
Programming With the Text Encoding Conversion Manager

[Carbon > Text & Fonts](#)



2005-07-07



Apple Inc.
© 2005 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

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

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

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

Apple, the Apple logo, Carbon, Chicago, Geneva, Mac, Mac OS, Macintosh, Monaco, New York, and TrueType are trademarks of Apple Inc., registered in the United States and other countries.

Numbers is a trademark of Apple Inc.

Adobe, Acrobat, and PostScript are trademarks or registered trademarks of Adobe Systems Incorporated in the U.S. and/or other countries.

Helvetica, Palatino, and Times are registered trademarks of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun

Microsystems, Inc. in the U.S. and other countries.

PowerPC and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

UNIX is a registered trademark of The Open Group

Xerox is a registered trademark of Xerox Corporation.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, 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 DOCUMENT, 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

Introduction **Introduction to Programming With the Text Encoding Conversion Manager**
7

- Why You Need to Convert Text From One Encoding to Another 7
- Deciding Which Encoding Converter to Use 8
 - The Text Encoding Converter 9
 - The Unicode Converter 10
- Character Encoding and Other Concepts Fundamental to Text Encoding Conversion 11
 - Characters 11
 - Coded Character Sets 12
 - Presentation Forms 12
 - Character Encoding Schemes 12
- Text Encoding Specifications 13
- Unicode and the Complexities of Conversion 14
 - About Unicode 14
 - Round-Trip Fidelity 15
- The Text Encoding Conversion Manager 17
 - About Earlier Releases 17
 - Checking the Version 17

Chapter 1 **Character Encoding Concepts In-Depth** 19

- Terminology 19
 - Character Sets and Encoding Schemes 19
 - Characters, Glyphs, and Related Terms 20
- Non-Unicode Character Encodings 21
 - General Character Set Structure 22
 - Simple Coded Character Sets 22
- Packing Schemes for Multiple Character Sets 24
- Code-Switching Schemes for Multiple Character Sets 25
- Unicode 26
- Character Set Features 28
 - Repertoire and Semantics 28
 - Combining and Conjoining Characters 29
 - Ordering Issues 30
- Character Data in Programming Languages 32

Chapter 2 **Writing Custom Plug-Ins 33**

Appendix A **Character Encodings and Internet Names 57**

- Identifying Character Encodings on the Internet 57
- Character Encodings Masquerading as Related Encodings 58
- Character Encodings and Their Internet Names 58

Appendix B **Mac OS Encoding Variants 65**

Document Revision History 69

Glossary 71

Figures and Tables

Introduction **Introduction to Programming With the Text Encoding Conversion Manager**
7

Figure I-1 A possible conversion path used by the Text Encoding Converter 10

Chapter 1 **Character Encoding Concepts In-Depth** **19**

Figure 1-1 Some glyph images for representing characters 20
Figure 1-2 Presentation forms 21
Figure 1-3 Comparison of 7-bit and 8-bit character set structures 22
Figure 1-4 Shift-JIS byte sequence 25
Figure 1-5 Unicode sequence expressed in UTF-16, UTF-8, and UTF-7 27
Figure 1-6 Some combining marks present in Unicode 29
Figure 1-7 Fraction slash and conjoining jamos 30
Figure 1-8 Implicit ordering 31
Figure 1-9 Character sequence and resulting display 31

Appendix A **Character Encodings and Internet Names** **57**

Table A-1 Character encoding Internet names and availability in Mac OS 58

Appendix B **Mac OS Encoding Variants** **65**

Table B-1 Mac OS Encoding Variants 65

Introduction to Programming With the Text Encoding Conversion Manager

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.

You can find descriptions of the basic text types for specifying text encodings and other aspects of conversion, the Text Encoding Converter, and the Unicode Converter in the following reference documents:

Text Encoding Conversion Manager Reference

Unicode Utilities Reference

The reference documents are meant to be used as you develop your applications. You can consult the 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, see *Handling Unicode Text Editing With MLTE*.

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, and 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 11) and in [“Character Encoding Concepts In-Depth”](#) (page 19).

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 document 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 11).) 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.

The Unicode Converter can convert most non-Unicode encodings to or from the no-subset variant of Unicode in either the UTF-16 or UTF-8 formats. For example, it can convert directly from Windows Latin-1 to UTF-8. It can also convert Mac encodings, most CJK encodings, and Latin-1 to or from the HFS+ decomposed variant of Unicode in either the UTF-16 or UTF-8 formats. Finally, it can convert the no-subset variant of Unicode (in either the UTF-16 or UTF-8 formats) to any of the normalized variants of Unicode in the UTF16 format.

You might want to 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.

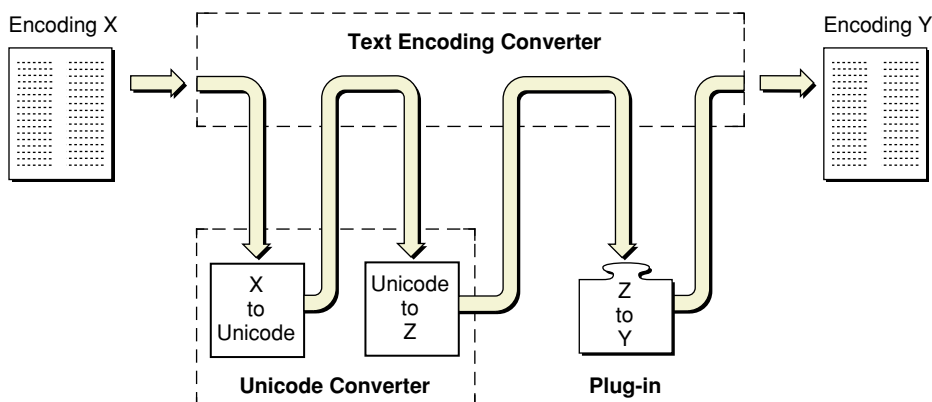
This section looks briefly at plug-ins while [“Writing Custom Plug-Ins”](#) (page 33) describes them in greater detail.

When you launch your application, the Text Encoding Converter scans the system 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](#) (page 10) 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 I-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 14) and [“Character Encoding Concepts In-Depth”](#) (page 19). For definition of some of the terms used in this section, see [“Character Encoding and Other Concepts Fundamental to Text Encoding Conversion”](#) (page 11).

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, and 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 [“Character Encoding Concepts In-Depth”](#) (page 19).

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 12).

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. *Inside Mac OS X: Text Encoding Converter Manager 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.

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.

The Text Encoding Conversion Manager

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 Conversion (TEC) Manager 1.0.x was released for use with Cyberdog 1.0 and 1.2 and with Mac OS Runtime for Java (MRJ) 1.0. TEC Manager 1.1 was released for use with Cyberdog 2.0.

TEC Manager 1.2 was included with Mac OS 8 in July 1997, and with MRJ 1.5; the corresponding interfaces were in Universal Interfaces 3.0. TEC Manager 1.2.1 was released as an SDK in September 1997.

TEC Manager 1.3 was included with Mac OS 8.1 in January 1998, and with MRJ 2.0. TEC Manager 1.3.1 (with one additional bug fix) was released as an SDK. The corresponding interfaces were in Universal Interfaces 3.1.

TEC Manager 1.4 was released as an SDK in September 1998, and was included with Mac OS 8.5 in October 1998. The corresponding interfaces were in Universal Interfaces 3.2. TEC Manager 1.4.2 was released as an SDK in February 1999, and was included with MRJ 2.1. TEC Manager 1.4.3 was included with Mac OS 8.6 in May 1999.

In older documentation for 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.

INTRODUCTION

Introduction to Programming With the Text Encoding Conversion Manager

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

Character Encoding Concepts In-Depth

This document is adapted from a tutorial created by Peter Edberg that was presented at the 11th International Unicode Conference. The original paper is published in the Proceedings of that conference with a notice indicating joint copyright by Apple Computer, Inc. and the Unicode Consortium.

The document explores some aspects of character encodings, including terms used, such as coded character sets, character encoding schemes, characters, glyphs, and related concepts. It discusses existing character encodings, focusing on important Internet encodings and how these encodings relate to the Unicode standard. The document also discusses special features of various character encodings and the use of character data in programming languages.

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.

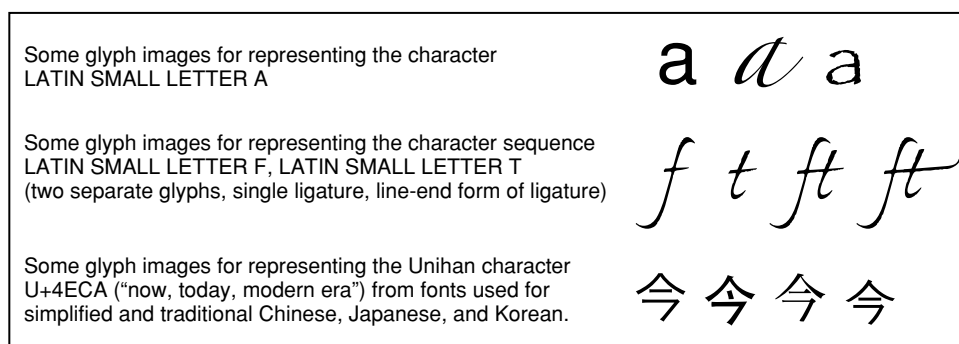
Characters, Glyphs, and Related Terms

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 2-1](#) (page 20) shows glyphs and their associated characters.













Figure 1-1 Some glyph images for representing characters



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 2-2](#) (page 21) shows some examples of presentation forms.

Figure 1-2 Presentation forms

Different contextual forms (final, medial, initial, isolated) for the character ARABIC LETTER HEH	   
	Fi Me In Is
Different ligature forms for the character sequence ARABIC LETTER LAM, ARABIC LETTER ALEF (final/medial form, isolated/initial form)	 
	Fi/Me Is/In
Ligatures for the character sequences LATIN SMALL LETTER F, LATIN SMALL LETTER I and LATIN SMALL LETTER F, LATIN SMALL LETTER L	 
	fi fl
CJK horizontal and vertical presentation variants	 
	[...] [⋮]
Katakana fullwidth and halfwidth variants	 
	アイウ アイウ

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.

“[Character Encodings and Internet Names](#)” (page 57) provides a more complete list of character encodings (but with less explanatory material), grouped by the writing systems they cover.

General Character Set Structure

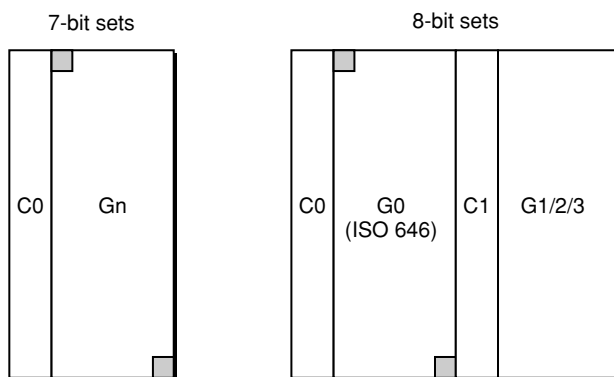
ISO 2022 and ISO 4873 define a structure for coded character sets using 7-bit or 8-bit values. These coded character sets provide a means of representing both graphic characters and control functions; control functions that can be represented with a single code point are also called control characters.

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 2-3](#) (page 22) shows these differences.

Figure 1-3 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 discontinuous 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.

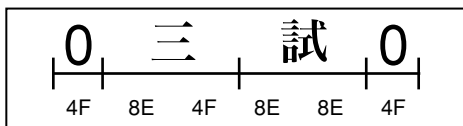
Packing Schemes for Multiple Character Sets

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 2-4](#) (page 25) shows this with a somewhat pathological Shift-JIS byte sequence using only two different byte values (the corresponding character images are also shown).

Figure 1-4 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 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 world's 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 2-5 (page 27) shows the same Unicode sequence in UTF-16, UTF-8, and UTF-7.

Figure 1-5 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 +Uxd0rA-

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 2-1 (page 20) 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 2-2](#) (page 21), 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 2-6](#) (page 29) gives an idea of the variety of combining marks present in Unicode:

Figure 1-6 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 + ◌́ + ◌̇	á
Combining double diacritic	o + ◌̃ + ◌	õ
Combining half diacritic	o + ◌̂ + ◌ + ◌̂	õ
Combining enclosing mark	左 + ◌	⓵

Notice that displayed position of the acute accent depends on the base character

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 2-7](#) (page 30) provides examples of the behavior of fraction slash and conjoining jamos.

Figure 1-7 Fraction slash and conjoining jamos

	Character sequence	Resulting display (may use one or multiple glyphs)
Fraction slash	11 + / + 16	11/16
Conjoining jamos	॥ + ॥ + ॥	॥

In [Figure 2-6](#) (page 29) and [Figure 2-7](#) (page 30), 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 2-6](#) (page 29) and [Figure 2-7](#) (page 30) 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 2-8](#) (page 31) gives an example of implicit ordering.

Figure 1-8 Implicit ordering

Character sequence	Resulting display (with global direction of right-left)
A B C א ב ג 1 2 3 4 5 6	ג ב א A B C 6 5 4 1 2 3

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 1-9 Character sequence and resulting display

	Character sequence	Resulting display
Devanagari	ह + ि + ं H(A) I nasal mark	हिं HIM
Thai	เ + อ + ้ EH H(AW) rising tone	เอ้ HEH (rising)

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).

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;
```

Writing Custom Plug-Ins

```

    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 and how large the output was. The plug-in should save all necessary state information so that it can continue the conversion where it left off in the event that 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 an 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(pluginContext->bufferContext.textOutputBuffer,
outBuf, &inBuf, inEnd, &escState, &status);
}
/* save state information */
pluginContext->pluginState.state1 = escState;
pluginContext->pluginState.state2 = shiftState;
pluginContext->pluginState.state3 = saveByte;

/* save new buffer positions */
pluginContext->bufferContext.textOutputBuffer = outBuf;
pluginContext->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 *pluginContext)
{
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 *pluginContext)
{
#pragma unused( encodingConverter )

OSStatus status = noErr;

if (pluginContext)
{
BytePtr outBuf = pluginContext->bufferContext.textOutputBuffer;
BytePtr outEnd = pluginContext->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,
```

Writing Custom Plug-Ins

```

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 {
            TECTextEncoding /* 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,

```

Writing Custom Plug-Ins

```

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;

```

Writing Custom Plug-Ins

```

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:

```

OSSStatus ConverterPluginClearSnifferContextInfo(
TECSnifferObjectRef encodingSniffer,
TECSnifferContextRec *snifContext)
{
#pragma unused (encodingSniffer)
OSSStatus 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,

```

OSSStatus SniffEUC_KR(
TECSnifferObjectRef encodingSniffer,
TECSnifferContextRec *snifContext)
{
#pragma unused (encodingSniffer)
OSSStatus 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 = sniffContext->maxFeatures;

if (inputBuffer && inputBufferEnd)
{
Byte c;
UInt8 isoState = sniffContext->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 */
sniffContext->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 *sniffContext)
{
#pragma unused (encodingSniffer, sniffContext)
/* 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:

```
#ifndef 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 */
    }
```

CHAPTER 2

Writing Custom Plug-Ins

```
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>

Character Encodings and Internet Names

This document provides information on character encodings and the Internet. It discusses the Internet Assigned Numbers Authority (IANA) registry, Internet names used for similar character encodings that could lead to confusion, and provides a list of common Internet names for character encodings along with their availability in the Mac OS.

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. Note that the IANA registry is updated periodically. [Table A-1](#) (page 58) 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 A-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. In the last two columns of the table, “N/A” means that the encoding is not supported.

Table A-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

Character Encodings and 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
ISO 8859-3 (Latin 3)	ISO-8859-3 , latin3		1.5	1.5
ISO-8859-15 (Latin 9)	ISO-8859-15, latin9	Latin-1 with EURO SIGN and CP 1252 letters	1.5	1.5
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
CP 850 (DOS Latin-1)	cp850		1.4	1.4
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
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

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
ISO 8859-4 (Latin-4)	ISO-8859-4, latin4		1.5	1.5
CP 1250 (Windows Latin-2)	windows-1250, cp 1250	Partly 8859-2, plus C1 additions	1.2	1.2
CP 1257 (Windows BalticRim)	windows-1257, cp 1257		1.5	1.5
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
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)		Similar to GBK	1.4	1.4
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

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
CP 950 (DOS and Windows Traditional)		Based on Big-5	1.4	1.4
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
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

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
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	1.4
ISO 2022-JP ("JIS")	ISO-2022-JP	JIS 201 + JIS 208 + JIS 212 (7-bit); Rfc 1468	1.2	N/A
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	1.4	1.4
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

Character encoding	Common Internet names	Related information	Version of Text Encoding Conversion Manager that first offered support in:	
			Text Encoding Converter	Unicode Converter
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
Thai				
TIS 620-2533			N/A	N/A
CP 874 (DOS + Windows)	cp874	Based on TIS 620-2533	1.4	1.4
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
CP 1258 (Windows Vietnamese)	windows-1258, cp1258		1.5	1.5

Mac OS Encoding Variants

For font-based Mac OS encoding variants, Table B-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 B-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	kMacIcelandicStdDefaultVariant
Geneva	Application font	kMacIcelandicStdDefaultVariant
Monaco		kMacIcelandicStdDefaultVariant
New York		kMacIcelandicStdDefaultVariant
Palatino		kMacIcelandicTTDefaultVariant
Times		kMacIcelandicTTDefaultVariant
Helvetica		kMacIcelandicTTDefaultVariant
Courier		kMacIcelandicTTDefaultVariant
Mac OS Japanese		
Osaka	System font, application font	kMacJapaneseStandardVariant
Osaka Tohaba		kMacJapaneseStandardVariant
MaruGothic		*
HonMincho		*
TohabaGothic		kMacJapaneseBasicVariant
TohabaMincho		kMacJapaneseBasicVariant
ChuGothic	PostScript	kMacJapanesePostScriptScrnVariant

Mac OS Encoding Variants

English Name	Comments	Constant for Variant
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.)		
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
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

APPENDIX B

Mac OS Encoding Variants

English Name	Comments	Constant for Variant
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

Document Revision History

This table describes the changes to *Programming With the Text Encoding Conversion Manager*.

Date	Notes
2005-07-07	Added information to the section "Deciding Which Encoding Converter to Use."
2003-05-14	Removed the chapters Basic Text Types Reference, Text Encoding Converter Reference, and Unicode Converter Reference. The information in these chapters is now documented in the following reference documents: <i>Text Encoding Converter Reference</i> and <i>Unicode Converter Reference</i>
	The appendixes are now called chapters.
	Performed edits as needed to fix cross references, update references to the Mac OS, and to fix introductory material to be compatible with current style.
	Updated art and formatting.
	Structured the document to enable it to be converted to XML.
1999-10-04	The following changes were made to reflect revisions for TEC 1.5:
	Removed script code <code>kTextEncodingMacUkrainian</code> from enumeration in Text Encoding Base.
	Added the following ISO 8-bit and 7-bit script codes to enumeration under Text Encoding Base. <code>kTextEncodingISOLatin6 = 0x020A, /* ISO 8859-10 */</code> <code>kTextEncodingISOLatin7 = 0x020D, /* ISO 8859-13, Baltic Rim */</code> <code>kTextEncodingISOLatin8 = 0x020E, /* ISO 8859-14, Celtic */</code> <code>kTextEncodingISOLatin9 = 0x020F, /* ISO 8859-15, 8859-1 + EURO etc */</code>

REVISION HISTORY

Document Revision History

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](#) (page 71).

Code Fragment Manager (CFM) In Mac OS 9 and earlier, refers to 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 In Mac OS 9 and earlier, a fragment refers to 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](#) (page 73).

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 no-subset variants of Unicode, in either UTF-16 or UTF-8, and many 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.

