result code if the internal mapping tables used for translation are invalid. For a list of other possible result codes, see “Text Encoding Conversion Manager Result Codes” (page 43).

DISCUSSION

If you have applications that use Mac OS Script Manager and Font Manager functions, you can use the `RevertTextEncodingToScriptInfo` function to convert information in a text encoding specification into the appropriate Mac OS script code, language code, and font name, if they can be unambiguously derived. Your application can then use this information to display text to a user on the screen.

SEE ALSO

The function `UpgradeScriptInfoToTextEncoding` (page 56)

"Text Encoding Base" (page 31)

Text Encoding Converter Reference

Contents

Text Encoding Converter Reference

This chapter describes the types, constants, and functions pertaining to the Text Encoding Converter.

For a description of types, constants, and functions that pertain to text encodings in general, see Chapter 2, “Basic Text Types Reference.” For a description of types, constants, and functions pertaining to the Unicode Converter, see Chapter 4, “Unicode Converter Reference.”

Chapter Overview

The Text Encoding Converter provides conversion between any two text encodings. This involves a combination of the following techniques:

- Table-based conversion to or from Unicode, using the Unicode Converter. A single X-to-Y conversion may involve converting X to Unicode, and then converting Unicode to Y. Intermediate storage for the Unicode form is handled by the Text Encoding Converter.
- Algorithmic conversion, using plug-in code modules. These plug-ins are implemented as code fragments.
- Maintaining and updating the current state (the current encoding and other relevant information) for multiple-encoding streams. It also handles detecting escape sequences, special control characters, and any other tags that change the current encoding state. This information is stored in the converter object and maintained by the plug-ins performing the conversion.

Conversion routines are handled using plug-in components implemented as code fragments. The main export symbol of each fragment is a routine that returns a pointer to a table containing a plug-in signature, table version information, and hooks to each of the plug-ins functions. Each plug-in can be polled for the encodings it supports and is responsible for handling all conversions between its supported encodings. The Text Encoding Converter decides how best to meet a caller’s conversion requirements using the conversion resources available to it. A conversion may involve combining several conversions in succession. The caller is shielded from this complexity and treats an encoding converter object as a single entity regardless of its actual structure.

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

Text Encoding Converter Constants

Text Encoding Converter Result Codes

Many of the Text Encoding Converter functions return result codes. These codes are listed in "Text Encoding Conversion Manager Result Codes" (page 43).

Text Encoding Converter Structures and Other Types

TECObjectRef

When making a text conversion, the Text Encoding Converter requires a reference to a converter object that indicates how to accomplish the conversion. Functions, such as `TECCreateConverter` (page 91), that create a converter object return this reference, which you can then pass to other functions when converting text. A converter object reference is defined by the `TECObjectRef` data type:

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

The structure of the `OpaqueTECObjectRef` data type is private, and a converter object is not accessible directly.

TECConversionInfo

When you call the function TECGetDirectTextEncodingConversions (page 72), you pass an array of text encoding conversion information structures. The function fills these structures 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 The text encoding specification for the source text.

destinationEncoding The text encoding specification for the destination text.

reserved1 Reserved.

reserved2 Reserved.

TECSnifferObjectRef

When analyzing text for possible encodings, the Text Encoding Converter requires a reference to a sniffer object that specifies what types of encodings can be detected. You receive this reference when calling the function TECCreateSniffer (page 85). A sniffer object reference is defined by the TECSnifferObjectRef data type:

```
typedef struct OpaqueTECSnifferObjectRef* TECSnifferObjectRef;
```

The structure of the OpaqueTECObjectRef data type is private, and a sniffer object is not accessible directly.

Text Encoding Converter Functions

Obtaining Information About Available Text Encodings

The number and kind of text encodings that the Text Encoding Converter supports depend on the conversion plug-ins currently installed in the user's system. Conversion plug-ins support conversion between encodings and are installed in the Text Encodings folder within the System Folder. For information about writing plug-ins, see Appendix A.

TECCountAvailableTextEncodings

Counts and returns the number of text encodings the Text Encoding Converter supports.

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

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error condition when polled by the Text Encoding Converter.

DISCUSSION

The `TECCountAvailableTextEncodings` function counts and returns the number of text encodings that you can use to perform conversions based on the current configuration of the Text Encoding Converter. This number indicates what size array you must allocate in a parameter of the function `TECGetAvailableTextEncodings` (page 67). Therefore, you should call this function before you call TECGetAvailableTextEncodings in order 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, this function includes each instance of the text encoding in its sum. Consequently, the same text encoding may be counted more than once. For example, the Japanese Encodings plug-in supports Mac OS Japanese, and so does the Unicode Encodings plug-in. However, since the TECGetAvailableTextEncodings function does not return duplicate text encoding specifications, TECCountAvailableTextEncodings may return a number greater than the number of array elements required.

TECGetAvailableTextEncodings

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

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

availableEncodings[]

An array of text encoding specifications. On output, the TECGetAvailableTextEncodings function fills the array with the specifications for the text encodings the Text Encoding Converter currently supports. To determine how large an array to allocate, use the function TECCountAvailableTextEncodings (page 66).

maxAvailableEncodings

The number of text encoding specifications the availableEncodings array can contain.

actualAvailableEncodings

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

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. 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 66). `TECGetAvailableTextEncodings` tells you the number of specifications it returns in the `actualAvailableEncodings` parameter.

TECCountSubTextEncodings

Counts and returns the number of subencodings a text encoding supports.

```
pascal OSStatus TECCountSubTextEncodings (
                TextEncoding inputEncoding,
                ItemCount *numberEncodings);
```

`inputEncoding`
The text encoding specification containing the subencodings.

`numberEncodings`
A pointer to a value of type `ItemCount`. On output, this value indicates the number of currently supported subencodings.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

The `TECCountSubTextEncodings` function counts and returns the number of subencodings that you can use to perform conversions based on the current configuration of the Text Encoding Converter. Subencodings are text encodings that are embedded as part of a larger text encoding specification. For example, EUC-JP contains JIS Roman or ASCII, JIS X0208, JIS X0212, and half-width Katakana from JIS X0201. Not every encoding that can be broken into multiple encodings necessarily supports this routine. It's up to the plug-in developer to decide which encodings might be useful to break up. Subencodings are *not* the same as text encoding variants.

The `numberEncodings` value returned tells you what size array you must allocate in a parameter of the function `TECGetSubTextEncodings` (page 69). Therefore, you should call this function before you call `TECGetSubTextEncodings` in order to accommodate the specifications for all of these text encodings.

If an encoding can be converted to multiple runs of encodings (as indicated by a destination base encoding of `kTextEncodingMultiRun`), you can call the function `TECGetSubTextEncodings` to get the list of output encodings. See the descriptions of the `TECCreateOneToManyConverter` (page 99) and `TECGetDestinationTextEncodings` (page 74) for information about multiple output encoding run conversions.

TECGetSubTextEncodings

Returns the text encoding specifications for the subencodings the encoding scheme supports.

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

`inputEncoding`
The text encoding specification containing the subencodings.

`subEncodings[]`
An array composed of text encoding specifications. On return, the `TECGetSubTextEncodings` function fills the array with the specifications for the text encodings, which are subencodings of the encoding specified in the `inputEncoding` parameter, given

the current configuration of the Text Encoding Converter. To determine how large an array to allocate, use the function `TECCountSubTextEncodings` (page 68).

`maxSubEncodings`
The number of text encoding specifications the `subEncodings` array can contain.

`actualSubEncodings`
A pointer to a value of type `ItemCount`. On output, this value indicates the number of subencodings the function returned in the `subEncodings` array.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

The `TECGetSubTextEncodings` function returns the text encoding specifications in the array you pass to the function as the `subEncodings` parameter. Subencodings are text encodings that are embedded as part of a larger text encoding specification. For example, EUC-JP contains JIS Roman or ASCII, JIS X0208, JIS X0212, and half-width Katakana from JIS X0201. Not every encoding that can be broken into multiple encodings necessarily supports this routine. It's up to the plug-in developer to decide which encodings might be useful to break up. Subencodings are *not* the same as text encoding variants.

If an encoding can be converted to multiple runs of encodings (as indicated by a destination base encoding of `kTextEncodingMultiRun)`, you can call the `TECGetSubTextEncodings` (page 69) function to get the list of output encodings. See the `TECCreateOneToManyConverter` (page 99) and `TECGetDestinationTextEncodings` (page 74) functions for information about multiple output encoding run conversions.

Identifying Direct Encoding Conversions

A direct conversion is one that can convert from a source encoding to a destination encoding in one step. Generally this means that a plug-in must be available that handles that specific conversion.

If more than one step is needed, the Text Encoding Converter can convert between any two encodings by using the available direct conversions to make intermediate conversions. The direct conversions available depend on the conversion plug-ins currently installed on the user's system.

TECCountDirectTextEncodingConversions

Counts and returns the number of direct conversions that the Text Encoding Converter supports in its current configuration.

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

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

You use the number that `TECCountDirectTextEncodingConversions` returns to determine how large to make the array you pass to the function `TECGetDirectTextEncodingConversions` (page 72).

`TECCountDirectTextEncodingConversions` counts each instance of an available conversion. That is, if different conversion plug-ins support the same direct conversion, this function includes each instance of the direct conversion in its sum. Consequently, the same direct conversion may be counted more than once. Because the `TECGetDirectTextEncodingConversions` (page 72) function does not return duplicate direct conversions, `TECCountDirectTextEncodingConversions` may return a number greater than the number of array elements required.

TECGetDirectTextEncodingConversions

Returns the types of direct conversions the Text Encoding Converter supports in its current configuration.

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

`directConversions[]`
An array composed of text encoding conversion information structures, each of which specifies a set of source and destination encodings. On return, each structure indicates one type of conversion the Text Encoding Converter supports. See `TECConversionInfo` (page 65) for more information. To determine how large an array to allocate, use the function `TECCountDirectTextEncodingConversions` (page 71).

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

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

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter

DISCUSSION

The `TECGetDirectTextEncodingConversions` function returns the text encoding specifications in the array you pass to the function as the `directConversions` parameter, eliminating any duplicate information in the process. Consequently, the number of encodings `TECGetDirectTextEncodingConversions` returns in the available encodings array may be fewer than the number of elements you allocated for the array based on your call to `TECCountDirectTextEncodingConversions` (page 71).

TECGetDirectTextEncodingConversions tells you the number of specifications it returns in the actualDirectConversions parameter.

Identifying Possible Destination Encodings

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

TECCountDestinationTextEncodings

Counts and returns the number of destination encodings possible for the specified source encoding using a single-step, direct conversion.

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

inputEncoding
The text encoding specification describing the source text.

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than noErr, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter

DISCUSSION

The TECCountDestinationTextEncodings function returns the number of direct conversions possible from the specified source encoding to any supported destination encodings. A direct encoding conversion consists of a conversion from the source encoding to a destination encoding without any intermediate conversions (that is, only one plug-in conversion function needs to be called). For example, suppose Mac OS Japanese is the source encoding. If a plug-in contains a Mac OS Japanese to ISO 2022-JP function, then conversion from Mac OS Japanese to ISO 2022-JP can be a direct (that is, a one-step) conversion. However, if no such function exists, the conversion must take place indirectly (for example, from Mac OS Japanese to EUC-JP and then from EUC-JP to ISO 2022-JP).

You can use the number that this function returns to determine how many text encoding specification elements to allocate for the array you pass to the function `TECGetDestinationTextEncodings` (page 74).

`TECCountDestinationTextEncodings` 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, this function includes each instance of the text encoding in its sum. Consequently, the same text encoding may be counted more than once. Since the `TECGetDestinationTextEncodings` function does not return duplicate text encoding specifications, `TECCountDestinationTextEncodings` may return a number greater than the number of array elements required.

TECGetDestinationTextEncodings

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

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

`inputEncoding`

The text encoding specification describing the source text.

`destinationEncodings[]`
An array of text encoding specifications. On return, this function fills the array elements with specifications for the destination encodings to which the converter can directly convert the source encoding given in the `inputEncoding` parameter. Your application allocates memory for this array to accommodate the encodings that this function returns. To determine how large an array to allocate, use the function `TECCountDestinationTextEncodings` (page 73).

`maxDestinationEncodings`
The maximum number of destination text encodings that the array can contain.

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

The `TECGetDestinationTextEncodings` function returns text encoding specifications for the possible destination encodings in the array you pass as the `directConversions` parameter, eliminating any duplicate information in the process. Consequently, the number of encodings `TECGetDestinationTextEncodings` returns in the available encodings array may be fewer than the number of elements you allocated for the array based on your call to the function `TECCountDestinationTextEncodings` (page 73). `TECGetDestinationTextEncodings` tells you the number of specifications it returns in the `actualDestinationEncodings` parameter.

You can display the names of these destination encodings to the user if desired.

Internet and Regional Text Encoding Names

The Internet has its own set of encoding names, defined in the Internet Assigned Numbers Authority (IANA) registry, that identify text encodings for

HTML, Web text, and Internet mail. The Text Encoding Converter uses numeric text encoding specifications. However, the Text Encoding Converter provides functions that let you translate between text encoding specifications and Internet name strings. In addition, the Text Encoding Converter also supplies functions that return a list of text encodings used for a particular language and geographic region.

TECGetTextEncodingFromInternetName

Returns the Mac OS text encoding specification that corresponds to the specified Internet encoding name.

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

`textEncoding` A pointer to a text encoding specification. On output, the structure contains the Mac OS text encoding specification that corresponds to the Internet name specified by the `encodingName` parameter.

`encodingName` A character string holding the Internet encoding name, in 7-bit US ASCII.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values.

DISCUSSION

The Internet uses names (strings) to identify Web or mail encodings, while the Text Encoding Converter uses numeric text encoding specifications. You use `TECGetTextEncodingFromInternetName` to obtain the text encoding specification for a text encoding whose Internet encoding name you provide. This function performs the opposite action of `TECGetTextEncodingInternetName` (page 77).

SEE ALSO

The function `GetTextEncodingName` (page 53)

TECGetTextEncodingInternetName

Returns the Internet encoding name that corresponds to the specified Mac OS text encoding.

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

`textEncoding` The text encoding specification for the encoding whose Internet encoding name you want to obtain.

`encodingName` The Internet encoding name. On return, this holds a character string in 7-bit US ASCII that represents the text encoding specified by the `textEncoding` parameter.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values.

DISCUSSION

If there are several Internet encoding names for the same text encoding, the function returns the preferred name. This function performs the opposite action of `TECGetTextEncodingFromInternetName` (page 76).

SEE ALSO

The function `GetTextEncodingName` (page 53)

TECCountWebTextEncodings

Counts and returns the number of currently supported text encodings for a specified region.

```
pascal OSStatus TECCountWebTextEncodings (
                        RegionCode locale,
                        ItemCount *numberEncodings);
```

`locale` A Mac OS region code indicating the locale for which you want to count encodings. A region code designates a combination of language, writing system, and geographic region; the region may not correspond to a particular country (for example, Swiss French or Arabic).

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

The `TECCountWebTextEncodings` function counts and returns the number of text encodings that you can use to perform conversions for the specified region. This number tells you what size array you must allocate in a parameter of the function `TECGetWebTextEncodings` (page 79). Therefore, you should call this function before you call `TECGetWebTextEncodings` in order to accommodate the specifications for all of these text encodings.

`TECCountWebTextEncodings` 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, this function includes each instance of the text encoding in its sum. Consequently, the same text encoding may be counted more than once. For example, the Japanese Encodings plug-in supports Mac OS Japanese and so does the Unicode Encodings plug-in. However, since the `TECGetWebTextEncodings` function does not return duplicate text encoding specifications, `TECCountWebTextEncodings` may return a number greater than the number of array elements required.

SEE ALSO

The function `TECCountAvailableTextEncodings` (page 66)

The function `TECGetAvailableTextEncodings` (page 67)

The function `TECGetTextEncodingFromInternetName` (page 76)

The function `GetTextEncodingName` (page 53)

The region codes section of "Script Manager" in *Inside Macintosh: Text*

TECGetWebTextEncodings

Returns the currently supported text encoding specifications for a specified region.

```
pascal OSStatus TECGetWebTextEncodings (
                    RegionCode locale,
                    TextEncoding availableEncodings[],
                    ItemCount maxAvailableEncodings,
                    ItemCount *actualAvailableEncodings);
```

`locale` A Mac OS region code indicating the locale for which you want to obtain encodings. A region code designates a combination of language, writing system, and geographic region; the region may not correspond to a particular country (for example, Swiss French or Arabic).

`availableEncodings[]`
An array of text encoding specifications. On return, the array contains specifications for the currently supported text encodings in the specified region. To determine how large an array to allocate, use the function `TECGetWebTextEncodings` (page 79).

`maxAvailableEncodings`
The number of text encoding specifications the `availableEncodings` array can contain.

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

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

For a specified Mac OS region code, `TECGetWebTextEncodings` fills in an array of type `TextEncoding` with a list of encodings commonly found on the World Wide Web for that region. The function eliminates any duplicate information in the process, so the number of encodings `TECGetWebTextEncodings` returns in the `availableEncodings` array may be fewer than the number of elements you allocated for the array based on your call to `TECCountWebTextEncodings` (page 77). `TECGetWebTextEncodings` tells you the number of specifications it returns in the `actualAvailableEncodings` parameter.

The list of available encodings could be used for an encoding selection menu found in many Web browsers.

SEE ALSO

The function `TECCountAvailableTextEncodings` (page 66)

The function `TECGetAvailableTextEncodings` (page 67)

The function `TECGetTextEncodingFromInternetName` (page 76)

The function `GetTextEncodingName` (page 53)

The region codes section of "Script Manager" in *Inside Macintosh: Text*

TECCountMailTextEncodings

Counts and returns the number of currently supported e-mail encodings for a specified region.

```
pascal OSStatus TECCountMailTextEncodings (
                    RegionCode locale,
                    ItemCount *numberEncodings);
```

`locale` A Mac OS region code indicating the locale for which you want to count encodings. A region code designates a combination of language, writing system, and geographic region; the region may not correspond to a particular country (for example, Swiss French or Arabic).

`numberEncodings`
: A pointer to a value of type `ItemCount`. On output, this value indicates the number of currently supported e-mail encodings for the region.

function result
: A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

The `TECCountMailTextEncodings` function counts and returns the number of e-mail text encodings currently available for a specified region. This number tells you what size array you must allocate in a parameter of the function `TECGetMailTextEncodings` (page 82). Therefore, you should call this function before you call `TECGetMailTextEncodings` in order to accommodate the specifications for all of these text encodings.

`TECCountMailTextEncodings` counts each instance of the same encoding. That is, if different conversion plug-ins support the same e-mail text encoding for any of the conversion processes they provide, this function includes each instance of the text encoding in its sum. Consequently, the same text encoding may be counted more than once. For example, the Japanese Encodings plug-in supports Mac OS Japanese and so does the Unicode Encodings plug-in. However, since the `TECGetMailTextEncodings` function does not return duplicate text encoding specifications, `TECCountMailTextEncodings` may return a number greater than the number of array elements required.

SEE ALSO

The function `TECCountAvailableTextEncodings` (page 66)

The function `TECGetAvailableTextEncodings` (page 67)

The function `TECCountWebTextEncodings` (page 77)

The function `TECGetTextEncodingFromInternetName` (page 76)

The function `TECGetWebTextEncodings` (page 79)

The function `GetTextEncodingName` (page 53)

The region codes section of "Script Manager" in *Inside Macintosh: Text*

TECGetMailTextEncodings

Returns the currently supported mail encoding specifications for a specified region.

```
pascal OSStatus TECGetMailTextEncodings (
                     RegionCode locale,
                     TextEncoding availableEncodings[],
                     ItemCount maxAvailableEncodings,
                     ItemCount *actualAvailableEncodings);
```

`locale` A Mac OS region code indicating the locale for which you want to obtain encodings. A region code designates a combination of language, writing system, and geographic region; the region may not correspond to a particular country (for example, Swiss French or Arabic).

`availableEncodings[]`
An array composed of text encoding specifications. On return, the array contains specifications for the currently supported e-mail text encodings in the specified region. To determine how large an array to allocate, use the function `TECCountMailTextEncodings` (page 80).

`maxAvailableEncodings`
The number of text encoding specifications the `availableEncodings` array can contain.

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

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled by the Text Encoding Converter.

DISCUSSION

The `TECGetMailTextEncodings` function returns the text encoding specifications currently supported from those that are commonly used for e-mail in the specified region. The text encoding specifications are returned in the array you pass to the function as the `availableEncodings` parameter, eliminating any duplicate information in the process. Consequently, the number of encodings `TECGetMailTextEncodings` returns in the `availableEncodings` array may be fewer than the number of elements you allocated for the array based on your call to `TECCountMailTextEncodings` (page 80).

SEE ALSO

The function `TECCountAvailableTextEncodings` (page 66)

The function `TECGetAvailableTextEncodings` (page 67)

The function `TECCountWebTextEncodings` (page 77)

The function `TECGetTextEncodingFromInternetName` (page 76)

The function `GetTextEncodingName` (page 53)

The region codes section of "Script Manager" in *Inside Macintosh: Text*

Investigating Encodings

Sniffer functions, optionally supported with plug-ins, check for unique or signifying characteristics that identify a particular text encoding. The Text Encoding Converter uses these functions to determine the most likely text encoding for a given piece of text.

TECCountAvailableSniffers

Counts and returns the number of sniffers available in all installed plug-ins.

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

`numberOfEncodings`

A pointer to a value of type `ItemCount`. On output, this value indicates the number of sniffers in all installed plug-ins.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled for available sniffers.

DISCUSSION

The `TECCountAvailableSniffers` function counts and returns the number of sniffers that you can use to perform a determination of the current text encoding. This number tells you what size array you must allocate in a parameter of the function `TECGetAvailableSniffers` (page 84).

`TECCountAvailableSniffers` counts each instance of the same sniffer. That is, if different conversion plug-ins support a sniffer for the same text encoding, this function includes each instance of the sniffer in its sum. Consequently, one type of sniffer may be counted more than once. However, since the `TECGetAvailableSniffers` function does not return duplicate text encoding specifications, `TECCountAvailableSniffers` may return a number greater than the number of array elements required.

TECGetAvailableSniffers

Returns the list of sniffers available in all installed plug-ins.

```
pascal OSStatus TECGetAvailableSniffers (
                     TextEncoding availableSniffers[],
                     ItemCount maxAvailableSniffers,
                     ItemCount *actualAvailableSniffers);
```

`availableSniffers[]`

An array composed of text encoding specifications. On output, the `TECGetAvailableSniffers` function fills the array with the text encoding specifications that the available sniffers currently support. To determine how large an array to allocate, use the function `TECCountAvailableSniffers` (page 83).

`maxAvailableSniffers`

The number of text encoding specifications the `availableSniffers` array can contain.

`actualAvailableSniffers`
A pointer to a value of type `ItemCount`. On output, this value indicates the number of text encodings the function returned in the `availableSniffers` array.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled for available sniffers.

DISCUSSION

The `TECGetAvailableSniffers` function returns the text encoding specifications that can be sniffed in the array you pass to the function as the `availableSniffers` parameter, eliminating any duplicate information in the process. Consequently, the number of encodings `TECGetAvailableSniffers` returns in the `availableSniffers` array may be fewer than the number of elements you allocated for the array based on your call to the function `TECCountAvailableSniffers` (page 83). `TECGetAvailableSniffers` tells you the actual number of specifications it returns in the `actualAvailableSniffers` parameter.

TECCreateSniffer

Creates a sniffer object and returns a reference to it.

```
pascal OSStatus TECCreateSniffer (
                    TECSnifferObjectRef *encodingSniffer,
                    TextEncoding testEncodings[],
                    ItemCount numTextEncodings);
```

`encodingSniffer`
A pointer to a sniffer object reference, which is of type `TECSnifferObjectRef` (page 65). On output, the reference pertains to the newly created sniffer object.

`testEncodings[]`
An array of text encoding specifications supplied by the caller; `TECCreateSniffer` will attempt to create a sniffer that is capable of sniffing for each of these encodings.

`numTextEncodings`
A value of type `ItemCount` that specifies the number of text encoding specifications in the `testEncodings[]` array.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If other than `noErr`, then one of the text conversion plug-ins encountered an error when polled for available sniffers.

DISCUSSION

The `TECCreateSniffer` function polls plug-ins for available sniffers, creates a sniffer object capable of sniffing each of the specified encodings that it can find a sniffer function for, and returns a reference to it. You use this sniffer object reference with sniffer functions such as `TECSniffTextEncoding` (page 86). If no sniffer function is available for a particular encoding, no error is returned and `TECSniffTextEncoding` indicates later that the encoding was not examined.

To remove a sniffer object, you must call the function `TECDisposeSniffer` (page 89).

SEE ALSO

The function `TECCountAvailableSniffers` (page 83)

The function `TECGetAvailableSniffers` (page 84)

TECSniffTextEncoding

Sniffs a text stream of unknown encoding, based on an array of possible encodings, and returns the probable encodings in a ranked list.

```
pascal OSStatus TECSniffTextEncoding (
                    TECSnifferObjectRef encodingSniffer,
                    TextPtr inputBuffer,
                    ByteCount inputBufferLength,
                    TextEncoding testEncodings[],
                    ItemCount numTextEncodings,
                    ItemCount numErrsArray[],
                    ItemCount maxErrs,
                    ItemCount numFeaturesArray[],
                    ItemCount maxFeatures);
```

`encodingSniffer`
A pointer to a sniffer object.

`inputBuffer` The text to be sniffed.

`inputBufferLength`
The length of the input buffer.

`testEncodings[]`
An array of text encoding specifications. On input, you must specify which text encodings you want to sniff for. On output, this array contains the input array rearranged in the order of most likely to least likely text encodings.

`numTextEncodings`
A value of type `ItemCount`. This value refers to the number of entries in the `testEncodings[]` parameter.

`numErrsArray[]`
An array of type `ItemCount`. This array must contain at least `numTextEncodings` elements. On return, `numErrsArray` holds the number of errors found for each possible text encoding. The entries are in the same order as the entries in the `testEncodings[]` parameter at output.

`maxErrs` The maximum number of errors allowed for a sniffer. The sniffer stops sniffing an encoding after this number is reached when creating the `numErrsArray` list.

`numFeaturesArray[]`
An array of type `ItemCount`. This array must contain at least `numTextEncodings` elements. On return, the `numFeaturesArray[]` parameter holds the number of features found for each possible text encoding. The entries are in the same order as the entries in the `testEncodings[]` parameter at output.

`maxFeatures` The maximum number of features allowed for a sniffer. The sniffer stops sniffing an encoding after this number is reached when creating the `numFeaturesArray` list.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If this function returns a result code other than `noErr`, then one of the conversion plug-ins accessed by the converter encountered an error condition while accessing a sniffer function.

DISCUSSION

For a specified stream of bytes in an unknown encoding and an array of possible encodings, `TECSniffTextEncoding` returns counts of "errors" and "features" for each of the encodings. Each error indicates a code point or sequence that is illegal in the specified encoding, and a feature indicates the presence of a sequence that is characteristic of that encoding. Table 3-1 shows sample output from a sniffer run.

Table 3-1 Sample Sniffer Output

Encoding	Errors	Features
EUC	0	8
JIS	0	0
Mac OS Japanese	20	20

For example, the byte sequence which is interpreted in Mac OS Roman as "äøéö" could legally be interpreted either as Mac OS Roman text or as Mac OS Japanese text. Both sniffers would return zero errors, but the Mac OS Japanese sniffer would also return two features of Mac OS Japanese (representing two legal 2-byte characters.)

The arrays are returned in a ranked list with the most likely text encodings first. The results are sorted first by number of errors (fewest to most), then by number of features (most to fewest), and then by the original order in the list. Upon return from the function, you can assume the correct encoding is in `testEncodings[0]`, or possibly `testEncodings[1]`.

If any of the available encodings are not examined, their number of errors and number of features are set to 0xFFFFFFFF, and they sort to the end of the list.

SEE ALSO

The function `TECCountAvailableSniffers` (page 83)

The function `TECGetAvailableSniffers` (page 84)

The function `TECCreateSniffer` (page 85)

TECDisposeSniffer

Disposes of a sniffer object.

```
pascal OSStatus TECDisposeSniffer (
                        TECSnifferObjectRef encodingSniffer);
```

`encodingSniffer`
A pointer to the sniffer object you want to remove.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If this function returns a result code other than `noErr`, then one of the conversion plug-ins accessed by the converter encountered an error condition.

DISCUSSION

This function releases all memory associated with the sniffer object created by the function `TECCreateSniffer` (page 85).

SEE ALSO

The function `TECCountAvailableSniffers` (page 83)

The function `TECGetAvailableSniffers` (page 84)

The function `TECSniffTextEncoding` (page 86)

TECClearSnifferContextInfo

Resets a sniffer object to its initial settings so it can be reused.

```
pascal OSStatus TECClearSnifferContextInfo (
                        TECSnifferObjectRef encodingSniffer);
```

`encodingSniffer`
A pointer to the sniffer object you want to reuse.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If this function returns a result code other than `noErr`, then one of the conversion plug-ins accessed by the converter encountered an error condition.

DISCUSSION

Sniffers maintain state information about the input encoding buffer and the number of errors and features found for each encoding; this information allows a caller to progressively sniff an input buffer in sequential chunks. Before sniffing a buffer than contains completely new information you must clear any state information by calling `TECClearSnifferContextInfo`.

SEE ALSO

The function `TECCountAvailableSniffers` (page 83)

The function `TECGetAvailableSniffers` (page 84)

The function `TECCreateSniffer` (page 85)

The function `TECSniffTextEncoding` (page 86)

The function `TECDisposeSniffer` (page 89)

Creating and Deleting Converter objects

The Text Encoding Converter functions use converter objects to describe the source, destination, and any intermediate encodings for a given conversion. These objects contain state information and references to plug-ins. When you call a text conversion function, you must pass a reference to the converter object. The functions in this section let you create a converter object and specify indirect conversion paths.

TECCreateConverter

Creates a text encoding converter object and returns a reference to it.

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

`newEncodingConverter`
A pointer to a converter object. On return, this reference points to the newly created text converter object.

`inputEncoding`
The text encoding specification for the source text encoding.

`outputEncoding`
The text encoding specification for the destination text encoding.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If this function returns a result code other than `noErr`, then it did not successfully create the converter object reference. If the current configuration of the converter does not support either the source or destination encoding, the function returns a `kTextUnsupportedEncodingErr` result code.

DISCUSSION

For a specified source encoding and destination encoding, `TECCreateConverter` determines a conversion path, creates a text encoding converter object, and returns a reference to it. You use this converter object reference with conversion functions such as `TECConvertText` (page 95) to convert text. This converter object describes the source, destination, and intermediate encodings; state information; and references to required plug-ins.

If no direct conversion path is found, then `TECCreateConverter` creates an indirect conversion automatically. You may also use the function `TECCreateConverterFromPath` (page 92) to explicitly specify a conversion path.

To remove a converter object, you must call the function `TECDisposeConverter` (page 93).

TECCreateConverterFromPath

Creates a converter 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 converter object reference. On return, the reference points to the newly created text converter object.

`inPath[]` An ordered array of text encoding specifications, beginning with the source encoding specification and ending with the destination encoding specification.

`inEncodings` The number of text encoding specifications in the `inPath` array.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If this function returns a result code other than `noErr`, then it did not successfully create the converter object reference. If the current configuration of the converter does not support all of the encodings in the array, the function returns a `kTextUnsupportedEncodingErr` result code.

DISCUSSION

You use `TECCreateConverterFromPath` to create and obtain a reference to a converter object that specifies a conversion path you define. This function is faster than the function TECCreateConverter (page 91) since it does not need to search for a conversion path.

You specify the conversion sequence from the source encoding through intermediate encodings to the destination encoding. To do so, you create an array of text encoding specifications that identify the encoding conversions through which the text to be converted should pass.

In this array, each adjacent pair of text encodings you specify must represent a conversion that is supported by the current configuration of the Text Encoding Converter. Otherwise, the function returns an error result code and does not create the converter object. To determine each subsequent step in the sequence from the source to the destination encoding, use the function TECGetDestinationTextEncodings (page 74).

To remove a converter object, you must call the function TECDisposeConverter (page 93).

TECDisposeConverter

Disposes of a converter object.

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

newEncodingConverter
: A reference to the text encoding converter object to be removed. This can be the reference returned by the function TECCreateConverter (page 91), TECCreateConverterFromPath (page 92), or TECCreateOneToManyConverter (page 99).

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values.

DISCUSSION

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

If you want to reuse the converter object for a different text stream with the same source and destination encoding, you should clear the converter object using the function `TECClearConverterContextInfo` (page 94) rather than dispose of it and create the object again.

TECClearConverterContextInfo

Resets a converter object to its initial state so it can be reused.

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

`encodingConverter`
The reference to the text encoding converter object whose context is to be cleared. This can be a reference returned by the function `TECCreateConverter` (page 91), `TECCreateOneToManyConverter` (page 99), or `TECCreateConverterFromPath` (page 92).

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values.

DISCUSSION

Creating a converter object and obtaining a reference to it entails some overhead and expense. It is more economical to reuse an existing converter object than to create a new one containing the same conversion information. You use the `TECClearConverterContextInfo` function to clear a converter object of any state and context information it contains and return it to its initial state. This does not, however, affect the source and destination encoding for which this converter object is intended to be used. If this function is unable to clear the context, it returns a result code passed through from one of the conversion plug-ins.

If you are converting multiple segments of a text string, you should not clear the converter object until you have completely converted all the text segments.

Converting Text Between Encodings

TECConvertText

Converts a stream of text from a source encoding to a destination encoding.

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

`encodingConverter`
The reference to the text encoding converter object to be used for the conversion.This can be a reference returned by the function `TECCreateConverter` (page 91) or `TECCreateConverterFromPath` (page 92).

`inputBuffer` The stream of text to be converted.

`inputBufferLength`
The length in bytes of the stream of text specified in the `inputBuffer` parameter. A byte is currently defined as a value of type `UInt8` or `unsigned char`.

`actualInputLength`
A pointer to a value of type `ByteCount`. On output, this value is the number of source text bytes from the input buffer that `TECConvertText` converted.

`outputBuffer` A pointer to a buffer for a byte stream. On output, this buffer holds the converted text.

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

`actualOutputLength`

A pointer to a value of type `ByteCount`. On output this value is the number of bytes of converted text returned in the buffer specified by the `outputBuffer` parameter.

function result

A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If there is not enough memory available for `TECConvertText` to convert the text when allocating internal buffers, the function returns the appropriate Memory Manager result code.

DISCUSSION

The `TECConvertText` function converts a buffer of text according to information contained in the converter object. The converter object specifies the source encoding, destination encoding, and any intermediate encodings required for conversion. 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 at least 32 bytes long. For the function to return successfully, the output buffer you allocate must be large enough to accommodate at least part of the converted text. Then, if the function cannot convert all of the text, it executes successfully, returns the portion of the text it converted, and 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 to determine how many bytes remain. To convert the remaining text, you simply call the function again with the remaining text and a new output buffer.

If the output buffer you allocate is too small to accommodate *any* of the converted text, the function fails. For example, if the destination encoding requires additional bytes in addition to the actual text (for instance, an escape sequence preceding the converted text), your buffer must be large enough to accommodate both.

IMPORTANT

If the destination encoding is a character encoding scheme—such as ISO-2022-JP, 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 that signal switches. ISO-2022-JP requires 3 to 5 bytes for an escape sequence preceding the 1-byte or 2-byte character it introduces. If you allocate a buffer that is less than 5 bytes, the `TECConvertText` function could fail, depending on the text being converted. ▲

To make sure that you receive all of the converted text, you should call the function `TECFlushText` (page 97) when you are finished converting all the text in a particular text stream.

TECFlushText

Flushes out any data that may be stored in a converter object's temporary buffers and returns the converter object to its default state.

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

`encodingConverter`
A reference to the text converter object, whose contents are to be flushed. This can be a reference returned by the function `TECCreateConverter` (page 91) or `TECCreateConverterFromPath` (page 92).

`outputBuffer` A pointer to a buffer for a byte stream. On output, this buffer holds the converted text.

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

`actualOutputLength`

A pointer to a value of type `ByteCount`. On output this value is the number of bytes of converted text returned in the buffer specified by the `outputBuffer` parameter.

function result

A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values.

DISCUSSION

The `TECFlushText` function flushes out any data that may be stored in the temporary buffers in the specified text encoding converter object. Always call `TECFlushText` when you finish converting a text stream. If you are converting a single stream in multiple chunks using multiple calls to `TECConvertText`, you need call `TECFlushText` only after the last call to `TECConvertText` for that stream. The function relies on the specified converter object for information identifying the source encoding, the destination encoding, and any intermediate encodings required for conversion.

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 at least 32 bytes long. For the function to return successfully, the output buffer you allocate must be large enough to accommodate the flushed text. If the output buffer you allocate is too small to accommodate *any* flushed text, the function will fail. For example, if the destination encoding requires additional bytes in addition to the actual text (for instance, an escape sequence preceding the converted text), your buffer must be large enough to accommodate both.

Encodings such as ISO-2022 that need to shift back to a certain default state at the end of a conversion can do so when this function is called.

Multiple Encoding Run Conversions

The following functions allow conversion to multiple encoding runs. This makes possible conversion of Unicode to multiple Mac OS encodings for display using the Script Manager.

TECCreateOneToManyConverter

Creates a converter object that can handle conversion from one source encoding to multiple destination encodings.

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

`newEncodingConverter`
: A pointer to a converter object. On return this points to the newly created one-to-many converter object.

`inputEncoding`
: The text encoding specification for the source text encoding.

`numOutputEncodings`
: The number of text encoding specifications in the `outputEncoding` array.

`outputEncodings[]`
: An ordered array of text encoding specifications for the destination text encodings.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If this function returns a result code other than `NoErr`, then it did not successfully create the converter object reference. If the current configuration of the converter does not support the source encoding or any of the destination encodings, the function returns `KTextUnsupportedEncodingErr`.

DISCUSSION

The `TECCreateOneToManyConverter` function determines a conversion path for the source encoding and destinations encodings you specify, creates a text encoding converter object, and returns a reference to it. You use this converter object reference with conversion functions such as `TECConvertTextToMultipleEncodings` (page 101). This converter object describes the source, destination, and intermediate encodings; state information; and references to required plug-ins.

To remove a converter object, you must call the function TECDisposeConverter (page 93).

SEE ALSO

The function TECFlushText (page 97)

TECCreateOneToManyConverterFromPath

This function is obsolete and no longer supported.

TECGetEncodingList

Gets the list of destination encodings specified in TECCreateOneToManyConverter.

```
pascal OSStatus TECGetEncodingList(
                    TECObjectRef encodingConverter,
                    ItemCount *numEncodings,
                    Handle *encodingList
                    );
```

encodingConverter

A reference to the text encoding conversion object returned by the TECCreateOneToManyConverter (page 99) function.

numEncodings

A pointer to a value of type ItemCount. On return, this value indicates the number of encodings specified by the encodingList handle.

`encodingList`

A handle to an array of text encoding specifications. On return, `encodingList` contains an array of destination text encoding specifications that the converter object can convert to. The memory is allocated by the Text Encoding Converter.

function result

A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) for a list of possible values.

DISCUSSION

The `TECGetEncodingList` function returns a list of destination encodings from a converter object created by `TECCreateOneToManyConverter` (page 99). The function returns the number of destination encodings and a handle to an array of text encoding specifications.

IMPORTANT

The `TECDisposeConverter` function automatically disposes of the handle, so you do not need to do so yourself. This also means that you should not attempt to reference the handle after you have disposed of the converter object. ▲

Plug-ins that handle one-to-many conversions use the `TECGetEncodingList` function to get the output encoding list from the converter object reference.

SEE ALSO

The function `TECDisposeConverter` (page 93)

TECConvertTextToMultipleEncodings

Converts text in the source encoding to runs of text in multiple destination encodings.

```
pascal OSStatus TECConvertTextToMultipleEncodings(
                    TECObjectRef encodingConverter,
                    ConstTextPtr inputBuffer,
                    ByteCount inputBufferLength,
                     ByteCount *actualInputLength,
                     TextPtr outputBuffer,
                     ByteCount outputBufferLength,
                     ByteCount *actualOutputLength,
                     TextEncodingRun outEncodingsBuffer[],
                     ItemCount maxOutEncodingRuns,
                     ItemCount *actualOutEncodingRuns);
```

`encodingConverter`
The reference to the text encoding converter object to be used for the conversion. This is the reference returned by the function `TECCreateOneToManyConverter` (page 99).

`inputBuffer` The stream of text to be converted.

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

`actualInputLength`
A pointer to a value of type `ByteCount`. On output, this value is the number of source text bytes that `TECConvertTextToMultipleEncodings` converted.

`outputBuffer` A pointer to a buffer for a byte stream. On output, this buffer holds the converted text.

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

`actualOutputLength`
A pointer to a value of type `ByteCount`. On output, this value is the number of bytes of the converted text returned in the buffer specified by the `outputBuffer` parameter.

`outEncodingsBuffer[]`
An array of text encoding runs for output. Note that the actual byte size of this buffer should be `actualOutEncodingRuns * sizeof(TextEncodingRun)`. See `TextEncodingRun` (page 46) for more information about the `TextEncodingRun` structure.

`maxOutEncodingRuns`
The maximum number of runs that will fit in the `outEncodingsBuffer` array.

actualOutEncodingRuns

A pointer to a value of type ItemCount. On output actualOutEncodingRuns contains the number of runs put in outEncodingsBuffer array.

function result

A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values. If the outEncodings buffer is filled while converting text, then function returns kTECOutputBufferFullStatus. If there is not enough memory available for TECConvertTextToMultipleEncodings to convert the text when allocating internal buffers, the function returns the appropriate Memory Manager result code.

DISCUSSION

The TECConvertTextToMultipleEncodings function converts the stream of text you pass it to runs of destination encodings specified in the converter object. The function relies on the specified converter object for the source encoding in which the text is expressed, the destination encodings, and the train of encodings forming the intermediate conversion path (if one is explicitly specified). 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 at least 32 bytes long. For the function to return successfully, the output buffer you allocate must be large enough to accommodate at least part of the converted text. Then, if the function cannot convert all of the text, it executes successfully, returns the portion of the text it converted, and 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 to determine how many bytes remain. To convert the remaining text, you simply call the function again with the remaining text and a new output buffer.

If the output buffer you allocate is too small to accommodate *any* of the converted text, the function fails. For example, if the destination encoding requires bytes in addition to the actual text (for instance, an escape sequence preceding the converted text), your buffer must be large enough to accommodate both.

IMPORTANT

If the destination encoding is a character encoding scheme—such as ISO-2022-JP, 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 that signal switches. ISO-2022-JP requires 3 to 5 bytes for an escape sequence preceding the 1-byte or 2-byte character it introduces. If you allocate a buffer that is less than 5 bytes, the `TECConvertText` function could fail, depending on the text being converted. ▲

The Text Encoding Converter creates internal buffers for all multistaged (that is, indirect) conversions that hold intermediate results.

SEE ALSO

The function `TECConvertText` (page 95)

TECFlushMultipleEncodings

Flushes out any encodings that may be stored in a converter object's temporary buffers and shifts encodings back to their default state, if any.

```
pascal OSStatus TECFlushMultipleEncodings(
                     TECObjectRef encodingConverter,
                     TextPtr outputBuffer,
                     ByteCount outputBufferLength,
                     ByteCount *actualOutputLength,
                     TextEncodingRun outEncodingsBuffer[],
                     ItemCount maxOutEncodingRuns,
                     ItemCount *actualOutEncodingRuns);
```

`encodingConverter`

The reference to the text encoding converter object whose contents are to be flushed. This is the reference returned by the function `TECCreateOneToManyConverter` (page 99).

`outputBuffer` A pointer to a buffer for a byte stream. On return, this buffer holds the converted text. If the buffer that you allocate is not large enough to hold the entire converted text stream, an error is returned.

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

`actualOutputLength`
A pointer to a value of type `ByteCount`. On output, this value is the actual number of bytes of the converted text returned in the buffer specified by the `outputBuffer` parameter.

`outEncodingsBuffer[]`
An ordered array of text encoding runs for the destination text encoding. Note that the actual byte size of this buffer should be `actualOutEncodingRuns * sizeof(TextEncodingRun)`. See `TextEncodingRun` (page 46) for more information about the `TextEncodingRun` structure.

`maxOutEncodingRuns`
The maximum number of encoding runs that will fit in `outEncodingsBuffer[]`.

`actualOutEncodingRuns`
A pointer to a value of type `ItemCount`. On return, `actualOutEncodingRuns` specifies how many runs were put in the buffer during conversion.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) for a list of possible values.

DISCUSSION

You should always call `TECFlushMultipleEncodings` at the end of the conversion process to flush out any data that may be stored in the temporary buffers of the text encoding converter object or perform other end-of-encoding conversion tasks. Encodings such as ISO-2022-JP that need to shift back to a certain default state at the end of a conversion can do so when this conversion function is called.

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 at least 32 bytes long. For the function to return successfully, the output buffer you allocate must be large enough to accommodate at least part of the converted text. Then, if the function cannot convert all of the text, it executes successfully, returns the portion of the text it converted, and 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 to determine how many bytes remain. To convert the remaining text, you simply call the function again with the remaining text and a new output buffer.

If the output buffer you allocate is too small to accommodate *any* of the converted text, the function fails. For example, if the destination encoding requires bytes in addition to the actual text (for instance, an escape sequence preceding the converted text), your buffer must be large enough to accommodate both.

IMPORTANT

If the destination encoding is a character encoding scheme—such as ISO-2022-JP, 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 that signal switches. ISO-2022-JP requires 3 to 5 bytes for an escape sequence preceding the 1-byte or 2-byte character it introduces. If you allocate a buffer that is less than 5 bytes, the `TECConvertText` function could fail, depending on the text being converted. ▲

SEE ALSO

The function `TECFlushText` (page 97)

The function `TECCreateOneToManyConverter` (page 99)

CHAPTER 4

Unicode Converter Reference

Contents

This chapter describes the types, constants, and functions pertaining to the Unicode Converter.

For a description of types, constants, and functions pertaining to text encodings and their specifications and for result codes returned by Unicode Converter functions, see Chapter 2, "Basic Text Types Reference." For a description of types, constants, and functions pertaining to the Text Encoding Converter, see Chapter 3, "Text Encoding Converter Reference."

The Unicode Converter provides mapping table conversion to or from Unicode. Its primary use is to convert text from any text encoding to Unicode or to convert Unicode text to any text encoding. It does not provide direct conversion between any two non-Unicode encodings, nor does it handle algorithmic code conversions (such as JIS to Shift-JIS) or code–switching schemes (ISO 2022, for example). However, when you want control over the conversion mapping process and extensive error reporting, you can use the Unicode Converter to convert between two text encodings using Unicode as the hub.

Unicode Converter Constants

Unicode Mapping Versions

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 structure `UnicodeMapping` (page 118) that is passed to a function. A Unicode mapping version is defined by the `UnicodeMapVersion` data type.

```
typedef SInt32  UnicodeMapVersion;
```

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. The following enumeration defines the use-latest-mapping constant:

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

Specific mapping versions are currently undefined. In the future specific mapping versions may be made available.

Conversion Control Flags

Your application uses control flags to determine how the conversion of text from one encoding to another is performed. The conversion functions `ConvertFromTextToUnicode` (page 129), `ConvertFromUnicodeToText` (page 139), `ConvertFromUnicodeToScriptCodeRun` (page 155) and `ConvertFromUnicodeToTextRun` (page 150) allow you to set control flags specifying the conversion process behavior. You can also specify control flags for the function `TruncateForUnicodeToText` (page 162).

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 a 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;
```

The following enumerations define constants for the control flag masks:

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

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

The following enumeration defines the possible settings for the directionality bits:

```
enum {
    kUnicodeDefaultDirection = 0,
    kUnicodeLeftToRight = 1,
    kUnicodeRightToLeft = 2
};

enum {
    kUnicodeDefaultDirectionMask =
                kUnicodeDefaultDirection << kUnicodeDirectionalityBits,
    kUnicodeLeftToRightMask =
                kUnicodeLeftToRight << kUnicodeDirectionalityBits,
    kUnicodeRightToLeftMask =
                UnicodeRightToLeft << kUnicodeDirectionalityBits
};
```

Constant descriptions

`kUnicodeUseFallbacksMask`

A mask for setting the Unicode-use-fallbacks conversion control flag. The Unicode Converter uses fallback mappings when it encounters a source text element for which there is no equivalent destination encoding. Fallback mappings are mappings that do not preserve the meaning or identity of the source character but represent a useful approximation of it. See the function `SetFallbackUnicodeToText` (page 172).

`kUnicodeKeepInfoMask`

A mask for setting the keep-information control flag which governs whether the Unicode Converter keeps the current state stored in the Unicode converter object before converting the text string.

If you clear this flag, the converter will initialize the Unicode converter object before converting the text string and assume that subsequent calls do not need any context, such as direction state for the current call.

If you set the flag, the converter uses the current state. This is useful if your application must convert a stream of text in pieces that are not block delimited. You should set this flag for each call in a series of calls on the same text stream.

`kUnicodeDefaultDirectionMask`

`kUnicodeLeftToRightMask`

`kUnicodeRightToLeftMask`

You can specify one of these masks to indicate 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 value `kUnicodeDefaultDirectionMask` tells the converter to use the value of the first strong direction character in the string, `kUnicodeLeftToRightMask` tells the converter that the base paragraph direction is left to right, and `kUnicodeRightToLeftMask` tells the converter that the base paragraph direction is right to left.

`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 destination encoding.

If set, the converter maps these text elements to their vertical forms, if they are available. For explanation of presentation forms, see Chapter 1, “About Text Encodings and Conversions.”

kUnicodeLooseMappingsMask

A mask that 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 destination encoding equivalent for the source text element.

If you clear this flag, the converter will use only the strict equivalence portion.

If 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 uses the loose mapping section. For explanation of strict and loose mapping, see Chapter 1, "About Text Encodings and Conversions."

kUnicodeStringUnterminatedMask

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

If you clear this bit, the converter treats the end of the buffer as the end of text.

If you set this bit, the converter assumes 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. For example, if the last character in the input buffer is 'A', ConvertFromUnicodeToText stops conversion at the 'A' and returns kTECIncompleteElementErr, because the next buffer could begin with a combining diacritical mark that should be treated as part of the same text element. If the last character in the input buffer is a control character, ConvertFromUnicodeToText does not return kTECIncompleteElementErr because a control character could not be part of a multiple character text element.

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 clear this flag, the converter treats the end of the buffer as a block separator for direction resolution. If you set this flag, it sets the direction as undetermined

`kUnicodeTextRunMask`

A mask for setting the text-run control flag which determines how the Unicode Converter converts Unicode text to a non-Unicode encoding when more than one possible destination encoding exists.

If you clear this flag, the function `ConvertFromUnicodeToTextRun` (page 150) or `ConvertFromUnicodeToScriptCodeRun` (page 155) attempts to convert the Unicode text to the single encoding from the list of encodings in the Unicode converter object that produces the best result, that is, that provides for the greatest amount of source text conversion.

If you set this flag, `ConvertFromUnicodeToTextRun` or `ConvertFromUnicodeToScriptCodeRun`, which are the only functions to which it applies, may generate a destination string that combines text in any of the encodings specified by the Unicode converter object.

`kUnicodeKeepSameEncodingMask`

A mask for setting the keep-same-encoding 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 destination encoding when multiple destination encodings exist. This control flag applies only if the `kUnicodeTextRunMask` control flag is set.

If you set this flag, the function `ConvertFromUnicodeToTextRun` (page 150) 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 destination 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.

Fallback-Handler Control Flags

A fallback mapping is a sequence of one or more characters in the destination 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 ©.

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. Using fallback control flags, described in "Fallback-Handler Control Flags" (page 115), you can specify the order in which both handlers are called if one fails.

You can set control flags to specify which fallback handler the Unicode Converter should use. If both are to be used, you can set the order in which they are called.

The following enumeration defines constants for the setting the `controlFlags` parameter of the functions `SetFallbackUnicodeToText` (page 172) and `SetFallbackUnicodeToTextRun` (page 175).

```
enum {
    kUnicodeFallbackDefaultOnly     = 0L,
    kUnicodeFallbackCustomOnly      = 1L,
    kUnicodeFallbackDefaultFirst    = 2L,
    kUnicodeFallbackCustomFirst     = 3L
};
```

Constant descriptions

`kUnicodeFallbackDefaultOnly`
Use the default fallback handler only.

`kUnicodeFallbackCustomOnly`
Use the custom fallback handler only.

`kUnicodeFallbackDefaultFirst`
Use the default fallback handler first, then the custom one.

`kUnicodeFallbackCustomFirst`
Use the custom fallback handler first, then the default one.

Filter Control Flags

You can query the Unicode Converter for a list of all available mappings between any two encodings whose base, variant, and format values match those you specify as filter control flags. For example, you can check for all available mappings between Unicode 2.0 and all encodings having a specific default variant. To identify possible mappings, you set filter control flags that give the matching criteria.

You call the `QueryUnicodeMappings` function (page 154) to obtain a list of mappings that meet your search criteria. When you call this function, you specify a pointer to a Unicode mapping structure (page 97). A Unicode mapping structure contains fields that specify text encodings, and a text encoding specification gives base, variant, and format values. For information about text encoding specifications, see "Basic Text Types Reference".

You can obtain a count of all possible mappings that match your criteria by calling the function `CountUnicodeMappings` (page 170). You might want to do this to determine how many elements to allocate for the array of mapping information `QueryUnicodeMappings` returns.

To identify the text encoding specification subfields of the Unicode mapping structure whose values you want to use as matching criteria, you set an `iFilter` parameter that you pass to the `QueryUnicodeMappings` function or the `CountUnicodeMappings` function. The iFilter parameter control flag settings identify the three fields of the Unicode encoding specification and the three fields of the other text encoding specification. The `QueryUnicodeMappings` function returns only those mappings whose corresponding field values match the ones you specify.

The following enumerations define constants that set the filter indicators of the `QueryUnicodeMappings` and `CountUnicodeMappings` `iFilter` fields:

```
enum {
    kUnicodeMatchUnicodeBaseBit     = 0,
    kUnicodeMatchUnicodeVariantBit  = 1,
    kUnicodeMatchUnicodeFormatBit   = 2,
    kUnicodeMatchOtherBaseBit       = 3,
    kUnicodeMatchOtherVariantBit    = 4,
    kUnicodeMatchOtherFormatBit     = 5,
};
```

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

Constant descriptions

kUnicodeMatchUnicodeBaseMask

If set, excludes mappings that do not match the text encoding base of the unicodeEncoding field of the structure UnicodeMapping (page 118). If not set, the function ignores the text encoding base of that field.

kUnicodeMatchUnicodeVariantMask

If set, excludes mappings that do not match the text encoding variant of the unicodeEncoding field of the specified Unicode mapping structure. If not set, the function ignores the text encoding variant of that field.

kUnicodeMatchUnicodeFormatMask

If set, excludes mappings that do not match the text encoding format of the unicodeEncoding field of the specified Unicode mapping structure. If not set, the function ignores the text encoding format of that field.

kUnicodeMatchOtherBaseMask

If set, excludes mappings that do not match the text encoding base of the otherEncoding field of the structure UnicodeMapping (page 118). If not set, the function ignores the text encoding base of that field.

kUnicodeMatchOtherVariantMask

If set, excludes mappings that do not match the text encoding variant of the otherEncoding field of the specified Unicode mapping structure. If not set, the function ignores the text encoding variant of that field.

kUnicodeMatchOtherFormatMask

If set, excludes mappings that do not match the text encoding format of the otherEncoding field of the specified

Unicode mapping structure. If not set, the function ignores the text encoding format of that field.

Unicode Converter Result Codes

Many of the Unicode Converter functions return result codes. These codes are listed in “Text Encoding Conversion Manager Result Codes” (page 43).

Unicode Converter Structures and Other Types

UnicodeMapping

A Unicode mapping structure contains a complete text encoding specification for a Unicode encoding, a complete non-Unicode text encoding specification giving the encoding for the text to be converted to or from Unicode, and the version of the mapping table to be used for conversion. You use a structure of this type to specify the text encodings to and from which the text string is to be converted. A Unicode mapping 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`. See “Text Encoding Base” (page 31).

`otherEncoding` A text encoding specification for the text to be converted to or from Unicode.

`mappingVersion` The version of the Unicode mapping table to be used.

Many Unicode Converter functions take a pointer to a Unicode mapping structure as a parameter. For functions that do not modify the Unicode mapping contents, the Unicode Converter provides a constant pointer to a Unicode mapping structure defined by the `ConstUnicodeMappingPtr` data type.

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

TextToUnicodeInfo

A **Unicode converter object** is a private object containing mapping and state information. Many of the Unicode Converter functions that perform conversions require a Unicode converter object containing information used for the conversion process. There are three types of Unicode converter objects, all serving the same purpose but used for different types of conversions. You use the `TextToUnicodeInfo` type, described here, for converting from non-Unicode text to Unicode text.

Because your application cannot directly create or modify the contents of the private Unicode converter object, the Unicode Converter provides functions to create and dispose of it. To create a Unicode converter object for converting from non-Unicode text to Unicode text, your application must first call either the function `CreateTextToUnicodeInfo` (page 125) or the function `CreateTextToUnicodeInfoByEncoding` (page 127) to provide the mapping information required for the conversion. You can then pass this object to the function `ConvertFromTextToUnicode` (page 129) or `ConvertFromPStringToUnicode` (page 163) to identify the information to be used in performing the actual conversion. After you have finished using the object, you should release the memory allocated for it by calling the function `DisposeTextToUnicodeInfo` (page 133). The `TextToUnicodeInfo` data type defines the Unicode converter object.

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

Another function, the function `TruncateForTextToUnicode` (page 160), also requires a Unicode converter object as a parameter. This function does not modify the contents of the private structure to which the Unicode converter object refers, so it uses the constant Unicode converter object defined by the `ConstTextToUnicodeInfo` data type.

```
typedef const TextToUnicodeInfo ConstTextToUnicodeInfo;
```

SEE ALSO

The Unicode converter object of type `UnicodeToTextInfo` (page 120) that you use for converting from Unicode text to non-Unicode text.

The Unicode converter object of type `UnicodeToTextRunInfo` (page 121) that you use to convert from Unicode text to runs of text expressed in various encodings.

UnicodeToTextInfo

Many of the Unicode Converter functions that perform conversions require a Unicode converter object containing information used for the conversion process. There are three types of Unicode converter objects used for different types of conversions. You use the `UnicodeToTextInfo` type, described here, for converting from Unicode to text.

Because your application cannot directly create or modify the contents of the private Unicode converter object, the Unicode Converter provides functions to create and dispose of it. To create a Unicode converter object for converting from Unicode to text, your application must first call either the function `CreateUnicodeToTextInfo` (page 135) or `CreateUnicodeToTextInfoByEncoding` (page 136).

You can then pass this object to the function `ConvertFromUnicodeToText` (page 139) or `ConvertFromUnicodeToPString` (page 165) to identify the information used to perform the actual conversion. After you have finished using the object, you should release the memory allocated for it by calling the function `DisposeUnicodeToTextInfo` (page 143).

A Unicode converter object for this purpose is defined by the `UnicodeToTextInfo` data type.

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

Another function, `TruncateForUnicodeToText` (page 162), also requires a Unicode converter object as a parameter. This function does not modify the contents of the private structure to which the Unicode converter object refers, so it uses the constant Unicode converter object defined by the `ConstUnicodeToTextInfo` data type.

```
typedef const UnicodeToTextInfo ConstUnicodeToTextInfo;
```

SEE ALSO

The Unicode converter object of type `TextToUnicodeInfo` (page 119) that you use for converting from non-Unicode text to Unicode text.

The Unicode converter object of type `UnicodeToTextRunInfo` (page 121) that you use to convert from Unicode text to runs of text expressed in various encodings.

UnicodeToTextRunInfo

Many of the Unicode Converter functions that perform conversions require a Unicode converter object containing information used for the conversion process. There are three types of Unicode converter objects used for different types of conversions. You use the `UnicodeToTextRunInfo` type, described here, for converting from Unicode to multiple encodings.

Because your application cannot directly create or modify the contents of the private Unicode converter object, the Unicode Converter provides functions to create and dispose of it. You can use any of three functions to create a Unicode converter object for converting from Unicode to multiple encodings. You can use `CreateUnicodeToTextRunInfo` (page 145), `CreateUnicodeToTextRunInfoByEncoding` (page 147), or `CreateUnicodeToTextRunInfoByScriptCode` (page 149).

You can then pass this object to the function `ConvertFromUnicodeToTextRun` (page 150) or `ConvertFromUnicodeToScriptCodeRun` (page 155) to identify the information used to perform the actual conversion. After you have finished using the object, you should release the memory allocated for it by calling the function `DisposeUnicodeToTextRunInfo` (page 159).

A Unicode converter object for this purpose is defined by the `UnicodeToTextRunInfo` data type.

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

SEE ALSO

The Unicode converter object of type `TextToUnicodeInfo` (page 119) that you use for converting from non-Unicode text to Unicode text.

The Unicode converter object of type `UnicodeToTextInfo` (page 120) that you use for converting from Unicode text to non-Unicode text.

ScriptCodeRun

To return the result of a multiple encoding conversion, the function `ConvertFromUnicodeToScriptCodeRun` (page 155) uses a script code run structure.

The script code run structure uses an extended script code with values in the range 0–254, which are the text encoding base equivalents to Mac OS encodings. Values 0–32 correspond directly to traditional script codes. This allows a script code run to distinguish Icelandic, Turkish, Symbol, Zapf Dingbats, and so on.

A script code run structure is defined by the `ScriptCodeRun` data type.

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

Field descriptions

`offset` The beginning character position of a text run and its script code in the converted text.

`script` The script code for the text that begins at the position specified.

UnicodeToTextFallbackProcPtr

In converting a text string, when the Unicode Converter encounters a source text element for which there is no destination encoding equivalent, it may use loose mappings and fallback characters to perform the conversion.

A fallback handler is a function that the Unicode Converter uses to perform fallback mapping. To assign your fallback handler to a Unicode converter object, you use the function `SetFallbackUnicodeToText` (page 172) or `SetFallbackUnicodeToTextRun` (page 175). Your own fallback handler must adhere to the following prototype function defined by the Unicode Converter:

```
typedef pascal OSStatus (*UnicodeToTextFallbackProcPtr)(
                            UniChar         *iSrcUniStr,
                            ByteCount       iSrcUniStrLen,
                            ByteCount       *oSrcConvLen,
                            TextPtr         oDestStr,
                            ByteCount       iDestStrLen,
                            ByteCount       *oDestConvLen,
                            LogicalAddress  iInfoPtr,
                            ConstUnicodeMappingPtr iUnicodeMappingPtr);
```

For information about creating a fallback handler function, see the description of the function `MyUnicodeToTextFallbackProc` (page 177).

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 you use to convert text to Unicode from any other encoding, and the other you use to convert text from Unicode to any other encoding. Each of these parts contains its own set of functions. By using the Unicode Converter with Unicode as an intermediary encoding, you can also convert text from any source encoding to any destination encoding.

Using a Static Library

The Text Encoding Conversion Manager provides 68K static libraries that include the Unicode Converter and Basic Text functions. You can use the static libraries if you do not want to use the CFM 68K version. The static libraries are provided directly to you for you to link into your applications, whereas the shared libraries that require the Code Fragment Manager are distributed with Mac OS, beginning with Mac OS 8. These static libraries use resources from the Text Encoding Converter extension and from the files in the Text Encodings folder, so both must be present whether you use the CFM 68K Unicode Converter or the 68K shared libraries.

Clients of the 68K static libraries must explicitly initialize and terminate the Unicode Converter using the functions described in this section. Initialization and termination are handled automatically for CFM clients.

InitializeUnicode

Initializes the 68K static library version of the Unicode Converter.

```
pascal OSStatus InitializeUnicode (StringPtr TECFileName);
```

`TECFileName` Text Encoding Conversion Manager extension file name. You may pass `NULL` for this parameter or you can pass the extension file name, which improves performance of the function. After you call `InitializeUnicode`, you can obtain the extension name for subsequent calls to `InitializeUnicode` using the function `TECGetInfo` (page 55).

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

You may call `InitializeUnicode` more than once. The function checks whether the converter has already been initialized, and if so, completes execution successfully.

TerminateUnicode

Terminates the 68K static library version of the Unicode Converter.

```
pascal void TerminateUnicode (void);
```

Converting to Unicode

For each stream of text in a single encoding that you want to convert to Unicode, your application must first call the function `CreateTextToUnicodeInfo` (page 125) or `CreateTextToUnicodeInfoByEncoding` (page 127) to create a Unicode converter object. When you call these functions, they locate and load the mapping tables required for the conversion, based on the mapping table information you provide.

After you finish converting a text stream using a Unicode converter object, you should dispose of the memory allocated for the Unicode converter object by calling function DisposeTextToUnicodeInfo (page 133).

You can use the same Unicode converter object to convert multiple text segments of a single text stream. A Unicode converter object persists until you dispose of it. You should use the same Unicode converter object only to convert segments of text belonging to the text stream for which you created the object.You should create a new Unicode converter object to convert another stream of text even if you intend to use the same mapping information stored in an existing Unicode converter object. This is because the Unicode Converter stores private state information in a Unicode converter object 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 Unicode converter object. Each time you call the ConvertFromTextToUnicode function to convert a segment of text belonging to the text stream, you pass the function the same object, repeating the process until the entire text stream is converted.

CreateTextToUnicodeInfo

Creates and returns a Unicode converter object containing information required for converting strings from a non-Unicode encoding to Unicode.

```
pascal OSStatus CreateTextToUnicodeInfo (
      ConstUnicodeMappingPtr iUnicodeMapping,
      TextToUnicodeInfo *oTextToUnicodeInfo);
```

iUnicodeMapping

A pointer to a structure of type UnicodeMapping (page 118). Your application provides this structure to identify the mapping to be used for the conversion. The unicodeEncoding field of this structure can specify a Unicode format of kUnicode16BitFormat or kUnicodeUTF8Format. Versions of the Unicode Converter prior to 1.2.1 do not support kUnicodeUTF8Format.

oTextToUnicodeInfo

A pointer to a Unicode converter object of type TextToUnicodeInfo (page 119), used for converting text to

Unicode. On output, the parameter returns a Unicode converter object that holds mapping table information you supply as the `UnicodeMapping` parameter and state information related to the conversion. This information is required for conversion of a text stream in a non-Unicode encoding to Unicode.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

You pass a Unicode converter object returned from the function `CreateTextToUnicodeInfo` to the function `ConvertFromTextToUnicode` (page 129) or `ConvertFromPStringToUnicode` (page 163) to identify the information to be used for the conversion. These two functions modify the contents of the object.

You pass a Unicode converter object returned from `CreateTextToUnicodeInfo` to the function `TruncateForTextToUnicode` (page 160) to identify the information to be used to truncate the string. This function does not modify the contents of the Unicode converter object.

In addition to various resource and memory errors that it can return, the function can return the following result codes:

- `kTextUnsupportedEncodingErr`
  One of the encodings specified by the Unicode mapping structure you supply is not currently supported.
- `kTECMissingTableErr`
  A resource associated with one of the encodings is missing.
- `kTECTableChecksumErr`
  A resource has an invalid checksum, indicating that it has become corrupted.

If an error is returned, the Unicode converter object is invalid

SEE ALSO

`CreateTextToUnicodeInfoByEncoding` (page 127)

CreateTextToUnicodeInfoByEncoding

Based on the given text encoding specification, creates and returns a Unicode converter object containing information required for converting strings from the specified non-Unicode encoding to Unicode.

```
pascal OSStatus CreateTextToUnicodeInfoByEncoding (
      TextEncoding iEncoding,
      TextToUnicodeInfo *oTextToUnicodeInfo);
```

`iEncoding` The text encoding specification for the source text. For text encoding specifications, see Chapter 2, "Basic Text Types Reference."

`oTextUnicodeInfo`
The Unicode converter object of type `TextToUnicodeInfo` (page 119) returned by the function.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

This function offers you an easier, alternative way to create a Unicode converter than with the function `CreateTextToUnicodeInfo` (page 125) because you do not need to create a Unicode mapping structure. You simply specify the text encoding of the source text. However, this method is less efficient because the text encoding parameter must be resolved internally into a Unicode mapping.

Using this function, you cannot specify a version of Unicode, so a default version of Unicode is used; 16-bit format is assumed.

You pass a Unicode converter object returned from `CreateTextToUnicodeInfoByEncoding` to the function `ConvertFromTextToUnicode` (page 129) or `ConvertFromPStringToUnicode` (page 163) to identify the information to be used for the conversion. These two functions modify the contents of the Unicode converter object.

You pass a Unicode converter object returned from `CreateTextToUnicodeInfoByEncoding` to the function `TruncateForTextToUnicode` (page 160) to identify the information to be used to truncate the string. This function does not modify the contents of the Unicode converter object.

If you are converting the text stream to Unicode as an intermediary encoding, and then from Unicode to the final destination encoding, you use the function `CreateUnicodeToTextInfo` (page 135) to create a Unicode converter object for the second part of the process.

SEE ALSO

`CreateTextToUnicodeInfo` (page 125)

"Text Encoding Base" (page 31)

ChangeTextToUnicodeInfo

Changes the mapping information for the specified Unicode converter object used to convert text to Unicode to the new mapping you provide.

```
pascal OSStatus ChangeTextToUnicodeInfo (
      TextToUnicodeInfo ioTextToUnicodeInfo,
      ConstUnicodeMappingPtr iUnicodeMapping);
```

`ioTextToUnicodeInfo`
The Unicode converter object of type `TextToUnicodeInfo` (page 119) containing the mapping to be modified. You use the function `CreateTextToUnicodeInfo` (page 125) to obtain one.

`iUnicodeMapping`
A structure of type `UnicodeMapping` (page 118) identifying the new mapping to be used. This is the mapping that replaces the existing mapping in the Unicode converter object.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

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 Unicode converter object pointed to by the `ioTextToUnicodeInfo` parameter with the information contained in the UnicodeMapping structure you supply as the iUnicodeMapping parameter.

ChangeTextToUnicodeInfo resets the Unicode converter object's fields as necessary.

In addition to various resource errors, the function can return the following result codes:

- paramErr
  The Unicode converter object you supplied is invalid.
- kTextUnsupportedEncodingErr
  One of the encodings specified by the Unicode mapping structure you supplied is not currently supported.
- kTECMissingTableErr
  A resource associated with one of the encodings is missing.
- kTECTableChecksumErr
  A resource has an invalid checksum, indicating that it has become corrupted.

If an error is returned, the Unicode converter object is invalid.

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 iOutputBufLen,
      ByteCount *oSourceRead,
      ByteCount *oUnicodeLen,
      UniCharArrayPtr oUnicodeStr);
```

`iTextToUnicodeInfo`
A Unicode converter object of type `TextToUnicodeInfo` containing mapping and state information used for the conversion. Your application obtains a Unicode converter object using the function `CreateTextToUnicodeInfo` (page 125).

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

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

`iControlFlags` Conversion control flags. The only control flag that you can use with `ConvertFromTextToUnicode` is `kUnicodeUseFallbacksMask`. See “Conversion Control Flags” (page 110).

`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 a value of type `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.

`iOutputBufLen` 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. The relationship between the size of the source string and the Unicode string is complex and depends on the source encoding and the contents of the string.

`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 `kTECUnmappableElementErr` result code, 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 Chapter 2, “Basic Text Types Reference.”
function result	A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

The `ConvertFromTextToUnicode` function converts a text string in a non-Unicode encoding to Unicode. You specify the source string’s encoding in the Unicode mapping structure that you pass to the function `CreateTextToUnicodeInfo` (page 125) to obtain a Unicode converter object for the conversion. You pass the Unicode converter object returned by `CreateTextToUnicodeInfo` to `ConvertFromTextToUnicode` as the `iTextToUnicodeInfo` parameter.

In addition to converting a text string in any encoding to Unicode, the `ConvertFromTextToUnicode` function can map offsets for 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 MacRoman, the Unicode string will be at least double the size (more if it uses noncomposed Unicode); for MacArabic and MacHebrew, the corresponding Unicode string could be up to six times as big. For most 2-byte encodings, for example Shift-JIS, the Unicode string will be less than double the size. For international robustness, your application should allocate a buffer three to four times larger than the source string. If the output Unicode text is actually UTF-8—which could occur beginning with the current release of the Text Encoding Conversion Manager, version 1.2.1—the UTF-8 buffer pointer must be cast to `UniCharArrayPtr` before it can be passed as the `oUnicodeStr` parameter. Also, the output buffer length will have a wider range of variation than for UTF-16; for ASCII input, the output will be the same size; for Han input, the output will be twice as big, and so on.

The function returns a `noErr` result code if it has completely converted the input string to Unicode without using fallback characters. If the function returns the `paramErr`, `kTECTableFormatErr`, or `kTECGlobalsUnavailableErr` result codes, it did not convert the string.

If the function returns `kTECBufferBelowMinimumSizeErr`, the output buffer was too small to allow conversion of any part of the input string. You need to increase the size of the output buffer and try again.

If the function returns the `kTECUsedFallbacksStatus` result code, the function has completely converted the string using one or more fallback characters. This can only happen if you set the Unicode-use-fallbacks control flag.

If the function returns `kTECOutputBufferFullErr`, the output buffer was not big enough to completely convert the input; `oSourceRead` indicates the amount of input converted. You can call the function again with another output buffer—or with the same output buffer, after copying its contents—to convert the remainder of the input string.

If the function returns `kTECPartialCharErr`, the input buffer ended with an incomplete multibyte character. If you have subsequent input text available, you can append the unconverted input from this call to the beginning of the subsequent input text and call the function again.

If the function returns `kTECUnmappableElementErr` because an input text element could not be mapped to Unicode, then the function did not completely convert the input string. This can only happen if you did not set the Unicode-use-fallbacks control flag. You can set this flag and then convert the remaining unconverted input, or take some other action.

SPECIAL CONSIDERATIONS

This function modifies the contents of the Unicode converter object you pass in the `iTextToUnicodeInfo` parameter.

DisposeTextToUnicodeInfo

Releases the memory allocated for the specified Unicode converter object.

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

`ioTextToUnicodeInfo`
A pointer to a Unicode converter object of type `TextToUnicodeInfo` (page 119), used for converting text to Unicode. On input, you specify the object to be disposed of, which your application created using the function `CreateTextToUnicodeInfo` (page 125) or `CreateTextToUnicodeInfoByEncoding` (page 127).

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

The `DisposeTextToUnicodeInfo` function disposes of the Unicode converter object and releases the memory allocated for it. Your application should not attempt to dispose of the same structure more than once.

You use this function only to release the memory for objects that your application created through the function `CreateTextToUnicodeInfo` (page 125) or `CreateTextToUnicodeInfoByEncoding` (page 127). You must not use it for any other type of Unicode converter object.

If your application specifies an invalid Unicode converter object, such as `NULL`, the function returns a `paramErr` result code.

Converting From Unicode

To convert text from Unicode to a non-Unicode encoding, you must first obtain a Unicode converter object containing the mapping and state information the Unicode Converter uses to perform the conversion. You use the function `CreateUnicodeToTextInfo` (page 135) or `CreateUnicodeToTextInfoByEncoding` (page 136) to obtain this object. These functions locate and load the mapping table resources required for the conversion. You then pass the Unicode converter object to the function `ConvertFromTextToUnicode` (page 129) to perform the conversion. When your application is finished using the Unicode converter object, you must dispose of it and the memory allocated for it by calling the function `DisposeUnicodeToTextInfo` (page 143).

You can use the same Unicode converter object to convert multiple Unicode strings belonging to the same text stream to the encoding specified in the mapping table.

If you use the same Unicode converter object for multiple segments of the same text stream, you should set the keep-information control flag when you call the conversion function. This is because how the conversion is performed might depend on the previous segment. The Unicode Converter might need to refer to the direction state from the previous segment—for example, to determine the text direction for Hebrew or Arabic text.

You should use the same Unicode converter object only to convert segments of text belonging to the single text stream for which you created the Unicode converter object. This is because the Unicode Converter stores private state information in a Unicode converter object that is relevant only to that particular Unicode converter object and the single text stream for which it is used.

When you are finished converting all the text reliant on the Unicode converter object, you must release the memory allocated for the Unicode converter object by calling the function `DisposeUnicodeToTextInfo` (page 143).

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 124) to convert the text to Unicode, then you use these functions to convert the text from Unicode to the final, destination encoding.

CreateUnicodeToTextInfo

Creates and returns a Unicode converter object containing information required for converting strings from Unicode to a non-Unicode encoding.

```
pascal OSStatus CreateUnicodeToTextInfo (
      ConstUnicodeMappingPtr iUnicodeMapping,
      UnicodeToTextInfo *oUnicodeToTextInfo);
```

`iUnicodeMapping`
A pointer to a structure of type `UnicodeMapping` (page 118). Your application provides this structure to identify the mapping to be used for the conversion. The `unicodeEncoding` field of this structure can specify a Unicode format of `kUnicode16BitFormat` or `kUnicodeUTF8Format`. Note that the versions of the Unicode Converter prior to 1.2.1 do not support `kUnicodeUTF8Format`.

`oUnicodeToTextInfo`
A pointer to a Unicode converter object of type `UnicodeToTextInfo` (page 120), used for converting Unicode strings to text. On output, the parameter returns a Unicode converter object that holds the mapping table information you supply as the `iUnicodeMapping` parameter and the state information related to the conversion. The information contained in the Unicode converter object is required for the conversion of a Unicode string to a non-Unicode encoding.

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

You pass the Unicode converter object returned from `CreateUnicodeToTextInfo` to the function `ConvertFromUnicodeToText` (page 139) or `ConvertFromUnicodeToPString` (page 165) to identify the information to be used for the conversion. These two functions modify the contents of the Unicode converter object.

In addition to various resource and memory errors, the function can return the following result codes:

- `kTextUnsupportedEncodingErr`
  One of the encodings specified by the Unicode mapping structure you supply is not currently supported.
- `kTECMissingTableErr`
  A resource associated with one of the encodings is missing.
- `kTECTableChecksumErr`
  A resource has an invalid checksum, indicating that it has become corrupted.

If an error is returned, the Unicode converter object is invalid.

SEE ALSO

The function `CreateUnicodeToTextInfoByEncoding` (page 136)

CreateUnicodeToTextInfoByEncoding

Based on the given text encoding specification for the converted text, creates and returns a Unicode converter object containing information required for converting strings from Unicode to the specified non-Unicode encoding.

```
pascal OSStatus CreateUnicodeToTextInfoByEncoding (
      TextEncoding iEncoding,
      TextToUnicodeInfo *oUnicodeToTextInfo);
```

`iEncoding` The text encoding specification for the destination, or converted, text.

`oUnicodeToTextInfo`
The Unicode converter object of type `UnicodeToTextInfo` (page 120) returned by the function.

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

This function offers you an easier, alternative way to create a Unicode converter than the function `CreateUnicodeToTextInfo` (page 135). However, this method is less efficient internally because the destination text encoding you specify must be resolved into a Unicode mapping. Using this function, you cannot specify a version of Unicode, so a default version of Unicode is used; 16-bit format is assumed.

You pass a Unicode converter object returned from the function `CreateUnicodeToTextInfoByEncoding` to the function `ConvertFromUnicodeToText` (page 139) or `ConvertFromUnicodeToPString` (page 165) to identify the information to be used for the conversion. These two functions modify the contents of the Unicode converter object.

You pass a Unicode converter object returned from `CreateUnicodeToTextInfoByEncoding` to the function `TruncateForUnicodeToText` (page 162) to identify the information to be used to truncate the string. This function does not modify the contents of the Unicode converter object.

SEE ALSO

"Text Encoding Base" (page 31)

ChangeUnicodeToTextInfo

Changes the mapping information contained in the specified Unicode converter object used to convert Unicode text to a non-Unicode encoding.

```
pascal OSStatus ChangeUnicodeToTextInfo (
      UnicodeToTextInfo ioUnicodeToTextInfo,
      ConstUnicodeMappingPtr iUnicodeMapping);
```

`ioUnicodeToTextInfo`

The Unicode converter object of type `UnicodeToTextInfo` (page 120) to be modified. You use the function `CreateUnicodeToTextInfo` (page 135) or `CreateUnicodeToTextInfoByEncoding` (page 136) to obtain a Unicode converter object of this type.

`iUnicodeMapping`

The structure of type `UnicodeMapping` (page 118) to be used. This is the new mapping that replaces the existing mapping in the Unicode converter object.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

The `ChangeUnicodeToTextInfo` function allows you to provide new mapping information for converting text from Unicode to a non-Unicode encoding. The function replaces the mapping table information that currently exists in the specified Unicode converter object with the information contained in the new Unicode mapping structure you provide.

`ChangeUnicodeToTextInfo` resets the Unicode converter object's fields as necessary. However, it does not initialize or reset the conversion state maintained by the Unicode converter object.

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

For example, you can change the other (destination) encoding of the Unicode mapping structure pointed to by the `iUnicodeMapping` parameter before you call the function `ConvertFromUnicodeToText` (page 139) to convert the next character or sequence of characters that require a different destination encoding.

In addition to various resource errors, the function can return the following result codes:

- `paramErr`
  The Unicode converter object you supplied is invalid.
- `kTextUnsupportedEncodingErr`
  One of the encodings specified by the Unicode mapping structure you supplied is not currently supported.
- `kTECMissingTableErr`
  A resource associated with one of the encodings is missing.
- `kTECTableChecksumErr`
  A resource has an invalid checksum, indicating that it has become corrupted.

If an error is returned, the Unicode converter object is invalid.

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 iOutputBufLen,
                    ByteCount *oInputRead,
                    ByteCount *oOutputLen,
                    LogicalAddress oOutputStr);
```

`iUnicodeToTextInfo`
A Unicode converter object of type `UnicodeToTextInfo` for converting text from Unicode. You use the function `CreateUnicodeToTextInfo` (page 135) or `CreateUnicodeToTextInfoByEncoding` (page 136) to obtain a Unicode converter object 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. If the input text is UTF-8, which is supported for versions 1.2.1 or later of the converter, you must cast the UTF-8 buffer pointer to `ConstUniCharArrayPtr` before you can pass it as this parameter.

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

```
kUnicodeUseFallbacksMask
kUnicodeKeepInfoMask
kUnicodeVerticalFormMask
kUnicodeLooseMappingsMask
kUnicodeStringUnterminatedMask
```

one of the following directionality masks:

`kUnicodeDefaultDirectionMask`
`kUnicodeLeftToRightMask`
`kUnicodeRightToLeftMask`

For a description of these control flags, see “Conversion Control Flags” (page 110).

`iOffsetCount` The number of offsets contained in the array provided by the `iOffsetArray` parameter. Your application supplies this value. 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`. All offsets must be less than `iUnicodeLen`.

`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 converted string in the new encoding.

`iOutputBufLen` 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 `kTECUnmappableElementErr` result code, 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 destination encoding without using fallback characters. If the function returns the `paramErr` or `kTECGlobalsUnavailableErr` result codes, it did not convert the string.

If the function returns `kTECTableFormatErr`, the code encountered a table in an unknown format. The function did not completely convert the input string (and may not have converted any of it).

If the function returns `kTECBufferBelowMinimumSizeErr`, the output buffer was too small to allow conversion of any part of the input string. You need to increase the size of the output buffer and try again.

If the function returns the `kTECUsedFallbacksStatus` result code, the function has completely converted the string using one or more fallback characters. This can only happen if you set the Unicode-use-fallbacks control flag.

If the function returns `kTECOutputBufferFullErr`, the output buffer was not big enough to completely convert the input; `oInputRead` indicates the amount of input converted. You can call the function again with another output buffer (or with the same output buffer, after copying its contents) to convert the remainder of the Unicode string.

If the function returns `kTECPartialCharErr`, the Unicode input string ended with an incomplete UTF-8 character (can only happen for UTF-8 input). If you have subsequent input text available, you can append the unconverted input from this call to the beginning of the subsequent input text and call the function again.

If the function returns `kTECUnmappableElementErr`, an input text element could not be mapped to the destination encoding. The function did not completely convert the Unicode input string. This can only happen if you did not set the

Unicode-use-fallbacks control flag. You can set this flag and convert the remaining unconverted input, or take some other action.

If the function returns kTextUndefinedElementErr, the Unicode input string included a value which is undefined for the specified Unicode version. The function did not completely convert the input string, and fallback handling will not be invoked. You can resume conversion from a point beyond the offending Unicode character, or take some other action.

If the function returns kTextIncompleteElementErr, then either the input string included a text element which is too long for the internal buffers, or the input string ended with a text element which may be incomplete (this latter case can only happen if you set the kUnicodeStringUnterminatedMask control flag). The function did not completely convert the input string, and fallback handling will not be invoked.

For additional information, see "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

The ConvertFromUnicodeToText function converts a Unicode text string to the destination encoding you specify in the Unicode mapping structure that you pass to the function CreateUnicodeToTextInfo (page 135) or CreateUnicodeToTextInfoByEncoding (page 136) when you call them to obtain a Unicode converter object for the conversion process. You pass the returned object to ConvertFromUnicodeToText as the iUnicodeToTextInfo parameter.

In addition to converting the Unicode string, ConvertFromUnicodeToText can map offsets for 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 MacRoman 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 MacArabic and MacHebrew, the result will usually be less than half the size of the Unicode string.

This function modifies the contents of the Unicode converter object you passed as the `iUnicodeToTextInfo` parameter.

SEE ALSO

The function `ConvertFromTextToUnicode` (page 129)

DisposeUnicodeToTextInfo

Releases the memory allocated for the specified Unicode converter object.

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

`ioUnicode`
A pointer to a Unicode converter object for converting from Unicode to a non-Unicode encoding. On input, you specify a Unicode converter object that your application created using the function `CreateUnicodeToTextInfo` (page 135) or `CreateUnicodeToTextInfoByEncoding` (page 136).

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

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

You must use this function only to release the memory for a Unicode converter object that your application created through the function

CreateUnicodeToTextInfo (page 135) or CreateUnicodeToTextInfoByEncoding (page 136). You must not use it for any other type of Unicode converter object.

The function returns noErr if it disposes of the Unicode converter object successfully. If your application specifies an invalid Unicode converter object, such as NULL, the function returns a paramErr result code.

Converting From Unicode to Multiple Encodings

It may not be possible to convert a Unicode string to a single destination encoding. To handle these cases, the Unicode Converter provides a function that allows you to specify a number of possible destination encodings and how the function should use these destination encodings, if necessary, when converting the Unicode string. Before you can use the function ConvertFromUnicodeToTextRun (page 150) for this purpose, you must first create and obtain a Unicode converter object containing the mapping and state information the Unicode Converter uses to perform the conversion. You use the function CreateUnicodeToTextRunInfo (page 145) CreateUnicodeToTextRunInfoByEncoding (page 147), or CreateUnicodeToTextRunInfoByScriptCode (page 149) for this purpose. You then pass the object to the ConvertUnicodeToTextRun function to perform the conversion. You can also convert a Unicode string to one or more scripts. For this purpose you use the function ConvertFromUnicodeToScriptCodeRun (page 155).

You can use the same Unicode converter object to convert multiple Unicode strings belonging to the same text stream to the encodings specified in the mapping table.

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

You should use the same Unicode converter object only to convert the text stream for which you created the Unicode converter object. This is because the Unicode Converter stores private state information in a Unicode converter object 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 Unicode converter object, release the memory allocated for the Unicode converter object by calling the function `DisposeUnicodeToTextRunInfo` (page 159).

CreateUnicodeToTextRunInfo

Creates and returns a Unicode converter object containing the information required for converting a Unicode text string to strings in one or more non-Unicode encodings.

```
pascal OSStatus CreateUnicodeToTextRunInfo (
      ItemCount iNumberOfMappings,
      const UnicodeMapping iUnicodeMappings[],
      UnicodeToTextRunInfo *oUnicodeToTextInfo);
```

`iNumberOfMappings`

The number of mappings specified by your application for converting from Unicode to any other encoding types, including other forms of Unicode. If you pass 0 for this parameter, the converter will use all of the scripts installed in the system. The primary script is the one with highest priority; `ScriptOrder` (`'itlm'` resource) determines the priority of the rest. If you set the high-order bit for this parameter, the Unicode converter assumes that the `iEncodings` parameter contains a single element specifying the preferred encoding. This feature is supported for versions 1.2 or later of the converter.

`iUnicodeMappings`

A pointer to an array of structures of type `UnicodeMapping` (page 118). Your application provides this structure to identify the mappings to be used for the conversion. The order in which you specify the mappings determines the priority of the destination encodings. For this function, the Unicode mapping structure can specify a Unicode format of `kUnicode16BitFormat` or `kUnicodeUTF8Format`. Note that the versions of the Unicode Converter prior to the Text Encoding Conversion Manager 1.2.1 do not support `kUnicodeUTF8Format`. Also, note that the `unicodeEncoding` field should be the same for all of the entries in `iUnicodeMappings`. If you pass `NULL` for the `iUnicodeMappings` parameter, the converter uses all of the scripts installed in the system, assuming the default version of Unicode with 16-bit format. The primary script is the one with the highest priority and `ScriptOrder`(`'itlm'` resource) determines the priority of the rest. This is supported beginning with version 1.2 of the Text Encoding Conversion Manager.

`oUnicodeToTextInfo`

A pointer to a Unicode converter object for converting Unicode text strings to strings in one or more non-Unicode encodings. On output, a Unicode converter object that holds the mapping table information you supply as the `iUnicodeMappings` parameter and the state information related to the conversion.

function result A result code. In addition to various resource and memory errors, the function can return the following result codes:

`kTextUnsupportedEncodingErr`
One of the encodings specified by the Unicode mapping structure you supply is not currently supported.

`kTECMissingTableErr`
A resource associated with one of the encodings is missing.

`kTECTableChecksumErr`
A resource has an invalid checksum, indicating that it has become corrupted.

If an error is returned, the Unicode converter object is invalid. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference" for other possible result code values.

DISCUSSION

You pass a Unicode converter object returned from the function `CreateUnicodeToTextRunInfo` to the function `ConvertFromUnicodeToTextRun` (page 150) or `ConvertFromUnicodeToScriptCodeRun` (page 155) to identify the information to be used for the conversion. These two functions modify the contents of the Unicode converter object.

SEE ALSO

The function `CreateUnicodeToTextInfoByEncoding` (page 136)

The function `CreateUnicodeToTextRunInfoByScriptCode` (page 149)

CreateUnicodeToTextRunInfoByEncoding

Based on the given text encoding specifications for the converted text runs, creates and returns a Unicode converter object containing information required for converting strings from Unicode to one or more specified non-Unicode encodings.

```
pascal OSStatus CreateUnicodeToTextRunInfoByEncoding (
      ItemCount iNumberOfEncodings,
      const TextEncoding iEncodings[],
      UnicodeToTextRunInfo *oUnicodeToTextInfo);
```

`iNumberOfEncodings`

The number of desired encodings. If you pass 0 for this parameter, the converter will use all of the scripts installed in the system. The primary script is the one with highest priority; `ScriptOrder`('`itlm`' resource) determines the priority of the rest. If you set the high-order bit for this parameter, the Unicode converter assumes that the `iEncodings` parameter contains a single element specifying the preferred encoding. This feature is supported for versions 1.2 or later of the converter.

`iEncodings` An array of text encoding specifications for the desired encodings. Your application provides this structure to identify the encodings to be used for the conversion. The order in which you specify the encodings determines the priority of the destination encodings. If you pass `NULL` for this parameter, the converter will use all of the scripts installed in the system. The primary script is the one with highest priority and `ScriptOrder`('`itlm`' resource) determines the priority of the rest.This feature is supported for versions 1.2 or later of the converter.

`oUnicodeToTextInfo`
A pointer to a Unicode converter object for converting Unicode text strings to strings in one or more non-Unicode encodings. On output, a Unicode converter object that holds the encodings you supply as the `iEncodings` parameter and the state information related to the conversion.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

You pass a Unicode converter object returned from `CreateUnicodeToTextRunInfoByEncoding` to the function `ConvertFromUnicodeToTextRun` (page 150) or `ConvertFromUnicodeToScriptCodeRun` (page 155) to identify the information to be used for the conversion. These two functions modify the contents of the Unicode converter object.

In addition to various resource and memory errors, the function can return the following result codes:

- `kTextUnsupportedEncodingErr`
  One of the encodings specified by the Unicode mapping structure you supply is not currently supported.
- `kTECMissingTableErr`
  A resource associated with one of the encodings is missing.
- `kTECTableChecksumErr`
  A resource has an invalid checksum, indicating that it has become corrupted.

If an error is returned, the converter object is invalid.

CreateUnicodeToTextRunInfoByScriptCode

Based on the given script codes for the converted text runs, creates and returns a Unicode converter object containing information required for converting strings from Unicode to one or more specified non-Unicode encodings.

```
pascal OSStatus CreateUnicodeToTextRunInfoByScriptCode(
      ItemCount iNumberOfScriptCodes,
      const ScriptCode iScripts[],
      UnicodeToTextRunInfo *oUnicodeToTextInfo);
```

`iNumberOfScriptCodes`
The number of desired scripts. If you pass 0 for this parameter, the converter uses all the scripts installed in the system. In this case, the primary script is the one with highest priority; `ScriptOrder` (`'itlm'` resource) determines the priority of the rest. If you set the high-order bit for this parameter, the Unicode converter assumes that the `iScripts` parameter contains a single element specifying the preferred script. This feature is supported beginning with the Text Encoding Conversion Manager 1.2.

`iScripts` An array of script codes for the desired scripts. Your application provides this structure to identify the scripts to be used for the conversion. The order in which you specify the scripts determines their priority. If you pass `NULL` for this parameter, the converter uses all of the scripts installed in the system. In this case, the primary script is the one with the highest priority and the priority order of the remaining scripts is defined by the `ScriptOrder(itlm resource)` resource. This feature is supported for versions 1.2 or later of the converter.

`oUnicodeToTextInfo`
A pointer to a Unicode converter object for converting Unicode text strings to strings in one or more non-Unicode encodings. On output, a Unicode converter object that holds the scripts you supply as the `iScripts` parameter and the state information related to the conversion.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

You pass a Unicode converter object returned from `CreateUnicodeToTextRunInfoByScriptCode` to the function `ConvertFromUnicodeToTextRun` (page 150) or `ConvertFromUnicodeToScriptCodeRun` (page 155) to identify the information to be used for the conversion. These two functions modify the contents of the Unicode converter object.

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 iOutputBufLen,
      ByteCount *oInputRead,
      ByteCount *oOutputLen,
      LogicalAddress oOutputStr,
      ItemCount iEncodingRunBufLen,
      ItemCount *oEncodingRunOutLen,
      TextEncodingRun oEncodingRuns[]);
```

`iUnicodeToTextInfo`
A Unicode converter object for converting Unicode text to one or more encodings. You use the function `CreateUnicodeToTextRunInfo` (page 145), `CreateUnicodeToTextRunInfoByEncoding` (page 147), or `CreateUnicodeToTextRunInfoByScriptCode` (page 149) to obtain a Unicode converter object 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:

```
kUnicodeUseFallbacksMask
kUnicodeKeepInfoMask
kUnicodeVerticalFormMask
kUnicodeLooseMappingsMask
kUnicodeStringUnterminatedMask
kUnicodeTextRunMask
kUnicodeKeepSameEncodingMask
```

one of the following directionality masks:

```
 kUnicodeDefaultDirectionMask
 kUnicodeLeftToRightMask
 kUnicodeRightToLeftMask
```

For a description of these control flags, see "Conversion Control Flags" (page 110).

If the text-run control flag is clear, `ConvertFromUnicodeToTextRun` attempts to convert the Unicode text to the single encoding it chooses from the list of encodings in the Unicode mapping structures array that you provide when you create the Unicode converter object. This is the encodings 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 always points to a value equal to 1.

If you set the use-fallbacks control flag, the converter uses 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, CreateUnicodeToTextRunInfoByEncoding, or CreateUnicodeToTextRunInfoByScriptCode when you created the Unicode converter object 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. What the converter does with the next text element depends on the setting of the keep-same-encoding control flag.

If the 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 keep-same-encoding control flag is set, the converter continues with the new destination encoding until it encounters a text element that cannot be converted using the new encoding. This attempts to minimize the number of encoding changes in the output text. 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.

iOffsetCount The number of offsets in the array pointed to by the iOffsetArray parameter. Your application supplies this value. 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. All offsets must be less than iUnicodeLen.

oOffsetCount A pointer to a value of type 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 resulting converted string.

iOutputBufLen 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 discussion following the parameter descriptions.)

iEncodingRunBufLen

The number of text encoding run elements you allocated for the encoding run array pointed to by the oEncodingRuns parameter. The converter returns the number of valid encoding runs in the location pointed to by oEncodingRunOutLen. Each entry in the encoding runs array specifies the beginning offset in the converted text and its associated text encoding.

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 On input, an array of structures of type TextEncodingRun (page 46). 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 beginning offset in the converted text string and the associated encoding specification.

function result A result code. The result codes are the same as those for the function `ConvertFromUnicodeToText` (page 139), with the following additional possibility: If the function returns `kTECArrayFullErr`, then the `oEncodingRuns` array was too small for all of the encodings runs in the output text, and the input was not completely converted. As you would if `kTECOutputBufferFullErr` was returned, you can call the function again with another output buffer—or with the same output buffer after copying its contents—to convert the remainder of the Unicode string.

DISCUSSION

To use the `ConvertFromUnicodeToTextRun` function, you must first set up an array of structures of type `UnicodeMapping` (page 118) containing, in order of precedence, the mapping information for the conversion. To create a Unicode converter object, you call the `CreateUnicodeToTextRunInfo` function passing it the Unicode mapping array, or you can the `CreateUnicodeToTextRunInfoByEncoding` or `CreateUnicodeToTextRunInfoByScriptCode` functions, which take arrays of text encodings or script codes instead of an array of Unicode mappings. You pass the returned Unicode converter object 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:

- To keep the converted text in a single encoding, clear the text-run control flag.
- To keep as much contiguous converted text as possible in one encoding, set the text-run control flag and clear the keep-same-encoding control flag.
- To minimize the number of resulting encoding runs and the changes of destination encoding, set both the text-run and 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 includes only elements for the beginning of each new encoding run in the converted string.

SEE ALSO

The function ConvertFromUnicodeToScriptCodeRun (page 155)

ConvertFromUnicodeToScriptCodeRun

Converts a string from Unicode to one or more scripts.

```
pascal OSStatus ConvertFromUnicodeToScriptCodeRun (
      UnicodeToTextRunInfo iUnicodeToTextInfo,
      ByteCount iUnicodeLen,
      ConstUniCharArrayPtr iUnicodeStr,
      OptionBits iControlFlags,
      ItemCount iOffsetCount,
      ByteOffset iOffsetArray[],
      ItemCount *oOffsetCount,
      ByteOffset oOffsetArray[],
      ByteCount iOutputBufLen,
      ByteCount *oInputRead,
      ByteCount *oOutputLen,
      LogicalAddress oOutputStr,
      ItemCount iScriptRunBufLen,
      ItemCount *oScriptRunOutLen,
      ScriptCodeRun oScriptCodeRuns[]
      );
```

`iUnicodeToTextInfo`
A Unicode converter object for converting Unicode text to one or more scripts. You use the function `CreateUnicodeToTextRunInfoByScriptCode` (page 149) to obtain a Unicode converter object 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
kUnicodeVerticalFormBit
kUnicodeLooseMappingsBit
kUnicodeStringUnterminatedBit
kUnicodeTextRunBit
kUnicodeKeepSameEncodingBit
```

one of the following directionality masks:

```
kUnicodeDefaultDirection
kUnicodeLeftToRightBit
kUnicodeRightToLeft
```

For a description of these control flags, see “Conversion Control Flags” (page 110).

If the text-run control flag is clear, `ConvertFromUnicodeToScriptCodeRun` attempts to convert the Unicode text to the single script from the list of scripts in the Unicode converter object 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 scripts specified in the array, then the converter chooses among them based on their order in the array. If this flag is clear, the `oScriptCodeRuns` parameter always points to a value equal to 1.

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

If you set the `kUnicodeTextRunBit` control flag, the converter attempts to convert the complete Unicode text string into the first script specified in the Unicode mapping structures array you passed to `CreateUnicodeToTextRunInfo`, `CreateUnicodeToTextRunInfoByEncoding`, or `CreateUnicodeToTextRunInfoByScriptCode` to create the Unicode converter object used for this conversion. If it cannot do this, the converter then attempts to convert the first text element that failed to the remaining scripts, in their specified order in the array. What the converter does with the next text element depends on the setting of the keep-same-encoding control flag:

If the keep-same-encoding control flag is clear, the converter returns to the original script and attempts to continue conversion with that script; this is equivalent to converting each text element to the first one that works, in the order specified.

If the Unicode-keep-same-encoding control flag is set, the converter continues with the new destination script until it encounters a text element that cannot be converted using the new script. This attempts to minimize the number of script code changes in the output text. When the converter cannot convert a text element using any of the scripts in the list and the Unicode-keep-same-encoding control flag is set, the converter uses the fallbacks default characters for the current script.

`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 a value of type `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 resulting converted string.

`iOutputBufLen` 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 buffer address. On input, this value points to the beginning 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 `ConvertFromUnicodeToScriptCodeRun` function returns the converted string up to the character that caused the error.

`iScriptRunBufLen`
The number of script code run elements you allocated for the script code run array pointed to by the `oScriptCodeRuns` parameter. The converter returns the number of valid script code runs in the location pointed to by `oScriptRunOutLen`. Each entry in the script code run array specifies the beginning offset in the converted text and its associated script code.

`oScriptRunOutLen`
A pointer to a value of type `ItemCount`. On output, this value contains the number of valid script code runs returned in the `oScriptCodeRuns` parameter.

`oScriptCodeRuns`
An array of elements of type `ScriptCodeRun`. Your application should allocate an array with the number of elements you specify in the `iScriptRunBufLen` parameter. On output, this array contains the script code runs for the converted text string. Each entry in the array specifies the beginning offset in the converted text string and the associated script code specification.

function result A result code. The result codes are the same as those for the function `ConvertFromUnicodeToText` (page 139).

DISCUSSION

To use the `ConvertFromUnicodeToScriptCodeRun` function, you must first set up an array of script codes containing in order of precedence the scripts to be used for the conversion. To create a Unicode converter object, you call the function `CreateUnicodeToTextRunInfoByScriptCode` (page 149). You pass the returned Unicode converter object as the `iUnicodeToTextInfo` parameter when you call the `ConvertFromUnicodeToScriptCodeRun` function.

The `ConvertFromUnicodeToScriptCodeRun` function returns the converted string in the array pointed to by the `oOutputStr` parameter.

DisposeUnicodeToTextRunInfo

Releases the memory allocated for the specified Unicode converter object.

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

`ioUnicodeToTextRunInfo`
A pointer to a Unicode converter object. On input, you specify a Unicode converter object that points to the conversion information to be disposed of, which your application created using the function `CreateUnicodeToTextRunInfo` (page 145), `CreateUnicodeToTextRunInfoByEncoding` (page 147), or `CreateUnicodeToTextRunInfoByScriptCode` (page 149).

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

The `DisposeUnicodeToTextRunInfo` function disposes of the Unicode converter object specified by the `ioUnicodeToTextRunInfo` parameter and releases the memory allocated for it. Your application should not attempt to dispose of the same Unicode converter object more than once.

You must use this function to release the memory only for a Unicode converter object that your application created through the function `CreateUnicodeToTextRunInfo` (page 145), `CreateUnicodeToTextRunInfoByEncoding` (page 147), or `CreateUnicodeToTextRunInfoByScriptCode` (page 149).

You must not use it for any other type of Unicode converter object.

If your application specifies an invalid Unicode converter object, such as `NULL`, the function returns `paramErr`.

Truncating Strings Before Converting Them

If you need to divide up a string, before converting it, your application can use the truncation functions to properly break the string so that the string to be converted is terminated with complete characters and complete text elements. To avoid the possibility of corrupting the contents of the string or breaking a string between the bytes of a multibyte character, it is best to use these functions instead of truncating the string yourself.

TruncateForTextToUnicode

Identifies where your application can safely break a multibyte string to be converted to Unicode so that the string is not broken in the middle of a multibyte character.

```
pascal OSStatus TruncateForTextToUnicode(
      ConstTextToUnicodeInfo iTextToUnicodeInfo,
      ByteCount iSourceLen,
    ConstLogicalAddress iSourceStr,
    ByteCount iMaxLen,
    ByteCount *oTruncatedLen);
```

`iTextToUnicodeInfo`
The Unicode converter object of type `TextToUnicodeInfo` (page 119) for the text string to be divided up with each segment properly truncated. The `TruncateForTextToUnicode` function does not modify the object's contents.

`iSourceLen` The length in bytes of the multibyte string to be divided up.

`iSourceStr` The address of the multibyte string to be divided up.

`iMaxLen` The maximum allowable length of the string to be truncated. This must be less than or equal to `iSourceLen`.

`oTruncatedLen` 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 `iSourceStr`, that is less than or equal to the length specified by `iMaxLen`. This identifies the byte after which you can break the string.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

Your application can use this function to break a string properly before you call the function `ConvertFromTextToUnicode` (page 129) so that the string you pass it is terminated with complete characters. You can call this function repeatedly to properly divide up 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. Each time you use the function, you get a properly terminated string within the allowable length range. You use the function as many times as necessary to be able to convert the entire text segment.

Because the `TruncateForTextToUnicode` function does not modify the contents of the Unicode converter object, you can call this function safely between calls to the function `ConvertFromTextToUnicode` (page 129).

If the function returns `paramErr`, `kTECGlobalsUnavailableErr`, or `kTECTableFormatErr` the value returned by the `oTruncatedLen` parameter is invalid.

TruncateForUnicodeToText

Identifies where your application can safely break 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 iUnicodeToTextInfo,
      ByteCount iSourceLen,
      ConstUniCharArrayPtr iSourceStr,
      OptionBits iControlFlags,
      ByteCount iMaxLen,
      ByteCount *oTruncatedLen);
```

`iUnicodeToTextInfo`
A Unicode converter object `UnicodeToTextInfo` (page 120) for the Unicode string to be divided up. The `TruncateForUnicodeToText` function does not modify the contents of this private structure.

`iSourceLen` The length in bytes of the Unicode string to be divided up.

`iSourceStr` A pointer to the Unicode string to be divided up.

`iControlFlags` Truncation control flags. Specify the flag `kUnicodeStringUnterminatedMask` if truncating a buffer of text that belongs to a longer stream containing a subsequent buffer of text that could have characters belonging to a text element that begins at the end of the current buffer. If you set this flag, typically you would set the `iMaxLen` parameter equal to `iSourceLen`. For information on the flag `kUnicodeStringUInterminatedMask`, see “Conversion Control Flags” (page 110).

`iMaxLen` The maximum allowable length of the string to be truncated. This must be less than or equal to `iSourceLen`.

`oTruncatedLen` 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 `iSourceStr` parameter, that is less than or equal to the value of the `iMaxLen` parameter. This returned parameter identifies the byte after which you can truncate the string.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

Your application can use this function to divide up a Unicode string properly truncating each portion before you call `ConvertFromUnicodeToText` or `ConvertFromUnicodeToScriptCodeRun` to convert the string. 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. Each time you use the function, you get a properly terminated string within the allowable length range. You use the function as many times as necessary to be able to convert the entire text segment.

Because this function does not modify the contents of the Unicode converter object, you can call this function between conversion calls.

In addition to resource errors, the functions may return any of the following result codes: `paramErr`, `kTECGlobalsUnavailableErr`, `kTECTableFormatErr`, `kTECPartialCharErr` (if truncating UTF-8), `kTECIncompleteElementErr`, or `kTextUndefinedElementErr`. If the result code is not `noErr`, then the value returned by the `oTruncatedLen` parameter is invalid.

Converting Between Unicode and Pascal Strings

ConvertFromPStringToUnicode

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

```
pascal OSStatus ConvertFromPStringToUnicode (
      TextToUnicodeInfo iTextToUnicodeInfo,
      ConstStr255Param iPascalStr,
      ByteCount iOutputBufLen,
      ByteCount *oUnicodeLen,
      UniCharArrayPtr oUnicodeStr);
```

iTextToUnicodeInfo
A Unicode converter object of type TextToUnicodeInfo (page 119) for the Pascal string to be converted. You can use the function CreateTextToUnicodeInfo (page 125) or CreateTextToUnicodeInfoByEncoding (page 127) to create the Unicode converter object.

iPascalStr The Pascal string to be converted to Unicode.

iOutputBufLen 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 string is smaller than the buffer size you allocated.

oUnicodeLen A pointer to a value of type ByteCount. On output, the length in bytes of the converted Unicode string returned in the oUnicodeStr parameter.

oUnicodeStr A pointer to a Unicode character array. On output, this buffer holds the converted Unicode string. For information on the Unicode character array, see “Unicode Character and String Pointer Data Types” (page 49).

function result A result code. The function returns the noErr result code if it has completely converted the Pascal string to Unicode without using fallback characters. If the function returns the paramErr, kTECTableFormatErr, or kTECGlobalsUnavailableErr result codes, it did not convert the string.

If the function returns kTECBufferBelowMinimumSizeErr, the output buffer was too small to allow conversion of any part of the input string. You need to increase the size of the output buffer and try again.

If the function returns the kTECUsedFallbacksStatus result code, the function has completely converted the Pascal string using one or more fallback characters.

If the function returns kTECOutputBufferFullErr, the output buffer was not big enough to completely convert the input. You can call the function again with another output buffer—or with the same output buffer, after copying its contents—to convert the remainder of the input string.

If the function returns kTECPartialCharErr, the input buffer ended with an incomplete multibyte character. If you have subsequent input text available, you can append the unconverted input from this call to the beginning of the subsequent input text and call the function again.

See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference" for other possible values.

DISCUSSION

The ConvertFromPStringToUnicode function provides an easy and efficient way to convert a short Pascal string to a Unicode string without incurring the overhead associated with the function ConvertFromTextToUnicode (page 129).

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

ConvertFromUnicodeToPString

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

```
pascal OSStatus ConvertFromUnicodeToPString (
      UnicodeToTextInfo iUnicodeToTextInfo,
      ByteCount iUnicodeLen,
      ConstUniCharArrayPtr iUnicodeStr,
      Str255 oPascalStr);
```

iUnicodeToTextInfo
A Unicode converter object. You use the CreateUnicodeToTextInfo or CreateUnicodeToTextInfoByEncoding function to obtain the Unicode converter object for the conversion.

iUnicodeLen
The length in bytes of the Unicode string to be converted. This is the string your application provides in the iUnicodeStr parameter.

`iUnicodeStr` A pointer to an array containing the Unicode string to be converted. For information on the Unicode character array, see Chapter 2, “Basic Text Types Reference.”

`oPascalStr` A buffer. On output, the converted Pascal string returned by the function.

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

The `ConvertFromUnicodeToPString` function provides an easy and efficient way to convert a Unicode string to a Pascal string in a Mac OS text encoding without incurring the overhead associated with use of the function `ConvertFromUnicodeToText` (page 139) or `ConvertFromUnicodeToScriptCodeRun` (page 155).

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 Unicode converter object.

The function returns a `noErr` result code if it has completely converted the Unicode string to the Pascal without using fallback characters. If the function returns the `paramErr` or `kTECGlobalsUnavailableErr` result codes, it did not convert the string.

If the function returns `kTECTableFormatErr`, the code encountered a table in an unknown format. The function did not completely convert the input string (and may not have converted any of it).

If the function returns `kTECBufferBelowMinimumSizeErr`, the output buffer was too small to allow conversion of any part of the input string. You need to increase the size of the output buffer and try again.

If the function returns the `kTECUsedFallbacksStatus` result code, the function has completely converted the string using one or more fallback characters.

If the function returns `kTECOutputBufferFullErr`, the output buffer was not big enough to completely convert the input. You can call the function again with another output buffer (or with the same output buffer, after copying its contents) to convert the remainder of the Unicode string.

If the function returns `kTECPartialCharErr`, the Unicode input string ended with an incomplete UTF-8 character, which can only happen for UTF-8 input. If you have subsequent input text available, you can append the unconverted input from this call to the beginning of the subsequent input text and call the function again.

If the function returns `kTextUndefinedElementErr`, the Unicode input string included a value that is undefined for the specified Unicode version. The function did not completely convert the input string, and fallback handling was not invoked. You can resume conversion from a point beyond the offending Unicode character, or take some other action.

If the function returns `kTextIncompleteElementErr`, then either the input string included a text element that is too long for the internal buffers, or the input string ended with a text element that may be incomplete. The latter case can happen only if you set the `kUnicodeStringUnterminatedMask` control flag. The function did not completely convert the input string, and fallback handling was not invoked.

Obtaining Mapping Information

The Unicode Converter provides functions you can use to obtain a list of the mappings available on the system that match specified criteria.

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 iFilter,
      ConstUnicodeMappingPtr iFindMapping,
      ItemCount iMaxCount,
      ItemCount *oActualCount,
      UnicodeMappingPtr oReturnedMappings);
```

`iFilter` Filter control flags representing the six values given in the Unicode mapping structure that this function uses to match against in determining which mappings on the system to return to your application. The filter control flag enumerations, described in “Filter Control Flags” (page 116), define the constants for the flags and their masks. You can include in the search criteria any of the three text encoding values—base, variant, and format—for both the Unicode encoding and the other specified encoding. For any flag not turned on, the value is ignored; the function does not check the corresponding value of the mapping tables on the system.

`iFindMapping`
A structure of type `UnicodeMapping` (page 118) containing the text encodings whose values are to be matched.

`iMaxCount`
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 `oReturnedMappings` 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 `kTECArrayFullErr` in this case.

`oActualCount`
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 `iMaxCount` if more matching mappings are found than can fit in the `oReturnedMappings` array.

`oReturnedMappings`
A pointer to an array of structures of type `UnicodeMapping` (page 118). On input, this pointer refers to an array for the matching mappings returned by the function. To allocate sufficient elements for the array, you can use the function `CountUnicodeMappings` (page 170) to determine the number of mappings returned for given values of the `iFilter` and `iFindMapping` parameters. 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 `kTECBufferBelowMinimumSizeErr` error result. The `oActualCount` parameter identifies the number of matching mappings actually found, which may be greater than the number returned.

function result
A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

DISCUSSION

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

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

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, you would set up the iFilter and iFindMappings 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 control flag bitmasks. For information on text encoding bases, text encoding default variants, and text encoding default formats and their constants, see the chapter "Basic Text Types Reference." In this example, the text encoding base field of the Unicode mapping 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.

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

        iFindMapping.otherEncoding = CreateTextEncoding(
        kTextEncodingMacRoman,
        kTextEncodingDefaultVariant,
        kTextEncodingDefaultFormat);
```

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

If the function returns a `noErr` result code, the value retuned in the `oActualCount` parameter is less than or equal to the value returned in the `iMaxCount` parameter and the `oReturnedMappings` parameter contains all of the matching mappings found. If the function returns a `kTECArrayFullErr`, the function found more mappings than your `oReturnedMappings` array could accommodate.

CountUnicodeMappings

Counts available mappings that meet the specified matching criteria.

```
pascal OSStatus CountUnicodeMappings (
      OptionBits iFilter
      ConstUnicodeMappingPtr iFindMapping
      ItemCount *oActualCount);
```

`iFilter` Filter control flags representing the six subfields of the Unicode mapping structure that this function uses to match against in determining which mappings on the system to return to your application. The filter control enumeration, described in “Filter Control Flags” (page 116), define the constants for the subfield’s flags and their masks. You can include in the search criteria any of the three text encoding subfields for both the Unicode encoding and the other specified encoding. For any flag not turned on, the subfield value is ignored and the function does not check the corresponding subfield of the mappings on the system.

`iFindMapping`

A structure of type `UnicodeMapping` (page 118) containing the text encodings whose field values are to be matched.

`oActualCount` A pointer to a value of type `ItemCount`. On output, the number of matching mappings found.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

You can use the function to obtain the count of mappings that meet specified criteria by setting individual filter control flags. You can filter on any of the three text encoding subfields of the Unicode mapping structure's `unicodeEncoding` specification and on any of the three text encoding subfields of the structure's `otherEncoding` specification. The `iFilter` parameter consists of a set of six control flags that you set to identify which of the corresponding six subfields to include in the match count. No filtering is performed on fields for which you do not set the corresponding filter control flag.

SEE ALSO

The function `QueryUnicodeMappings` (page 167)

Setting the Fallback Handler

A fallback handler is a function that the Unicode Converter uses to perform fallback mapping from Unicode to another encoding. Fallback mapping is invoked if the `kUnicodeUseFallbacksMask` control flag is set and the converter encounters a Unicode character that cannot be mapped to the destination encoding using either strict mappings or—if the `kUnicodeLooseMappingsMask` control flag is set—loose mappings.

Fallback mapping from Unicode is a process in which a Unicode character is mapped to a sequence of one or more characters in another encoding 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. The fallback mapping depends on the destination encoding. In general, fallback mappings are not reversible, and therefore, do not provide round-trip fidelity.

The Unicode Converter supplies a default fallback handler for mapping from Unicode to other encodings. Using `SetFallbackUnicodeToText` or `SetFallbackUnicodeToTextRun`, you can also install your own application-defined fallback handler and use it alone, or you can use yours in combination with the default fallback handler. If you use both, you can specify which one gets called first; the other one gets called only if the first one fails. If fallback mapping is invoked and the specified fallback handler fails—or if both handlers fail when both are used—then the Unicode Converter uses a default fallback sequence obtained from the mapping table to represent the unmappable Unicode character. The default fallback sequence is usually a question mark character in the destination encoding.

SetFallbackUnicodeToText

Associates an application-defined fallback handler with a specific `UnicodeToTextInfo` Unicode converter object for a single text run to be used with either the function `ConvertFromUnicodeToText` (page 139) or `ConvertFromUnicodeToPString` (page 165).

```
pascal OSStatus SetFallbackUnicodeToText (
      UnicodeToTextInfo iUnicodeToTextInfo,
      UnicodeToTextFallbackUPP iFallback,
      OptionBits iControlFlags,
      LogicalAddress iInfoPtr);
```

`iUnicodeToTextInfo`
The Unicode converter object with which the fallback handler is to be associated. You use the function `CreateUnicodeToTextInfo` (page 135) or `CreateUnicodeToTextInfoByEncoding` (page 136) to obtain a Unicode converter object of this type.

`iFallback`
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 `UnicodeToTextFallbackProcPtr` (page 122). For a description of how to create your own fallback handler, see `MyUnicodeToTextFallbackProc` (page 177). You should use the `NewUnicodeToTextFallbackProc` macro to convert a pointer to your fallback handler into a `UnicodeToTextFallbackUPP`. See the example in this function's discussion.

`iControlFlags`
Control flags 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. See "Fallback-Handler Control Flags" (page 115).

`iInfoPtr` The address of a block of memory 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 memory block to store data required by your fallback handler whenever it is called. This is similar in use to a reference constant (refcon). If you don't need to use a memory block, specify `NULL` for this parameter.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in "Basic Text Types Reference."

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 Unicode converter object passed to the functions `ConvertFromUnicodeToText` (page 139), `ConvertFromUnicodeToTextRun` (page 150), `ConvertFromUnicodeToPString` (page 165), and `ConvertFromUnicodeToScriptCodeRun` (page 155). You can define multiple fallback handlers and associate them with different Unicode converter objects, depending on your requirements.

The following example shows how to install an application-defined fallback handler. You can name your application-defined fallback handler anything you choose. The name, `MyUnicodeToTextFallbackProc`, used in this example is not significant. However, you must adhere to the parameters, the return type, and the calling convention as expressed in this example, which follows the prototype, because a pointer to this function must be of type `UnicodeToTextFallbackProcPtr` as defined in the `Unicode.h` header file.

The `Unicode.h` header file also defines the `UnicodeToTextFallbackUPP` type and the `NewUnicodeToTextFallbackProc` macro. See "Application-Defined Function" (page 176) for a description of the parameters of an application-defined fallback handler.

Listing 4-1 Installing an Application-Defined Fallback Handler

```
#include <Types.h>
#include <Errors.h>
#include <MixedMode.h>
#include <TextCommon.h>
#include <Unicode.h>
pascal OSStatus MyUnicodeToTextFallbackProc(
    UniChar *iSrcUniStr, ByteCount iSrcUniStrLen, ByteCount *oSrcConvLen,
    TextPtr oDestStr, ByteCount iDestStrLen,
    ByteCount *oDestConvLen, LogicalAddress iInfoPtr,
    ConstUnicodeMappingPtr iUnicodeMappingPtr) {
                .
                .
                .

        /* include your actual fallback handler implementation here */
}
            .
            .
            .
main () {
            .
            .
            .
    UnicodeMapping mapping;
    UnicodeToTextInfo unicodeToTextInfo;
    UnicodeToTextFallbackUPP fallbackProc;
    OSStatus status;
            .
            .
            .
    mapping.unicodeEncoding =
        CreateTextEncoding(kTextEncodingUnicodeDefault,
        kTextEncodingDefaultVariant, kUnicode16BitFormat);
    mapping.otherEncoding =
        CreateTextEncoding(kTextEncodingMacRoman,
        kTextEncodingDefaultVariant, kTextEncodingDefaultFormat);
    mapping.mappingVersion = kUnicodeUseLatestMapping;
    status = CreateUnicodeToTextInfo(&mapping, &unicodeToTextInfo);
            .
            .
            .
    fallbackProc =
        NewUnicodeToTextFallbackProc(MyUnicodeToTextFallbackProc);
    status = SetFallbackUnicodeToText(unicodeToTextInfo, fallbackProc,
        kUnicodeFallbackCustomFirst, NULL);
            .
            .
            .
    status = ConvertFromUnicodeToText(unicodeToTextInfo,
            .
            .
            .
}
```

SetFallbackUnicodeToTextRun

Associates an application-defined fallback handler with a specific Unicode converter object for multiple text runs to be used with the function `ConvertFromUnicodeToTextRun` (page 150) or `ConvertFromUnicodeToScriptCodeRun` (page 155).

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

`iUnicodeToTextInfo`
The Unicode converter object with which the fallback handler is to be associated. You use the function `CreateUnicodeToTextRunInfo` (page 145), `CreateUnicodeToTextRunInfoByEncoding` (page 147), or `CreateUnicodeToTextRunInfoByScriptCode` (page 149) to obtain a Unicode converter object to specify for this parameter.

`iFallback` 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

UnicodeToTextFallbackProcPtr (page 122). For a description of how to create your own fallback handler, see MyUnicodeToTextFallbackProc (page 177). You should use the NewUnicodeToTextFallbackProc macro described in the discussion of the function SetFallbackUnicodeToText (page 172).

iControlFlags Control flags 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. See “Fallback-Handler Control Flags” (page 115).

iInfoPtr The address of a block of memory 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 block to store data required by your fallback handler whenever it is called. This is similar in use to a reference constant (refcon). If you don’t need to use a memory block, specify NULL for this parameter.

function result A result code. See “Text Encoding Conversion Manager Result Codes” (page 43) in the chapter “Basic Text Types Reference.”

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 Unicode converter object passed to the function ConvertFromUnicodeToText (page 139), ConvertFromUnicodeToTextRun (page 150), ConvertFromUnicodeToPString (page 165), or ConvertFromUnicodeToScriptCodeRun (page 155). You can define multiple fallback handlers and associate them with different Unicode converter objects, depending on your requirements.

Application-Defined Function

You can name the application-defined fallback handler anything you choose—the name is not significant. Here, the name MyUnicodeToTextFallbackProc is used for illustrative purposes. However, the parameters, the return type, and the calling convention are all important, since a pointer to the application-defined fallback handler function must be of type `UnicodeToTextFallbackProcPtr` as defined in `Unicode.h`.

MyUnicodeToTextFallbackProc

Converts a Unicode text element for which there is no destination 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 *iSrcUniStr,
      ByteCount iSrcUniStrLen,
      ByteCount *oSrcConvLen,
      TextPtr *oDestStr,
      ByteCount iDestStrLen,
      ByteCount *oDestConvLen,
      LogicalAddress *iInfoPtr
      ConstUnicodeMappingPtr iUnicodeMappingPtr);
```

`iSrcUniStr` A pointer to a single UTF-16 character to be mapped by the fallback handler.

`iSrcUniStrLen` The length in bytes of the UTF-16 character indicated by the `iSrcUniStr` parameter. Usually this is 2 bytes, but it could be 4 bytes for a non-BMP character.

`oSrcConvLen` A pointer to a value of type `ByteCount`. On output, the length in bytes of the portion of the Unicode character that was actually processed by your fallback handler. Your fallback handler returns this value. It should set this to 0 if none of the text was handled, or 2 or 4 if the Unicode character was handled. This value is initialized to 0 before the fallback handler is called.

`oDestStr` A pointer to the output buffer where your handler should place any converted text.

`iDestStrLen` The maximum size in bytes of the buffer provided by the `oDestStr` parameter.

`oDestConvLen` 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. It is initialized to 0 (zero) before the fallback handler is called.

`iInfoPtr` A pointer to a block of memory allocated by your application, which can be used by your fallback handler in any way that you like. This is the same pointer passed as the last parameter of `SetFallbackUnicodeToText` or `SetFallbackUnicodeToTextRun`. How you use the data passed to you in this memory block is particular to your handler. This is similar in use to a reference constant (refcon).

`iUnicodeMappingPtr`
A constant pointer to a structure of type `UnicodeMapping` (page 118). This structure identifies a Unicode encoding specification and a particular base encoding specification.

function result A result code. See "Text Encoding Conversion Manager Result Codes" (page 43) in the chapter "Basic Text Types Reference."

DISCUSSION

The Unicode Converter calls your fallback handler when it cannot convert a text string using the mapping table specified by the Unicode converter object passed to either `ConvertFromUnicodeToText` or `ConvertFromUnicodeToPString`. The control flags you set for the `controlFlags` parameter of the function `SetFallbackUnicodeToText` (page 172) or the `SetFallbackUnicodeToTextRun` (page 175) 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 character to be converted and its length, a buffer for the converted string you return and the buffer length, and a pointer to a block of memory 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 `UnicodeToTextFallbackProcPtr` (page 122).

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 function CreateUnicodeToTextInfoByEncoding (page 136), ConvertFromUnicodeToPString (page 165), ConvertFromUnicodeToTextRun (page 150), or ConvertFromUnicodeToScriptCodeRun (page 155). You associate an application-defined fallback handler with a particular Unicode converter object you intend to pass to the conversion function when you call it.

Your handler should return noErr if it can handle the fallback, or kTECUnmappableElementErr if it cannot. It can return other errors for exceptional conditions, such as when the output buffer is too small. If your handler returns kTECUnmappableElementErr, then oSrcConvLen and oDestConvLen are ignored because either the default handler will be called or the default fallback sequence will be used.

Text converted from UTF-8 will already have been converted to UTF-16 before the fallback handler is called to process it. Your fallback handler should do all of its processing on text encoded in UTF-16.

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

To associate a fallback-handler function with a Unicode converter object you use the SetFallbackUnicodeToText (page 172) and SetFallbackUnicodeToTextRun (page 175) 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 versions of the Unicode Converter prior to 1.2, the fallback handler may receive a multiple character text element, so the source string length value could be greater than 2 and the fallback handler may set srcConvLen to a value greater than 2. In versions earlier than 1.2.1, the srcConvLen and destConvLen variables are not initialized to 0; both values are ignored unless the fallback handler returns noErr.

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

Appendixes

APPENDIX A

Writing Custom Plug-Ins

This document provides information on writing plug-ins for text encoding conversion on Mac OS–based computers.

Text encoding conversion plug-ins, which provide conversion services between pairs of encodings, inform the Text Encoding Conversion Manager about their conversion and encoding analysis capabilities. The Text Encoding Conversion Manager sets up plug-ins and tears them down; the plug-ins perform conversions, handle caller options, and examine text encodings.

Support for new encodings is provided by writing new text encoding plug-ins. Plug-ins are implemented as Code Fragment Manager (CFM) libraries.

The number and kind of text encodings that the Text Encoding Conversion Manager supports depends on the conversion plug-ins that are currently installed in the system. Text encoding conversion plug-ins are installed in the Text Encodings folder within the System Folder.

Generally, plug-ins provide algorithmic conversions, although plug-ins can also provide mapping-table-based conversions. Mapping-table-based conversions provided by the Unicode Converter are available through a provided plug-in which calls the Unicode Converter.

The Text Encoding Conversion Manager provides mechanisms to create converter objects to communicate with the plug-ins.

Plug-ins are implemented as code fragments. The main export symbol of the code fragment is a routine that returns the address of a structure of type `TECPluginDispatchTable`. The structure is a plug-in dispatch table that contains a dispatch table format version number, a signature for the plug-in, and hooks for the methods each plug-in needs to support.

The filename of a plug-in does not affect the actual text conversion performed by the Text Encoding Conversion Manager.

Export symbols of the code fragment plug-in include the standard CFM initialization and termination routines as well as the main routine.

The initialization routine is called by the Text Encoding Conversion Manager when the plug-in is loaded. It must return `noErr` or the plug-in is not installed. For example,

```
OSErr INIT_KoreanPlugin(InitBlockPtr initBlkPtr){
                    return noErr;
                    }
```

The termination routine performs cleanup before the plug-in is unloaded. For example,

```
void TERM_KoreanPlugin(void)
                    {
                    }
```

The main export symbol is the name of the routine that returns the address of the `TECPluginDispatchTable`. Because this is the main export symbol, the table is loaded after the plug-in has been installed by the Text Encoding Conversion Manager. For example,

```
TECPluginDispatchTable *GetKoreanDispatchTable(void)
                    {
                    return &KoreanPluginDispatchTable;
                    }
```

The table consists of a dispatch table format version number, a signature that uniquely identifies the plug-in, and routine pointers to the plug-in's methods. The methods are discussed later in this appendix. The compatible version number is always less than or equal to the current version number.

```
struct TECPluginDispatchTable {
    /* version information */
    TECPluginVersion version;
    TECPluginVersion compatibleVersion;
    TECPluginSignature PluginID;

    /* converter hooks */
    TECPluginNewEncodingConverterPtr        PluginNewEncodingConverter;
    TECPluginClearContextInfoPtr            PluginClearContextInfo;
    TECPluginConvertTextEncodingPtr         PluginConvertTextEncoding;
    TECPluginFlushConversionPtr             PluginFlushConversion;
    TECPluginDisposeEncodingConverterPtr    PluginDisposeEncodingConverter;
    /* sniffer hooks */
    TECPluginNewEncodingSnifferPtr          PluginNewEncodingSniffer;
    TECPluginClearSnifferContextInfoPtr     PluginClearSnifferContextInfo;
    TECPluginSniffTextEncodingPtr           PluginSniffTextEncoding;
    TECPluginDisposeEncodingSnifferPtr      PluginDisposeEncodingSniffer;

    /* Support encoding information. These hooks can be implemented as resources. */
    TECPluginGetCountAvailableTextEncodingsPtr
                                            PluginGetCountAvailableTextEncodings;
    TECPluginGetCountAvailableTextEncodingPairsPtr
                                            PluginGetCountAvailableTextEncodingPairs;
    TECPluginGetCountDestinationTextEncodingsPtr
                                            PluginGetCountDestinationTextEncodings;
    TECPluginGetCountSubTextEncodingsPtr    PluginGetCountSubTextEncodings;
    TECPluginGetCountAvailableSniffersPtr   PluginGetCountAvailableSniffers;
    TECPluginGetCountWebEncodingsPtr        PluginGetCountWebTextEncodings;
    TECPluginGetCountMailEncodingsPtr       PluginGetCountMailTextEncodings;

    TECPluginGetTextEncodingInternetNamePtr PluginGetTextEncodingInternetName;
    TECPluginGetTextEncodingFromInternetNamePtr
                                            PluginGetTextEncodingFromInternetName;
};
typedef struct TECPluginDispatchTable TECPluginDispatchTable;
```

Each plug-in must implement routines for creating the converter object, resetting the state of the converter object, encoding conversions, and disposing of the converter object. That is, the following routine pointers in the dispatch table should be valid for a basic plug-in:

```
TECPluginNewEncodingConverterPtr
TECPluginClearContextInfoPtr
TECPluginConvertTextEncodingPtr
TECPluginDisposeEncodingConverterPtr

/* You can implement the following routine pointers or use their
corresponding resources. */
TECPluginGetCountAvailableTextEncodingsPtr
TECPluginGetCountAvailableTextEncodingPairsPtr
TECPluginGetCountDestinationTextEncodingsPtr
```

Example:

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

      &ConverterPluginNewEncodingConverter,
      &ConverterPluginClearContextInfo,
      &ConverterPluginConvertTextEncoding,
      &ConverterPluginFlushConversion,
      &ConverterPluginDisposeEncodingConverter,

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

      nil, // &ConverterPluginGetAvailableTextEncodings,
      nil, // &ConverterPluginGetAvailableTextEncodingPairs,
      nil, // &ConverterPluginGetDestinationTextEncodings,
      nil, // PluginGetSubTextEncodings,

      nil, // PluginGetSniffers;
      nil, // PluginGetWebTextEncodings;
      nil, // PluginGetMailTextEncodings;

      nil, // PluginGetTextEncodingMIMEName,
      nil, // PluginGetTextEncodingFromMIMEName,
      };
```

The Text Encoding Conversion Manager communicates with its plug-ins through structures of type `TECConverterContextRec`. Context structures are created and disposed of by the Text Encoding Conversion Manager. Plug-ins are called to construct and dispose of their own data. The Text Encoding Conversion Manager and plug-ins communicate with each other in the following ways:

1. The Text Encoding Conversion Manager supplies input and output buffers to plug-ins.
2. Plug-ins report back how much text they have converted.

Note
TECConverterContextRec is used by encoding converter objects. TECSnifferContextRec is used by encoding sniffers. Encoding sniffers are discussed in later sections.

```
struct TECConverterContextRec {
    /* public - manipulated externally and within plug-in */
    Ptr pluginRec;
    TextEncoding sourceEncoding;
    TextEncoding destEncoding;
    UInt32 reserved1;
    UInt32 reserved2;
    TECBufferContextRec bufferContext;

    /* private - manipulated only within plug-in */
    UInt32 contextRefCon;
    ProcPtr conversionProc;
    ProcPtr flushProc;
    ProcPtr clearContextInfoProc;
    UInt32 options1;
    UInt32 options2;
    TECPluginStateRec pluginState; /* state information */
};
typedef struct TECConverterContextRec TECConverterContextRec;
```

Most of the public section of the TECConverterContextRec structure is maintained by the Text Encoding Conversion Manager and should not be modified by the plug-in. The bufferContext field is set up by the Text Encoding Conversion Manager to point to the input and output buffers before the conversion routine, pointed to by PluginConvertTextEncoding (a routine pointer defined in the plug-in dispatch table), is called. On exit from that routine, the plug-in should update this structure to indicate how much of the input buffer was consumed and how much text was placed in the output buffer.

```
struct TECBufferContextRec {
    TextPtr textInputBuffer;
    TextPtr textInputBufferEnd;
    TextPtr textOutputBuffer;
    TextPtr textOutputBufferEnd;
    TextPtr encodingInputBuffer; /* currently not used */
    TextPtr encodingInputBufferEnd; /* currently not used */
    TextPtr encodingOutputBuffer; /* currently not used */
    TextPtr encodingOutputBufferEnd; /* currently not used */
};
typedef struct TECBufferContextRec TECBufferContextRec;
```

The private section of the `TECConverterContextRec` structure provides persistent storage for a plug-in between conversion routine calls. It isn't modified by the Text Encoding Conversion Manager. For example, the private section can be used to store state information during a multi-pass encoding conversion. If a plug-in requires more space than is provided in this structure to keep its local data, it can maintain a pointer or a handle to its data in the `contextRefCon` field.

The fields in the private section can be used in any way a particular plug-in requires. All current Apple plug-ins set up these fields with the routine pointed to by `PluginNewEncodingConverter`, a routine pointer defined in the plug-in dispatch table, in the following way:

The `contextRefCon` field is set to `nil`. It can be used to store a handle to additional information handled by the plug-in.

The `conversionProc` field points to a routine within the plug-in that performs a specific conversion, for example, EUC to ISO-2022-JP.

The `flushProc` field points to a routine within the plug-in that flushes the output buffer with some text sequence in order to set the output buffer state to a certain text mode, such as ASCII mode. It is currently used in EUC to ISO-2022-JP conversion.

The `clearContextInfoProc` field points either to a generic routine that clears all state information in the private section or to custom routines that clear the conversion context for each specific conversion.

Only `state1`, `state2`, `state3`, and `state4` of the `TECPluginStateRec` structure are used for storing plug-in state information. But you can use the rest in any way you want.

```
struct TECPluginStateRec {
                     UInt8 state1;
                     UInt8 state2;
                     UInt8 state3;
                     UInt8 state4;
                     UInt32 longState1;
                     UInt32 longState2;
```

```
                    UInt32 longState3;
                    UInt32 longState4;
                    };

typedef struct TECPluginStateRec TECPluginStateRec;
```

When a converter object is created, the creation routine pointed to by PluginNewEncodingConverter, a routine pointer defined in the plug-in dispatch table, is called by the Text Encoding Conversion Manager to allow the plug-in to set up its TECConverterContextRec structure. This creation routine sets up the conversion routine pointer, clear context information routine pointer, flush routine pointer, and the context reference value.

The TECConverterContextRec structure needs to contain all the information the plug-in required to perform conversions between the encodings specified in inputEncoding and outputEncoding.

Note that text encoding specifications (type TextEncoding) are considered private structures. They are defined as of type UInt32 and can be passed by value. Text encoding specifications are persistent objects. For example,

```
static OSStatus ConverterPluginNewEncodingConverter(
    TECObjectRef *newEncodingConverter,
    TECConverterContextRec *plugContext,
    TextEncoding inputEncoding,
    TextEncoding outputEncoding)
{
#pragma unused( newEncodingConverter )

    OSStatus status = noErr;
    TextEncoding encodingKSC_5601_87 = CreateTextEncoding(kTextEncodingKSC_5601_87,
            kTextEncodingDefaultVariant, kTextEncodingDefaultFormat);
    TextEncoding encodingISO_2022_KR = CreateTextEncoding(kTextEncodingISO_2022_KR,
            kTextEncodingDefaultVariant, kTextEncodingDefaultFormat);
    TextEncoding encodingEUC_KR = CreateTextEncoding(kTextEncodingEUC_KR,
            kTextEncodingDefaultVariant, kTextEncodingDefaultFormat);
    TextEncoding encodingMacKorean = CreateTextEncoding(kTextEncodingMacKorean,
            kTextEncodingDefaultVariant, kTextEncodingDefaultFormat);

    /* initialize private data in plugContext */
    plugContext->conversionProc = nil;
    plugContext->clearContextInfoProc = nil;
plugContext->flushProc = nil;
plugContext->contextRefCon = (unsigned long)nil;

/* create the converter if possible */
if (inputEncoding == encodingKSC_5601_87) {

    if (outputEncoding == encodingEUC_KR || outputEncoding == encodingMacKorean) {
        plugContext->conversionProc = (ProcPtr) &ConvertKSC_5601toEUC_KR;
        plugContext->clearContextInfoProc = (ProcPtr) &ClearConverterContext;
    } else{
        status = kTextUnsupportedEncodingErr;
    }

} else if (inputEncoding ==  encodingISO_2022_KR) {
    if (outputEncoding == encodingEUC_KR || outputEncoding == encodingMacKorean) {
        plugContext->conversionProc = (ProcPtr) &ConvertISO2022KRtoEUC_KR;
        plugContext->clearContextInfoProc = (ProcPtr) &ClearConverterContext;
} else {
    status = kTextUnsupportedEncodingErr;
    }
} else if (inputEncoding ==  encodingEUC_KR ||
            inputEncoding == encodingMacKorean) {

if (outputEncoding == encodingKSC_5601_87) {
    plugContext->conversionProc = (ProcPtr) &ConvertEUC_KRtoKSC_5601;
    plugContext->clearContextInfoProc = (ProcPtr) &ClearConverterContext;
} else if (outputEncoding == encodingISO_2022_KR) {
    plugContext->conversionProc = (ProcPtr) &ConvertEUC_KRtoISO2022KR;
    plugContext->clearContextInfoProc = (ProcPtr) &ClearConverterContext;
    plugContext->flushProc = (ProcPtr) &FlushTextEUC_KRtoISO_2022_KR;
} else{status = kTextUnsupportedEncodingErr;
}
} else {
    status = kTextUnsupportedEncodingErr;
}
return status;
}
```

The clear context routine pointed to by `PluginClearContextInfo`, a routine pointer defined in the plug-in dispatch table, is called to clear out the plug-in context or state information to prepare for a new conversion of the same type.

It is always called by the Text Encoding Conversion Manager right after creating the converter object. For example,

```
static OSStatus ConverterPluginClearContextInfo(
      TECObjectRef encodingConverter,
      TECConverterContextRec *plugContext)
      {
      OSStatus status = noErr;
      status = (
      *((TECPluginClearContextInfoPtr)
      (plugContext->clearContextInfoProc))
      ) (encodingConverter, plugContext);
      return status;
      }
```

The pointer `plugContext->clearContextInfoProc` points to a clear context routine. It is set up in the `ConverterPluginNewEncodingConverter` routine above when a converter object is created. For example,

```
OSStatus ClearConverterContext(
      TECObjectRef encodingConverter,
      TECConverterContextRec *plugContext)
      {
      #pragma unused (encodingConverter)
      OSStatus status = noErr;
      if (plugContext)
      {

      // for normal state
      plugContext->pluginState.state1 = kASCIIState;

      // for shift in/out state
      plugContext->pluginState.state2 = kShiftInState;

      // for saved byte
      plugContext->pluginState.state3 = kNullSaveByte;

      // for pure KSC <-> EUC conversion
      plugContext->pluginState.state4 = kKSC5601_92State;
      plugContext->pluginState.longState1 = 0;
      plugContext->pluginState.longState2 = 0;
```

```
        plugContext->pluginState.longState3 = 0;
        plugContext->pluginState.longState4 = 0;
        }

        else
        {
        status = paramErr;
        }
        return status;
        }
```

Note that you may directly call a particular `ClearConverterContext` routine in the `ConverterPluginClearContextInfo` routine for clearing the converter context if you don't care what the conversion is. The Text Encoding Conversion Manager provides a convenient way, using the routine pointer `plugContext->clearContextInfoProc`, to call a clear context routine that is set up according to the input and output encodings when the converter object is created.

The conversion routine pointed to by `PluginConvertTextEncoding`, a routine pointer defined in the plug-in dispatch table, is called to perform the actual encoding conversion.

The `bufferContext` field of a structure of type `TECBufferContextRec`—used for the `TECConverterContextRec` parameter of the conversion routine—points to the beginning and end of the input and output buffers.

The plug-in should convert the text in the input buffer to the desired encoding and place it in the output buffer, deciding how much of the input text it can convert and fit in the output buffer. Upon exit, the plug-in needs to update the `inputBuffer` and `outputBuffer` pointers to reflect how much of the text was converted an how large the output was. The plug-in should save all necessary state information so that it can continue the conversion where it left off in the event that all of the input text could not fit, after conversion, in the output buffer. When converting the text, convert as much of the input text as you can and still fit the converted text in the output buffer. For example,

```
static OSStatus ConverterPluginConvertTextEncoding(
        TECObjectRef encodingConverter, TECConverterContextRec
        *plugContext)
        {
        OSStatus status = noErr;
```

```
      status =  (
      *((TECPluginConvertTextEncodingPtr) (plugContext->conversionProc)))

      (encodingConverter, plugContext);
      return status;
      }
```

The pointer `plugContext->conversionProc` points to a encoding conversion routine. It is setup in the `ConverterPluginNewEncodingConverter` routine above when a converter object is created. For example,

```
OSStatus ConvertISO2022KRtoEUC_KR(
      TECObjectRef encodingConverter, TECConverterContextRec
      *plugContext)
      {
      #pragma unused (encodingConverter)
      OSStatus status = noErr;

      if (plugContext) {
      BytePtr inBuf  = plugContext->bufferContext.textInputBuffer;
      BytePtr inEnd  = plugContext->bufferContext.textInputBufferEnd;
      BytePtr outBuf = plugContext->bufferContext.textOutputBuffer;
      BytePtr outEnd = plugContext->bufferContext.textOutputBufferEnd;
      Byte saveByte;
      UInt8 escState, shiftState;

      /* get state information */
      escState = plugContext->pluginState.state1;
      shiftState = plugContext->pluginState.state2;
      saveByte = plugContext->pluginState.state3;

      /* perform conversion */
      /* no error message yet if there is no input */
      while ((inBuf < inEnd) && (status == noErr))
      {
      status = HandleState(*inBuf, &escState, &shiftState,
      &saveByte, &outBuf, outEnd);

      /* Check if the buffer full status is actually */
      /* a buffer below minimum size error. */
      /* And advance the input buffer if appropriate. */
      PostProcess(plugContext->bufferContext.textOutputBuffer,
      outBuf, &inBuf, inEnd, &escState, &status);
      }
      /* save state information */
      plugContext->pluginState.state1 = escState;
      plugContext->pluginState.state2 = shiftState;
      plugContext->pluginState.state3 = saveByte;

      /* save new buffer positions */
      plugContext->bufferContext.textOutputBuffer = outBuf;
      plugContext->bufferContext.textInputBuffer  = inBuf;
      }

      else
      {
      status = paramErr;
      }

      return status;
      }
```

Note that you may not directly use the ConverterPluginConvertTextEncoding routine for converting the encodings because you don't have the conversion information. The Text Encoding Conversion Manager provides a convenient way to call a conversion routine that is set up according to the input and output encodings.

The destruction routine pointed to by PluginDisposeEncodingConverter, a routine pointer defined in the plug-in dispatch table, is called for each plug-in referenced in a converter object when it is disposed of. The plug-in is responsible for disposing of any memory or other resources such as conversion tables it may have created or loaded from disk in the creation routine. For example,

```
static OSStatus ConverterPluginDisposeEncodingConverter(
      TECObjectRef newEncodingConverter,
      TECConverterContextRec *plugContext)
      {
      OSStatus status = noErr;
      return status;
      }
```

The flush routine pointed to by PluginFlushConversion, a routine pointer defined in the plug-in dispatch table, is called to flush the output buffer to certain mode. For example, this is needed in the EUC_KR to ISO2022_KR conversion because after an input buffer has been consumed, a shift in sequence may be needed to change back to ASCII mode in the output buffer.

```
OSStatus FlushTextEUC_KRtoISO_2022_KR(
      TECObjectRef encodingConverter,
      TECConverterContextRec *plugContext)
      {
      #pragma unused( encodingConverter )

      OSStatus status = noErr;

      if (plugContext)
      {
      BytePtr outBuf = plugContext->bufferContext.textOutputBuffer;
      BytePtr outEnd = plugContext->bufferContext.textOutputBufferEnd;
      UInt8 isoState, shiftState;
      Byte saveByte;

      isoState = plugContext->pluginState.state1;
      shiftState = plugContext->pluginState.state2;
      saveByte = plugContext->pluginState.state3;
      if (shiftState != kShiftInState) {
      /* Shift in sequence */
      status = OutputEscapeSequence(
      kShiftInState, &outBuf, outEnd);

      if (status == noErr)
      {

      /* Remember to reset back to shift in mode if no error */
      isoState = kDesignationState;
      shiftState = kShiftInState;
      saveByte = kNullSaveByte;
      }

      /* Check if the buffer full status is actually */
      /* a buffer below minimum size error */
      if ((status == kTECOutputBufferFullStatus) &&
(outBuf == plugContext->bufferContext.textOutputBuffer))
status = kTECBufferBelowMinimumSizeErr;

/* Save state information & new buffer positions */
plugContext->pluginState.state1 = isoState;
plugContext->pluginState.state2 = shiftState;
plugContext->pluginState.state3 = saveByte;
plugContext->bufferContext.textOutputBuffer = outBuf;
}
}

else
{
status = paramErr;
}

return status;
}
```

Note
UTF7 maintains an internal bit buffer that needs to be flushed. ◆

The following routines, defined in the plug-in dispatch table, provide information to the Text Encoding Conversion Manager to find out what services are available to it in each of its plug-ins. These services include which encodings the plug-in knows about and which conversions it can perform on those encodings.

Some routines may be replaced by resources. Resources are preferable. However, in some cases, you might want to use the routines—for example, for the Unicode plug-in, which needs to scan tables.

The routine pointed to by `PluginGetCountAvailableTextEncodings`, a routine pointer defined in the plug-in dispatch table, counts the actual number of available text encodings and fills in an array of type `TextEncoding` with the encodings supported by the plug-in. This is used by the

TECGetAvailableTextEncodings routine in the Text Encoding Conversion Manager.

```
typedef OSStatus (*TECPluginGetCountAvailableTextEncodingsPtr)
                     (TextEncoding *availableEncodings,
                     ItemCount maxAvailableEncodings,
                     ItemCount *actualAvailableEncodings);
```

The routine pointed to by PluginGetCountAvailableTextEncodingPairs, a routine pointer defined in the plug-in dispatch table, counts the actual number of available text encoding conversions and fills in an array of type TECConversionInfo with the encoding conversions supported by the plug-in. This is used by the TECGetAvailableTextEncodings routine in the Text Encoding Conversion Manager.

```
typedef OSStatus (*TECPluginGetCountAvailableTextEncodingPairsPtr)
      (TECConversionInfo *availableEncodings,
      ItemCount maxAvailableEncodings,
      ItemCount *actualAvailableEncodings);
```

A TECConversionInfo structure is used to describe conversion services available in a plug-in. Each plug-in is required to provide information about the actual encoding conversions in a given buffer. This is used by TECGetDirectTextEncodingConversions in the Text Encoding Conversion Manager.

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

Each structure contains a pair of source and destination encodings that describes the kind of conversion the plug-in can perform. An encoding is created by using the `CreateTextEncoding` function. For example,

```
TextEncoding encodingKSC_5601_87 = CreateTextEncoding(
                    kTextEncodingKSC_5601_87,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat
                    );
```

The variant and format are discussed in conjunction with the resource of type `kTECAvailableEncodingsResType` later in this appendix.

The routine pointed to by `PluginGetCountDestinationTextEncodings`, a routine pointer defined in the plug-in dispatch table, counts the actual number of available destination text encodings. The routine also fills in an array of type `TextEncoding` with all the text encodings that the parameter `inputEncoding` can be directly converted to in one step. This routine is used by the Text Encoding Conversion Manager to find and evaluate paths from one encoding to another.

Note
A conversion may go through many intermediate encodings. ◆

```
typedef OSStatus (*TECPluginGetCountDestinationTextEncodingsPtr)
                    (TextEncoding inputEncoding,
                    TextEncoding *destinationEncodings,
                    ItemCount maxDestinationEncodings,
                    ItemCount *actualDestinationEncodings
                    );
```

The routine pointed to by `PluginGetCountSubTextEncodings`, a routine pointer defined in the plug-in dispatch table, finds out which subencodings are packaged within a text encoding. For example EUC-JP and ISO 2022-JP both contain JIS X0208, JIS X0212, JIS Roman, and half-width Katakana.

```
typedef OSStatus (*TECPluginGetCountSubTextEncodingsPtr)
                    (TextEncoding inputEncoding,
                    TextEncoding subEncodings[],
                    ItemCount maxSubEncodings,
                    ItemCount *actualSubEncodings);
```

The routine pointed to by PluginGetCountAvailableSniffers, a routine pointer defined in the plug-in dispatch table, counts the actual number of available sniffers and fills in an array of type TextEncoding with the encodings that can be sniffed by the plug-in.

```
typedef OSStatus (*TECPluginGetCountAvailableSniffersPtr)
                     (TextEncoding *availableEncodings,
                     ItemCount maxAvailableEncodings,
                     ItemCount *actualAvailableEncodings);
```

The routine pointed to by PluginGetTextEncodingInternetName, a routine pointer defined in the plug-in dispatch table, finds the name of a text encoding as it would appear in a Multipurpose Internet Mail Extensions (MIME) header. The routine pointed to by PluginGetTextEncodingFromInternetName performs the inverse.

```
typedef OSStatus (*TECPluginGetTextEncodingInternetNamePtr)
                     (TextEncoding textEncoding,
                     Str255 encodingName);

typedef OSStatus (*TECPluginGetTextEncodingFromInternetNamePtr)
                     (TextEncoding *textEncoding,
                     ConstStr255Param encodingName);
```

The routine pointed to by PluginGetCountWebTextEncodings, a routine pointer defined in the plug-in dispatch table, counts the actual number of available Web encodings and fills in an array of type TextEncoding with the Web encodings. These encodings might appear in a Web browser encoding menu.

```
typedef OSStatus (*TECPluginGetCountWebEncodingsPtr)
                     (TextEncoding *availableEncodings,
                     ItemCount maxAvailableEncodings,
                     ItemCount *actualAvailableEncodings);
```

The routine pointed to by PluginGetCountMailTextEncodings, a routine pointer defined in the plug-in dispatch table, counts the actual number of available mail encodings and fills in an array of type TextEncoding with the mail encodings. These encodings might appear in an email transfer encoding menu.

```
typedef OSStatus (*TECPluginGetCountMailEncodingsPtr)
                    (TextEncoding *availableEncodings,
                    ItemCount maxAvailableEncodings,
                    ItemCount *actualAvailableEncodings);
```

To facilitate plug-in development, avoid duplicate code, and eventually avoid unnecessarily loading a plug-in, certain data access plug-in methods can be implemented as resources. If these resources are present, the corresponding routines are never called. If this information is not available until runtime, such as is the case with the Unicode plug-in, which needs to find out which conversion tables are available, then the plug-in is loaded and the corresponding routine is called instead. If all of these are implemented as resources, then initialization of the Text Encoding Conversion Manager occurs more quickly because you don't need to load your plug-in fragment until it is required.

All resource IDs are kTECResourceID.

Resource macro	Replaces Routines
kTECAvailableEncodingsResType	PluginGetCountAvailableTextEncodings
kTECConversionInfoResType	PluginGetCountAvailableTextEncodingPairs
	PluginGetCountDestinationTextEncodings
kTECInternetNamesResType	PluginGetTextEncodingInternetName PluginGetTextEncodingFromInternetName
kTECLocalizedNamesResType	PluginGetTextEncodingLocalizedName
kTECAvailableSniffersResType	PluginGetCountAvailableSniffers
kTECWebEncodingsResType	PluginGetCountWebTextEncodings
kTECMailEncodingsResType	PluginGetCountMailTextEncodings
kTECSubTextEncodingsResType	PluginGetCountSubTextEncodings

The above resources are discussed below.

The following resource type provides information that tells which encodings the plug-in knows about.

```
/* supported encodings list */

type kTECAvailableEncodingsResType {
    longint = $$CountOf (memberArray);
    Array memberArray {
        memberStart:
        TECTextEncoding /* encoding */
        memberEnd:
    };
};
```

For example,

```
resource kTECAvailableEncodingsResType (kTECResourceID) {
                    {
                    kTextEncodingKSC_5601_87,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingISO_2022_KR,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingMacKorean,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingEUC_KR,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    }
                    };
```

The above example shows that there are four encodings, namely, kTextEncodingKSC_5601_87, kTextEncodingISO_2022_KR, kTextEncodingMacKorean, and kTextEncodingEUC_KR, that this plug-in knows about. Since the encodings do not have special variants and formats, default variants and formats are used. If a plug-in supports different variants and formats, the text encodings must appear in the list.

The first value in the resource entries above, kTextEncodingKSC_5601_87 (0x0640), with type TextEncodingBase (UInt32), as defined in TextCommon.h, is the primary specification of the source or destination encoding. The values 0 through 32 (0x00 through 0x0020) correspond to Mac OS script codes.

The second value, with type TextEncodingVariant (UInt32), specifies the minor variant of the base encoding. For a given TextEncodingBase, the enumeration of variants always begins with 0. The value kTextEncodingDefaultVariant specifies the default variant of the base encoding.

The last value, with type TextEncodingFormat (UInt32), designates a particular way of algorithmically transforming a particular encoding, say for transmission through communication channels that may handle only 7-bit values. These transformations are not viewed as different encodings, but merely as different formats for representing the same encoding. The value kTextEncodingDefaultFormat specifies the default format of the base encoding.

Note
Only Unicode encodings can take non-zero formats currently. ◆

The following resource type provides information identifying which encoding conversions the plug-in can perform.

```
/* Conversion pairs */

type kTECConversionInfoResType {
    longint = $$CountOf (memberArray);
    Array memberArray {
        memberStart:
        TECTextEncoding     /* source encoding */
        TECTextEncoding     /* dest encoding */

        longint res1;      /* reserved - free */
        longint res2;      /* reserved - free   */
        memberEnd:
    };
};
```

For example,

```
resource kTECConversionInfoResType (kTECResourceID) {
                    {
                    /* Round trip KSC 5601 to MacKorean */
                    kTextEncodingKSC_5601_87,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingMacKorean,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat, 0, 0,
                    kTextEncodingMacKorean,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingKSC_5601_87,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat, 0, 0,

                    /* Round trip ISO 2022 to MacKorean */
                    kTextEncodingISO_2022_KR,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingMacKorean,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat, 0, 0,
                    kTextEncodingMacKorean,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingISO_2022_KR,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat, 0, 0,
                    ...
                    }
                    };
```

The following resource type provides the name of a text encoding as it would appear in a Multipurpose Internet Mail Extensions (MIME) header. Multiple encodings can map to one Internet MIME name, but an Internet MIME name maps only to the first encoding found.

```
/* Internet names */

type kTECInternetNamesResType {
    longint = $$CountOf (memberArray);
    Array memberArray {

    memberStart:
    ListStart:
    longint = (ListEnd[$$ArrayIndex(memberArray)] -
        ListStart[$$ArrayIndex(memberArray)]) / 8 - 4;
                                    /* offset to next item */
    TECTextEncoding                 /* text encoding of name */
    pstring;                        /* encoding name */
    align long;                     /* match size to C structure size */

    ListEnd:
    memberEnd:
    };
};
```

For example,

```
resource kTECInternetNamesResType (kTECResourceID) {
                     {
                     kTextEncodingKSC_5601_87,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat,
                     "KS_C_5601-1987",
                     kTextEncodingKSC_5601_87,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat,
                     "KSC_5601",
                     kTextEncodingISO_2022_KR,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat,
                     "ISO-2022-KR",
                     kTextEncodingEUC_KR,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat,
                     "EUC-KR”
                     }
                     };
```

The above example shows that there are three encodings, namely, kTextEncodingKSC_5601_87, kTextEncodingISO_2022KR, and kTextEncodingEUC_KR, for which this plug-in knows the Internet names. Because the encodings do not have special variants and formats, default variants and formats are used. One of the encodings, kTextEncodingKSC_5601_87, has two Internet names, namely, KS_C_5601-1987 and KSC_5601.

The following resource type provides information about the available sniffers.

```
/* supported sniffers list */

type kTECAvailableSniffersResType {
    longint = $$CountOf (memberArray);
    Array memberArray {
    memberStart:
    TECTextEncoding /* encoding */
    memberEnd:
    };
};
```

For example,

```
resource kTECAvailableSniffersResType (kTECResourceID) {
                    {
                    kTextEncodingKSC_5601_87,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingISO_2022_KR,
                    kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    kTextEncodingEUC_KR, kTextEncodingDefaultVariant,
                    kTextEncodingDefaultFormat,
                    }
                    };
```

The following resource type provides information about the available Web encodings.

```
/* Web encodings */

type kTECWebEncodingsResType {
    longint = $$CountOf (memberArray); /* number of sets in resource */
    Array memberArray {
        memberStart:
        ListStart:

        longint = (ListEnd[$$ArrayIndex(memberArray)] -
            ListStart[$$ArrayIndex(memberArray)]) / 8 - 4;
                                /* offset to next item */
        longint = $$CountOf (localesArray);
```

```
                              /* number of encodings in resource */
        Array localesArray {
            TECLocale           /* search locales */
        };

        longint = $$CountOf (webEncodingsArray);
                                /* number of encodings in resource *
        Array webEncodingsArray {
            TECTextEncoding     /* Web encodings */
        };

        ListEnd:
        memberEnd:
        };
    };
```

For example,

```
resource kTECWebEncodingsResType (kTECResourceID) {
      {

      /* Korean encodings */
      {
      verKorea, /* Korean Republic of Korea */
      },

      {
      kTextEncodingISO_2022_KR,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
      kTextEncodingEUC_KR,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat
      },

      }
      };
```

The following resource type provides information about the available encodings for electronic mail (e-mail) by region.

```
/* mail encodings */

type kTECMailEncodingsResType {
    longint = $$CountOf (memberArray); /* number of sets in resource */
    Array memberArray {
        memberStart:
        ListStart:

    longint = (ListEnd[$$ArrayIndex(memberArray)] -
        ListStart[$$ArrayIndex(memberArray)]) / 8 - 4;
                                /* offset to next item */
        longint = $$CountOf (localesArray);
                                /* number of encodings in resource */
        Array localesArray {
            TECLocale               /* search locales */
        };
        longint = $$CountOf (mailEncodingsArray);
                                /* number of encodings in resource */
        Array mailEncodingsArray {
                TECTextEncoding     /* mail encodings */
        };

        ListEnd:
        memberEnd:
    };
};
```

For example,

```
resource kTECMailEncodingsResType (kTECResourceID) {
      {

      /* Korean encodings */
      {
      verKorea, /* Korean Republic of Korea */
      },

      {
      kTextEncodingMacKorean,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
        kTextEncodingISO_2022_KR,
        kTextEncodingDefaultVariant,
        kTextEncodingDefaultFormat,
        kTextEncodingEUC_KR,
        kTextEncodingDefaultVariant,
        kTextEncodingDefaultFormat,
        kTextEncodingUnicodeV2_0,
        kTextEncodingDefaultVariant,
        kUnicodeUTF7Format,
        kTextEncodingUnicodeV2_0,
        kTextEncodingDefaultVariant,
        kUnicodeUTF8Format
        },

        }
        };
```

The following resource type provides information about which subencodings are packaged within a text encoding. For example ISO 2022-JP and EUC-JP both contain JIS Roman, JIS X0208, JIS X0212, and half-width Katakana.

```
/* subencodings */

type kTECSubTextEncodingsResType {
    longint = $$CountOf (memberArray);
                        /* number of sets of subencodings in resource */
    Array memberArray {
        memberStart:
        ListStart:
    longint = (ListEnd[$$ArrayIndex(memberArray)] -
            ListStart[$$ArrayIndex(memberArray)]) / 8 - 4;
                        /* offset to next item */
    TECTextEncoding     /* search encoding */
    longint = $$CountOf (subEncodingsArray);
                         /* number of subencodings in resource */

    Array subEncodingsArray {
        TECTextEncoding /* search encoding */
    };

    ListEnd:
```

```
    memberEnd:
    };
};
```

For example,

```
resource kTECSubTextEncodingsResType (kTECResourceID) {
      {
      kTextEncodingISO_2022_JP,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,

      {
      kTextEncodingISOLatin1,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
      kTextEncodingJIS_X0208_90,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
      kTextEncodingJIS_X0212_90,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,

      /* half-width katakana */
      kTextEncodingJIS_X0201_76,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
      },

      kTextEncodingEUC_JP,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,

      {
      kTextEncodingISOLatin1,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
      kTextEncodingJIS_X0208_90,
      kTextEncodingDefaultVariant,
      kTextEncodingDefaultFormat,
      kTextEncodingJIS_X0212_90,
        kTextEncodingDefaultVariant,
        kTextEncodingDefaultFormat,

        /* half-width katakana */
        kTextEncodingJIS_X0201_76,
        kTextEncodingDefaultVariant,
        kTextEncodingDefaultFormat,
        ...
        }

        }
        };
```

Sniffers allow the Text Encoding Conversion Manager to detect the encoding characteristics of a text stream. A context record of the sniffer is provided for plug-ins and Text Encoding Conversion Manager communication. A sniffer is created by the Text Encoding Conversion Manager and the routine pointed to by `PluginNewEncodingSniffer`, a routine pointer defined in the plug-in dispatch table, is called. All sniffer routines are defined in the plug-in dispatch table. They are discussed below.

The sniffer context structure `TECSnifferContextRec` is similar to `TECConverterContextRec`. Its public section contains information set up by the Text Encoding Conversion Manager and returns information to the caller. The private section is available for plug-in use.

```
struct TECSnifferContextRec {
        /* public - manipulated externally and by plug-in */
        Ptr pluginRec;
        TextEncoding encoding;
        ItemCount maxErrors;
        ItemCount maxFeatures;
        TextPtr textInputBuffer;
        TextPtr textInputBufferEnd;
        ItemCount numFeatures;

        /* will be output to caller */
        ItemCount numErrors;

        /* private - manipulated only within plug-in */
        UInt32 contextRefCon;
        ProcPtr sniffProc;
```

```
      ProcPtr clearContextInfoProc;
      TECPluginStateRec pluginState; /* state information */
      };

typedef struct TECSnifferContextRec TECSnifferContextRec;
```

When a sniffer object is created in the Text Encoding Conversion Manager, the routine pointed to by `PluginNewEncodingSniffer`, a routine pointer defined in the plug-in dispatch table, is called by the Text Encoding Conversion Manager to allow the plug-in to set up its sniffer context structure `TECSnifferContextRec`.

Example:

```
OSStatus ConverterPluginNewEncodingSniffer(
                     TECSnifferObjectRef *encodingSniffer,
                     TECSnifferContextRec *snifContext,
                     TextEncoding inputEncoding)
                     {
                     #pragma unused (encodingSniffer)
                     OSStatus status = noErr;

                     TextEncoding encodingKSC_5601_87 =
                     CreateTextEncoding(kTextEncodingKSC_5601_87,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat);

                     TextEncoding encodingISO_2022_KR =
                     CreateTextEncoding( kTextEncodingISO_2022_KR,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat);

                     TextEncoding encodingEUC_KR =
                     CreateTextEncoding( kTextEncodingEUC_KR,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat);

                     TextEncoding encodingMacKorean =
                     CreateTextEncoding( kTextEncodingMacKorean,
                     kTextEncodingDefaultVariant,
                     kTextEncodingDefaultFormat);

                     if (snifContext)
                     {
                     if (inputEncoding == encodingKSC_5601_87)
                     snifContext->sniffProc = (ProcPtr) SniffKSC_5601;

                     else if (inputEncoding == encodingISO_2022_KR)
                     snifContext->sniffProc = (ProcPtr) SniffISO2022KR;

                     else if (inputEncoding == encodingEUC_KR ||
                     inputEncoding == encodingMacKorean)
                     snifContext->sniffProc = (ProcPtr) SniffEUC_KR;

                     else
                     status = kTextUnsupportedEncodingErr;
                     }

                     else
                     {
                     status = paramErr;
                     }

                     return status;
                     }
```

The routine pointed to by `PluginClearSnifferContextInfo`, a routine pointer defined in the plug-in dispatch table, is called to clear the sniffer context state information for sniffing a new input buffer. This is always called by the Text Encoding Conversion Manager right after creating the sniffer.

Example:

```
OSStatus ConverterPluginClearSnifferContextInfo(
                     TECSnifferObjectRef encodingSniffer,
                     TECSnifferContextRec *snifContext)
                     {
                     #pragma unused (encodingSniffer)
                     OSStatus status = noErr;

                     if (snifContext) {
                     snifContext->pluginState.state1 = kASCIIState;
                     snifContext->pluginState.state2 = kShiftInState;
                     snifContext->pluginState.state3 = 0;
                     snifContext->pluginState.state4 = 0;
```

```
                    snifContext->numFeatures = 0;
                    snifContext->numErrors = 0;
                    }

                    else
                    {
                    status = paramErr;
                    }

                    return status;
                    }
```

The routine pointed to by `PluginSniffTextEncoding`, a routine pointer defined in the plug-in dispatch table, is called to perform the actual sniffing. To sniff text encodings, loop through the input buffer and count errors and features. The Text Encoding Conversion Manager looks at the number of errors and features to determine the encoding of the given text. The routine is pointed to by `snifContext->sniffProc` to `ConverterPluginNewEncodingSniffer`, which is also defined in the plug-in dispatch table, when the sniffer is created. For example,

```
OSStatus SniffEUC_KR(
      TECSnifferObjectRef encodingSniffer,
      TECSnifferContextRec *snifContext)
      {
      #pragma unused (encodingSniffer)
      OSStatus status = noErr;

      if (snifContext)
      {
      BytePtr inputBuffer = snifContext->textInputBuffer;
      BytePtr inputBufferEnd = snifContext->textInputBufferEnd;
      ItemCount *numErrs = &snifContext->numErrors;
      ItemCount maxErrs = snifContext->maxErrors;
      ItemCount *numFeatures = &snifContext->numFeatures;
      ItemCount maxFeatures = snifContext->maxFeatures;

      if (inputBuffer && inputBufferEnd)
      {
      Byte c;
      UInt8 isoState = snifContext->pluginState.state1;
ItemCount errs = *numErrs;
ItemCount features = *numFeatures;

while(errs < maxErrs && features < maxFeatures &&
inputBuffer < inputBufferEnd)
{
c = *inputBuffer++; /* count errors and features in encoding */
/* set status when appropriate */
...
}

/* save state information */
snifContext->pluginState.state1 = isoState;

/* save number of errors and features */
*numErrs = errs;
*numFeatures = features;
} else {
status = paramErr;

/* Initialization.  Just in case. */
*numErrs = 0;
*numFeatures= 0;
}
}

else
{
status = paramErr;
}

return status;
}
```

The destruction routine pointed to by `PluginDisposeEncodingSniffer`, a routine pointer defined in the plug-in dispatch table, is called when the sniffer is disposed of. To dispose of the sniffer, simply dispose of any memory or resources that may have been allocated in the creation routine.

Example:

```
OSStatus ConverterPluginDisposeEncodingSniffer(
                    TECSnifferObjectRef encodingSniffer,
                    TECSnifferContextRec *snifContext)
                    {
                    #pragma unused (encodingSniffer, snifContext)
                    /* nothing to do */

                    return noErr;
                    }
```

All plug-in routines should return values with `OSStatus` type, except the three routines named by the plug-in library symbols.

Some common status and error codes that may be returned to the Text Encoding Conversion Manager using type `OSStatus` are listed below:

- `kTECOutputBufferFullStatus`—Output buffer is full before all text could be converted.
- `noErr`—No error occurred or status is normal.
- `paramErr`—One or more of the input parameters has an invalid value.
- `kTextUnsupportedEncodingErr`—The given encoding is not supported in the current plug-in.
- `kTECBufferBelowMinimumSizeErr`—The output text buffer is too small to allow processing of the first input text element.
- `kTECPartialCharErr`—The input text ends in the middle of a multi-byte character, conversion stopped. In this case, the plug-in code should save the state in its private space and the input pointer should back up to the beginning of the multi-byte character.
- `kTextMalformedInputErr`—The text input contained a sequence that is not legal in the specified encoding.

The plug-in should have `'encv'` for file creator and '`ecpg`' for file type.

The `'cfrg'` resource serves to inform the Process Manager and Code Fragment Manager of code fragments. The resource ID must be zero.

Example:

```
#ifdef PPC
                     resource 'cfrg' (0) {

                     {
                     kPowerPC, /* instruction set architecture */
                     kFullLib, /* base-level library */
                     kNoVersionNum, /* no implementation version number*/
                     kNoVersionNum, /* no definition version number */
                     kDefaultStackSize, /* use default stack size */
                     kNoAppSubFolder, /* no library directory */
                     kIsDropIn, /* fragment is a drop-in library */
                     kOnDiskFlat, /* fragment is in the data fork */
                     kZeroOffset, /* fragment starts at offset 0 */
                     kWholeFork, /* fragment occupies entire fork */
                     "KoreanPlugin" /* name of the library fragment */
                     }

                     };

                     #else
                     resource 'cfrg' (0) {

                     {
                     kMotorola, /* instruction set architecture */
                     kFullLib, /* base-level library */
                     kNoVersionNum, /* no implementation version number*/
                     kNoVersionNum, /* no definition version number */
                     kDefaultStackSize, /* use default stack size */
                     kNoAppSubFolder, /* no library directory */
                     kIsDropIn, /* fragment is a drop-in library */
                     kOnDiskFlat, /* fragment is in the data fork */
                     kZeroOffset, /* fragment starts at offset 0 */
                     kWholeFork, /* fragment occupies entire fork */
                     "KoreanPlugin" /* name of the library fragment */
                     };
                     #endif
```

The `'vers'` resource provides the version information. The resource ID must be 1.

Example:

```
resource 'vers' (1, purgeable)
                    {
                    0x01, 0x20, final, 0x00,
                    verUS,
                    "1.2",
                    "1.2, Copyright Apple Computer, Inc. 1994-1997."
                    };
```

Here is the URL of a Web site that gives useful encoding conversion information:

`http://www.ora.com/people/authors/lunde/cjk-char.html`

The Request For Comments (RFC) documents can be found at:

`http://www.cis.ohio-state.edu/hypertext/information/rfc.html`

Character Encodings Concepts

This appendix is adapted from a tutorial created by Peter Edberg that was presented at the 11th International Unicode Conference. The original paper will be published in the Proceedings of that conference with a notice indicating joint copyright by Apple Computer, Inc. and the Unicode Consortium.

The appendix explores some aspects of character encodings, including

- terms used, such as coded character sets, character encoding schemes, characters, glyphs, and related concepts
- existing character encodings, focusing on important Internet encodings and how these encodings relate to the Unicode standard
- special features of various character encodings
- character data in programming languages
- Internet character encoding registry and encoding naming conventions

Terminology

Many of the terms defined in this section are used informally. They are defined in order to facilitate the discussion in the remainder of this appendix.

Character Sets and Encoding Schemes

A recent meeting on character sets organized by the Internet Architecture Board proposed a 7-layer architectural model for the transmission of text data. The first three layers are required for specifying the content of a transmitted text stream “on the wire”; higher layers specify language, locale, and so forth. As specified in the minutes of that meeting, the first three layers are

- **coded character set (CCS)**, a mapping from a set of abstract characters to a set of integers. Examples include ISO 10646, ASCII, and the ISO 8859 series.

- **character encoding scheme (CES)**, a mapping from one or more CCSs to a set of octets. Examples include ISO 2022 and UTF-8. A given CES is typically associated with a single CCS; for example, UTF-8 applies only to ISO 10646.
- **transfer encoding syntax (TES)**, a transformation applied to character data encoded using a CCS and possibly a CES to allow it to be transmitted by a specific protocol or set of protocols. Examples include base64 and quoted-printable.

Note
The term integer is used in this appendix in its mathematical sense; that is, it does not refer to the integer size on a particular CPU. Also, the term octet is used here instead of byte because the latter has not always meant an 8-bit unit; octet is explicitly defined to be an ordered sequence of 8 bits considered as a unit (the term is from ISO character set standards. ◆

Other documents offer slightly different definitions of characteristics of a CCS, for example, a repertoire of abstract characters, range of numbers, and a mapping from numbers to characters (not necessarily invertible). Each of the integers in the set used to represent a CCS is called a code point.

A CES might be more accurately described as a mapping from a sequence of elements in one or more CCSs to a sequence of octets. This definition suggests that the mapping from a single CCS element to its representation in the CES does not fully characterize the CES, which may include additional octets to set or change state information.

A TES is usually used to send 8-bit data through a transport mechanism that is only safe for 7-bit data, and even then may perform special handling for certain 7-bit values.

This appendix frequently uses the shorter term *character set* to mean coded character set and *character encoding* or *encoding scheme* to encompass both character sets and more complex character encoding schemes.

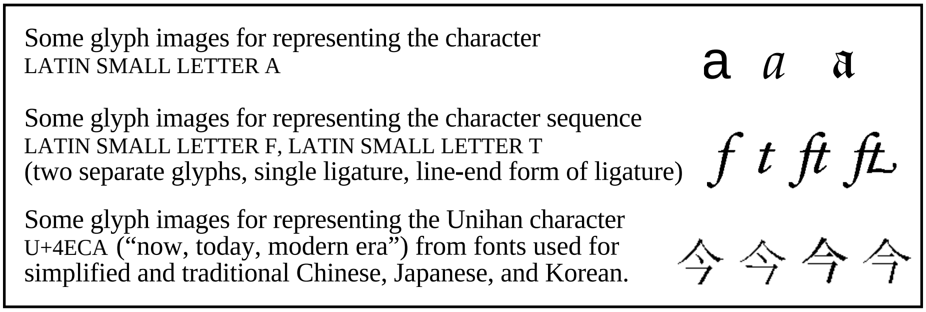
Characters, Glyphs, and Related Terms

Characters are the atomic units of content for text data; they include letters, digits, punctuation, and symbols. A **character** is an abstract entity without any particular appearance. A **coded character** is a character together with its numeric representation in a particular CCS.

A **text element** is a group of one or more characters that is treated as a single entity for a particular process such as collation, display, or transcoding. The way that characters are grouped into text elements depends on the process; each process may group characters differently.

Glyph images are the visual elements used to represent characters; aspects of text presentation such as font and style apply to glyph images, not to characters. The mapping from a sequence of coded characters to a sequence of glyph images on a display device is complex. In general there is not a one-to-one mapping from character to glyph image; a particular glyph image may correspond to more or less than one character. Figure B-1 shows glyphs and their associated characters.

Figure B-1 Some glyph images for representing characters



Some glyph images for representing the character LATIN SMALL LETTER A

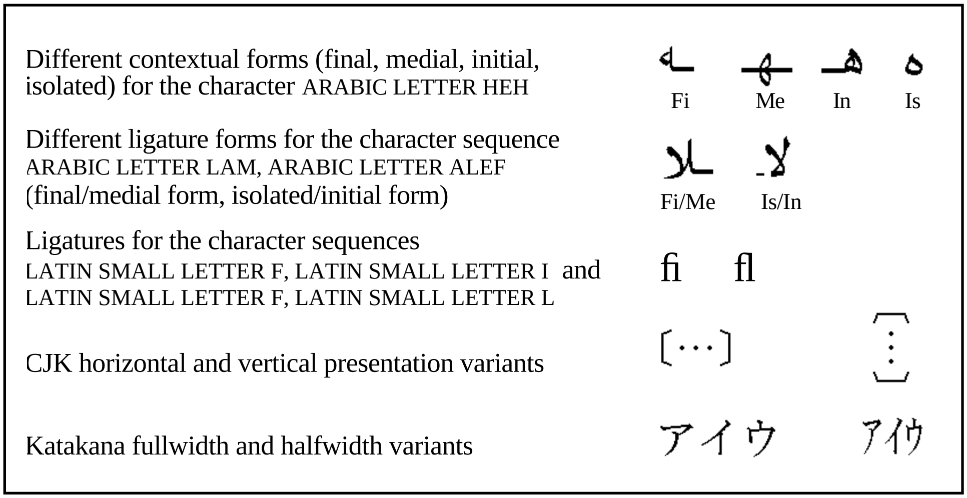
Some glyph images for representing the character sequence LATIN SMALL LETTER F, LATIN SMALL LETTER T (two separate glyphs, single ligature, line-end form of ligature)

Some glyph images for representing the Unihan character U+4ECA (“now, today, modern era”) from fonts used for simplified and traditional Chinese, Japanese, and Korean.

A **script** is a collection of related characters, subsets of which are required to write a particular language. Some examples of scripts are Latin, Greek, Hiragana, Katakana, and Han. A **writing system** consists of a set of characters from one or more scripts that are used to write a particular language and the rules that govern the presentation of those characters. Punctuation, digits, and symbols that are shared across many writing systems can be considered as one or more separate pseudo-scripts. For example, the Japanese writing system includes a Kanji subset of Han characters, plus Hiragana, Katakana, some Latin, and various punctuation and symbols, some of which are specific to CJK—Chinese, Japanese, Korean—or even just to Japanese, and some of which are more general.

The term **presentation form** is generally used to mean a kind of abstract shape that represents a standard way to display a particular character or group of characters in a particular context as specified by a particular writing system. The term **glyph** by itself may refer either to presentation forms or to glyph images. This appendix assumes the latter convention. Figure B-2 shows some examples of presentation forms.

Figure B-2 Presentation forms



Different contextual forms (final, medial, initial, isolated) for the character ARABIC LETTER HEH

Different ligature forms for the character sequence ARABIC LETTER LAM, ARABIC LETTER ALEF (final/medial form, isolated/initial form)

Ligatures for the character sequences LATIN SMALL LETTER F, LATIN SMALL LETTER I and LATIN SMALL LETTER F, LATIN SMALL LETTER L

CJK horizontal and vertical presentation variants

Katakana fullwidth and halfwidth variants

The determination of what is a character in a CCS should be based on what is best for implementing the range of text processes for which that CCS will be used. The characters in a CCS need not correspond to what a user or linguist might consider a character. In fact, if the CCS will be used for more than one writing system, this might be impossible to do anyway, since each writing system has its own notion of what constitutes a natural character. Well-designed software should provide users with the behavior they expect or prefer, regardless of the details of the underlying character encoding, and without exposing users to those details.

Some character sets that were intended primarily for display using less sophisticated display software have encoded presentation forms as characters. For example, the DOS Arabic character set (code page 864) encodes Arabic contextual forms and ligatures instead of abstract letters.

Non-Unicode Character Encodings

Most of these encodings are designed to support one writing system, or a group of writing systems that use the same script. As a result, in some cases certain encodings are treated as implying a particular language, which is information that should be several layers higher in the architectural model described previously in this appendix.

Appendix C provides a more complete list of character encodings (but with less explanatory material), grouped by the writing systems they cover.

General Character Set Structure

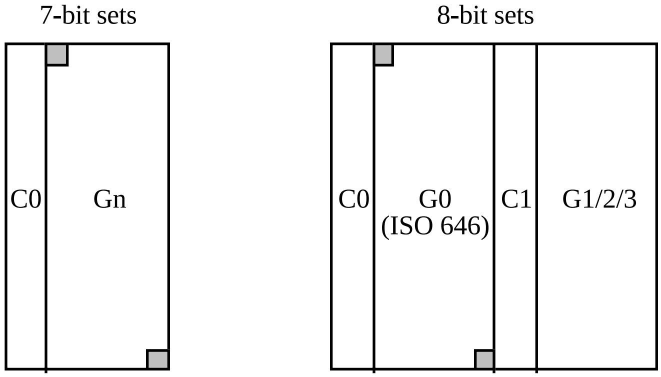
ISO 2022 and ISO 4873 define a structure for coded character sets using 7-bit or 8-bit values. These coded character sets provide a means of representing both graphic characters and control functions; control functions that can be represented with a single code point are also called control characters.

For character sets using 7-bit values, the range 0x00–0x1F is reserved for a set of 32 control characters, designated C0; another set of 32 control functions, designated C1, may be represented with escape sequences. The range 0x20–0x7F (96 code points) is reserved for up to four sets of graphic characters, designated G0–G3 (in some graphic sets, each code point requires two or three 7-bit values). Most Gn sets use only the 94 code points 0x21–0x7E, in which case 0x20 is reserved for SPACE, and 0x7F is reserved for DELETE. ISO 2022 specifies a protocol for

- assigning real sets of control functions, drawn from another standard, to C0 and possibly C1
- assigning real sets of graphic characters, drawn from another standard, to G0 and possibly G1, G2, and G3
- switching among the Gn sets for use of the range 0x20–0x7F

For 8-bit character sets, the C0 set uses 0x00–0x1F, but the C1 set uses 0x80–0x9F. The G0 set uses 0x21–0x7E (with SPACE and DELETE reserved), but the G1, G2, and G3 sets share the range 0xA0–0xFF (96 code points). Figure B-3 shows these differences.

Figure B-3 Comparison of 7-bit and 8-bit character set structures



The G0 set is typically the ISO 646 international reference version (ASCII). The C0 and C1 control functions are typically from ISO 6429, although other control sets can be used.

Simple Coded Character Sets

All of these use a fixed number of 7-bit or 8-bit values to represent the code point. Here are some examples for different code point sizes.

- One 7-bit value (these can provide a Gn set that adheres to the ISO structure):
  - ASCII, as specified by ANSI X3.4. This is a U.S. national standard, and is the U.S. national variant of ISO 646.
  - ISO 646, an international standard. It is similar to ASCII, except that for ten code points (corresponding to ASCII characters @ [\ ] ^ ` { | } ~) it does not designate a specific character, and for two other code points (corresponding to ASCII characters $ #) it allows either of two specified characters. National variants are defined by designating some of these code points to represent specific non-ASCII characters needed for a particular language. A sender and receiver can agree on a particular variant; in the absence of such an agreement, ISO specifies an international reference version, which is now the same as ASCII. For example, the Japanese national variant (known as JIS Roman) replaces ASCII \ with ¥ , and replaces ASCII ~ with ‾ .

- □ Some older national and regional standards that are not ISO 646 variants, such as SI 960 for Hebrew and ASMO 449 for Arabic.

■ One 8-bit value:

- □ ISO 8859-x. This international standard has multiple parts. ISO 8859-1 is well known as Latin-1, the most common encoding on the Web. ISO 8859 includes other Latin parts, such as Latin-5 (ISO 8859-9, used for Turkish), as well as parts for Cyrillic, Greek, Arabic, Hebrew, and other scripts. These adhere to the ISO 8-bit structure: The range 0x00–0x1F is reserved for C0 controls, 0x20 is SPACE, the range 0x21–0x7E is identical to ASCII, x7F is DELETE, the range 0x80–0x9F is reserved for C1 controls, and the range 0xA0–0xFF contains a 96-character G1 set that depends on the 8859 part.
- □ ASCII-based vendor character sets for non-East-Asian scripts: DOS code pages such as 437, Windows code pages such as 1252, Mac OS character sets, and so on. These support the ASCII graphic characters directly, but they typically do not follow the full 8-bit structure used for ISO standards; for example, they typically encode graphic characters in the C1 area. Windows 1252, for example, is ISO 8859-1 plus additional characters in the C1 area.
- □ National standards such as TIS (Thai Industrial Standard) 620-2533 and JIS (Japanese Industrial Standard) X0201. JIS X0201, for example, combines JIS Roman with a set of Katakana and punctuation characters in the range 0xA1–0xDF.
- □ ISO character sets for bibliographic use, such as ISO 5426, which often use nonspacing diacritic characters (in these standards, nonspacing marks precede the base character).
- □ EBCDIC character sets used on IBM mainframes and midrange machines. The layout is based on Hollerith card codes, and is quite different from ASCII. The basic Latin letters are in six discontiguous ranges a–i, j–r, s–z, A–I, J–R, S–Z, all with code points above 0x80; control characters are 0x00–0x3F and 0xFF. The original EBCDIC-US had a graphic character repertoire somewhat different from ASCII: it did not include square brackets or a circumflex accent, but did include cent sign, broken bar, not sign, and no-break space; it also had 95 undefined code points scattered about. Fourteen of the original EBCDIC-US code points could be changed for national variants (as with ISO 646). Newer versions of EBCDIC fill in the undefined code points with characters from ISO 8859-1 or other standards.

- Two 7-bit values (Any of these can be used as a Gn set within the ISO framework):
    - □ Japan: The original Japanese 2-byte national standard was JIS C6226-1978. This was significantly revised as JIS X0208-1983, with a minor update in 1990. It includes punctuation and symbols (some specific to CJK or to Japanese), Hiragana, Katakana, and 6356 Kanji (Han), as well as basic letters for Latin, Greek, and Cyrillic (all in 2-byte form). JIS X0212 (1990) is an add-on set with additional Kanji (5801), additional Latin characters, and so forth. JIS C6226 provided a model for other East Asia national standards.
    - □ China: GB 2312-1980 is the basic national standard, with 6763 Hanzi (Han), punctuation and symbols, Katakana, Hiragana, basic Latin, Greek, and Cyrillic, plus Bopomofo.
    - □ Korea: KSC 5601-1987 is the most widely known of the Korean national standards. It includes 2350 composed Hangul syllables, 4620 distinct Hanja (Han), punctuation and symbols, Katakana, Hiragana, basic Latin, Greek, and Cyrillic; some of the Hanja are encoded multiple times, once for each pronunciation. This standard was updated in 1992; the basic standard was not significantly changed, but a new annex defined a complete "Johab" set of the 11,172 possible composed Hangul syllables.
    - □ Taiwan: CNS 11643-1992 defines a set of 2-byte standards, something like the parts of ISO 8859. Each part is called a plane, and the standard defines 16 planes. Only 7 planes currently have character assignments; altogether they include 48,027 Hanzi and ~700 other characters.
- Three 7-bit values (these are mainly for bibliographic usage):
    - □ CCCII (Chinese Character Code for Information Interchange): The high-order value specifies the plane; planes are grouped into sets of 6, called layers. The first layer (53,016 code points) contains basic characters; most of the other layers are reserved for variant forms, which are assigned code points that correspond to the position of the equivalent basic character. The remaining layers contain Kana and Hangul (for Japanese and Korean).
    - □ EACC (East Asia Character Code): This is a U.S. standard (ANSI Z39.64) based on CCCII.

Packing Schemes for Multiple Character Sets

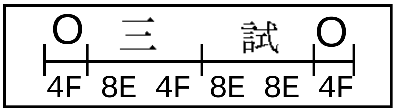
Packing schemes use a sequence of 8-bit values, so they are generally not suitable for mail (although they are often used on the Web). In these schemes, certain characters function as a *local shift* that controls the interpretation of the next 1–3 bytes.

The most well-known packing scheme is probably Shift-JIS, which was originally developed by Microsoft for use with MS-DOS. It includes the following:

- The characters from JIS X0201, represented as single bytes, with same code points as in JIS X0201: 0x00–0x7F and 0xA1–0xDF.
- The characters from JIS X0208, represented as 2 bytes, with the first byte in the range 0x81–0x9F or 0xE0–0xEF and the second byte in the range 0x40–0x7E or 0x80–0xFC.
- Space for 2444 user-defined characters, represented as 2 bytes, with the first byte in the range 0xF0–0xFC, and the second byte in the range 0x40–0x7E or 0x80–0xFC.

The 2-byte units all begin with byte values that are not used for JIS X0201, so it is possible to distinguish them if the text is processed serially from the beginning of a buffer. However, the second bytes of 2-byte units use values that can be confused either with the first byte of a 2-byte unit or with a single-byte code point from JIS X0201; when pointing into an arbitrary location in the middle of Shift-JIS text, it may be impossible to determine character boundaries. Figure B-4 shows this with a somewhat pathological Shift-JIS byte sequence using only two different byte values (the corresponding character images are also shown).

Figure B-4 Shift-JIS byte sequence



Moreover, Shift-JIS contains multiple representations of the Katakana and basic Latin repertoires, which are available in 1-byte form via JIS X0201, and in 2-byte form via JIS X0208. Shift-JIS has a well-deserved reputation as a troublesome encoding scheme.

The EUC (Extended UNIX Code) packing schemes were originally developed for UNIX systems; they use units of 1 to 4 bytes.

- EUC-JP (Japanese) combines JIS-Roman, the JIS X0201 Katakana and related punctuation, JIS X0208, and JIS X0212:

Character Set	**Range of Corresponding EUC Sequence**
JIS-Roman	0x21–0x7E (same as JIS-Roman code point)
JIS X0208	0xA1A1–0xFEFE (X0208 code point + 0x8080)
JIS X0201, Katakana, etc.	0x8EA1–0x8EDF (0x8E, then X0201 code point)
JIS X0212	0x8FA1A1–0x8FFEFE (0x8F, then X0212 code point + 0x8080)

- EUC-CN (simplified Chinese) combines ASCII, GB 2312 (adds 0x8080 to GB code point)
- EUC-KR (Korean) combines ASCII, KSC 5601-1987 (adds 0x8080 to KSC code point)
- EUC-TW (traditional Chinese) combines ASCII and all 16 planes of CNS 11643-1992. The 16 planes are encoded as 0x8E, then the plane number + 0xA0, then the CNS code point + 0x8080. In addition, Plane 1 is redundantly encoded as simply the CNS code point + 0x8080.

The Big 5 encoding is a special case. This is not a national standard, but a de facto encoding used for traditional Chinese. It combines ASCII—represented as 1-byte units—with 2-byte units that represent Hanzi, CJK punctuation and symbols, and other characters. There is no separate specification for the set of characters represented by the 2-byte units, although the Hanzi repertoire matches the CNS 11643 Plane 1 repertoire. For the 2-byte units, the first byte is in the range 0xA1–0xFE, and the second byte is in the range 0x40–0x7E or 0xA1–0xFE.

The acronym MBCS (multi-byte character set) is used for encoding schemes that mix character units of different byte lengths (as in the packing schemes mentioned above), in contrast to SBCS (single-byte character set). The acronym DBCS (double-byte character set) is sometimes used for pure two-byte encodings such as JIS X0208, and sometimes used synonymously with MBCS.

Code-Switching Schemes for Multiple Character Sets

Code-switching schemes generally use a sequence of 7-bit values, so they are suitable for mail. ISO 2022 specifies a general code-switching scheme. In its general 7-bit form, it uses

- escape sequences to specify the character sets currently assigned to G0–G3 and C0–C1
- certain C0 and C1 controls to switch the current character set to be any of G0–G3 (using the character sets previously assigned to G0–G3)
- other C1 controls for a temporary character set switch that applies only to the next character

However, ISO 2022 it is rarely used in this form on the Internet. Instead, for certain languages there are one or more predefined combinations of character sets and protocols for use with ISO 2022: for example, ISO-2022-JP (Japanese), ISO-2022-KR (Korean), and ISO-2022-CN (simplified Chinese). Each of these specifies the character sets to be used, the escape sequences or controls used to switch among them, and necessary defaults and reset behavior (such as initial state and the end-of-line reset).

Another common code-switching scheme is HZ, used for Chinese mail and news. This uses ~} and ~{ for switching between ASCII and GB 2312.

The EBCDIC Host encodings used on IBM mainframes for CJK text are a special case and use a sequence of 8-bit values. These encodings combine a single-byte EBCDIC character set and a double-byte IBM character set with graphic characters in the range 0x41–0xFE. The EBCDIC control character Shift Out (SO, 0x0E) is used to switch to the double-byte character set, and the control character Shift In (SI, 0x0F) is used to switch to the single-byte character set.

Unicode

Unicode is a universal character set whose goal is to include characters for all of the worlds written languages, plus a large set of technical symbols, math operators, and so on—everything that needs to be encoded in text. It originated in work by Apple and Xerox in 1988, which was in turn based on the Xerox XCCS universal character set. At about the same time, the ISO/IEC joint technical committee JTC1 was developing a separate universal character set. These efforts were merged beginning in 1991 to produce what is essentially a single character set.

There are actually two parallel standards. The Unicode Consortium is responsible for Unicode, while ISO/IEC JTC1 is responsible for ISO 10646. The goal is to keep the character repertoire and code point assignments synchronized. However, beyond that there are some differences.

The Unicode standard specifies character properties and some rendering behavior, and includes conformance criteria. It clarifies character usage and semantics, and provides a set of guidelines for implementing Unicode. Mapping tables for converting other character sets to Unicode are also provided.

ISO/IEC 10646, like most ISO character set standards, does not specify character properties or rendering behavior. On the other hand, it identifies three implementation levels and many subset repertoires to permit software to indicate precisely what it can and cannot support.

Basic Unicode uses 16-bit code points. Two ranges, each consisting of 1024 16-bit code points, are reserved for high-half surrogates and low-half surrogates; these can be combined to function as a 32-bit code point. This scheme, known as UTF-16, adds a million additional code points.

ISO 10646 supports a 16-bit form (including UTF-16), called UCS-2, as well as a full 32-bit form, called UCS-4. In UCS-4, the high-order byte indicates the group and the next highest order byte indicates the plane. UTF-16 can represent UCS-4 code points from group 0, planes 0 through 16, but uses different numeric values for the characters in planes 1 through 16. Characters that can be represented using a single 16-bit code point are said to be on the Base Multilingual Plane (BMP).

All of these forms can use the full range of 16-bit values. No attempt is made to avoid 16-bit values that contain bytes that may be interpreted in special ways on byte-oriented systems. The first 256 Unicode characters parallel ISO 8859-1; but since the Unicode code points are 16 bits, the high-order byte is 0, which might be interpreted as a C-string terminator on a byte-oriented system.

To permit transmission of Unicode over byte-oriented 8-bit and 7-bit channels, two transformation formats have been devised.

UTF-8 is intended for 8-bit protocols (such as the Web). All of the ASCII repertoire maps to single-byte characters using the ASCII code points. Other Unicode BMP characters map to a sequence of 2 or 3 bytes; the initial bytes of these sequences, as well as the following bytes, are all in distinct ranges so they can be distinguished from each other and from the ASCII range. This makes it relatively easy to process (much easier than Shift-JIS, for example).

UTF-7 is intended for 7-bit protocols (such as mail). Certain characters in the ASCII repertoire are preserved intact. Other Unicode characters are mapped using a modified base 64 encoding. The character + is used to switch to modified base 64, and - is used to switch back out.

Figure B-5 shows the same Unicode sequence in UTF-16, UTF-8, and UTF-7.

Figure B-5 Unicode sequence expressed in UTF-16, UTF-8, and UTF-7

Text:	Beijing 北京
UTF-16 (hex):	`0042 0065 0069 006A 0069 006E 0067 0020 5317 4EAC`
UTF-8 (hex):	`42 65 69 6A 69 6E 67 20 E5 8C 97 E4 BA AC`
UTF-7 (ASCII):	`Beijing +UxdOrA-`

Unicode provides a single encoding that can be used to represent multilingual text. Using a single encoding is much easier than supporting the multitude of encodings otherwise required for multilingual text. Unicode is also much easier to process than many of the other encodings.

The use of Unicode does not by itself imply any particular language or group of languages, unlike the use of, say, ISO 2022-JP, which implies Japanese, or EUC-KR, which implies Korean. A Unicode code point represents a character that may be common to several languages. For example, Figure B-1 (page 221) shows a single Unicode Han character that is used in Chinese, Japanese, and Korean. Unicode encodes plain text—that is, the minimum information for preservation of text content and basic text legibility. It does not explicitly encode higher-level information such as language or font. Note, however, that Unicode does distinguish among characters in different scripts that may have the same appearance, such as LATIN CAPITAL LETTER A and GREEK CAPITAL LETTER ALPHA; this is necessary for preservation of text content.

The Unicode repertoire is a superset of the repertoires of a large number of important standards. Thus, it can also serve as a hub for conversion among multiple encoding systems. For a specific set of source standards, Unicode ensures round-trip fidelity: Every character that is distinct in one of those standards is also distinct in Unicode (for this and other reasons, Unicode includes a number of compatibility characters that would not otherwise have been separately encoded). However, for other standards there may not be a one-to-one mapping from their repertoire onto Unicode; the other standards may include multiple characters that all correspond to the same Unicode character, or they may include characters for which there is no corresponding Unicode character. For example, the Adobe symbol set includes separate code points for upper, center, and lower sections of multiline parentheses, square brackets, and curly brackets; there are no corresponding characters in Unicode.

Unicode provides considerable advantages over other encodings, and Unicode is moving into widespread use. This is especially true on the Internet, where the profusion of character encodings has created the most acute problems. Examples of Unicode use include:

- the character encoding for Java
- the document character set for HTML 3.2
- LDAP and other Internet services
- UDF (the Universal Disk Format adopted for DVD)
- the base encoding for Windows NT
- the base encoding for NextStep and Rhapsody text

Character Set Features

Repertoire and Semantics

The notion of character repertoire becomes a bit fuzzy when a single character in one repertoire has a range of interpretations that matches several characters in another repertoire. Consider the following:

- ASCII 0x2D, HYPHEN-MINUS. Unicode has a HYPHEN-MINUS, but also separate HYPHEN and MINUS SIGN characters. In effect the Unicode repertoire has three characters matching the single ASCII character.
- JIS X0208 0x2142, specified as «double vertical line, parallel.» Unicode has separate characters for DOUBLE VERTICAL LINE and PARALLEL TO. There is no single Unicode character that exactly matches the JIS character; each of the Unicode characters matches one interpretation of the JIS character.

Some character encodings explicitly represent presentation forms. All of the forms shown in Figure B-2 (page 222), for example, are explicitly encoded in one or another encodings. This also creates a situation where multiple characters in one encoding match a smaller number of characters in another encoding.

Finally, there are many nonstandard additions to various encodings. For example:

- Many vendors have their own versions of Shift-JIS that add characters at various code points that are unused in standard Shift-JIS. These may be treated as separate encodings.
- Users in certain fields, such as law or medicine, may have their own standard set of «gaiji» characters that are added to Shift-JIS using custom fonts. Even without gaiji additions, different fonts on a platform may implement slightly different versions of a character encoding (usually the differences are in less commonly used characters).
- Many encodings permit the addition of user-defined characters in unused code points. A glyph editor may be provided so users can create a custom glyph and assign it to a code point.

Combining and Conjoining Characters

The Unicode standard defines a combining character as «a character that graphically combines with a preceding base character» and a nonspacing mark as «a combining character whose positioning in presentation is dependent on its base character». A nonspacing mark generally does not consume space along the visual baseline in and of itself.

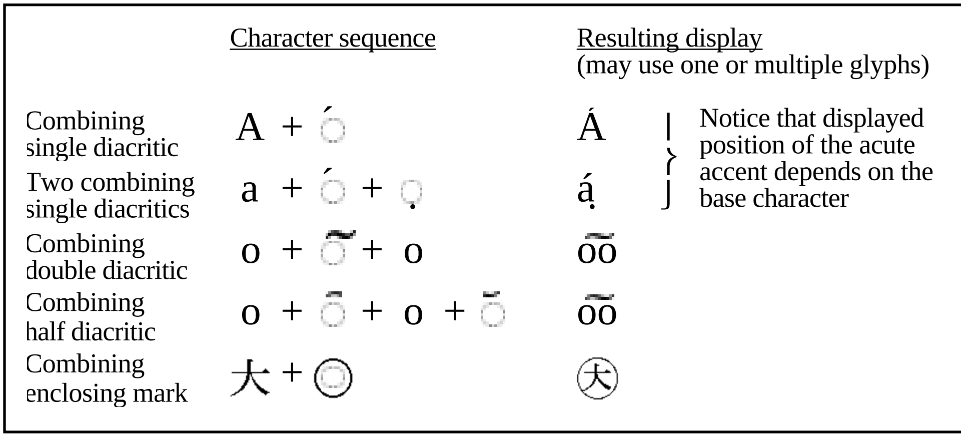
Similar nonspacing marks have been used in bibliographic standards for some time. Many of these standards are derived from the USMARC set developed by the Library of Congress in the 1960s. In these standards, nonspacing marks precede the base character so they can be handled by the primitive text layout techniques that were characteristic of the 1960s. The MARC sets and ISO 5426 allow one or two combining marks; these sets support many Latin-script languages and transliteration of several non-Latin-script languages. ISO 6937 allows one combining diacritic before a base character and allows only certain combinations of diacritics and base characters.

In ASMO 449 (Arabic), ISCII-88 and ISCII-91 (Indic), and TIS 620-2529 and TIS 620-2533 (Thai), combining marks for vowels, tones, and so on follow the base character. Unicode adopted this approach and extended it to nonspacing marks for Latin, Greek, and other scripts, so that all combining characters could be handled consistently.

The USMARC and ISO 5426 sets included characters for right and left halves of diacritics that span two base characters (these are used in Tagalog, for example). Unicode included these for compatibility, but also included single characters for the full diacritic.

Unicode also includes a set of combining enclosing marks for symbols, such as COMBINING ENCLOSING CIRCLE. Figure B-6 gives an idea of the variety of combining marks present in Unicode:

Figure B-6 Some combining marks present in Unicode


Character sequence
Resulting display
(may use one or multiple glyphs)
Combining single diacritic
A +
Á
Two combining single diacritics
a + +
ạ́
Notice that displayed position of the acute accent depends on the base character
Combining double diacritic
o + + o
o͠o
Combining half diacritic
o + + o +
o︢o︣
Combining enclosing mark
大 +
大⃝

There are other sorts of characters that combine graphically for display, but that—strictly speaking—are not combining characters.

Unicode and some other character sets (such as Mac OS Roman) include a FRACTION SLASH character for composing fractions. A digit (or digit sequence), followed by a fraction slash, followed by another digit (sequence) should be displayed as a single composed fraction.

Unicode also includes a set of conjoining Korean jamos. These constitute the Korean alphabet and are graphically combined into square syllable blocks for display according to well-defined rules (The Unicode standard provides an algorithm for this). This is similar to the process of ligature formation in Arabic or Devanagari (although in those scripts the set of ligatures and the rules are typically more font-dependent); but Unicode also has a set of nonconjoining jamos. Figure B-7 provides examples of the behavior of fraction slash and conjoining jamos.

Figure B-7 Fraction slash and conjoining jamos

	Character sequence	Resulting display (may use one or multiple glyphs)
Fraction slash	3 + ⁄ + 4	${}^{3}/_{4}$
Conjoining Jamos	ㅍ + ㅡ + ㄹ	플

In Figure B-6 and Figure B-7, the character sequences shown on the left side are called decomposed character sequences; they generally correspond to a single displayed text element. Some character encodings may represent that displayed text element with a single character code, in addition to or instead of using the decomposed representation. Single code points for text elements such as the ones on the right side of Figure B-6 and Figure B-7 are called precomposed characters. Unicode includes many precomposed characters as well as combining and conjoining characters that can be used for decomposed sequences; the former accommodate backward compatibility requirements, while the latter are better suited to modern graphics and text processing systems.

As a result, Unicode includes multiple representations (or «multiple spellings») for the same text elements. Multiple representations of the same text elements should generally be treated as equivalent for most text processing purposes. Also, when converting among encodings, there may be multiple representations in Unicode that correspond to a given character in another encoding.

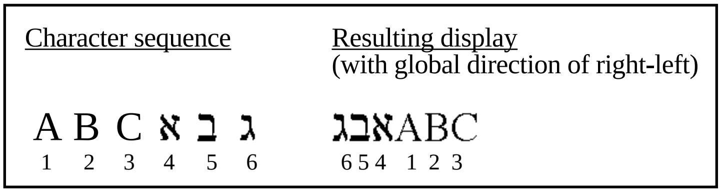
Ordering Issues

For Arabic and Hebrew, there are three conventions for the order in which text is encoded:

- Implicit or logical order, in which the text is stored in memory in the same order it would be spoken or typed. Characters have an inherent direction attribute, and this attribute is used by a display algorithm to determine the proper (or most likely) display order for the corresponding glyphs. The algorithm may make use of global line direction information if available.
- Explicit order, in which all display ordering is determined by explicit controls.
- Visual order, in which text is stored line-by-line in left-to-right display order (that is, the Arabic and Hebrew non-numeric text is encoded in reverse order). This is typically used for older systems or when no real support for bidirectional text is provided, and requires explicit line breaks.

Unicode uses implicit order, with the addition of optional controls for unusual cases or fine-tuning, and specifies the reordering algorithm for display. The Windows and Mac OS Hebrew and Arabic encodings also assume implicit order. Figure B-8 gives an example of implicit ordering.

Figure B-8 Implicit ordering



Characters that are otherwise identical in different character encodings may have different direction attributes in the two encodings, and this creates another "fuzzy" problem for matching character repertoires. For example, Unicode has a single PLUS SIGN character, with direction class European Number Terminator; the Mac OS Hebrew and Arabic encodings have two plus sign characters, one with strong left-right direction, and one with strong right-left direction. This is because the Mac OS encodings were designed in 1986 for a reordering model that was less sophisticated than the current Unicode reordering model.

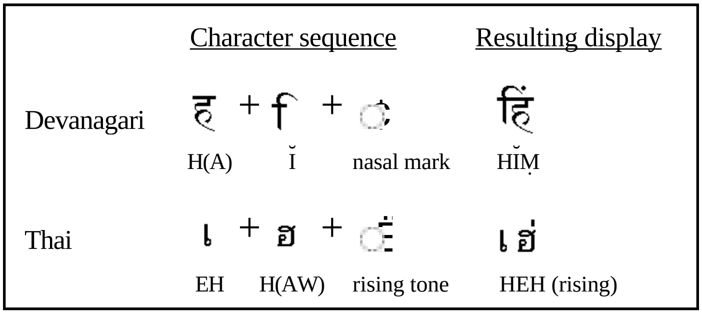
There are also two different ordering conventions for characters in Indic and related Southeast Asian scripts. In these scripts, consonants have an inherent vowel, which is pronounced after the consonant. A vowel mark may be used with the consonant to change the vowel; this vowel mark may be displayed above, below, to the left or to the right of the consonant; it may even surround the consonant or have components that appear on either side.

The scripts of India are generally encoded in logical order, so that any dependent vowel (and other marks related to the consonant) follows the consonant in memory. The consonant, together with any dependent vowel and other marks, constitutes a «consonant cluster». Successive clusters are displayed in left-to-right order, but within a cluster the ordering may be complex. (Clusters may also include vowel-less dead consonants that precede the main consonant.)

Thai consonants have an inherent tone as well as an inherent vowel; tone marks may be added to change the tone, in addition to any vowel signs. Thai is generally encoded in visual order, unlike the scripts of India, so a vowel that modifies a consonant's inherent vowel may precede or follow that consonant in memory.

Unicode follows the above conventions for encoding Indic and Thai (Lao is related to Thai, and is encoded similarly).

Figure B-9 Character sequence and resulting display



Character Data in Programming Languages

The C `char` type is supposed to be large enough to store any member of the execution character set. If a genuine character from that set is stored in a `char` object, its value is equivalent to the integer code for the character and is non-negative. The `char` type is also equivalent to a single byte and may be signed or unsigned (implementation dependent).

C does not actually define the size of a byte, so in principle a byte could be made large enough so a `char` would accommodate multi-octet characters and Unicode characters. However, in most implementations, bytes and `char` objects are 8 bits, and multi-octet characters require a sequence of `char` objects.

Instead, C provides the wide character or `wchar_t` type. This is really supposed to be large enough to hold the largest character in any extended execution set supported by the implementation (including MBCS encodings). It permits internal processing using fixed-size characters; C library functions such as `mbstowcs( )` and `wcstombs()` convert between SBCS/MBCS strings and wide character strings. However, the size of `wchar_t` is implementation specific; although it is usually 16 or 32 bits, on some implementations it is equivalent to `char`.

Java takes a different approach: Bytes remain 8 bits, but a Java `char` is a 16-bit unit intended to contain a Unicode character.

Finally, programming languages generally provide some abstraction away from encoding details. For example, the C character constant 'A' may have the value 0x41 for an ASCII-based implementation, but 0xC1 for an EBCDIC-based implementation. Nevertheless, programs may make more subtle assumptions about character encodings, such as assuming that A–Z have sequential contiguous code points (not true in EBCDIC).

A P P E N D I X C

Some Character Encodings and Their Common Internet Names

Identifying Character Encodings on the Internet

In many Internet protocols, a `charset` parameter may be used in certain contexts to specify both a character set and a character encoding scheme. The value of the `charset` parameter is a case-insensitive string limited to the characters A–Z, a–z, 0–9, hyphen–minus, underscore, period, and colon. The character encoding names specified for this parameter are generally expressed in US–ASCII octet values.

The character encoding name may be an experimental name beginning with `x-`; if it is not an experimental name, it must be a name registered with the Internet Assigned Numbers Authority (IANA) that corresponds to a character encoding that has a formal specification. Multiple names exist for most character encodings in the registry. The IANA registry is updated periodically; for example, the name EUC-JP was added to it in January. Table C-1 (page 243) identifies character encodings for various languages, gives some of their common Internet names, and tells when the character encoding was first supported for the Text Encoding Converter and the Unicode Converter. To preview the style of character set name used on the Internet, here are a few sample names:

```
ISO-8859-1  latin1  UNICODE-1-1-UTF-7  Shift_JIS  X-EUC-CN
```

Many of the character encodings in use on the Internet are not registered with IANA and do not have official Internet names, although they may have names that have become de facto standards. Moreover, even when an encoding is registered, the name specified by IANA may not be the one that is actually used on the Internet. For example, EUC-JP has been registered for some time with the unwieldy name `Extended_UNIX_Code_Packed_Format_for_Japanese`, but the name actually used is the unofficial `X-EUC-JP`. Another example, `Shift_JIS`, is the official name, but the names commonly used in its stead are `x-shift-jis` and `x-sjis`. In many cases, mail and browser software recognizes only the unofficial names, not the official ones.

In some cases, the names for unregistered encodings follow a pattern established by other, registered encodings. For example, some IBM/Microsoft code pages are registered with names consisting of `cp` followed by the code page number: `cp437`, `cp850`, `cp852`. Code page `874` is not registered, but the name `cp874` would be expected. Most Windows code pages are registered using the form used in these examples: `windows-1250`, `windows-1251`. Windows Latin-1 is, oddly enough, not registered as either `windows-1252` or `cp1252`, although both forms are in use.

Character Encodings Masquerading as Related Encodings

Some Internet names used for similar character encodings could lead to confusion. For example, the Windows Latin-1 character encoding is commonly labeled `ISO-8859-1` on the Internet because it is a superset of ISO 8859-1. Clients that actually treat it as ISO 8859-1 may be confused by the extra characters in the C1 area.

The Mac OS Roman character set used for Western European languages was created several years before ISO 8859-1. It does not have exactly the same repertoire, and many of the characters it does share with ISO 8859-1 have different code points. Many Mac OS Internet applications use an encoding developed by André Pirard in which the Mac OS Roman repertoire is assigned new code points to align as much as possible with ISO 8859-1; this character encoding is referred to as Mac Latin-1 or Mac Mail and is usually labeled as `ISO-8859-1` on the Internet.

Character Encodings and Their Internet Names

Table C-1 lists character encodings for various languages, gives some of their common Internet names, and identifies the version of the Text Encoding Conversion Manager for which character encoding was first supported for use by the Text Encoding Converter and the Unicode Converter.

Some Character Encodings and Their Common Internet Names

Table C-1 Character encoding Internet names and availability in Mac OS

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
Universal				
Unicode 2.0 (16 bit)	UTF-16		1.2	1.2
Unicode 2.0 UTF-8	UTF-8		1.2	1.2.1
Unicode 2.0 UTF-7	UTF-7		1.2	N/A
Unicode 1.1 (16-bit)	UNICODE 1-1		1.2	1.2
Unicode 1.1 UTF-8	UNICODE-1-1-UTF-8		1.2	1.2.1
Unicode 1.1 UTF-7	UNICODE-1-1-UTF-7		1.2	N/A
Western European languages				
ASCII	US-ASCII		1.2.1	1.2.1
ISO 8859-1 (Latin-1)	ISO-8859-1, latin1		1.2.1	1.2.1
CP 1252 (Windows Latin-1)	windows-1252, cp1252	ISO 8859-1, plus additions in C1 area	1.2	1.2
CP 437 (DOS Latin-US)	cp437		1.2	1.2
Mac OS Roman	mac, macintosh, x-mac-roman		1.2	1.2
Mac OS Icelandic	x-mac-icelandic	based on Mac OS Roman	1.2	1.2
Mac OS Latin-1, Mac OS Mail	x-mac-latin1 (commonly sent as ISO-8859-1)	Mac OS Roman permuted to align with 8859-1	1.2	1.2
NextStep Latin			1.2	1.2

Some Character Encodings and Their Common Internet Names

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
CP 037 (EBCDIC-US)	`cp037`	ISO 8859-1 repertoire, different layout	1.2.1	1.2.1
Arabic				
ISO 8859-6 (Latin/Arabic)	`ISO-8859-6`, `arabic`		1.2	1.2
CP 1256 (Windows Arabic)	`windows-1256`, `cp1256`	Partly 8859-6, plus C1 additions	1.2	1.2
CP 864 (DOS Arabic)	`cp864`	Encodes Arabic presentation forms	1.2	1.2
Mac OS Arabic	`x-mac-arabic`		1.2	1.2
Mac OS Farsi	`x-mac-farsi`		1.2	1.2
Central European languages				
ISO 8859-2 (Latin-2)	`ISO-8859-2`, `latin2`		1.2	1.2
CP 1250 (Windows Latin-2)	`windows-1250`, `cp 1250`	Partly 8859-2, plus C1 additions	1.2	1.2
Mac OS Central European Roman	`x-mac-centraleurroman`		1.2	1.2
Mac OS Croatian	`x-mac-croatian`	Based on Mac OS Roman	1.2	1.2
Mac OS Romanian	`x-mac-romanian`	Based on Mac OS Roman	1.2	1.2

Some Character Encodings and Their Common Internet Names

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
Chinese				
GB 2312-80			1.2	N/A
EUC-CN	`GB2312`, `X-EUC-CN`	ASCII + GB 2312- 80 (8-bit)	1.2	1.2
CP 936 (DOS and Windows Simplified)		Same as EUC-CN	N/A	N/A
Mac OS Chinese Simplified		Based on EUC-CN	1.2	1.2
ISO 2022-CN ("GB")	`ISO-2022-CN`	ASCII + GB 2312-80 (7-bit) (see RFC1922)	1.2	N/A
HZ	`HZ-GB-2312`	ASCII + GB 2312-80 (7-bit) (see RFC1842);	1.2	N/A
GBK (extended GB)		EUC-CN + Unihan repertoire (8-bit)	1.2	1.2
CNS 11643 plane 1	`x-cns11643-1`		N/A	N/A
CNS 11643 plane 2	`x-cns11643-2`		N/A	N/A
EUC-TW	`X-EUC-TW`	ASCII + CNS 11643-1992 (8-bit)	1.2	1.2
Big-5	`Big5`	(8-bit)	1.2	1.2
CP 950 (DOS and Windows Traditional)		Based on Big-5	N/A	N/A

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
Chinese (contd.)				
Mac OS Chinese Traditional		Based on Big-5	1.2	1.2
CCCII			N/A	N/A
EACC			N/A	N/A
Cyrillic				
ISO 8859-5 (Latin/Cyrillic)	`ISO-8859-5`, `cyrillic`		1.2	1.2
KOI8-R	`KOI8-R`	See Rfc 1489	1.2	1.2
CP 1251 (Windows Cyrillic)	`windows-1251`, `cp1251`	Not based on ISO 8859-5	1.2	1.2
CP 866 (DOS Russian)	`cp866`		N/A	N/A
Mac OS Cyrillic	`x-mac-cyrillic`		1.2	1.2
Mac OS Ukrainian	`x-mac-ukrainian`	Mac OS Cyrillic with two replacements	1.2	1.2
Greek				
ISO 8859-7	`ISO-8859-7`, `greek`		1.2	1.2
ISO 5428	`ISO_5428:1980`		N/A	N/A
CP 1253 (Windows Greek)	`windows-1253`, `cp1253`	Nearly 8859-7, plus C1 additions	1.2	1.2
Mac OS Greek	`x-mac-greek`		1.2	1.2
Greek CCITT	`greek-ccitt`		N/A	N/A

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
Hebrew				
ISO 8859-8 (Latin/Hebrew)	`ISO-8859-8`, `hebrew`		1.2	1.2
CP 1255 (Windows Hebrew)	`windows-1255`,`cp1255`	Mostly 8859-8, plus C1 additions	1.2	1.2
Mac OS Hebrew (2 variants)	`x-mac-hebrew`		1.2	1.2
Indic				
ISCII-91		Parallel encodings for all Indic scripts	N/A	N/A
Mac OS Gujarati			1.2	1.2
Mac OS Devanagari			1.2	1.2
Mac OS Gurmukhi			1.2	1.2
Japanese				
JIS X0208			1.2	N/A
JIS X0212			N/A	N/A
EUC-JP	`EUC-JP`, `X-EUC-JP`	JIS 201 + JIS 208 + JIS 212 (8-bit)	1.2	1.2
ISO 2022-JP ("JIS")	`ISO-2022-JP`	JIS 201 + JIS 208 + JIS 212 (7-bit); Rfc 1468	1.2	N/A

Some Character Encodings and Their Common Internet Names

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
Japanese (contd.)				
Shift-JIS	`Shift_JIS`, `x-sjis`, `x-shift-jis`	JIS 201 + JIS 208 (8-bit)	1.2	1.2
CP 932 (DOS + Windows)		Based on Shift-JIS	N/A	N/A
Mac OS Japanese		Based on Shift-JIS	1.2	1.2
Korean				
KSC 5601-1987			1.2	N/A
EUC-KR	`EUC-KR`	ASCII + KSC 5601-87 (8-bit); Rfc 1557	1.2	1.2
CP 949 (DOS + Windows)		Unified Hangul Code: EUC-KR + Johab	N/A	N/A
Mac OS Korean		Based on EUC-KR	1.2	1.2
ISO 2022-KR ("KSC")	`ISO-2022-KR`	ASCII + KSC 5601-87 (7-bit): Rfc 1557	1.2	N/A
KSC 5700			N/A	N/A
Symbols encoding				
Adobe Symbol	`Adobe-Symbol-Encoding`		N/A	N/A
Mac OS Symbol	`x-mac-symbol`	Based on Adobe Symbol	1.2	1.2
Mac OS dingbats	`x-mac-dingbats`	Based on Adobe Zapf Dingbats	1.2	1.2

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
Thai				
TIS 620-2533			N/A	N/A
CP 874 (DOS + Windows)	`cp874`	Based on TIS 620-2533	N/A	N/A
Mac OS Thai	`x-mac-thai`	Based on TIS 620-2533	1.2	1.2
Turkish				
ISO 8859-9 (Latin-5)	`ISO-8859`, `latin5`		1.2	1.2
ISO 8859-3 (Latin-3)	`ISO-8859-3`		N/A	N/A
CP 1254 (Windows Latin-5)	`windows-1254`, `cp1254`		1.2	1.2
Mac OS Turkish	`x-mac-turkish`	Based on Mac OS Roman	1.2	1.2
Vietnamese				
VISCII	`VISCII`	Rfc 1456	N/A	N/A
TCVN-n			N/A	N/A

Mac OS Encoding Variants

For font-based Mac OS encoding variants, Table D-1 gives the variant name in the English language, comments about the variant (such as whether it is the system or application font), and the constant used to represent the variant.

Note
Except for the names listed under Mac OS Icelandic, the English font names given below are not the names used by the Font Manager or displayed in menus. ◆

Table D-1 Mac OS Encoding Variants

English Name	Comments	Constant for Variant
Mac OS Icelandic Script = `smRoman` (0), language = `langIcelandic` (15), region = `verIceland` (21)		
Chicago	System font	`kMacIcelandicStandardVariant`
Geneva	Application font	`kMacIcelandicStandardVariant`
Monaco		`kMacIcelandicStandardVariant`
New York		`kMacIcelandicStandardVariant`
Palatino		`kMacIcelandicTrueTypeVariant`
Times		`kMacIcelandicTrueTypeVariant`
Helvetica		`kMacIcelandicTrueTypeVariant`
Courier		`kMacIcelandicTrueTypeVariant`

English Name	Comments	Constant for Variant
Mac OS Japanese		
Osaka	System font, application font	`kMacJapaneseStandardVariant`
Osaka Tohaba		`kMacJapaneseStandardVariant`
MaruGothic		*
HonMincho		*
TohabaGothic		`kMacJapaneseBasicVariant`
TohabaMincho		`kMacJapaneseBasicVariant`
ChuGothic	PostScript	`kMacJapanesePostScriptScrnVariant`
SaiMincho	PostScript	`kMacJapanesePostScriptScrnVariant`
HeiseiKakuGothic		`kMacJapaneseStandardVariant`
HeiseiMincho		`kMacJapaneseStandardVariant`
ChuGothic BBB		`kMacJapaneseStandardVariant`
ChuGothic BBB Tohaba		`kMacJapaneseStandardVariant`
Ryumin Light-KL		`kMacJapaneseStandardVariant`
Ryumin Light-KL-Tohaba		`kMacJapaneseStandardVariant`

* (For System 7.1-J, the constant is `kMacJapaneseVertAtKuPlusTenVariant`, otherwise it's `kMacJapaneseStandardVariant`.)

English Name	Comments	Constant for Variant
Mac OS Arabic		
Cairo	System font	`kMacArabicStandardVariant`
Geeza	Application font	`kMacArabicTrueTypeVariant`
Nadeem		`kMacArabicTrueTypeVariant`
Baghdad		`kMacArabicTrueTypeVariant`
Kufi		`kMacArabicTrueTypeVariant`
Thuluth	PostScript	`kMacArabicThuluthVariant`
Thuluth Bold	PostScript	`kMacArabicThuluthVariant`
AlBayan		`kMacArabicAlBayanVariant`

English Name	Comments	Constant for Variant
Mac OS Farsi		
Script = smArabic (4), language = langFarsi (31), region = verIran (48)		
Tehran	System font	kMacFarsiStandardVariant
Asfahan		kMacFarsiTrueTypeVariant
Mashad		kMacFarsiTrueTypeVariant
NadeemFarsi		kMacFarsiTrueTypeVariant
Kamran		kMacFarsiTrueTypeVariant
Amir		kMacFarsiTrueTypeVariant
Mac OS Hebrew		
Eilat	System font	kMacHebrewStandardVariant
Hermon	Application font	kMacHebrewStandardVariant
Arial		kMacHebrewStandardVariant
New Peninim		kMacHebrewStandardVariant
Corsiva		kMacHebrewStandardVariant
Raanana		kMacHebrewStandardVariant
RamatGan		kMacHebrewStandardVariant
Sinai Book		kMacHebrewFigureSpaceVariant
Ramat Sharon		kMacHebrewFigureSpaceVariant
Carmel		kMacHebrewFigureSpaceVariant
Caesarea		kMacHebrewFigureSpaceVariant
Gilboa		kMacHebrewFigureSpaceVariant

APPENDIX E

Conventions for Unicode Text in the Mac OS

This appendix describes the conventions for plain text documents and Clipboard content in the Mac OS.

File Requirements

Most documents created on a Mac OS-based system use a richer text model than pure Unicode, so the emphasis here is on easy interchange with other platforms. In particular, an application should be able to

- import and export Unicode plain text files from other platforms with no data loss
- easily import a Unicode plain text file into a rich text environment

File Types

The file type `'utxt'` has been registered for UTF-16 plain text documents. The (optional) scrap type `'utxt'` is also registered for UTF-16 Clipboard text.

Whether it is useful to register a file type or scrap type for UTF-8 text is currently under discussion. As do other documents and text that use WorldScript encodings, plain UTF-8 documents could use the file and scrap type `'TEXT'`. UTF-8 is compatible with the assumptions that govern WorldScript encodings; these encodings are not specifically identified in `'TEXT'` files and Clipboard contents.

File Content

A plain text Unicode document, in a file or on the Clipboard, can contain any valid character from Unicode 2.0 or later. In particular, it can contain control characters in the range `U+0000` through `U+001F` and `U+0080` through `U+009F`. It may also contain codes in the Corporate and Private Use Zones although these may not interchange properly.

The byte-order mark `U+FEFF` may be present at the beginning of the content. If it is absent in UTF-16 content, big-endian order is assumed.

Creating Content

When creating file content, write line and paragraph separators using the special Unicode characters intended for this purpose—`U+2028` and `U+2029`—instead of using some combination of `CR` and `LF`. This makes the content more portable; when the content is read on a particular platform, these Unicode separators can be converted to the separators customary for that platform.

Reading Content

When reading file content, accept and treat the Unicode line and paragraph separators as such. In addition, also treat any of the following as paragraph separators: `LF`, `CR`, `CRLF`.

When converting content to Mac OS encodings, set the `kUnicodeLooseMappingsBit` control flag. (You may use other control bits in addition to this one).

Glossary

character An atomic unit of content for text data. A character is an abstract entity without any particular appearance; characters include letters, digits, punctuation, and symbols.

character encoding scheme A text encoding that maps a sequence of characters (from one or more coded character sets) to a sequence of bytes, in order to combine characters from multiple coded character sets or to permit easier handling of some coded character sets. Compare **coded character set.**

code fragment See **fragment.**

Code Fragment Manager (CFM) The part of the Macintosh system software that prepares fragments for execution.

code point An integer value that represents (or can represent) a character.

coded character A character together with its numeric representation in a particular coded character set.

coded character set A text encoding that maps each character in a set of characters to a particular integer from a set of integers. Compare **character encoding scheme.**

code-switching scheme A character encoding scheme that allows switching between different coded character sets, usually signaled by escape or other special sequences. See also **character coding scheme.**

content transfer encoding See **transfer encoding syntax.**

converter object An instance of data that tells a text converter how to convert text from a particular source encoding to a particular destination encoding, and maintains any necessary state information that applies to the conversion of a particular stream of text.

destination encoding The text encoding that describes the desired encoding of the text after conversion. Compare **source encoding.**

direct conversion A text conversion by the Text Encoding Converter that can be handled in one step (that is, by one call to a single plug-in). Compare **indirect conversion.**

Extended UNIX Code (EUC) A type of packing scheme that is used as the text encoding for UNIX workstations that handle East Asian languages. See also **packing scheme.**

fallback mapping A character or sequence of characters used to replace a character that has no direct equivalent in the destination encoding. For example, if the target encoding does not contain “å,” a possible fallback mapping would be “aa.”

fragment On the Mac OS, a block of executable code or data. Fragments are handled by the Code Fragment Manager. See also **Code Fragment Manager.**

glyph image A visual element used to represent one or more characters.

indirect conversion A text conversion by the Text Encoding Converter that requires stepping through one or more intermediate conversions before reaching the desired destination encoding. Compare **direct conversion.**

Internet The name given to the world-wide network of computers.

loose mapping A mapping between text encodings that preserves the information content of text but does not permit round-trip fidelity.

Multipurpose Internet Mail Extensions (MIME) Mechanisms for specifying and describing the format of Internet message bodies.

packing scheme A type of character encoding scheme where characters are encoded using a variable number of bytes. Typically certain bytes signal the beginning of a character and how many additional bytes are used to encode the character. Character sets with a large number of elements are often stored using a packing scheme. See also **character encoding scheme.**

perfect round-trip conversion This occurs when mapping a character from a particular source encoding to a particular destination encoding (usually Unicode) and then back to the source encoding again yields the original character.

plug in See **text encoding conversion plug-in.**

presentation form An abstraction of a range of glyph images, which represents a standard way to display a particular character or group of characters in a particular context as specified by a particular writing system. See also **glyph image**.

script A collection of related characters, subsets of which are required to write a particular language. Some examples of scripts are Latin, Greek, Hiragana, Katakana, and Han.

sniffer A function included with a text conversion plug-in that scans text for features that identify a particular text encoding.

source encoding The text encoding that describes the encoding of the text before conversion. Compare **destination encoding.**

strict mapping A mapping between text encodings that preserves the information content of text and permits round-trip fidelity.

text element A group of one or more characters that is treated as a single entity for a particular process such as collation, display, or transcoding.

text encoding The coded character set or character encoding scheme used to represent a particular piece of text. See also **coded character set, character encoding scheme.**

text encoding base The primary specification of a text encoding, and one component of a text encoding specification. See also **text encoding specification, text encoding variant, text encoding format.**

text encoding conversion plug-in A code fragment that provides conversion services between pairs of encodings. A text encoding conversion plug-in informs the Text Encoding Conversion Manager about its conversion and encoding analysis capabilities

text encoding format A subset of the text encoding specification that specifies the byte format of the encoding. For example, a format might specify that the encoding take up only 7-bits for transmission over 7-bit channels. See also **text encoding specification, universal transformation format.**

text encoding specification A scalar value that defines a text encoding to be used in a conversion. It includes information about the **text encoding base,** the **text encoding variant,** and the **text encoding format.**

text encoding variant A specification of one among possibly several minor variants or subsets of a particular text encoding base. See also **text encoding specification, text encoding base.**

Text Encoding Conversion Manager A pair of shared library extensions—namely, the Text Encoding Converter and the Unicode Converter—that facilitate text encoding conversion on Mac OS–based computers

Text Encoding Converter A shared library extension that provides the services for general and algorithmic encoding conversions or multi-encoding streams. The Text Encoding Converter sometimes uses the Unicode Converter.

transfer encoding syntax A transformation applied to text encoded using a character encoding scheme to allow it to be transmitted by a specific protocol or set of protocols. This is normally used to permit 8-bit data to be sent through channels that can only handle 7-bit values. Also called **content transfer encoding**. Compare **character encoding scheme, universal transformation format.**

Unicode A universal character set that includes tens of thousands of characters covering the world’s major written languages along with many symbols.

Unicode Converter A shared library extension that provides table-based conversion between Unicode and other encodings.

universal transformation format Special formats that allow transmission of Unicode characters over 7-bit (UTF-7) and 8-bit (UTF-8) channels. See also **transfer encoding syntax.**

writing system A set of characters from one or more scripts that are used to write a particular language and the rules that govern the presentation of those characters.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Peter Edberg, Judy Melanson, and Jun Suzuki

ILLUSTRATOR
Karin Stroud

DEVELOPMENTAL EDITOR
Laurel Rezeau

PRODUCTION EDITOR
Glen Frank

Special thanks to Julio Gonzalez and Jim Chan.