Chinese, Japanese, Korean & Vietnamese Computing

2nd Edition
Broader Unicode Coverage

CJKV

Information Processing

O'REILLY®

Ken Lunde

O'REILLY®

CJKV Information Processing

First published a decade ago, *CJKV Information Processing* quickly became the unsurpassed source of information on processing text in Chinese, Japanese, Korean, and Vietnamese. It has now been thoroughly updated to provide web and application developers with the latest techniques and tools for disseminating information directly to audiences in East Asia. This second edition reflects the considerable impact that Unicode, XML, OpenType, and operating systems such as Windows XP, Vista, Mac OS X, and Linux have had on East Asian text processing in recent years. With this book, you will:

- Learn about CJKV writing systems and scripts, and their transliteration methods
- Explore trends and developments in character sets and encodings, particularly Unicode
- Examine the world of typography, specifically how CJKV text is laid out on a page
- Learn information-processing techniques, such as code conversion algorithms, and apply them using different programming languages
- Process CJKV text using different platforms, text editors, and word processors
- Become more informed about CJKV dictionaries and dictionary software
- Manage CJKV content and presentation when publishing in print or for the Web

Internationalizing and localizing applications is paramount today—especially for audiences in East Asia, the fastest-growing segment of the computing world. *CJKV Information Processing*, Second Edition, will help you develop web and other applications effectively in a field that many find difficult to master.

"Our world is naturally complex, and the East Asian writing systems are perhaps doubly so. Their scripts pose challenges for all forms of communication and publishing, whether it is for the Web or for print. I highly recommend Ken Lunde's book as a companion and guide for navigating the intricacies of East Asian text processing."

—Dr. John Warnock, Co-founder, Adobe Systems Incorporated

Ken Lunde is a senior computer scientist in CJKV Type Development at Adobe Systems Incorporated. He has a Ph.D. in linguistics from the University of Wisconsin–Madison.

www.oreilly.com

US $59.99 **CAN $59.99**

ISBN: 978-0-596-51447-1

9 780596 514471 55999

CJKV Information Processing

SECOND EDITION

CJKV Information Processing

Ken Lunde

O'REILLY®
Běijīng · Cambridge · Farnham · Köln · Sebastopol · Táiběi · Tōkyō

CJKV Information Processing, Second Edition
by Ken Lunde

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 USA.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Julie Steele
Production Editors: Ken Lunde and Rachel Monaghan
Copyeditor: Mary Brady
Proofreader: Genevieve d'Entremont
Indexer: Ken Lunde

Production Services: Ken Lunde
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

January 1999: First Edition.
December 2008: Second Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *CJKV Information Processing*, the image of a blowfish, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book uses Repkover,™ a durable and flexible lay-flat binding.

ISBN: 978-0-596-51447-1
[C]

This book is dedicated to the four incredible women who have touched—and continue to influence—my life in immeasurable ways:

My mother, Jeanne Mae Lunde*, for bringing me into this world and for putting me on the path I am on.*

My mother-in-law, Sadae Kudo*, for unknowingly bringing together her daughter and me.*

My wife, friend, and partner, Hitomi Kudo*, for her companionship, love, and support, and for constantly reminding me about what is truly important.*

Our daughter, Ruby Mae Lunde*, for showing me that actions and decisions made today impact and influence our future generations.*

I shall be forever in their debt….

Contents

Foreword

The noncommittal phrase *information processing* covers a lot of ground, from generating mailing lists and tabulating stock exchange transactions to editing and typesetting Lady Murasaki's *Tale of Genji*, the meditations of Lao Zi, or the poems of Han Shan. There is a lot of information in the world, and it is stored, handled, and processed in a lot of different ways.

The oldest human writing systems known—including Sumerian, early Egyptian, Chinese, and early Mayan—seem to have sprung up independently. They thrived in different places, serving unrelated languages, and they look thoroughly different from one another, but they all have something in common: they all employ large numbers of signs for meanings, supplemented with signs for sounds. The only such script that has survived in common use to the present day is Han Chinese. All the other scripts now in general use for writing natural human languages are essentially confined to the writing of sounds. They are all syllabic, consonantal, or alphabetic.

Here in the West, we often speak of Chinese as if it were a single language. Once upon a time, perhaps it was—but for thousands of years things have been more complicated than that. There are now more than a dozen Chinese languages, each with several dialects, spoken in China and by people of Chinese origin living elsewhere in the world. The most successful member of the group, often called Mandarin, is spoken as a first or second language by roughly a billion people. Add all those who speak at least one of the other Chinese languages (such as Yuè, the common language of Hong Kong, or Mǐn Nán, the most common language in Taiwan, or Wú, which is common in Shanghai), and the total gets closer to a billion and a half. Add also the speakers of Japanese, Korean, and Vietnamese, whose languages belong to different families but whose scripts and literatures have a long and rich association with Chinese, and the number is larger yet: about 25% of the human population as a whole.

You can see from this alone why a book giving clear and thorough guidance on the handling of text in Chinese, Japanese, Korean, and Vietnamese might be important. But even if these scripts weren't part of daily life for a quarter of humanity, there would be good reasons to study them. Compared to the Latin alphabet, they are wonderfully complex

and polymorphous. That human speech can be recorded and transmitted in all these different ways tells us something about language, something about the mind, and something about how many different ways there are to be a human being.

There is always a certain tension between written and spoken language, because speech continues to change while writing is used to preserve it. Wherever reading and writing are normal parts of daily life, there are things that people know how to say but aren't sure how to write, and things that they know how to write, and may write fairly often, but might never find any occasion to say. (*Sincerely yours* is one example.) If we wrote only meanings and no sounds, the gulf between speech and writing would be wider, but the tension between them would often be less. This is what happens, in fact, when we use mathematical notation. We write basically nothing but symbols for meanings, along with a lot of punctuation to keep the meanings from getting confused, and what we have written can be read with equal ease, and with equal accuracy, in English, Hungarian, Arabic, or Chinese. It is not really possible to write spoken language in this way—but it is possible to write a close correlative. We can write (as logicians do when they use symbolic logic) the sequence of meanings that a string of spoken sentences would carry; then we can read what we have written by pronouncing the names of those meanings in any language we choose and filling in the holes with whatever inflections, links, and lubricants our spoken grammar requires. That in fact is how Japanese, Vietnamese, and Korean were first written: using Chinese characters to represent the meanings, then reading back the meanings of these Chinese signs in a language other than Chinese. (Most Chinese languages other than classical Mandarin are written even now in the same way.)

In time, the Japanese devised their delicate and supple syllabic scripts (*hiragana* and *katakana*) as a supplement to *kanji*, which are Han Chinese glyphs used to write Japanese; the Koreans devised their ingeniously analytical Hangul script, in which alphabetic information is nested into discrete syllabic units; and the Vietnamese, after spending a thousand years building their own large lexicon of redefined Chinese glyphs and new glyphs on the Chinese model, embraced a complex variant of the Latin alphabet instead. But in all three cultures, the old associations with Chinese script and Chinese literature run deep and have never disappeared. The connection is particularly vivid in the case of Japanese, whose script is what a linguistic geologist would call a kind of breccia or conglomerate: chunks of pure Chinese script and angular syllabics (and often chunks of Latin script as well) cemented into a matrix of cursive syllabics.

Ken Lunde is an enthusiast for all these complications and an expert on their electronic outcome—expert enough that his book is relied on by professionals and amateurs alike, in both Asia and the West.

Many North Americans and Europeans have written books about the Orient, but very few of those books have been translated into Asian languages, because so few of them can tell Asians anything about themselves. Once in a while, though, outsiders really know their stuff, and insiders see that this is so. The first edition of this book (which ran to 1,100 pages) was published in California in 1999. It was recognized at once as the definitive work in

the field and was promptly translated into both Chinese and Japanese. Its shorter predecessor, *Understanding Japanese Information Processing*—Ken Lunde's first book, published in 1993, when he was only 28—had been greeted the same way: it was recognized as the best book of its kind and promptly published, unabridged, in Japanese. As a reader, I am comforted by those endorsements. The subject is, after all, complex. Some would call it daunting. I know how useful this book has been to me, and it pleases me to know that native speakers of Chinese and Japanese have also found it useful.

Robert Bringhurst

Quadra Island, British Columbia · 21 August 2008

Preface

Close to 16 years have elapsed since *Understanding Japanese Information Processing* was published, and perhaps more importantly, 10 years have gone by since *CJKV Information Processing* was first published. A lot has changed in those 10 years. I should point out that I was first inspired to undertake the initial "CJKV" expansion sometime in 1996, during a lengthy conversation I had at a Togo's near the UC Berkeley campus with Peter Mui, my editor for *Understanding Japanese Information Processing*.

Join me in reading this thick tome of a book, which also serves as a reference book, and you shall discover that "CJKV" (Chinese, Japanese, Korean, and Vietnamese) will become a standard term in your arsenal of knowledge. But, before we dive into the details, allow me to discuss some terms with which you are no doubt familiar. Otherwise, you probably would have little need or desire to continue reading.

Known to more and more people, *internationalization*, *globalization*, and *localization* seem to have become household or "buzz" words in the field of computing and software development, and have also become very hot topics among high-tech companies and researchers due to the expansion of software markets to include virtually all parts of the planet. This book is specifically about CJKV-enabling, which is the adaptation of software for one or more CJKV locales. It is my intention that readers will find relevant and useful CJKV-enabling information within the pages of this book.

Virtually every book on internationalization, globalization, or localization includes information on character sets and encodings, but this book is intended to provide much more. In summary, it provides a brief description of writing systems and scripts, a thorough background of the history and current state of character sets, detailed information on encoding methods, code conversion techniques, input methods, keyboard arrays, font formats, glyph sets, typography, output methods, algorithms with sample source code, tools that perform useful information processing tasks, and how to handle CJKV text in the context of email and for web and print publishing. Expect to find plenty of platform-independent information and discussions about character sets, how CJKV text is encoded and handled on a number of operating systems, and basic guidelines and tips for developing software targeted for CJKV markets.

Now, let me tell you what this book is *not* about. Don't expect to find out how to design your own word-processing application, how to design your own fonts for use on a computer (although I provide sources for such tools), or how to properly handle formats for CJKV numerals, currency, dates, times, and so on. This book is not, by any stretch of the imagination, a complete reference manual for internationalization, globalization, or localization, but should serve remarkably well as a companion to such reference works, which have fortunately become more abundant.

It is my intention for this book to become the definitive source for information related to CJKV information processing issues.* Thus, this book focuses heavily on how CJKV text is handled on computer systems in a very platform-independent way, with an emphasis or bias toward Unicode and other related and matured technologies. Most importantly, everything that is presented in this book can be programmed, categorized, or easily referenced.

This book was written to fill the gap in terms of information relating to CJKV information processing, and to properly and effectively guide software developers. I first attempted to accomplish this over the course of several years by maintaining an online document that I named *JAPAN.INF* (and entitled *Electronic Handling of Japanese Text*). This document had been made publicly available through a number of FTP sites worldwide, and had gained international recognition as the definitive source for information relating to Japanese text handling on computer systems. *Understanding Japanese Information Processing* excerpted and further developed key information contained in *JAPAN.INF*. However, since the publication of *Understanding Japanese Information Processing* in 1993, *JAPAN.INF*, well, uh, sort of died. Not a horrible death, mind you, but rather to prepare for its reincarnation as a totally revised and expanded online document that I entitled *CJK.INF* (the CJK analog to *JAPAN.INF*). The work I did on *CJK.INF* helped to prepare me to write the first edition of this book, which provided updated material plus significantly more information about Chinese, Korean, and Vietnamese, to the extent that granting the book a new title was deemed appropriate and necessary. The second edition, which you have in your hands, represents a much-needed update, and I hope that it becomes as widely accepted and enjoyed as the first edition.

Although I have expended great effort to provide sufficient amounts of information for Chinese, Japanese, Korean, and Vietnamese computing, you may feel that some bias toward Japanese still lingers in many parts of this book. Well, if your focus or interest happens to be Japanese, chances are you won't even notice. In any case, you can feel at ease knowing that almost everything discussed in this book can apply equally to all of these languages. However, the details of Vietnamese computing in the context of using ideographs are still emerging, so its coverage is still somewhat limited and hasn't changed much since the first edition.

* The predecessor of this book, *Understanding Japanese Information Processing*, which had a clear focus on Japanese (hence its title) apparently became the definitive source for Japanese information processing issues, and was even translated into Japanese.

What Changed Since the First Edition?

Several important events took place during the 10 years since the first edition was published. These events could be characterized as technologies that were in the process of maturing, and have now fully matured and are now broadly used and supported.

First and foremost, *Unicode* has become the preferred way in which to represent text in digital format, meaning when used on a computer. Virtually all modern OSes and applications now support Unicode, and in a way that has helped to trivialize many of the complexities of handling CJKV text. As you read this book, you may feel a bias toward Unicode. This is for a good reason, because unless your software embraces Unicode, you are going down the wrong path. This also means that if you are using an application that doesn't handle Unicode, chances are it is outdated, and perhaps a newer version exists that supports Unicode.

Second, *OpenType* has become the preferred font format due to its cross-platform nature, and how it allows what were once competing font formats, Type 1 and TrueType, to exist in harmony. Of course, OpenType fonts support Unicode and have strong multilingual capabilities. OpenType fonts can also provide advanced typographic functionality.

Third, *PDF* (*Portable Document Format*) has become the preferred way in which to publish for print, and is also preferred for the Web when finer control over presentation is desired. In addition to supporting Unicode, PDF encapsulates documents in such a way that they can be considered a reliable digital master, and the same file can be used for displaying and printing. In short, PDF has become the key to the modern publishing workflow.

Last but not least, the *Web* itself has matured and advanced in ways that could not be predicted. The languages used to build the Web, which range from languages that describe the content and presentation of web documents, such as CSS, HTML, XHTML, and XML, to scripting languages that enable dynamic content, have matured, and they all have one thing in common: they all support Unicode. In case it is not obvious, Unicode will be a recurring theme throughout this book.

Audience

Anyone interested in how CJKV text is processed on modern computers will find this book useful, including those who wish to enter the field of CJKV information processing, and those who are already in the field but have a strong desire for additional reference material. This book will also be useful for people using any kind of computer and any type of operating system, such as FreeBSD, the various Linux distributions, Mac OS X, MS-DOS, Unix, and the various flavors of Windows.

Although this book is specifically about CJKV information processing, anyone with an interest in creating multilingual software or a general interest in I18N (*internationalization*), G11N (*globalization*), or L10N (*localization*) will learn a great deal about the issues involved in handling complex writing systems and scripts on computers. This is particularly

true for people interested in working with CJKV text. Thankfully, information relating to the CJKV-enabling of software has become less scarce.

I assume that readers have little or no knowledge of a CJKV language (Chinese, Japanese, Korean, or Vietnamese) and its writing system. In Chapter 2, *Writing Systems and Scripts*, I include material that should serve as a good introduction to CJKV languages, their writing systems, and the scripts that they use. If you are familiar with only one CJKV language, Chapter 2 should prove to be quite useful for understanding the others.

Conventions Used in This Book

Kanji, hanzi, hanja, kana, hiragana, katakana, hangul, jamo, and other terms will come up, time and time again, throughout this book. You will also encounter abbreviations and acronyms, such as ANSI, ASCII, CNS, EUC, GB, GB/T, GBK, ISO, JIS, KS, and TCVN. Terms, abbreviations, and acronyms—along with many others words—are usually explained in the text, and again in Appendix D, *Glossary*, which I encourage you to study.

When hexadecimal values are used in the text for lone or single bytes, and to remove any ambiguity, they are prefixed with `0x`, such as `0x80`. When more than one byte is specified, hexadecimal values are instead enclosed in angled brackets and separated by a space, such as <80 80> for two instances of `0x80`. Unicode scalar values follow the convention of using the `U+` prefix followed by four to six hexadecimal digits. Unicode encoding forms are enclosed in angled brackets and shown as hexadecimal code units, except when the encoding form specifies byte order, in which case the code units are further broken down into individual bytes. `U+20000`, for example, is expressed as <D840 DC00> in UTF-16, but as <D8 40 DC 00> in UTF-16BE (UTF-16 big-endian) and as <40 D8 00 DC> in UTF-16LE (UTF-16 little-endian). Its UTF-8 equivalent is <F0 A0 80 80>. The angled brackets are generally omitted when such values appear in a table. Furthermore, Unicode sequences are expressed as Unicode scalar values separated by a comma and enclosed in angled brackets, such as <U+304B, U+309A>.

Decimal values are almost always clearly identified as such, and when used, appear as themselves without a prefix, and unenclosed. For those who prefer other notations, such as binary or octal, Appendix B, *Notation Conversion Table*, can be consulted to convert between all four notations.

Throughout this book I generically use short suffixes such as "J," "K," "S," "T," "V," and "CJKV" to denote locale-specific or CJKV-capable versions of software products. I use these suffixes for the sake of consistency, and because software manufacturers often change the way in which they denote CJKV versions of their products. In practice, you may instead encounter the suffix 日本語版 (*nihongoban*, meaning "Japanese version"), the prefix "Kanji," or the prefix 日本語 (*nihongo*, meaning "Japanese") in Japanese product names. For Chinese software, 中文 (*zhōngwén*, meaning "Chinese") is a common prefix. I also refrain from using version numbers for software described in this book (as you know, this sort of information becomes outdated very quickly). I use version numbers only when they represent a significant advancement or development stage in a product.

References to "China" in this book refer to the People's Republic of China (PRC; 中华人民共和国 *zhōnghuá rénmín gònghé guó*), also commonly known as Mainland China. References to "Taiwan" in this book refer to the Republic of China (ROC; 中華民國 *zhōnghuá mínguó*). Quite often this distinction is necessary.

Name ordering in this book, when transliterated in Latin characters, follows the convention that is used in the West—the given name appears first, followed by the surname. When the name is written using CJKV characters—in parentheses following the transliterated version—the surname appears first, followed by the given name.

"ISO 10646" and "Unicode" are used interchangeably throughout this book. Only in some specific contexts are they different.

Italic is used for pathnames, filenames, program names, new terms where they are defined, newsgroup names, and web addresses, such as domain names, URLs, and email addresses.

`Constant width` is used in examples to illustrate output from commands, the contents of files, or the text of email messages.

`Constant width bold` is used in examples to indicate commands or other text that should be typed literally by the user; occasionally it is also used to distinguish parts of an example.

The `%` (percent) character is used to represent the Unix shell prompt for Unix and similar command lines.

Footnotes are used for parenthetical remarks and for providing URLs. Sometimes what is written in the text proper has been simplified or shortened for the purpose of easing the discussion or for practical reasons (especially in Chapter 2 where I introduce the many CJKV writing systems), and the footnotes—usually, but not always—provide additional details.

How This Book Is Organized

Let's now preview the contents of each chapter in this book. Don't feel compelled to read this book linearly, but feel free to jump around from section to section. Also, the index is there for you to use.

Chapter 1, *CJKV Information Processing Overview*, provides a bird's eye overview of the issues that are addressed by this book, and is intended to give readers an idea of what they can expect to learn. This chapter establishes the context in which this book will become useful in your work or research.

Chapter 2, *Writing Systems and Scripts*, contains information directly relating to CJKV writing systems and their scripts. Here you will learn about the various types of characters that compose CJKV texts. This chapter is intended for readers who are not familiar with the Chinese, Japanese, Korean, or Vietnamese languages (or who are familiar with

only one or two of those languages). Everyone is bound to learn something new in this chapter.

Chapter 3, *Character Set Standards*, describes the two classes of CJKV character set standards: coded and noncoded. Coded character set standards are further divided into two classes: national and international. Comparisons are also drawn between CJKV character set standards, and the coverage of Unicode is extensive.

Chapter 4, *Encoding Methods*, contains information on how the character set standards described in Chapter 3 are encoded on computer systems. Emphasis is naturally given to the encoding forms of Unicode, but information about legacy encoding methods is also provided. Encoding is a complex but important step in representing and manipulating human-language text in a computer. Other topics include software for converting from one CJKV encoding to another, and instructions on how to repair damaged CJKV text files.

Chapter 5, *Input Methods*, contains information on how CJKV text is input. First I discuss CJKV input in general terms, and then describe several specific methods for entering CJKV characters on computer systems. Next, we move on to the hardware necessary for CJKV input, specifically keyboard arrays. These range from common keyboard arrays, such as the QWERTY array, to ideograph tablets containing thousands of individual keys.

Chapter 6, *Font Formats, Glyph Sets, and Font Tools*, contains information about bitmapped and outline font formats as they relate to CJKV, with an emphasis toward OpenType. The information presented in this chapter represents my daily work at Adobe Systems, so some of its sections may suffer from excruciating detail, which explains the length of this chapter.

Chapter 7, *Typography*, contains information about how CJKV text is properly laid out on a line and on a printed page. Merely having CJKV fonts installed is not enough—there are rules that govern where characters can and cannot be used, and how different character classes behave, in terms of spacing, when in proximity. The chapter ends with a description of applications that provide advanced page composition functionality.

Chapter 8, *Output Methods*, contains information about how to display, print, or otherwise output CJKV text. Here you will find information relating to the latest printing and display technologies.

Chapter 9, *Information Processing Techniques*, contains information and algorithms relating to CJKV code conversion and text-handling techniques. The actual mechanics are described in detail, and, where appropriate, include algorithms written in C, Java, and other programming languages. Though somewhat dated, the chapter ends with a brief description of three Japanese code-processing tools that I have written and maintained over a period of several years. These tools demonstrate how the algorithms can be applied in the context of Japanese.

Chapter 10, *OSes, Text Editors, and Word Processors*, contains information about operating systems, text editors, and word processors that are CJKV-capable, meaning that they support one or more CJKV locale.

Chapter 11, *Dictionaries and Dictionary Software*, contains information about dictionaries, both printed and electronic, that are useful when dealing with CJKV text. Also included are tips on how to more efficiently make use of the various indexes used to locate ideographs in dictionaries.

Chapter 12, *Web and Print Publishing*, contains information on how CJKV text is best handled electronically over networks, such as when using email clients. Included are tips on how to ensure that what you send is received intact, as well as information about the Internet domains that cover the CJKV locales. Web and print publishing, through the use of HTML (*HyperText Markup Language*), XML (*Extensible Markup Language*), and PDF (*Portable Document Format*) are also discussed in detail.

Appendix A, *Code Conversion Tables*, provides a code conversion table between decimal Row-Cell, hexadecimal ISO-2022, hexadecimal EUC, and hexadecimal Shift-JIS (Japanese-specific) codes. Also included is an extension that handles the Shift-JIS user-defined range.

Appendix B, *Notation Conversion Table*, lists all 256 8-bit byte values in the four common notations: binary, octal, decimal, and hexadecimal.

Appendix C, *Perl Code Examples*, provides Perl equivalents of many algorithms found in Chapter 9—along with other goodies.

Appendix D, *Glossary*, defines many of the concepts and terms used throughout this book (and other books).

Finally, the *Bibliography* lists many useful references, many of which were consulted while writing this book.

Although not included in the printed version of this book, the following appendixes are provided as downloadable and printable PDFs. This book does include placeholder pages for them, which serve to specify their URLs.

Appendix E, *Vendor Character Set Standards*, is reference material for those interested in vendor-specific extensions to CJKV character set standards. To a great extent, Unicode has effectively deprecated these standards.

Appendix F, *Vendor Encoding Methods*, is reference material for those interested in how the vendor character sets in Appendix E are encoded.

Appendix G, *Chinese Character Sets—China*, provides character set tables, character lists, mapping tables, and indexes that relate to standards from China.

Appendix H, *Chinese Character Sets—Taiwan*, provides character set tables, character lists, mapping tables, and indexes that relate to standards from Taiwan.

Appendix I, *Chinese Character Sets—Hong Kong*, provides character set tables and character lists that relate to standards from Hong Kong.

Appendix J, *Japanese Character Sets*, provides character set tables, character lists, mapping tables, and indexes that relate to standards from Japan.

Appendix K, *Korean Character Sets*, provides character set tables, character lists, mapping tables, and indexes that relate to standards from Korea, specifically South Korea.

Appendix L, *Vietnamese Character Sets*, provides character set tables that relate to standards from Vietnam.

Appendix M, *Miscellaneous Character Sets*, provides character set tables for standards such as ASCII, ISO 8859, EBCDIC, and EBCDIK.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM or DVD of examples from this book does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*CJKV Information Processing*, Second Edition, by Ken Lunde. Copyright 2009 O'Reilly Media, Inc., 978-0-596-51447-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
USA
800-998-9938 (in the United States or Canada)
+1-707-829-0515 (international/local)
+1-707-829-0104 (facsimile)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://oreilly.com/catalog/9780596514471/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

Safari® Books Online

When you see a Safari Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com/*.

Acknowledgments

To write a book or reference work this thick requires interaction with and cooperation from people across our planet. It is not possible for me to list all the people who have helped or guided me over the years—there are literally hundreds.

In some cases, people simply come to me for help on a particular subject (that's what happens, I guess, when people are aware of your email address—to ensure that I will receive a ton of email, in various parts of this book you will find that my email address is *lunde@adobe.com*). Sometimes I may not know the answer, but the question usually inspires me to seek out the truth. The truth is out there.

The year 2008 marks 17 wonderful years at Adobe Systems, a company that provides me with daily CJKV-related challenges. Its always-advancing technologies and commitment to customers is what initially attracted me, and these are the qualities and values that keep me here. Besides, they let me display on the wall of my office the antelopes that I have harvested* annually in Wyoming since 2003. I firmly believe that "diversity in the workforce" is a door that swings both ways, and the antelopes that I display in my office serve as a tribute to that belief. Speaking of tributes, all aspects of the production of this book are a tribute to Adobe Systems' publishing technologies.

To all the people who have read and criticized my previous writings, tolerated my sometimes dull but otherwise charming personality at work, pointed out errors in my work, exchanged email with me for whatever reason, or otherwise helped me to grow and become a better person: *thank you!* You should know who you are.

Special thanks go to Tim O'Reilly (the president and founder of O'Reilly Media) and Peter Mui for believing in my first book, *Understanding Japanese Information Processing*, and to Peter for serving as its editor. It was Peter who encouraged me to expand it to cover the complete CJKV framework. Thanks go to Edie Freedman for sticking with my idea

* Harvesting is another way to express hunting.

of a blowfish for the cover.* Ron Bilodeau graciously helped me through the layout of the book, and nurtured my desire to embrace Adobe InDesign's particular paradigm. Robert Romano also deserves a lot of credit for the work he did on the figures that are found throughout this book. Julie Steele, my editor, continually pushed and prodded me to get this book done as close to schedule as possible. Mary Brady performed the copyedit, exposing various errors and oddities that crept from my fingers to the keyboard. Genevieve d'Entremont proofread the entire book, and discovered additional gems. Rachel Monaghan reviewed the index, and also provided production assistance.

Attempting to write a book of any length, while holding down a day job and doing so as an activity above and beyond it, takes a lot of effort and perseverance. I'd like to specifically thank David Lemon, Karen Catlin, and Digby Horner for their support and encouragement while writing this book. My colleagues and coworkers at Adobe Systems, especially those in Type Development, deserve the same recognition. Miguel Sousa, a coworker in Type Development, needs to be singled out and especially thanked for providing to me custom versions of the Minion Pro and Myriad Pro fonts that include the glyphs necessary for Pinyin transliteration and for implementing tabular hexadecimal digits, both of which are used extensively throughout this book. The tabular headecimal digits are especially nice, especially for a book such as this one.

The following individuals were incredibly helpful by reviewing various parts of this book, from specific pages to entire chapters, during the various stages of its prolonged development: Tom Bishop, Jim Breen, Robert Bringhurst, Seong Ah Choi (최성아), Richard Cook, Gu Hua (顾华), Paul Hackett, Jerry Hall, Taichi Kawabata (川幡太一), John Knightley, Tatsuo Kobayashi (小林龍生), Mark Leisher, David Lemon, Lu Qin (陸勤), Nat McCully, Dirk Meyer, Charles Muller, Eric Muller, Mihai Nita, Thomas Phinney, Read Roberts, Markus Scherer, Jungshik Shin (신정식), Miguel Sousa, Frank Tang (譚永鋒), Margie Vogel, Taro Yamamoto (山本太郎), and Retarkgo Yan (甄炯輝). I am indebted to each and every one of them for providing me with useful insights and inspirational ideas, and in some deserving cases, sharp criticism. I am, of course, ultimately responsible for any errors, omissions, or oddities that you may encounter while reading this book.

Finally, I wish to thank my wonderful parents, Vernon Delano Lunde and Jeanne Mae Lunde, for all of their support throughout the years; my son, Edward Dharmputra Lunde; my step-son, Ryuho Kudo (工藤龍芳); my beautiful daughter, Ruby Mae Lunde (工藤瑠美); and my beloved and caring wife, Hitomi Kudo (工藤仁美). I treasure the time that I spend with my parents, which includes varmint and larger game hunting with my father. Having my own family has great rewards.

* Michael Slinn made the astute observation that the *Babel Fish* would have been more appropriate as a cover creature for this book—according to Douglas Adams' *The Hitchhiker's Guide to the Galaxy*, you simply stick a Babel Fish in your ear, it connects with your brain, and you can suddenly understand all languages.

CHAPTER 1
CJKV Information Processing Overview

Here I begin this book by stating that a lot of mystique and intrigue surrounds how CJKV—*Chinese, Japanese, Korean, and Vietnamese*—text is handled on computer systems, ranging from handheld mobile devices to mainframe computers. Although I agree with there being sufficient intrigue, there is far too much mystery in my opinion and experience. Much of this is due to a lack of information, or perhaps simply due to a lack of information written in a language other than Chinese, Japanese, Korean, or Vietnamese. Nevertheless, many fine folks, such as you, the reader of this book, would like to know how this all works. To confirm some of your worst fears and speculations, CJKV text *does* require special handling on computer systems. However, it should not be very mysterious after having read this book. In my experience, you merely need to break the so-called *one-byte-equals-one-character* barrier—most CJKV characters are represented by more than a single byte (or, to put it in another way, more than eight bits).*

English information processing was a reality soon after the introduction of early computer systems, which were first developed in England and the United States. Adapting software to handle more complex scripts, such as those used to represent CJKV text, is a more recent phenomenon. This adaptation developed in various stages and continues today.

Listed here are several key issues that make CJKV text a challenge to process on computer systems:

- CJKV writing systems use a mixture of different, but sometimes related, scripts.
- CJKV character set standards enumerate thousands or tens of thousands of characters, which is orders of magnitude more than used in the West—*Unicode now includes more than 100,000 characters, adequately covering CJKV needs.*
- There is no universally recognized or accepted CJKV character set standard such as ASCII for writing English—*I would claim that Unicode has become such a character set, hence its extensive coverage in this book.*

* For a greater awareness of, and appreciation for, some of the complexities of dealing with multiple-byte text, you might consider glancing now at the section entitled "Byte Versus Character Handling" in Chapter 9.

- There is no universally recognized or accepted CJKV encoding method such as ASCII encoding—*again, the various Unicode encoding forms have become the most widely used encodings, for OSes, applications, and for web pages.*
- There is no universally recognized or accepted input device such as the QWERTY keyboard array—this same keyboard array, through a method of transliteration, is frequently used to input most CJKV text through reading or other means.
- CJKV text can be written horizontally or vertically, and requires special typographic rules not found in Western typography, such as spanning tabs and unique line-breaking rules.

Learning that the ASCII character set standard is not as universal as most people think is an important step. You may begin to wonder why so many developers assume that everyone uses ASCII. This is okay. For some regions, ASCII is sufficient. Still, ASCII has its virtues. It is relatively stable, and it forms the foundation for many character sets and encodings. UTF-8 encoding, which is the most common encoding form used for today's web pages, uses ASCII as a subset. In fact, it is this characteristic that makes its use preferred over the other two Unicode encoding forms, specifically UTF-16 and UTF-32.

Over the course of reading this chapter, you will encounter several sections that explain and illustrate some very basic, yet important, computing concepts, such as notation and byte order, all of which directly relate to material that is covered in the remainder of this book. Even if you consider yourself a seasoned software engineer or expert programmer, you may still find value in those sections, because they carry much more importance in the context of CJKV information processing. That is, how these concepts relate to CJKV information processing may be slightly different than what you had previously learned.

Writing Systems and Scripts

CJKV text is typically composed of a mixture of different scripts. The Japanese writing system, as an example, is unique in that it uses four different scripts. Others, such as Chinese and Korean, use fewer. Japanese is one of the few, if not the only, languages whose writing system exhibits this characteristic of so many scripts being used together, even in the same sentence (as you will see very soon). This makes Japanese quite complex, orthographically speaking, and poses several problems.*

Unicode's definitions of *script* and *writing system* are useful to consider. *Script* is defined as a collection of letters and other written signs used to represent textual information in one or more writing systems. For example, Russian is written with a subset of the Cyrillic script; Ukranian is written with a different subset. The Japanese writing system uses several scripts. *Writing system* is defined as a set of rules for using one or more scripts to write a particular language. Examples include the American English writing system, the British English writing system, the French writing system, and the Japanese writing system.

* *Orthography* is a linguistic term that refers to the writing system of a language.

The four Japanese scripts are *Latin characters*, *hiragana*, *katakana*, and *kanji* (collectively referred to as *ideographs* regardless of the language). You are already familiar with Latin characters, because the English language is written with these. This script consists of the upper- and lowercase Latin alphabet, which consists of the characters often found on typewriter keys. Hiragana and katakana are native Japanese syllabaries (see Appendix D for a definition of "syllabary"). Both hiragana and katakana represent the same set of syllables and are collectively known as *kana*. Kanji are ideographs that the Japanese borrowed from China over 1,600 years ago. Ideographs number in the tens of thousands and encompass meaning, reading, and shape.

Now let's look at an example sentence composed of these four scripts, which will serve to illustrate how the different Japanese scripts can be effectively mixed:

EUC等のエンコーディング方法は日本語と英語が混交しているテキストをサポートします。

In case you are curious, this sentence means "Encoding methods such as EUC can support texts that mix Japanese and English." Let's look at this sentence again, but with the Latin characters underlined:

EUC等のエンコーディング方法は日本語と英語が混交しているテキストをサポートします。

In this case, there is a single abbreviation, EUC (short for *Extended Unix Code*, which refers to a locale-independent encoding method, a topic covered in Chapter 4). It is quite common to find Latin characters used for abbreviations in CJKV texts. Latin characters used to transliterate Japanese text are called ローマ字 (*rōmaji*) or ラテン文字 (*raten moji*) in Japanese.

Now let's underline the katakana characters:

EUC等のエンコーディング方法は日本語と英語が混交しているテキストをサポートします。

Each katakana character represents one syllable, typically a lone vowel or a consonant-plus-vowel combination. Katakana characters are commonly used for writing words borrowed from other languages, such as English, French, or German. Table 1-1 lists these three underlined katakana words, along with their meanings and readings.

Table 1-1. Sample katakana

Katakana	Meaning	Reading[a]
エンコーディング	encoding	*enkōdingu*
テキスト	text	*tekisuto*
サポート	support	*sapōto*

a. The macron is used to denote long vowel sounds.

Note how their readings closely match that of their English counterparts, from which they were derived. This is no coincidence: it is common for the Japanese readings of borrowed words to be spelled out with katakana characters to closely match the original.

Next we underline the hiragana characters:

EUC等のエンコーディング方法は日本語と英語が混交しているテキストをサポートします。

Hiragana characters, like katakana as just described, represent syllables. Hiragana characters are mostly used for writing grammatical words and inflectional endings. Table 1-2 illustrates the usage or meaning of the hiragana in the preceding example.

Table 1-2. Sample hiragana

Hiragana	Meaning or usage	Reading
の	Possessive marker	*no*
は	Topic marker	*wa*[a]
と	and (conjunction)	*to*
が	Subject marker	*ga*
している	doing… (verb)	*shite-iru*
を	Object marker	*o*
します	do… (verb)	*shimasu*

a. This hiragana character is normally pronounced *ha*, but when used as a topic marker, it becomes *wa*.

That's a lot of grammatical stuff! Japanese is a postpositional language, meaning that grammatical markers, such as the equivalent of prepositions as used in English, come after the nouns that they modify. These grammatical markers are called *particles* (助詞 *joshi*) in Japanese.

Finally, we underline the ideographs (called *hànzì* in Chinese, *kanji* in Japanese, *hanja* in Korean, and *chữ Hán* and *chữ Nôm* in Vietnamese):

EUC等のエンコーディング方法は日本語と英語が混交しているテキストをサポートします。

At first glance, ideographs appear to be more complex than the other characters in the sentence. This happens to be true most of the time. Ideographs represent meanings and are often called *Chinese characters*, *Han characters*, *pictographs*, or *logographs*.* Ideographs are also assigned one or more readings (pronunciations), each of which is determined by context. While their readings differ depending on the language (meaning Chinese, Japanese, Korean, or Vietnamese), ideographs often have or convey the same or similar meaning. This makes it possible for Japanese to understand—but not necessarily pronounce—

* Being a widespread convention, this is beyond critique. However, linguists use these terms for different classes of ideographs, depending on their etymology.

some very basic Chinese, Korean, and Vietnamese texts. Table 1-3 provides a listing of the underlined ideographs and ideograph compounds (words composed of two or more ideographs) from our example sentence, and supplies their meanings and readings.

Table 1-3. Sample ideographs and ideograph compounds

Ideographs	Meaning	Reading
等	such as…	*nado*
方法	method	*hōhō*
日本語	Japanese (language)	*nihongo*
英語	English (language)	*eigo*
混交	(to) mix	*konkō*

Of course, this example includes only those types of characters that are used in Japanese—other locales use different types of characters. Table 1-4 lists the four CJKV locales, along with what scripts their writing systems use.

Table 1-4. CJKV locales and their scripts

Locale	Scripts
China	Latin and hanzi (simplified)
Taiwan	Latin, zhuyin, and hanzi (traditional)
Japan	Latin, hiragana, katakana, and kanji
Korea[a]	Latin, jamo, hangul, and hanja
Vietnam	Latin (Quốc ngữ), chữ Nôm, and chữ Hán

a. Jamo are the alphabet-like components that make up hangul.

Table 1-5 lists some sample characters from each of the scripts used in CJKV locales. We discuss the scripts of these writing systems in much greater detail in Chapter 2.

Table 1-5. Sample CJKV characters

Script	Sample characters
Latin characters	A B C D E F G H I J … q r s t u v w x y z
Zhuyin	ㄅㄆㄇㄈㄉㄊㄋㄌㄍㄎ … ㄠㄡㄢㄣㄤㄥㄦㄧㄨㄩ
Hiragana	ぁあぃいぅうぇえぉお … りるれろゎわゐゑをん
Katakana	ァアィイゥウェエォオ … ロヮワヰヱヲンヴヵヶ
Jamo	ㄱㄲㄳㄴㄵㄶㄷㄸㄹㄺ … ㅀㅁㅂㅃㅄㅅㅆㅇㅈㅉ・ㆍㅣ
Hangul syllables	가각간갇갈갉갊감갑값 … 휩휭히힉힌힐힘힙힛힝

Table 1-5. Sample CJKV characters

Script	Sample characters
Hanzi (simplified)	啊阿埃挨哎唉哀皑癌蔼 … 黪黯鼢鼬鼯鼹鼷鼽鼾齄
Hanzi (traditional)	一乙丁七乃九了二人儿 … 驫鱺鸝灩灪爩麤齾齉龘
Kanji	亜唖娃阿哀愛挨姶逢葵 … 齶龕龜龠堯槇遙瑤凜熙
Hanja	伽佳假價加可呵哥嘉嫁 … 晞曦熙熹熺犧禧稀羲詰

But, how frequently are each of these scripts used? Given an average sampling of Japanese writing, one normally finds 30% kanji, 60% hiragana, and 10% katakana. Actual percentages depend on the nature of the text. For example, you may find a higher percentage of kanji in technical literature, and a higher percentage of katakana in fields such as fashion and cosmetics, which make extensive use of loan words written in katakana. Most Korean texts consist of nothing but hangul syllables, and most Chinese texts are composed of only hanzi.* Latin characters are used the least, except in Vietnam, where they represent the primary script.

So, how many characters do you need to learn in order to read and write CJKV languages effectively? Here are some *very* basic guidelines:

- You must learn hiragana and katakana if you plan to deal with Japanese—this constitutes approximately 200 characters.
- Learning hangul is absolutely necessary for Korean, but you can get away with not learning hanja.
- You need to have general knowledge of about 1,000 kanji to read over 90% of the kanji in typical Japanese texts—more are required for reading Chinese texts because only hanzi are used.

If you have not already learned Chinese, Japanese, Korean, or Vietnamese, I encourage you to learn one of them so that you can better appreciate the complexity of their writing systems. Although I discuss character dictionaries, and learning aids to a lesser extent, in Chapter 11, they are no substitute for a human teacher.

Character Set Standards

A character set simply provides a common *bucket*, *repertoire*, or *collection* of characters. You may have never thought of it this way, but the English alphabet is an example of a character set standard. It specifies 52 upper- and lowercase letters. Character set standards are used to ensure that we learn a minimum number of characters in order to communicate with others in society. In effect, they limit the number of characters we need to

* Well, you will also find symbol-like characters, such as punctuation marks.

learn. There are only a handful of characters in the English alphabet, so nothing is really being limited, and as such, there really is no character set standard per se. In the case of languages that use ideographs, however, character set standards play an especially vital role. They specify which ideographs—out of the tens of thousands in existence—are the most important to learn for each locale. The current Japanese set, called *Jōyō Kanji* (常用漢字 *jōyō kanji*), although advisory, limits the number of ideographs to 1,945.* There are similar character sets in China, Taiwan, and Korea. These character set standards were designed with education in mind, and are referred to as *noncoded* character sets.

Character set standards designed for use on computer systems are almost always larger than those used for the purpose of education, and are referred to as *coded* character sets. Establishing coded character set standards for use with computer systems is a way to ensure that everyone is able to view documents created by someone else. ASCII is a Western character set standard, and ensures that their computer systems can communicate with each other. But, as you will soon learn, ASCII is not sufficient for the purpose of professional publishing (neither is its most common extension, ISO 8859-1:1998).

Coded character set standards typically contain characters above and beyond those found in noncoded ones. For example, the ASCII character set standard contains 94 printable characters—42 more than the upper- and lowercase alphabet. In the case of Japanese, there are thousands of characters in the coded character sets in addition to the 1,945 in the basic noncoded character set. The basic coded Japanese character set standard, in its most current form, enumerates 6,879 characters and is designated JIS X 0208:1997. There are four versions of this character set, each designated by the year in which it was established: 1978, 1983, 1990, and 1997. There are two typical compatibility problems that you may encounter when dealing with different versions of the same character set standard:

- Some of these versions contain different numbers of characters—later versions generally add characters.
- Some of these versions are not 100% compatible with each other due to changes.

In addition, there may be an extended character set standard, such as Japan's JIS X 0212-1990, that defines 6,067 additional characters, most of which are kanji, or China's GB 18030-2005, which adds tens of thousands of characters to its original GB 2312-80 standard.

Additional incompatibility has occurred because operating system (OS) developers took these coded character set standards one step further by defining their own extensions. These vendor character set standards are largely, but not completely, compatible, and almost always use one of the national standards as their base. When you factor in vendor character set standards, things appear to be a big mess. Fortunately, the broad adoption of Unicode has nearly brought the development of vendor-specific character sets to a halt. This book documents these character sets, primarily for historical purposes.

* The predecessor of this character set, *Tōyō Kanji* (当用漢字 *tōyō kanji*), was prescriptive.

Encoding Methods

Encoding is the process of mapping a character to a numeric value, or more precisely, assigning a numeric value to a character. By doing this, you create the ability to uniquely identify a character through its associated numeric value. The more unique a value is among different encoding methods, the more likely that character identification will be unambiguous. Ultimately, the computer needs to manipulate the character as a numeric value. Independent of any CJKV language or computerized implementations thereof, indexing encoded values allows a numerically enforced ordering to be mapped onto what might otherwise be a randomly ordered natural language.

While there is no universally recognized encoding method, many have been commonly used—for example, ISO-2022-KR, EUC-KR, Johab, and Unified Hangul Code (UHC) for Korean. Although Unicode does not employ a single encoding form, it is safe to state that the encoding forms for Unicode—UTF-8, UTF-16, and UTF-32—have become universally recognized. In addition, each one has become the preferred encoding form for specific uses. For the Web, UTF-8 is the most common encoding form. Applications prefer to use the UTF-16 encoding form internally for its overall space-efficiency for the majority of multilingual text. OpenType fonts, when they include glyphs that map from characters outside the *Basic Multilingual Plane* (BMP), make use of and prefer the UTF-32 encoding form.

Data Storage Basics

First, before describing these encoding methods, here's a short explanation of how memory is allocated on computer systems. Computer systems process data called *bits*. These are the most basic units of information, and they can hold or store one of two possible values: *on* or *off*. These are usually mapped to the values 1 or 0, respectively. Bits are strung together into units called *bytes*. Bytes are usually composed of 7 or 8 bits. Seven bits allow for up to 128 unique combinations, or values; 8 bits allow for up to 256. While these numbers are sufficient for representing most characters in Western writing systems, it does not even come close to accommodating large character sets whose characters number in the thousands, such as those required by the CJKV locales. It is also possible to use more than 8 bits, and some encoding methods use 16 or 32 bits as their code units, which are equivalent to 2 and 4 bytes, respectively.

The first attempt to encode an Asian language script on computer systems involved the use of Japanese half-width katakana characters. This is a limited set of 63 characters that constitutes a minimal set for representing Japanese text. But there was no support for kanji. The solution to this problem, at least for Japanese, was formalized in 1978, and employed the notion of using 2 bytes to represent a single character. This did not eliminate the need for one-byte characters, though. The Japanese solution was to extend the notion of one-byte character encoding to include two-byte characters. This allows for text with mixed one- and two-byte characters. How one- and two-byte characters are distinguished depends on the encoding method. Two bytes equal 16 bits, and thus can provide up to

65,536 unique values. This is best visualized as a 256×256 matrix. See Figure 1-1 for an illustration of such a matrix.

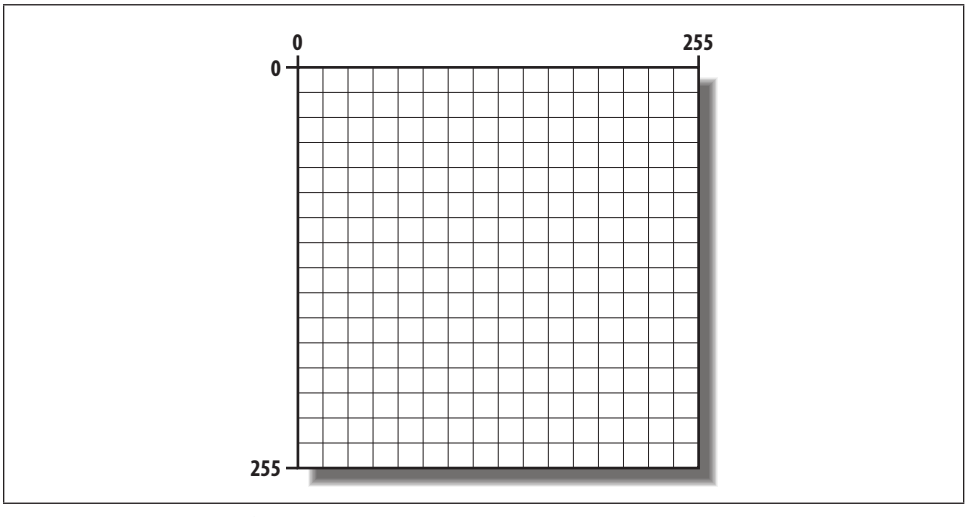


Figure 1-1. 256×256 encoding matrix

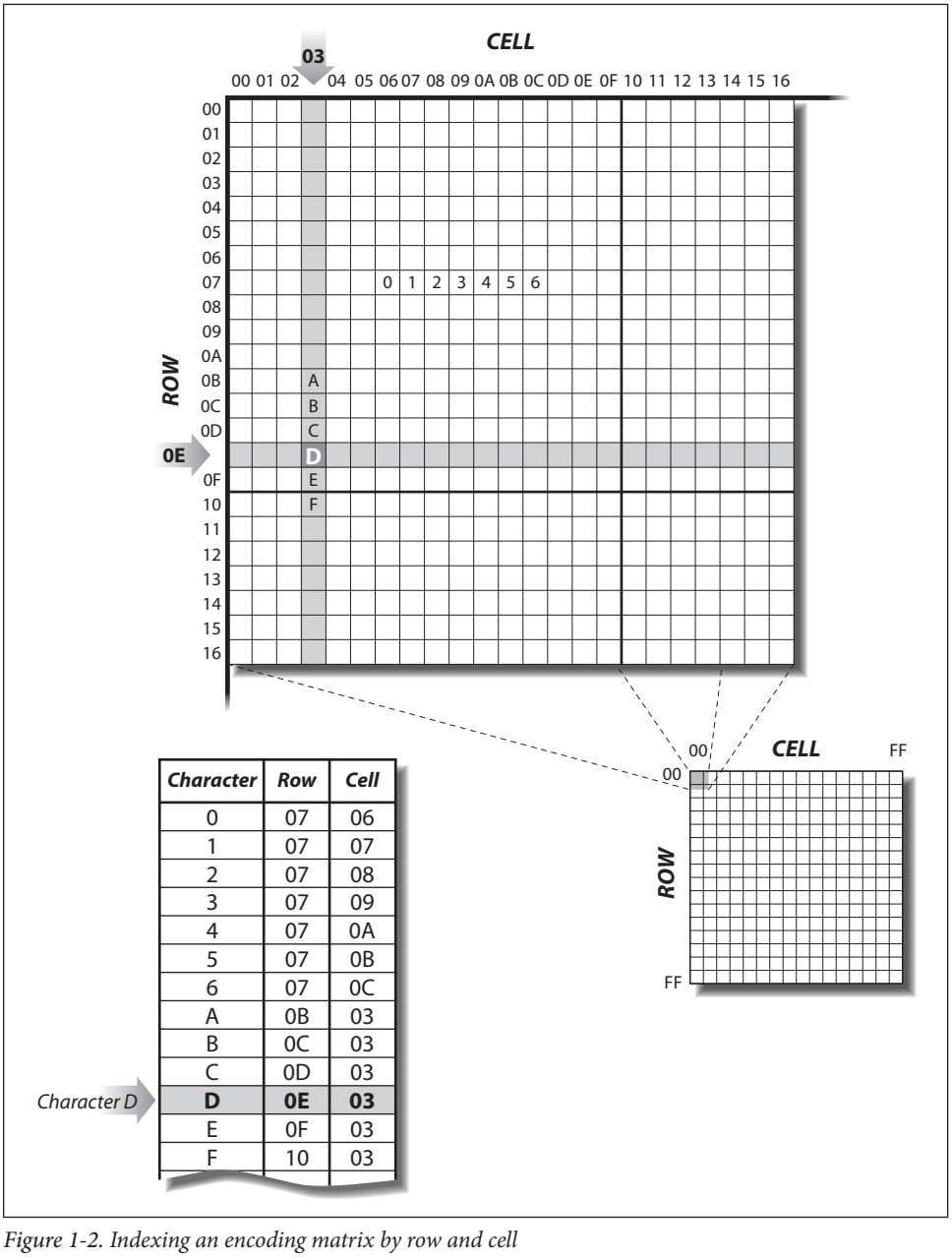
However, not all of these 65,536 cells can be used for representing displayable characters. To enable the mixture of one- and two-byte characters within a single text stream, some characters needed to be reserved as control characters, some of which then serve as the characters that signify when a text stream shifts between one- and two-byte modes. In the case of ISO-2022-JP encoding, the upper limit of displayable characters was set at 8,836, which is the size of the code space made from a 94×94 matrix.*

But why do you need to mix one- and two-byte characters anyway? It is to support existing one-byte encoding standards, such as ASCII, within a two-byte (or sometimes larger) encoding system. One-byte encoding methods are here to stay, and it is still a rather efficient means to encode the characters necessary to write English and many other languages. The most common encoding method for web pages, a mixed one- through four-byte encoding form called UTF-8, includes ASCII as its one-byte portion. However, languages with large character sets—those spoken in the CJKV locales—require two or more bytes to encode characters. Some encoding methods treat all characters, including ASCII, the same, meaning that they consume or require the same amount of encoding space. UTF-16 and UTF-32 use 16- and 32-bit code units, meaning that ASCII "A" (0x41) is represented by 16 or 32 bits: <0041> or <00000041>.

Along with discussions about character sets and encodings, you will encounter the terms "row" and "cell" again and again throughout this book. These refer to the axes of a matrix

* Code space refers to the area within the (usual) 256×256 encoding matrix that is used for encoding characters. Most of the figures in Chapter 4 and Appendix F illustrate code spaces that fall within this 256×256 matrix.

used to hold and encode characters. A matrix is made up of rows, and rows are made up of cells. The first byte specifies the row, and the second specifies the cell of the row. Figure 1-2 illustrates a matrix and how characters' positions correspond to row and cell values.



Character	Row	Cell
0	07	06
1	07	07
2	07	08
3	07	09
4	07	0A
5	07	0B
6	07	0C
A	0B	03
B	0C	03
C	0D	03
D	**0E**	**03**
E	0F	03
F	10	03

Figure 1-2. Indexing an encoding matrix by row and cell

In an attempt to allow for a mixture of one- and two-byte characters, several CJKV encoding methods have been developed. As you will learn in Chapter 4, these encoding methods are largely, but not completely, compatible. You will also see that there are encoding methods that use three or even four bytes to represent a single character!

The most common Japanese encoding methods are ISO-2022-JP, Shift-JIS, and EUC-JP. ISO-2022-JP, the most basic, uses seven-bit bytes (or, seven bits of a byte) to represent characters, and requires special characters or sequences of characters (called *shifting characters* or *escape sequences*) to shift between one- and two-byte modes. Shift-JIS and EUC-JP encodings make generous use of eight-bit characters, and use the value of the first byte as the way to distinguish one- and multiple-byte characters. Now, Unicode has become the most common encoding for Japanese.

Input Methods

Those who type English text have the luxury of using keyboards that can hold all the keys to represent a sufficient number of characters. CJKV characters number in the thousands, though, so how does one type CJKV text? Large keyboards that hold thousands of individual keys exist, but they require special training and are difficult to use. This has led to software solutions: *input methods* and the *conversion dictionaries* that they employ.

Most Chinese and Japanese text is typically input in two stages:

1. The user types raw keyboard input, which the computer interprets by using the input method and the conversion dictionary to display a list of *candidate* characters (the word "candidate" here refers to the character or characters that are mapped to the input string in the conversion dictionary).
2. The user selects one choice from the list of candidate characters, or requests more choices.

How well each stage of input is handled on your computer depends greatly on the quality (and vintage) of the input software you are using. Typical Korean and Vietnamese input is much more direct, and it varies by the keyboard layout that is used.

Software called an *input method* handles both of these input stages. It is so named because it grabs the user's keyboard input before any other software can use it (specifically, it is the first software to process keyboard input).

The first stage of input requires keyboard input, and can take one of two usual forms:

- Transliteration using Latin characters (type "k" plus "a" to get か, and so on), which is common for Chinese and Japanese, but relatively rare for Korean.
- Native-script input—zhuyin for Chinese as used in Taiwan, hiragana for Japanese, hangul for Korean, and so on.

The form used depends on user preference and the type of keyboard in use. For Japanese, the input method converts transliterated Japanese into hiragana on-the-fly, so it doesn't

really matter which keyboard you are using. In fact, studies show that over 70% of Japanese computer users prefer transliterated Japanese input.

Once the input string is complete, it is then parsed in one of two ways: either by the user during input or by a parser built into the input method. Finally, each segment is run through a conversion process that consists of a lookup into a conversion dictionary. This is very much like a *key-value* lookup. Typical conversion dictionaries have tens of thousands of entries. It seems that the more entries, the better the conversion quality. However, if the conversion dictionary is too large, users are shown a far too lengthy list of candidates. This reduces input efficiency.

Can ideographs be input one at a time? While single-ideograph input is possible, there are three basic units that can be used. These units allow you to limit the number of candidates from which you must choose. Typically, the larger the input unit, the fewer candidates. The units are as follows:

- Single ideograph
- Ideograph compound
- Ideograph phrase

Early input programs required that each ideograph be input individually, as single ideographs. Nowadays it is much more efficient to input ideographs as they appear in compounds or even phrases. This means that you may input two or more ideographs at once by virtue of inputting their combined reading. For example, the ideograph compound 漢字 (the two ideographs for writing the word meaning "ideograph") can be input as two separate characters, 漢 (pronounced *kan* in Japanese) and 字 (pronounced *ji* in Japanese). Table 1-6 shows the two target ideographs, along with other ideographs that share the same reading.

Table 1-6. Single ideograph input—Japanese

Character	Reading	Ideographs with identical readings
漢	KAN	乾侃冠寒刊勘勧巻喚堪姦完官寛干幹患感慣憾換敢柑桓棺款歓汗漢澗潅環甘監看竿管簡緩缶翰肝艦莞観諫貫鑑鑑間閑関陥韓館舘
字	JI	事似侍児字寺慈持時次滋治爾璽痔磁示而耳自蒔辞

You can see that there are many other ideographs with the same readings, so you may have to wade through a long list of candidate ideographs before you find the correct one. A more efficient way is to input them as one unit, called an ideograph compound. This produces a much shorter list of candidates from which to choose. Table 1-7 illustrates the two ideographs input as a compound, along with candidate compounds with the same reading.

Table 1-7. Ideograph compound input—Japanese

Compound	Reading	Compounds with identical readings
漢字	KANJI	漢字 感じ 幹事 監事 完司

Note how the list of ideograph compounds is much shorter in this case. There is an even higher-level input unit called an *ideograph phrase*. This is similar to inputting two or more ideographs as a single compound, but it adds another element, similar to a preposition in English, that makes the whole string into a phrase. An example of an ideograph phrase is 漢字は, which means "the ideograph" in Japanese. Because Chinese-language text is composed solely of hanzi, ideograph phrases apply only to Japanese, and possibly Korean.

Some of you may know of input software that claims to let you convert whole sentences at once. This is not really true. Such software allows you to input whole sentences, but the sentence is then parsed into smaller units, usually ideograph phrases, and then converted. Inputting whole sentences before any conversion is merely a convenience for the user.

Korean input has some special characteristics that are related to how their most widely used script, hangul syllables, is composed. Whether input is by a QWERTY or a Korean keyboard array, Korean input involves entering hangul elements called *jamo*. As the jamo are input, the operating system or input software attempts to compose hangul syllables using an automaton. Because of how hangul syllables are composed of jamo, the user may have up to three alternatives for deleting characters:

- Delete entire hangul syllable
- Delete by jamo
- Delete by word

This particular option is specific to Korean, and depends on whether the input method supports it.

Typography

Typography is a broad topic, and covers the way in which characters are set together, or composed, to form words, lines, and pages. What makes CJKV text different from Western text is the fact that it can usually be written or set in one of two orientations, described as follows:

- Left to right, top to bottom—horizontal setting, as in this book
- Top to bottom, right to left—vertical setting

Chapter 7 provides plenty of examples of horizontal versus vertical writing. More often than not, vertical writing orientation causes problems with Western-language applications. This is because vertical writing does not come for free, and some nontrivial amount of effort is required to implement this support in applications. Luckily, it is generally

acceptable to set CJKV text in the same horizontal orientation as most Western languages. Traditional novels and short stories are often set vertically, but technical materials, such as science textbooks and the like, are set horizontally.

Vertically set CJKV text is not a simple matter of changing writing direction. Some characters require special handling, such as a different orientation (90° clockwise rotation) or a different position within the *em-square*.* Chapter 7 provides some sample text set both horizontally and vertically, and it illustrates the characters that require special treatment.

In addition to the two writing directions for CJKV text, there are other special layout or composition considerations, such as special rules for line breaking, special types of justification, metrics adjustments, methods for annotating characters, and so on.

Basic Concepts and Terminology FAQ

Now I'll define some basic concepts which will help carry you through this entire book. These concepts are posed as questions because they represent questions you might raise as you read this book. If at any time you encounter a new term, please glance at the glossary toward the back of the book: new terms are included and explained there.

What Are All These Abbreviations and Acronyms?

Most technical fields are flooded with abbreviations and acronyms, and CJKV information processing is no exception. Some of the more important, and perhaps confusing, ones are explained in the following section, but when in doubt, consult Appendix D.

What is the difference between GB and GB/T? What about GBK?

Most references to "GB" refer to the GB 2312-80 character set standard, which represents the most widely implemented character set for Chinese. Now, of course, a much larger character set, designated GB 18030-2005, is considered the standard character set.

GB stands for *Guo Biao* (国标 *guóbiāo*), which is short for *Guojia Biaozhun* (国家标准 *guójiā biāozhǔn*), and simply means "National Standard."

Because GB/T character set standards are traditional analogs of existing GB character set standards, some naturally think that the "T" stands for *Traditional*. This represents yet another myth to expose as untrue. The "T" in "GB/T" actually stands for *Tui* (推 *tuī*), which is short for *Tuijian* (推荐 *tuījiàn*) and means "recommended" in the sense that it is the opposite of "forced" or "mandatory."

The "K" in GBK (an extension to GB 2312-80) comes from the Chinese word 扩展 (*kuòzhǎn*), which means "extension." As you will learn in Chapter 3, while GBK is an

* The term *em-square* refers to a square-shaped space whose height and width roughly correspond to the width of the letter "M." The term *design space* is actually a more accurate way to represent this typographic concept.

extension to GB 2312-80, and thus appropriately named, GBK itself was extended to become GB 18030-2005.

What are JIS, JISC, and JSA? How are they related?

In much of the literature in the field of Japanese information processing, you will quite often see references to JISC, JIS, and JSA. The most common of these is JIS; the least is JISC. What these refer to can sometimes be confusing and is often contradicted in reference works.

JIS stands for *Japanese Industrial Standard* (日本工業規格 *nihon kōgyō kikaku*), the name given to the standards used in Japanese industry.* The character ㊐ is the original symbol for JIS, which was used through September 30, 2008. The character Ⓙ is the new symbol for JIS, which was issued on October 1, 2005, and its use became mandatory on October 1, 2008. JIS can refer to several things: the character set standards established by JISC, the encoding method specified in these character set standards, and even the keyboard arrays described in specific JIS standards. Context should usually make its meaning clear. Of course, JIS appears frequently in this book.

JISC stands for *Japanese Industrial Standards Committee* (日本工業標準調査会 *nihon kōgyō hyōjun chōsakai*).† This is the name of the governing body that establishes JIS standards and publishes manuals through JSA. The committee that develops and writes each JIS manual is composed of people from Japanese industry who have a deep technical background in the topic to be covered by the manual. Committee members are listed at the end of each JIS manual.

JSA stands for *Japanese Standards Association* (日本規格協会 *nihon kikaku kyōkai*).‡ This organization publishes the manuals for the JIS standards established by JISC, and generally oversees the whole process.

JIS is often used as a blanket term covering JIS, JISC, and JSA, but now you know what they *genuinely* mean.

Several JIS "C" series standards changed designation to "X" series standards on March 1, 1987. Table 1-8 lists the JIS standards—mentioned in this book—that changed designation from "C" to "X" series.

* There is even a JIS standard for manufacturing toilet paper! It is designated JIS P 4501:2006 and is entitled トイレットペーパー (*toiretto pēpā*). Its English title is *Toilet tissue papers*. The "P" series JIS standards are for the pulp and paper industries.

† *http://www.jisc.go.jp/*

‡ *http://www.jsa.or.jp/*

Table 1-8. JIS standard designation changes

JIS "C" series	JIS "X" series
JIS C 6220	JIS X 0201
JIS C 6228	JIS X 0202
JIS C 6225	JIS X 0207
JIS C 6226	JIS X 0208
JIS C 6233	JIS X 6002
JIS C 6235	JIS X 6003
JIS C 6236	JIS X 6004
JIS C 6232	JIS X 9051
JIS C 6234	JIS X 9052

Because these changes took place well over two decades ago, they have long been reflected in software and other documentation.

What is KS?

KS simply stands for *Korean Standard* (한국 공업 규격/韓國工業規格 *hanguk gongeop gyugyeok*). All Korean character set standard designations begin with the two uppercase letters "KS." The character ㉿ is the symbol for KS.

All KS standards also include another letter in their designation. Those that are discussed in this book all include the letter "X," which now indicates electric or electronic standards.*

Several KS "C" series standards changed designation to "X" series standards on August 20, 1997. Table 1-9 lists the KS standards—mentioned in this book—that changed designation from the "C" to "X" series.

Table 1-9. KS standard designation changes

KS "C" series	KS "X" series
KS C 5601	KS X 1001
KS C 5657	KS X 1002
KS C 5636	KS X 1003
KS C 5620	KS X 1004
KS C 5700	KS X 1005

* Other letter designations for KS standards include "B" (mechanical), "D" (metallurgy), and "A" (general guidelines).

Table 1-9. KS standard designation changes

KS "C" series	KS "X" series
KS C 5861	KS X 2901
KS C 5715	KS X 5002

It took several years until these new KS standard designations were consistently reflected in software and documentation. Especially for KS X 1001, its original designation, KS C 5601, is still seen often.

Are VISCII and VSCII identical? What about TCVN?

Although both VISCII and VSCII are short for *Vietnamese Standard Code for Information Interchange*, they represent completely different character sets and encodings. VISCII is defined in RFC 1456,* and VSCII is derived from TCVN 5712:1993 (specifically, VN2), which is a Vietnamese national standard. VSCII is also known as ISO IR 180. The differences among VISCII and VSCII are described in Chapter 3. Appendix L provides complete encoding tables for VISCII and VSCII, which better illustrate their differences.

TCVN stands for *Tiêu Chuẩn Việt Nam*, which translates into English as "Vietnamese Standard." Like CNS, GB, JIS, and KS, it represents the first portion of Vietnamese standard designations.

What Are Internationalization, Globalization, and Localization?

Internationalization—often abbreviated as I18N, composed of the initial letter "I" followed by the middle eighteen (18) letters followed by the final letter "N"—is a term that usually refers to the process of preparing software so that it is ready to be used by more than one culture, region, or locale.† Internationalization is thus what engineers do.

Globalization, similarly abbreviated as G11N, is often used synonymously with internationalization, but encompasses the business aspects, such as entering a foreign market, conducting market research, studying the competition, strategizing, becoming aware of legal restrictions, and so on.‡ Globalization is thus what companies as a whole do.

Localization—often abbreviated as L10N under the same auspices as I18N and G11N—is the process of adapting software for a specific culture, region, or locale. *Japanization*—often abbreviated as J10N—is thus a locale-specific instance of L10N. While this book does not necessarily address all of these issues, you will find information pertinent to internationalization and localization within its pages.

* *http://www.ietf.org/rfc/rfc1456.txt*

† Quiz time. Guess what CJKV6N, C10N, K11N, M17N, S32S, and V12N stand for.

‡ But, globalization should not be confused with *global domination*, which is an entirely different matter.

Internationalization and localization are different processes. For example, it is possible to develop an application that properly handles the various scripts of the world, and is thus internationalized, but provides an English-language *user interface* (UI), and is thus not localized. The converse is also possible, specifically an application with a UI that has been translated into a large number of languages, and is thus localized, but fails to properly handle the various scripts of the world, and is thus not internationalized.

In any case, market demand forces or encourages developers to embrace I18N, G11N, and L10N, because doing so results in the functionality that their customers demand, or it provides menus and documentation written in the language of the target locale. They often require special handling because many non-Latin writing systems include a large number of characters, have complex rendering issues, or both.

What Are the Multilingual and Locale Models?

There have been two basic models for internationalization: the *locale model* and the *multilingual model*. The locale model was designed to implement a set of attributes for specific locales. The user must explicitly switch from one locale to another. The character sets implemented by the locale model were specific to a given culture or region, thus locale.

The multilingual model, on the other hand, was designed or expected to go one step further by not requiring the user to flip or switch between locales. Multilingual systems thus implement a character set that includes all the characters necessary for several cultures or regions. But still, there are cases when it is impossible to correctly render characters without knowing the target locale.

For the reasons just pointed out, a combination of these two models is ideal, which is precisely what has happened in the field of software internationalization. Unicode is the ideal character set because it is truly multilingual, and it effectively bridges and spans the world's writing systems. Of course, Unicode is not perfect. But then again, nothing made by man, by definition, can be perfect. Still, no other character set spans the world's writing systems as effectively and broadly as Unicode has done. This is why it is wise to embrace Unicode, and I encourage you to do so. Embracing Unicode also requires that its data be tagged with locale attributes so that the characters behave accordingly and appropriately for each culture or region. The *Common Locale Data Repository* (CLDR) is the most complete and robust source for locale data, and will be explored in Chapter 9.*

Thus, both of these models for internationalization have succeeded, not by competing, but rather by combining into a single solution that has proven to work well.

What Is a Locale?

Locale is a concept for specifying the language and country or region, and is significant in that it affects the way in which an OS, application, or other software behaves. A locale,

* *http://www.unicode.org/cldr/*

as used by software, typically consists of a language identifier and a country or region identifier.

What Is Unicode?

Unicode is the first truly successful multilingual character set standard, and it is supported by three primary encoding forms, UTF-8, UTF-16, and UTF-32. Unicode is also a major focus of this book.

Conceived 20 years ago by my friend Joe Becker, Unicode has become the preferred character set and has been successful in enabling a higher level of internationalization. In other words, Unicode has trivialized many aspects of software internationalization.

How Are Unicode and ISO 10646 Related?

Make no mistake, Unicode and ISO 10646 are different standards.* The development of Unicode is managed by *The Unicode Consortium*, and that of ISO 10646 is managed by the *International Organization for Standardization* (ISO). But, what is important is that they are equivalent, or rather kept equivalent, through a process that keeps them in sync.

ISO 10646 increases its character repertoire through new versions of the standard, additionally designated by year, along with amendments. Unicode, on the other hand, does the same through new versions. It is possible to correlate Unicode and ISO 10646 by indicating the version of the former, and the year and amendments of the latter.

More detailed coverage of Unicode can be found in Chapter 3, and details of the various Unicode encoding forms can be found in Chapter 4.

What Are Row-Cell and Plane-Row-Cell?

Row-Cell is the translated form of the Japanese word 区点 (*kuten*), which literally means "ward [and] point," or more intuitively as "row [and] cell."† This notation serves as an encoding-independent method for referring to characters in most CJKV character set standards. A Row-Cell code usually consists of four decimal digits. The "Row" portion consists of a zero-padded, two-digit number with a range from 01 to 94. Likewise, the "Cell" portion also consists of a zero-padded, two-digit number with a range from 01 to 94. For example, the first character in most CJKV character set standards is represented as 01-01 in Row-Cell notation, and is more often than not a full-width "space" character, which is typically referred to as an *ideographic space*.

Bear in mind that some character sets you will encounter include or span more than a single 94×94 matrix, each of which is referred to as a *plane*. CNS 11643-2007 and JIS X

* *http://unicode.org/faq/unicode_iso.html*

† In Chinese, Row-Cell is expressed as 区位 (*qūwèi*); in Korean, as 행렬/行列 (*haengnyeol*). Note that if the "Cell" portion of "Row-Cell" is in isolation in Korean, then it is expressed instead with the hangul 열 (*yeol*), not 렬 (*nyeol*).

0213:2004 are examples of legacy character set standards that include two or more planes. Obviously, Row-Cell notation must be expanded to include the notion of plane, meaning *Plane-Row-Cell*. In Japanese, this is expressed as 面区点 (*menkuten*) and is used as the preferred notation for referencing the characters in JIS X 0213:2004.

When I provide lists of characters throughout this book, I usually include Row-Cell (or Plane-Row-Cell) codes. These are useful for future reference of these data, and so that you don't have to hunt for the codes yourself. Now that Unicode plays an important role in today's software development efforts, I also provide Unicode scalar values when appropriate.

What Is a Unicode Scalar Value?

Like Row-Cell notation, Unicode scalar values serve as an encoding-independent method of referring to specific Unicode characters, or sequences of Unicode characters. It is a notation, but instead of using decimal values such as Row-Cell notation, hexadecimal values are used due to the larger 256×256 encoding space afforded by Unicode. Still, Unicode scalar values are not tied to a specific encoding, such as UTF-8, UTF-16, nor UTF-32.

The syntax for Unicode scalar values is simple and consists of a prefix followed by four to six hexadecimal digits. The prefix is "U+," and the length of the hexadecimal digits varies by whether the character is in the *Basic Multilingual Plane* (BMP) or in one of the 16 supplementary planes. I prefer to think of Unicode scalar values as equivalent to UTF-32 encoding when expressed in hexadecimal notation, but zero-padded to four digits. The ASCII "space" is thus represented as U+0020 (not U+20), and the last code point in Plane 16 is represented as U+10FFFF.

Unicode scalar values are incredibly useful. When their prefix is removed and replaced with the appropriate *Numeric Character Reference* (NCR) syntax, such as `&#x4E00;` for U+4E00, they become immediately usable in contexts that support HTML or XML.* They can also be used within angled brackets and separated with a comma to indicate standardized Unicode sequences, such as <U+528D, U+E0101>.

Unicode scalar values are used throughout this book and are provided for your convenience.

Characters Versus Glyphs: What Is the Difference?

Now here's a topic that is usually beaten to death! The term *character* is an abstract notion indicating a class of shapes declared to have the same meaning or abstract shape. The term *glyph* refers to a specific instance of a character.

* NCRs are SGML constructs that have carried through into SGML derivations, such as HTML and XML. I cannot guarantee that all HTML and XML implementations support NCRs, but the vast majority do.

Interestingly, more than one character can constitute a single glyph, such as the two characters *f* and *i*, which can be fused together as the single entity *fi*. This fi glyph is called a *ligature*. The dollar currency symbol is a good example of a character with several glyphs. There are at least four distinct glyphs for the dollar currency symbol, described and illustrated here:

- An "S" shape with a single vertical bar: $
- An "S" shape with a single broken vertical bar: $
- An "S" shape with two vertical bars: $
- An "S" shape with two broken vertical bars: $

The differences among these four glyphs are minor, but you cannot deny that they still represent the same character, specifically the dollar currency symbol. More often than not, you encounter a difference in this glyph as a difference in typeface.

However, there are some characters that have a dozen or more variant forms. Consider the kanji 辺 (*hen*, used in the common Japanese family name 渡辺 *watanabe*), which has only two variant forms that are included in JIS X 0208:1997, and are thus included in Unicode: 邉 (U+9089) and 邊 (U+908A). These are considered the traditional forms of the kanji 辺. Table 1-10 lists the additional variant forms that are included in the Adobe-Japan1-6 character collection, which will be covered in Chapter 6.

Table 1-10. Standard versus variant forms

Standard Form	Variant forms	Additional variant forms
辺	邉 邊	邉邉邉邉邉邉邉邉邉邉邉邉邉邉 邊邊邊邊邊邊邊

Clearly, these variant forms all appear to represent that same character, but are simply different glyphs.

You will discover that CJKV character set standards do not define the glyphs for the characters contained within their specifications. Unfortunately (or, fortunately, as the case may be), many think that the glyphs that appear in these manuals are the official ones, and to some extent they become the default, or *prototypical*, glyphs for the characters.

Note, however, that the official Jōyō Kanji Table *does* define the glyph shape, at least for the 1,945 kanji contained within its specification. Interestingly, JSA at one time published two standards that did, in fact, define glyphs for characters by virtue of specifying precise bitmap patterns for every character: JIS X 9051-1984* and JIS X 9052-1983.† The glyphs

* Previously designated JIS C 6232-1984

† Previously designated JIS C 6234-1983

set forth in these two standards were designed for the JIS X 0208-1983 standard, which was current at the time. However, these glyphs have not been widely accepted in industry, mainly due to the introduction of outline fonts. It seems as though JSA has no intention of ever revising these documents, and some speculate that this may be their way of not enforcing glyphs.

The one Japanese organization that established a definitive Japanese glyph standard in Japan is the now-defunct FDPC, which is an abbreviation for *Font Development and Promotion Center* (文字フォント開発・普及センター *moji fonto kaihatsu fukyū sentā*). FDPC was a MITI (*Ministry of International Trade and Industry*—通商産業省 *tsūshō sangyō shō*)-funded organization, and has since been folded in with JSA. This government organization, with the help of members, developed a series of Japanese outline fonts called Heisei (平成 heisei) typefaces. The first two Heisei typefaces that were released were *Heisei Mincho W3* (平成明朝 W3 *heisei minchō W3*) and *Heisei Kaku (squared) Gothic W5* (平成角ゴシック W5 *heisei kaku goshikku W5*). In fact, the standard Japanese typeface used in the production of the first edition of this book was Heisei Mincho W3. A total of seven weights of both designs were produced, ranging from 3 (W3) to 9 (W9). Two weights of *Heisei Maru (rounded) Gothic* (平成丸ゴシック *heisei maru goshikku*), specifically 4 and 8, also were developed. The Heisei typefaces have become somewhat commonplace in the Japanese market.

Stability versus correctness

I have learned that changes to prototypical glyphs are inevitable and unavoidable. There are two forces or notions at work. One is *stability*, and the other is the notion of *correctness*. Unfortunately, the notion of correctness can—and ultimately will—change over time. Languages and their writing systems change, which is part of their nature. In Japan, the first series of prototypical glyph changes took place in 1983, when JIS X 0208-1983 was published. Bear in mind that the first version of the standard was published in 1978. Somewhat subtle prototypical glyph changes took place when it was revised in 1990 and designated JIS X 0208-1990. Stability ruled the day when the 1997 revision, designated JIS X 0208:1997, was published, along with its extension in 2000, designated JIS X 0213:2000. A new set of 1,022 kanji, called *NLC Kanji* (表外漢字 *hyōgai kanji*), introduced a new notion of correctness, and directly influenced prototypical glyph changes that were introduced in 2004, as a revision to the existing standard, designated JIS X 0213:2004. I have found it interesting that some of these prototypical glyph changes have caused the glyphs for some characters to come full circle, meaning that they reverted to their original forms as found in the 1978 version of the standard. Table 1-11 provides but one example of a JIS X 0208:1997 kanji that came full circle—specifically 36-52. Its Unicode code point, regardless of glyph, is U+8FBB.

Table 1-11. Prototypical glyph changes over time—Japan

Code point	1978	1983	1990	1997	2000	2004
36-52	辻	辻	辻	辻	辻	辻

China takes glyph issues very seriously and expended the effort to develop a series of standards, published in a single manual entitled *32×32 Dot Matrix Font Set and Data Set of Chinese Ideograms for Information Interchange* (信息交换用汉字 32×32 点阵字模集及数据集 *xìnxī jiāohuàn yòng hànzì 32×32 diǎnzhèn zìmújí jí shújùjí*). This explicitly defined glyphs for the GB 2312-80 character set standard in various typeface styles. These standards are listed in Table 1-12.

Table 1-12. Chinese glyph standards

Standard	Pages	Title (in English)
GB 6345.1-86	1–27	*32×32 Dot Matrix Font Set of Chinese Ideograms for Information Interchange*
GB 6345.2-86	28–31	*32×32 Dot Matrix Font Data Set of Chinese Ideograms for Information Interchange*
GB 12034-89	32–55	*32×32 Dot Matrix Fangsongti Font Set and Data Set of Chinese Ideograms for Information Interchange*
GB 12035-89	56–79	*32×32 Dot Matrix Kaiti Font Set and Data Set of Chinese Ideograms for Information Interchange*
GB 12036-89	80–103	*32×32 Dot Matrix Heiti Font Set and Data Set of Chinese Ideograms for Information Interchange*

Song (specified in GB 6345.1-86), Fangsong, Kai, and Hei are the most common typeface styles used in Chinese. When the number of available pixels is reduced, it is impossible to completely represent all of an ideograph's strokes. These standards are useful because they establish bitmapped patterns that offer a compromise between accuracy and legibility. The GB 16794.1-1997 standard (信息技术一通用多八位编码字符集 48 点阵字形 *xìnxī jìshù—tōngyòng duōbāwèi biānmǎ zìfújí 48 diǎnzhèn zìxíng*) is similar to the GB standards listed in Table 1-12, but covers the complete GBK character set and provides 48×48 bitmapped patterns for every character. An older set of GB standards, GB 5007.1-85 (信息交换用汉字 24×24 点阵字模集 *xìnxī jiāohuàn yòng hànzì 24×24 diǎnzhèn zìmújí*) and GB 5007.2-85 (信息交换用汉字 24×24 点阵字模数据集 *xìnxī jiāohuàn yòng hànzì 24×24 diǎnzhèn zìmú shújùjí*), provided 24×24 bitmapped patterns for a single design, and obviously covered GB 2312-80, not GBK, given that they were published in 1985, long before GBK was developed.

Exactly how the terms *character* and *glyph* are defined can differ depending on the source. Table 1-13 provides the ISO and The Unicode Consortium definitions for the terms *abstract character*, *character*, *glyph*, *glyph image*, and *grapheme*.

Table 1-13. Abstract character, character, glyph, glyph image, and grapheme definitions

Terminology	ISO	Unicode[a]
Abstract character	n/a	A unit of information used for the organization, control, or representation of textual data. (See definition D7 in Section 3.4, Characters and Encoding.)
Character	A member of a set of elements used for the organization, control, or representation of data.[b] An atom of information with an individual meaning, defined by a character repertoire.[c]	(1) The smallest component of written language that has semantic value; refers to the abstract meaning and/or shape, rather than a specific shape (see also *glyph*), though in code tables some form of visual representation is essential for the reader's understanding. (2) Synonym for abstract character. (3) The basic unit of encoding for the Unicode character encoding. (4) The English name for the ideographic written elements of Chinese origin. [See *ideograph* (2).]
Glyph	A recognizable abstract graphical symbol which is independent of any specific design.[c]	(1) An abstract form that represents one or more glyph images. (2) A synonym for *glyph image*. In displaying Unicode character data, one or more glyphs may be selected to depict a particular character. These glyphs are selected by a rendering engine during composition and layout processing. (See also *character*.)
Glyph image	An image of a glyph, as obtained from a glyph representation displayed on a presentation surface.[c]	The actual, concrete image of a glyph representation having been rasterized or otherwise imaged onto some display surface.
Grapheme	n/a	(1) A minimally distinctive unit of writing in the context of a particular writing system. For example, ‹b› and ‹d› are distinct graphemes in English writing systems because there exist distinct words like *big* and *dig*. Conversely, a lowercase italiform letter *a* and a lowercase Roman letter a are not distinct graphemes because no word is distinguished on the basis of these two different forms. (2) What a user thinks of as a character.

a. *The Unicode Standard, Version 5.0* (Addison-Wesley, 2006)

b. ISO 10646:2003

c. ISO 9541-1:1991

As usual, the standards—in this case ISO 10646 and Unicode—provide clear and precise definitions for otherwise complex and controversial terms. Throughout this book, I use the terms *character* and *glyph* very carefully.

What Is the Difference Between Typeface and Font?

The term *typeface* refers to the design characteristics of a collection of glyphs, and is often comprised of multiple fonts. Thus, a *font* refers to a single instance of a typeface, such as a specific style, relative weight, relative width, or other design attributes, and in the case of bitmapped fonts, point size. This is why the commonly used term *outline font* is somewhat of a misnomer—the mathematical outlines are, by definition, scalable, which means that they are not specific to a point size. A better term is *outline font instance*. But, I digress.

Western typography commonly uses serif, sans serif, and script typeface styles. Table 1-14 lists the common CJKV typeface styles, along with correspondences across locales.

Table 1-14. Western versus CJKV typeface styles

Western	Chinese[a]	Japanese	Korean
Serif[b]	Ming (明體 *míngtǐ*) Song (宋体 *sòngtǐ*)	Mincho (明朝体 *minchōtai*)	Batang (바탕 *batang*)[c]
Sans serif	Hei (黑体 *hēitǐ*)	Gothic (ゴシック体 *goshikkutai*)	Dotum (돋움 *dotum*)[d]
Script	Kai (楷体 *kǎitǐ*)	Kaisho (楷書体 *kaishotai*) Gyosho (行書体 *gyōshotai*) Sosho (草書体 *sōshotai*)	Haeseo (해서 체/楷書體 *haeseoche*) Haengseo (행서 체/行書體 *haengseoche*) Choseo (초서 체/草書體 *choseoche*)
Other	Fangsong (仿宋体 *fǎngsòngtǐ*)	Kyokasho (教科書体 *kyōkashotai*)	

a. Replace 体 with 體 in these typeface style names for Traditional Chinese.

b. The convention has been that Ming is used for Traditional Chinese, and Song is used for Simplified Chinese.

c. In the mid-1990s, the Korean Ministry of Culture specified this term, replacing *Myeongjo* (명조체/明朝體 *myeongjoche*).

d. In the mid-1990s, the Korean Ministry of Culture specified this term, replacing *Gothic* (고딕 체/고딕體 *godikche*).

Table 1-14 by no means constitutes a complete list of CJKV typeface styles—there are numerous typeface styles for hangul, for example. To provide a sample of typeface variation within a locale, consider the four basic typeface styles used for Chinese, as illustrated in Table 1-15.

Table 1-15. Chinese typeface styles—examples

Song	Hei	Kai	Fangsong
中文简体字	**中文简体字**	中文简体字	中文简体字

Clearly, the glyphs shown in Table 1-15 represent the same characters, in that they convey identical meanings, and differ only in that their glyphs are different by virtue of having different typeface designs.

What Are Half- and Full-Width Characters?

The terms *half-* and *full-width* refer to the relative glyph size of characters. These are referred to as *hankaku* (半角 *hankaku*) and *zenkaku* (全角 *zenkaku*), respectively, in

Japanese.* Half-width is relative to full-width. Full-width refers to the glyph size of standard CJKV characters, such as zhuyin, kana, hangul syllables, and ideographs. Latin characters, which appear to take up approximately half the display width of CJKV characters, are considered to be half-width by this standard. The very first Japanese characters to be processed on computer systems were half-width katakana. They have the same approximate display width as Latin characters. There are now full-width Latin and katakana characters. Table 1-16 shows the difference in display width between half- and full-width characters (the katakana character used as the example is pronounced *ka*).

Table 1-16. Half- and full-width characters

Width	Katakana	Latin
Half	ｶｶｶｶｶ	12345
Full	カカカカカ	１２３４５

As you can see, full-width characters occupy twice the display width as their half-width versions. At one point in time there was a clear-cut relationship between the display width of a glyph and the number of bytes used to encode it (the *encoding length*)—the number of bytes simply determined the display width. Half-width katakana characters were originally encoded with one byte. Full-width characters were encoded with two bytes. Now that there is a much richer choice of encoding methods available, this relationship no longer holds true. Table 1-17 lists several popular encoding methods, along with the number of bytes required to represent half- and full-width characters.

Table 1-17. Half- and full-width character representations—Japanese

Width	Script	ASCII	ISO-2022-JP	Shift-JIS	EUC-JP	UTF-8	UTF-16
Full	Katakana	n/a	2 bytes	2 bytes	2 bytes	3 bytes	16 bits
	Latin	n/a	2 bytes	2 bytes	2 bytes	3 bytes	16 bits
Half	Katakana	n/a	1 byte	1 byte	2 bytes	3 bytes	16 bits
	Latin	1 byte	1 byte	1 byte	1 byte	1 byte	16 bits

I should also point out that in some circumstances, half-width may also mean not full-width, and can refer to characters that are intended to be proportional.

* In Chinese, specifically in Taiwan, these terms are 半形 (*bànxíng*) and 全形 (*quánxíng*), respectively. But, in China, they are the same as those used for Japanese, specifically 半角 (*bànjiǎo*) and 全角 (*quánjiǎo*), respectively. In Korean, these terms are 반각/半角 (*bangak*) and 전각/全角 (*jeongak*), respectively.

Latin Versus Roman Characters

To this day, many people debate whether the 26 letters of the English alphabet should be referred to as *Roman* or *Latin* characters. While some standards, such as those published by ISO, prefer the term Latin, other standards prefer the term Roman. In this book, I prefer to use Latin over Roman, and use it consistently, but readers are advised that they should treat both terms synonymously.

When speaking of typeface designs, the use of the term Roman is used in contrast with the term *italic*, and refers to the upright or nonitalic letterforms.

What Is a Diacritic Mark?

A diacritic mark is an attachment to a character that typically serves as an accent, or indicates tone or other linguistic information. Diacritic marks are important in the scope of this book, because many transliteration and Romanization systems make use of them. Japanese long vowels, for example, are indicated through the use of the diacritic mark called a *macron*. Vietnamese makes extensive use of multiple diacritic marks, meaning that some characters can include more than one diacritic mark. Many non-Latin scripts, such as Japanese hiragana and katakana, also use diacritic marks. In the case of Japanese kana, one such diacritic mark serves to indicate the voicing of consonants.

What Is Notation?

The term notation refers to a method of representing units. A given distance, whether expressed in miles or kilometers, is, after all, the same physical distance. In computer science, common notations for representing the value of bit arrays, bytes, and larger units are listed in Table 1-18, and all correspond to a different numeric base.

Table 1-18. Decimal 100 in common notations

Notation	Base	Value range	Example
Binary	2	0 and 1	01100100
Octal	8	0–7	144
Decimal	10	0–9	100
Hexadecimal	16	0–9 and A–F	64

While the numbers in the Example column all have the same underlying value, specifically 100 (in decimal), they have been expressed using different notations, and thus take on a different form. Most people—that is, non-nerds—think in decimal notation, because that is what we were taught. However, computers—and some nerds—process information

using binary notation.* As discussed previously, computers process bits, which have two possible values. In the next section, you will learn that hexadecimal notation does, however, have distinct advantages when dealing with computers.

What Is an Octet?

We have already discussed the terms *bits* and *bytes*. But what about the term *octet*? At a glance, you can tell it has something to do with the number eight. An octet represents eight bits, and is thus an eight-bit byte, as opposed to a seven-bit one. This becomes confusing when dealing with 16-bit encodings. Sixteen bits can be broken down into two eight-bit bytes, or two octets. Thirty-two bits, likewise, can be broken down into four eight-bit bytes, or four octets.

Given 16 bits in a row:

```
0110010001011111
```

this string of bits can be broken down into two eight-bit units, specifically octets (bytes):

```
01100100
01011111
```

The first eight-bit unit represents decimal 100 (0x64), and the second eight-bit unit represents decimal 95 (0x5F). All 16 bits together as a single unit are usually equal to 25,695 in decimal, or <645F> in hexadecimal (it may be different depending on a computer's specific architecture). Divide 25,695 by 256 to get the first byte's value as a decimal octet, which results in 100 in this case; the remainder from this division is the value of the second byte, which, in this case, is 95. Table 1-19 lists representations of two octets (bytes), along with their 16-bit unit equivalent. This is done for you in the four different notations.

Table 1-19. Octets and 16-bit units in various notations

Notation	First octet	Second octet	16-bit unit
Binary	01100100	01011111	01100100 01011111
Octal	144	137	62137
Decimal	100	95	25,695
Hexadecimal	64	5F	64 5F

Note how going from two octets to a 16-bit unit is a simple matter of concatenation in the case of binary and hexadecimal notation. This is not true with decimal notation, which requires multiplication of the first octet by 256, followed by the addition of the second octet. Thus, the ease of converting between different representations—octets versus 16-bit units—depends on the notation that you are using. Of course, string concatenation

* Now that I think about it, the *Bynars*, featured in the *Star Trek: The Next Generation* episode entitled "11001001," represent an entire race that processes information in binary form.

is easier than two mathematical operations. This is precisely why hexadecimal notation is used very frequently in computer science and software development.

In some cases, the order in which byte concatenation takes place matters, such as when the byte order (also known as *endianness*) differs depending on the underlying computing architecture. Guess what the next section is about?

What Are Little- and Big-Endian?

There are two basic computer architectures when it comes to the issue of byte order: little-endian and big-endian. That is, the order in which the bytes of larger-than-byte storage units—such as integers, floats, doubles, and so on—appear.* One-byte storage units, such as *char* in C/C++, do not need this special treatment. That is, unless your particular machine or implementation represents them with more than one byte. The following is a synopsis of little- and big-endian architectures:

- Little-endian machines use computing architectures supported by Vax and Intel processors. Historically, MS-DOS and Windows machines are little-endian.
- Big-endian machines use computing architectures supported by Motorola and Sun processors. Historically, Mac OS and most Unix workstations are big-endian. Big-endian is also known as *network byte order*.

Linux, along with Apple's rather recent switch to Intel processors, has blurred this distinction, to the point that platform or OS are no longer clear indicators of whether little- or big-endian byte order is used. In fact, until the 1970s, virtually all processors used big-endian byte order. The introduction of microprocessors with their (initially) simple logic circuits and use of byte-level computations led to the little-endian approach. So, from the viewpoint of history, the mainframe versus microprocessor distinction gave birth to byte order differences. I should also point out that runtime detection of byte order is much more robust and reliable than guessing based on OS.

Table 1-20 provides an example two-byte value as encoded on little- and big-endian machines.

Table 1-20. Little- and big-endian representation

Notation	High byte	Low byte	Little-endian	Big-endian
Binary	01100100	01011111	01011111 01100100	01100100 01011111
Hexadecimal	64	5F	5F 64	64 5F

A four-byte example, such as 0x64, 0x5F, 0x7E, and 0xA1, becomes <A1 7E 5F 64> on little-endian machines, and <64 5F 7E A1> on big-endian machines. Note how the bytes

* A derivation of little- and big-endian came from *Gulliver's Travels*, in which there were civil wars fought over which end of a boiled egg to crack.

themselves—not the underlying bits of each byte—are reversed depending on endianness. This is precisely why endianness is also referred to as byte order. The term *endian* is used to describe what impact the byte at the end has on the overall value. The UTF-16 value for the ASCII "space" character, U+0020, is <00 20> for big-endian machines and <20 00> for little-endian ones.

Now that you understand the concept of endianness, the real question that needs answering is when endianness matters. Please keep reading….

What Are Multiple-Byte and Wide Characters?

If you have ever read comprehensive books and materials about ANSI C, you more than likely came across the terms multiple-byte and wide characters. Those documents typically don't do those terms justice, in that they are not fully explained. Here you'll get a definitive answer.

When dealing with encoding methods that are processed on a per-byte basis, endianness or byte order is irrelevant. The bytes that compose each character have only one order, regardless of the underlying architecture. These encoding methods support what are known as multiple-byte characters. In other words, these encoding methods use the byte as their code unit.

So, what encoding methods support multiple-byte characters? Table 1-21 provides an incomplete yet informative list of encoding methods that support multiple-byte characters. Some encoding methods are tied to a specific locale, and some are tied to CJKV locales in general.

Table 1-21. Multiple-byte characters—encoding methods

Encoding	Encoding length	Locale
ASCII	One-byte	n/a
ISO-2022	One- and two-byte	CJKV
EUC	One- through four-byte, depending on locale	CJKV
GBK	One- and two-byte	China
GB 18030	One-, two-, and four-byte	China
Big Five	One- and two-byte	Taiwan and Hong Kong
Shift-JIS	One- and two-byte	Japan
Johab	One- and two-byte	Korea
UHC	One- and two-byte	Korea
UTF-8	One- through four-byte	n/a

There are some encodings that require special byte order treatment, and thus cannot be treated on a per-byte basis. These encodings use what are known as wide characters, and

almost always provide a facility for indicating the byte order. Table 1-22 lists some encoding methods that make use of wide characters, all of which are encoding methods for Unicode.

Table 1-22. Wide characters—encoding methods

Encoding	Encoding length
UCS-2	16-bit fixed
UTF-16	16-bit variable-length
UTF-32	32-bit fixed

Sometimes the encodings listed in Table 1-22 are recommended to use the *Byte Order Mark* (BOM) at the beginning of a file to explicitly indicate the byte order. The BOM is covered in greater detail in Chapter 4.

It is with endianness or byte order that we can more easily distinguish multiple-byte from wide characters. Multiple-byte characters have the same byte order, regardless of the underlying processor architecture. The byte order of wide characters is determined by the underlying processor architecture and must be flagged or indicated in the data itself.

Advice to Readers

This chapter serves as an introduction to the rest of this book, and is meant to whet your appetite for what lies ahead in the pages that follow. When reading the chapters of this book, I suggest that you focus on the sections that cover or relate to Unicode, because they are likely to be of immediate value and benefit. Information about legacy character sets and encodings is still of great value because it relates to Unicode, often directly, and also serves to chronicle how we got to where we are today.

In any case, I hope that you enjoy reading this book as much as I enjoyed writing the words that are now captured within its pages.

CHAPTER 2

Writing Systems and Scripts

Reading the introductory chapter provided you with a taste of what you can expect to learn about CJKV information processing in this book. Let's begin the journey with a thorough description of the various CJKV writing systems that serve as the basis for the characters set standards that will be covered in Chapter 3.

Mind you, we have already touched upon this subject, though briefly, in the introductory chapter, but there is a lot more to learn! After reading this chapter, you should have a firm grasp of the types of characters, or character classes, used to write CJKV text, specifically the following:

- Latin characters—including transliteration and romanization systems
- Zhuyin—also called *bopomofo*
- Kana—*hiragana* and *katakana*
- Hangul syllables—including *jamo*, the elements from which they're made
- Ideographs—originating in China
- Non-Chinese ideographs—originating in Japan, Korea, and Vietnam

Knowing that each of these character classes exhibits its own special characteristics and often has locale-specific usages is important to grasp. This information is absolutely crucial for understanding discussions elsewhere in this book. After all, many of the problems and issues that caused you to buy this book are the result of the complexities of these writing systems. This is not a bad thing: the complexities and challenges that we face are what make our lives interesting, and to some extent, unique from one another.

Latin Characters, Transliteration, and Romanization

Latin characters (拉丁字母 *lādīng zìmǔ* in Chinese, ラテン文字 *raten moji* or ローマ字 *rōmaji* in Japanese, 로마자 *romaja* in Korean, and *Quốc ngữ*/國語 in Vietnamese) used in the context of CJKV text are the same as those used in Western text, specifically the 52 upper- and lowercase letters of the Latin alphabet, sometimes decorated with accents to

indicate length, tone, or other phonetic attributes, and sometimes set with full-width metrics. Also included are the 10 numerals 0 through 9. Accented characters, usually vowels, are often required for transliteration or Romanization purposes. Table 2-1 lists the basic set of Latin characters.

Table 2-1. Latin characters

Character class	Characters
Lowercase	abcdefghijklmnopqrstuvwxyz
Uppercase	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Numerals	0123456789

There is really nothing special about these characters. Latin characters are most often used in tables (numerals), in abbreviations and acronyms (alphabet), or for transcription or transliteration purposes, sometimes with accented characters to account for tones or other phonetic attributes.

Transliteration systems are distinguished from Romanization systems in that they are not the primary way to write a language, and serve as a prounounciation aid for those who are not familiar with the primary scripts of the language.

Commonly used transliteration systems for CJKV text that use characters beyond the standard set of Latin characters illustrated in Table 2-1 include *Pinyin* (Chinese), *Hepburn* (Japanese), *Kunrei* (Japanese), and *Ministry of Education* (Korean). These and other CJKV transliteration systems are covered in the following sections. *Quốc ngữ*, on the other hand, is a Romanization system, because it is the accepted way to write Vietnamese.

Special cases of transliteration are covered in Chapter 9, specifically in the section entitled "Special Transliteration Considerations."

Chinese Transliteration Methods

Chinese uses two primary transliteration methods: *Pinyin* (拼音 *pīnyīn*) and *Wade-Giles* (韋氏 *wéishì*). There is also the *Yale* method, which is not covered in this book. There are many similarities between these two transliteration methods; they mainly differ in where they are used. Pinyin is used in China, whereas Wade-Giles is popular in Taiwan. Historically speaking, Wade-Giles was the original Chinese transliteration system recognized during the nineteenth century.

Table 2-2 lists the consonant sounds as transliterated by Pinyin and Wade-Giles—zhuyin/bopomofo symbols, also a transliteration system, and described later in this chapter, are included for the purpose of cross-reference.

Table 2-2. Chinese transliteration—consonants

Zhuyin/bopomofo	Pinyin	Wade-Giles
ㄅ	B	P
ㄆ	P	P′
ㄇ	M	M
ㄈ	F	F
ㄉ	D	T
ㄊ	T	T′
ㄋ	N	N
ㄌ	L	L
ㄍ	G	K
ㄎ	K	K′
ㄏ	H	H
ㄐ	J	CH[a]
ㄑ	Q	CH′[a]
ㄒ	X	HS[a]
ㄓ	ZH	CH
ㄔ	CH	CH′
ㄕ	SH	SH
ㄖ	R	J
ㄗ	Z	TS
ㄘ	C	TS′
ㄙ	S	S

a. Only before *i* or *ü*

Table 2-3 lists the vowel sounds as transliterated by Pinyin—zhuyin/bopomofo are again included for reference. Note that this table is constructed as a matrix that indicates what zhuyin vowel combinations are possible and how they are transliterated. The two axes themselves serve to indicate the transliterations for single zhuyin vowels.

Table 2-3. Chinese transliteration—vowels

	ㄧ I	ㄨ U	ㄩ Ü
ㄚ A	ㄧㄚ IA	ㄨㄚ UA	
ㄛ O		ㄨㄛ UO	

Table 2-3. Chinese transliteration—vowels

	ㄧ I	ㄨ U	ㄩ Ü
ㄜ E	ㄧㄝ IE		ㄩㄝ ÜE
ㄞ AI		ㄨㄞ UAI	
ㄟ EI		ㄨㄟ UEI	
ㄠ AO	ㄧㄠ IAO		
ㄡ OU	ㄧㄡ IOU		
ㄢ AN	ㄧㄢ IAN	ㄨㄢ UAN	ㄩㄢ ÜAN
ㄣ EN	ㄧㄣ IN	ㄨㄣ UEN	ㄩㄣ ÜN
ㄤ ANG	ㄧㄤ IANG	ㄨㄤ UANG	
ㄥ ENG	ㄧㄥ ING	ㄨㄥ UENG or ONG	ㄩㄥ IONG

The zhuyin character ㄦ, which deserves separate treatment from the others, is usually transliterated *er*.

It is sometimes necessary to use an apostrophe to separate the Pinyin readings of individual hanzi when the result can be ambiguous. Consider the transliterations for the words 先 and 西安, which are *xiān* and *xī'ān*, respectively. Note the use of the apostrophe to distinguish them.

More details about the zhuyin characters themselves appear later in this chapter. Po-Han Lin (林伯翰 *lín bóhàn*) has developed a Java applet that can convert between the Pinyin, Wade-Giles, and Yale transliteration systems.* He also provides additional details about Chinese transliteration.†

Chinese tone marks

Also of interest is how tone marks are rendered when transliterating Chinese text. Basically, there are two systems for indicating tone. One system, which requires the use of special fonts, employs diacritic marks that serve to indicate tone. The other system uses the numerals 1 through 4 immediately after each hanzi transliteration—no special fonts are required. Pinyin transliteration generally uses diacritic marks, but Wade-Giles uses numerals.

Table 2-4 lists the names of the Chinese tone marks, along with an example hanzi for each. Note that there are cases in which no tone is required.

* *http://www.edepot.com/java.html*

† *http://www.edepot.com/taoroman.html*

Table 2-4. Chinese tone mark examples

Tone	Tone name	Number[a]	Example	Meaning
None	轻声/輕聲 (*qīngshēng*)	None	*ma* (吗)	Question particle
Flat	阴平/陰平 (*yīnpíng*)	1	*ma1* or *mā* (妈)	mother
Rising	阳平/陽平 (*yángpíng*)	2	*ma2* or *má* (麻)	hemp, flax
Falling-Rising	上声/上聲 (*shǎngshēng*)	3	*ma3* or *mǎ* (马)	horse
Falling	去声/去聲 (*qùshēng*)	4	*ma4* or *mà* (骂)	cursing, swearing

a. Microsoft's Pinyin input method uses the numeral 5 to indicate no tone.

It is also common to find reference works in which Pinyin readings have no tone marks at all—that is, no numerals and no diacritic marks. I have observed that tone marks can be effectively omitted when the corresponding hanzi are in proximity, such as on the same page; the hanzi themselves can be used to remove any ambiguity that arises from no indication of tones. Pinyin readings provided throughout this book use diacritic marks to indicate tone.

Japanese Transliteration Methods

There are four Japanese transliteration systems worth exploring in the context of this book:

The Hepburn system (ヘボン式 *hebon shiki*)
: Popularized by James Curtis Hepburn, an American missionary, in 1887 in the third edition of his dictionary, this is considered the most widely used system. This transliteration system was developed two years earlier, in 1885, by the *Roman Character Association* (羅馬字会 *rōmajikai*).

The Kunrei system (訓令式 *kunrei shiki*)
: Developed in 1937, this is considered the official transliteration system by the Japanese government.

The Nippon system (日本式 *nippon shiki*)
: Developed by Aikitsu Tanakadate (田中館愛橘 *tanakadate aikitsu*) in 1881—nearly identical to the Kunrei system, but the least used.

The Word Processor system (ワープロ式 *wāpuro shiki*)
: Developed in a somewhat *ad hoc* fashion over recent years by Japanese word processor and input method manufacturers. Whereas the other three transliteraton systems are largely phonemic, the Word Processor system more closely adheres to a one-to-one transcription of the kana.

The Japanese transliterations in this book adhere to the Hepburn system. Because the Word Processor system allows for a wide variety of transliteration possibilities, which is the nature of input methods, it is thus a topic of discussion in Chapter 5.

Table 2-5 lists the basic kana syllables (shown here and in other tables of this section using hiragana), transliterated according to the three transliteration systems. Those that are transliterated differently in the three systems have been highlighted for easy differentiation. Table 2-18 provides similar information, but presented in a different manner.

Table 2-5. Single syllable Japanese transliteration

Kana	Hepburn	Kunrei	Nippon
あ	A	A	A
い	I	I	I
う	U	U	U
え	E	E	E
お	O	O	O
か	KA	KA	KA
が	GA	GA	GA
き	KI	KI	KI
ぎ	GI	GI	GI
く	KU	KU	KU
ぐ	GU	GU	GU
け	KE	KE	KE
げ	GE	GE	GE
こ	KO	KO	KO
ご	GO	GO	GO
さ	SA	SA	SA
ざ	ZA	ZA	ZA
し	SHI	SI	SI
じ	JI	ZI	ZI
す	SU	SU	SU
ず	ZU	ZU	ZU
せ	SE	SE	SE
ぜ	ZE	ZE	ZE

Table 2-5. Single syllable Japanese transliteration

Kana	Hepburn	Kunrei	Nippon
そ	SO	SO	SO
ぞ	ZO	ZO	ZO
た	TA	TA	TA
だ	DA	DA	DA
ち	CHI	TI	TI
ぢ	JI	ZI	DI
つ	TSU	TU	TU
づ	ZU	ZU	DU
て	TE	TE	TE
で	DE	DE	DE
と	TO	TO	TO
ど	DO	DO	DO
な	NA	NA	NA
に	NI	NI	NI
ぬ	NU	NU	NU
ね	NE	NE	NE
の	NO	NO	NO
は	HA	HA	HA
ば	BA	BA	BA
ぱ	PA	PA	PA
ひ	HI	HI	HI
び	BI	BI	BI
ぴ	PI	PI	PI
ふ	FU	HU	HU
ぶ	BU	BU	BU
ぷ	PU	PU	PU
へ	HE	HE	HE

Table 2-5. Single syllable Japanese transliteration

Kana	Hepburn	Kunrei	Nippon
べ	BE	BE	BE
ぺ	PE	PE	PE
ほ	HO	HO	HO
ぼ	BO	BO	BO
ぽ	PO	PO	PO
ま	MA	MA	MA
み	MI	MI	MI
む	MU	MU	MU
め	ME	ME	ME
も	MO	MO	MO
や	YA	YA	YA
ゆ	YU	YU	YU
よ	YO	YO	YO
ら	RA	RA	RA
り	RI	RI	RI
る	RU	RU	RU
れ	RE	RE	RE
ろ	RO	RO	RO
わ	WA	WA	WA
ゐ	WI	WI	WI
ゑ	WE	WE	WE
を	O	O	WO
ん	N or M[a]	N	N

a. An *m* was once used before the consonants *b*, *p*, or *m*—an *n* is now used in all contexts.

Table 2-6 lists what are considered to be the palatalized syllables—although they signify a single syllable, they are represented with two kana characters. Those that are different in the three transliteration systems are highlighted.

Table 2-6. Japanese transliteration—palatalized syllables

Kana	Hepburn	Kunrei	Nippon
きゃ	KYA	KYA	KYA
ぎゃ	GYA	GYA	GYA
きゅ	KYU	KYU	KYU
ぎゅ	GYU	GYU	GYU
きょ	KYO	KYO	KYO
ぎょ	GYO	GYO	GYO
しゃ	SHA	SYA	SYA
じゃ	JA	ZYA	ZYA
しゅ	SHU	SYU	SYU
じゅ	JU	ZYU	ZYU
しょ	SHO	SYO	SYO
じょ	JO	ZYO	ZYO
ちゃ	CHA	TYA	TYA
ぢゃ	JA	ZYA	DYA
ちゅ	CHU	TYU	TYU
ぢゅ	JU	ZYU	DYU
ちょ	CHO	TYO	TYO
ぢょ	JO	ZYO	DYO
にゃ	NYA	NYA	NYA
にゅ	NYU	NYU	NYU
にょ	NYO	NYO	NYO
みゃ	MYA	MYA	MYA
みゅ	MYU	MYU	MYU
みょ	MYO	MYO	MYO
ひゃ	HYA	HYA	HYA
びゃ	BYA	BYA	BYA
ぴゃ	PYA	PYA	PYA

Table 2-6. Japanese transliteration—palatalized syllables

Kana	Hepburn	Kunrei	Nippon
ひゅ	HYU	HYU	HYU
びゅ	BYU	BYU	BYU
ぴゅ	PYU	PYU	PYU
ひょ	HYO	HYO	HYO
びょ	BYO	BYO	BYO
ぴょ	PYO	PYO	PYO
りゃ	RYA	RYA	RYA
りゅ	RYU	RYU	RYU
りょ	RYO	RYO	RYO

Table 2-7 lists what are considered to be long (or doubled) vowels. The first five rows are hiragana, and the last five are katakana. Note that only the long hiragana *i*—written いい, and transliterated *ii*—is common to all three systems, and that the Kunrei and Nippon systems are identical in this regard.

Table 2-7. Japanese transliteration—long vowels

Kana	Hepburn	Kunrei	Nippon
ああ	Ā	Â	Â
いい	II	II	II
うう	Ū	Û	Û
ええ	Ē	Ê	Ê
えい	EI	EI	EI
おう	Ō	Ô	Ô
アー	Ā	Â	Â
イー	Ī	Î	Î
ウー	Ū	Û	Û
エー	Ē	Ê	Ê
オー	Ō	Ô	Ô

The only difference among these systems' long vowel transliterations is the use of a macron (Hepburn) versus a circumflex (Kunrei and Nippon). Almost all Latin fonts include circumflexed vowels, but those with macroned vowels are still relatively rare.

Finally, Table 2-8 shows some examples of how to transliterate Japanese double consonants, all of which use a small つ or ッ (*tsu*).

Table 2-8. Japanese transliteration—double consonants

Example	Transliteration
かっこ	*kakko*
いっしょ	*issho*
ふっそ	*fusso*
ねっちゅう	*netchū*
しって	*shitte*
ビット	*bitto*
ベッド	*beddo*
バッハ	*bahha*

Korean Transliteration Methods

There are now four generally accepted methods for transliterating Korean text: *The Revised Romanization of Korean*[*] (국어의 로마자 표기법/國語의 로마字 表記法 *gugeoui romaja pyogibeop*), established on July 7, 2000; *Ministry of Education* (문교부/文敎部 *mungyobu*, derived from and sometimes referred to as *McCune-Reischauer*), established on January 13, 1984;[†] *Korean Language Society* (한글 학회/한글 學會 *hangeul hakhoe*), established on February 21, 1984;[‡] and ISO/TR 11941:1996 (*Information Documentation—Transliteration of Korean Script into Latin Characters*), established in 1996. The transliterated Korean text in this book adheres to the RRK transliteration method because it represents the official way in which Korean text is transliterated, at least in South Korea.[§] Other transliteration methods, not covered in this book, include Yale, Lukoff, and Horne.

Table 2-9 lists the jamo that represent consonants, along with their representation in these three transliteration methods. Also included are the ISO/TR 11941:1996 transliterations when these jamo serve as the final consonant of a syllable. ISO/TR 11941:1996 Method 1

* *http://www.mct.go.kr/english/roman/roman.jsp*

† *http://www.hangeul.or.kr/24_1.htm*

‡ *http://www.hangeul.or.kr/hnp/hanroma.hwp*

§ Notable exceptions include words such as *hangul*, which should really be transliterated as *hangeul*.

is used for North Korea (DPRK), and Method 2 is used for South Korea (ROK). Uppercase is used solely for clarity.

Table 2-9. Korean transliteration—consonants

Jamo	RRK[a]	MOE	KLS	ISO (DPRK)	Final	ISO (ROK)	Final
ㄱ	G/K[b]—G	K/G	G	K	K	G	G
ㄴ	N	N	N	N	N	N	N
ㄷ	D/T[c]—D	T/D	D	T	T	D	D
ㄹ	R/L[d]—L	R/L	L	R	L	R	L
ㅁ	M	M	M	M	M	M	M
ㅂ	B/P[e]—B	P/B	B	P	P	B	B
ㅅ	S	S/SH	S	S	S	S	S
ㅇ	None/NG	None/NG	None/NG	None	NG	None	NG
ㅈ	J	CH/J	J	C	C	J	J
ㅊ	CH	CH′	CH	CH	CH	C	C
ㅋ	K	K′	K	KH	KH	K	K
ㅌ	T	T′	T	TH	TH	T	T
ㅍ	P	P′	P	PH	PH	P	P
ㅎ	H	H	H	H	H	H	H
ㄲ	KK	KK	GG	KK	KK	GG	GG
ㄸ	TT	TT	DD	TT	n/a	DD	n/a
ㅃ	PP	PP	BB	PP	n/a	BB	n/a
ㅆ	SS	SS	SS	SS	SS	SS	SS
ㅉ	JJ	TCH	JJ	CC	n/a	JJ	n/a

a. When Clause 8 of this system is invoked, the character shown after the dash shall be used.

b. *G* is used before vowels, and *K* is used when followed by another consonant or to form the final sound of a word.

c. *D* is used before vowels, and *T* is used when followed by another consonant or to form the final sound of a word.

d. *R* is used before vowels, and *L* is used when followed by another consonant or to form the final sound of a word.

e. *B* is used before vowels, and *P* is used when followed by another consonant or to form the final sound of a word.

Note that some of the double jamo do not occur at the end of syllables. Also, some of these transliteration methods, most notably the Ministry of Education system, have a number of rules that dictate how to transliterate certain jamo depending on their context.

This context dependency arises because the MOE and RRK (without Clause 8 invoked) are transcription systems and not transliteration systems, and because the hangul script is morphophonemic (represents the underlying root forms) and not phonemic (represents the actual sounds). If a one-to-one correspondence and round-trip conversion are desired, invoking Clause 8 of the RRK system accomplishes both. ICU's Hangul-Latin transliteration function does this.*

For example, the ㄱ jamo is transliterated as *k* when voiceless (such as at the beginning of a word or not between two voiced sounds), and as *g* when between two voiced sounds (such as between two vowels, or between a vowel and a voiced consonant). Thus, ㄱ in 강물 is pronounced *k* because it is voiceless, but the same ㄱ in 한강 is pronounced *g* because it is between a voiced consonant (ㄴ) and a vowel (ㅏ), and becomes voiced itself.

Clause 8 of RRK states that when it is necessary to convert transliterated Korean back into hangul for special purposes, such as for academic papers, transliteration is done according to hangul spelling and not by pronunciation. The jamo ㄱ, ㄷ, ㅂ, and ㄹ are thus always written as *g*, *d*, *b*, and *l*. Furthermore, when ㅇ has no phonetic value, it is replaced by a hyphen, which may also be used to separate or distinguish syllables.

ISO/TR 11941:1996 also defines transliterations for compound consonant jamo that appear only at the end of hangul syllables, all of which are listed in Table 2-10.

Table 2-10. ISO/TR 11941:1996 compound jamo transliteration

Jamo	DPRK	ROK
ㄱㅅ	KS	GS
ㄴㅈ	NJ	NJ
ㄴㅎ	NH	NH
ㄹㄱ	LK	LG
ㄹㅁ	LM	LM
ㄹㅂ	LP	LB
ㄹㅅ	LS	LS
ㄹㅌ	LTH	LT
ㄹㅍ	LPH	LP
ㄹㅎ	LH	LH
ㅂㅅ	PS	BS

* *http://icu-project.org/userguide/Transform.html*

Table 2-11 lists the jamo that represent vowels and diphthongs, along with their representations in the three transliteration methods. Again, uppercase is used for clarity, and differences have been highlighted.

Table 2-11. Korean transliteration—vowels

Jamo	RRK	MOE	KLS	ISO (DPRK and ROK)
ㅏ	A	A	A	A
ㅑ	YA	YA	YA	YA
ㅓ	EO	ŏ	EO	EO
ㅕ	YEO	Yŏ	YEO	YEO
ㅗ	O	O	O	O
ㅛ	YO	YO	YO	YO
ㅜ	U	U	U	U
ㅠ	YU	YU	YU	YU
ㅡ	EU	ŭ	EU	EU
ㅣ	I	I	I	I
ㅐ	AE	AE	AE	AE
ㅒ	YAE	YAE	YAE	YAE
ㅔ	E	E	E	E
ㅖ	YE	YE	YE	YE
ㅘ	WA	WA	WA	WA
ㅙ	WAE	WAE	WAE	WAE
ㅚ	OE	OE	OE	OE
ㅝ	WO	Wŏ	WEO	WEO
ㅞ	WE	WE	WE	WE
ㅟ	WI	WI	WI	WI
ㅢ	UI	ŭI	EUI	YI

Note that the ISO/TR 11941:1996 transliteration method is identical for both North and South Korea (DPRK and ROK, respectively).

As with most transliteration methods, there are countless exceptions and special cases. Tables 2-9 and 2-11 provide only the basic transliterations for jamo. It is when you start

combining consonants and vowels that exceptions and special cases become an issue. In fact, a common exception is the transliteration of the hangul used for the Korean surname "Lee." I suggest that you try Younghong Cho's *Korean Transliteration Tools*.*

Vietnamese Romanization Methods

Writing Vietnamese using Latin characters—called *Quốc ngữ* (國語)—is considered the most acceptable method for expressing Vietnamese today. As a result, Quốc ngữ is not considered a transliteration method. As with using Latin characters to represent Chinese, Japanese, and Korean text, it is the currently acceptable means to express Vietnamese in writing. Quốc ngữ is thus a Romanization system.

This writing system is based on Latin script, but is decorated with additional characters and many diacritic marks. This complexity serves to account for the very rich Vietnamese sound system, complete with tones.

In addition to the upper- and lowercase English alphabet, Quốc ngữ requires two additional consonants and 12 additional base characters (that is, characters that do not indicate tone), as shown in Table 2-12.

Table 2-12. Additional Quốc ngữ consonants and base characters

Character class	Consonants	Base characters
Lowercase	đ	ăâêôơư
Uppercase	Đ	ĂÂÊÔƠƯ

The modifiers that are used for the base vowels, in the order shown in Table 2-12, are called *breve* or *short* (*trăng* or *mũ ngược* in Vietnamese), *circumflex* (*mũ* in Vietnamese), and *horn* (*móc* or *râu* in Vietnamese).

While these additional base characters include diacritic marks and other attachments, they do not indicate tone. There are six tones in Vietnamese, five of which are written with a tone mark. Every Vietnamese word must have a tone. The diacritic marks for these six tones are shown in Table 2-13, along with their names.

Table 2-13. The six Vietnamese tones

Tone mark	Name in Vietnamese	Name in English
none	Không dấu	none
◌̀	Huyền	Grave

* *http://www.sori.org/hangul/conv2kr.cgi*

Table 2-13. The six Vietnamese tones

Tone mark	Name in Vietnamese	Name in English
◌̉	Hỏi	Hook above, curl, or *hoi*
◌̃	Ngã	Tilde
◌́	Sắc	Acute
◌̣	Nặng	Dot below, underdot, or *nang*

All of the diacritic-annotated characters that are required for the Quốc ngữ writing system, which are combinations of base characters plus tones, are provided in Table 2-14.

Table 2-14. Quốc ngữ base characters and tone marks

		Base characters																			
		a	A	ă	Ă	â	Â	e	E	ê	Ê	i	I	o	O	ô	Ô	ơ	Ơ	u	U	ư	Ư	y	Y
Tone marks	◌̀	à	À	ằ	Ằ	ầ	Ầ	è	È	ề	Ề	ì	Ì	ò	Ò	ồ	Ồ	ờ	Ờ	ù	Ù	ừ	Ừ	ỳ	Ỳ
	◌̉	ả	Ả	ẳ	Ẳ	ẩ	Ẩ	ẻ	Ẻ	ể	Ể	ỉ	Ỉ	ỏ	Ỏ	ổ	Ổ	ở	Ở	ủ	Ủ	ử	Ử	ỷ	Ỷ
	◌̃	ã	Ã	ẵ	Ẵ	ẫ	Ẫ	ẽ	Ẽ	ễ	Ễ	ĩ	Ĩ	õ	Õ	ỗ	Ỗ	ỡ	Ỡ	ũ	Ũ	ữ	Ữ	ỹ	Ỹ
	◌́	á	Á	ắ	Ắ	ấ	Ấ	é	É	ế	Ế	í	Í	ó	Ó	ố	Ố	ớ	Ớ	ú	Ú	ứ	Ứ	ý	Ý
	◌̣	ạ	Ạ	ặ	Ặ	ậ	Ậ	ẹ	Ẹ	ệ	Ệ	ị	Ị	ọ	Ọ	ộ	Ộ	ợ	Ợ	ụ	Ụ	ự	Ự	ỵ	Ỵ

In summary, Quốc ngữ requires 134 additional characters beyond the English alphabet. Fourteen are additional base characters (see Table 2-12), and the remaining 120 include diacritic marks that indicate tone (see Table 2-14). Although the U+1E*xx* block of Unicode provides the additional characters necessary for Vietnamese in precomposed form, they can still be represented as sequences that are composed of a base character followed by one or more diacritic marks. Windows Code Page 1258 includes some, but not all, of these precomposed characters.

ASCII-based Vietnamese transliteration methods

When only the ASCII character set is available, it is still possible to represent Vietnamese text using well-established systems. The two most common ASCII-based transliteration methods are called *VIetnamese Quoted-Readable* (VIQR) and *VSCII MNEMonic* (VSCII-MNEM). The VIQR system is documented in RFC 1456.* Table 2-15 illustrates how Quốc ngữ base characters and tones are represented in these two systems.

* *http://www.ietf.org/rfc/rfc1456.txt*

Table 2-15. VIQR and VSCII-MNEM transliteration methods

	Quốc ngữ		VIQR		VSCII-MNEM	
Base characters	ă	Ă	a(	A(	a<	A<
	â	Â	a^	A^	a>	A>
	ê	Ê	e^	E^	e>	E>
	ô	Ô	o^	O^	o>	O>
	ơ	Ơ	o+	O+	o*	O*
	ư	Ư	u+	U+	u*	U*
	đ	Đ	dd	DD	dd	DD
Tones	à	À	a`	A`	a!	A!
	ả	Ả	a?	A?	a?	A?
	ã	Ã	a~	A~	a"	A"
	á	Á	a'	A'	a'	A'
	ạ	Ạ	a.	A.	a.	A.

Table 2-16 illustrates how base characters and tones are combined in each system. Note how the base character's ASCII-based annotation comes before the ASCII-based tone mark.

Table 2-16. Base character plus tones using VIQR and VSCII-MNEM methods

Quốc ngữ	VIQR		VSCII-MNEM	
ờ Ờ	o+`	O+`	o*!	O*!
ở Ở	o+?	O+?	o*?	O*?
ỡ Ỡ	o+~	O+~	o*"	O*"
ớ Ớ	o+'	O+'	o*'	O*'
ợ Ợ	o+.	O+.	o*.	O*.

Zhuyin/Bopomofo

Zhuyin, developed in the early part of the 20th century, is a method for transcribing Chinese text using ideograph elements for their reading value. In other words, it is a transliteration system. It is also known as the *National Phonetic System* (注音符号 *zhùyīn fúhào*) or *bopomofo*. The name *bopomofo* is derived from the readings of the first four characters in the character set: *b*, *p*, *m*, and *f*. There are a total of 37 characters (representing 21

consonants and 16 vowels), along with five symbols to indicate tone (one of which has no glyph) in the zhuyin character set.

Table 2-17 illustrates each of the zhuyin characters, along with the ideograph from which they were derived, and their reading. Those that represent vowels are at the end of the table.

Table 2-17. Zhuyin characters

Zhuyin	Ideograph	Reading—Pinyin
ㄅ	勹	B
ㄆ	攵	P
ㄇ	冂	M
ㄈ	匚	F
ㄉ	力	D
ㄊ	去	T
ㄋ	乃	N
ㄌ	力	L
ㄍ	巛	G
ㄎ	亏	K
ㄏ	厂	H
ㄐ	丩	J
ㄑ	く	Q
ㄒ	丅	X
ㄓ	屮	ZH
ㄔ	彳	CH
ㄕ	尸	SH
ㄖ	日	R
ㄗ	卩	Z
ㄘ	专	C
ㄙ	ム	S
ㄚ	丫	A

Table 2-17. Zhuyin characters

Zhuyin	Ideograph	Reading—Pinyin
ㄛ	己	O
ㄜ	左	E
ㄝ	也	EI
ㄞ	歹	AI
ㄟ	飞	EI
ㄠ	幺	AO
ㄡ	又	OU
ㄢ	弓	AN
ㄣ	乜	EN
ㄤ	尤	ANG
ㄥ	厶	ENG
ㄦ	儿	ER
ㄧ or 一	丨	I
ㄨ	乂	U
ㄩ	凵	IU

The zhuyin character set is included in character sets developed in China (GB 2312-80 and GB/T 12345-90, Row 8) and Taiwan (CNS 11643-2007, Plane 1, Row 5). This set of characters is identical across these two Chinese locales, with one exception, which is indicated in Table 2-17 with two different characters: "丨" is used in China, and "一" is used in Taiwan.

Kana

The most frequently used script found in Japanese text is *kana*. It is a collective term for two closely related scripts, as follows:

- Hiragana
- Katakana

Although one would expect to find kana characters only in Japanese character sets, they are, in fact, part of some Chinese and Korean character sets, in particular GB 2312-80 and KS X 1001:2004. In fact, kana are encoded at the same code points in the case of GB 2312-80! Why in the world would Chinese and Korean character sets include kana? Most likely

for the purposes of creating Japanese text using a Chinese or Korean character set.* After all, many of the ideographs are common across these locales.

The following sections provide detailed information about kana, along with how they were derived from ideographs.

Hiragana

Hiragana (平仮名 *hiragana*) are characters that represent sounds, specifically syllables. A syllable is generally composed of a consonant plus a vowel—sometimes a single vowel will do. In Japanese, there are five vowels: *a*, *i*, *u*, *e*, and *o*; and 14 basic consonants: *k*, *s*, *t*, *n*, *h*, *m*, *y*, *r*, *w*, *g*, *z*, *d*, *b*, and *p*. It is important to understand that hiragana is a syllabary, not an alphabet: you cannot decompose a hiragana character into a part that represents the vowel and a part that represents the consonant. Hiragana (and katakana, covered in the next section) is one of the only true syllabaries still in common use today. Table 2-18 illustrates a matrix containing the basic and extended hiragana syllabary.

Table 2-18. The hiragana syllabary

		K	S	T	N	H	M	Y	R	W	G	Z	D	B	P
A	あ	か	さ	た	な	は	ま	や	ら	わ	が	ざ	だ	ば	ぱ
I	い	き	し	ち	に	ひ	み		り	ゐ	ぎ	じ	ぢ	び	ぴ
U	う	く	す	つ	ぬ	ふ	む	ゆ	る		ぐ	ず	づ	ぶ	ぷ
E	え	け	せ	て	ね	へ	め		れ	ゑ	げ	ぜ	で	べ	ぺ
O	お	こ	そ	と	の	ほ	も	よ	ろ	を	ご	ぞ	ど	ぼ	ぽ
N	ん														

The following are some notes to accompany Table 2-18:

- Several hiragana have smaller versions, and are as follows (the standard version is in parentheses): ぁ (あ), ぃ (い), ぅ (う), ぇ (え), ぉ (お), っ (つ), ゃ (や), ゅ (ゆ), ょ (よ), and ゎ (わ).
- Two hiragana, ゐ and ゑ, are no longer commonly used.
- The hiragana を is read as *o*, not *wo*.
- The hiragana ん is considered an independent syllable and is pronounced approximately *ng*.

* There is, however, one fatal flaw in the Chinese and Korean implementations of kana. They omitted five symbols used with kana, all of which are encoded in row 1 of JIS X 0208:1997 (Unicode code points also provided): ヽ (01-19; U+30FD), ヾ (01-20; U+30FE), ゝ (01-21; U+309D), ゞ (01-22; U+309E), and ー (01-28; U+30FC).

Notice that some cells do not contain any characters. These sounds are no longer used in Japanese, and thus no longer need a character to represent them. Also, the first block of characters is set in a 5×10 matrix. This is sometimes referred to as the *50 Sounds Table* (50 音表 *gojūon hyō*), so named because it has a capacity of 50 cells. The other blocks of characters are the same as those in the first block, but with diacritic marks.

Diacritic marks serve to annotate characters with additional information—usually a changed pronunciation. In the West you commonly see accented characters such as *á*, *à*, *â*, *ä*, *ã*, and *å*. The accents are called *diacritic marks*.

In Japanese there are two diacritic marks: *dakuten* (also called *voiced* and *nigori*) and *handakuten* (also called *semi-voiced* and *maru*). The dakuten (濁点 *dakuten*) appears as two short diagonal strokes (゛) in the upper-right corner of some kana characters. The dakuten serves to voice the consonant portion of the kana character to which it is attached.* Examples of voiceless consonants include *k*, *s*, and *t*. Their voiced counterparts are *g*, *z*, and *d*, respectively. Hiragana *ka* (か) becomes *ga* (が) with the addition of the dakuten. The *b* sound is a special voiced version of a voiced *h* in Japanese.

The handakuten (半濁点 *handakuten*) appears as a small open circle (゜) in the upper-right corner of kana characters that begin with the *h* consonant. It transforms this *h* sound into a *p* sound.

Hiragana were derived by cursively writing kanji, but no longer carry the meaning of the kanji from which they were derived. Table 2-22 lists the kanji from which the basic hiragana characters were derived.

In modern Japanese, hiragana are used to write grammatical words, inflectional endings for verbs and adjectives, and some nouns.† They can also be used as a fallback (read "crutch") in case you forget how to write a kanji—the hiragana that represent the reading of a kanji are used in this case. In summary, hiragana are used to write some native Japanese words.

Table 2-19 enumerates the hiragana characters that are included in the JIS X 0208:1997 and JIS X 0213:2004 character set standards. For both character set standards, all of these characters are in Row 4.

* *Voicing* is a linguistic term referring to the vibration of the vocal bands while articulating a sound.

† Prior to the Japanese writing system reforms that took place after World War II, hiragana and katakana were used interchangeably, and many legal documents used katakana for inflectional endings and for purposes now used exclusively by hiragana.

Table 2-19. Hiragana characters in JIS standards

Standard	Characters
JIS X 0208:1997	ぁあぃいぅうぇえぉおかがきぎくぐけげこごさざしじすずせぜそぞただちぢっつづてでとどなにぬねのはばぱひびぴふぶぷへべぺほぼぽまみむめもゃやゅゆょよらりるれろゎわゐゑをん
JIS X 0213:2004	ゔゕゖ

Note how these characters have a cursive or calligraphic look to them (cursive and calligraphic refer to a smoother, handwritten style of characters). Keep these shapes in mind while we move on to katakana.

Katakana

Katakana (片仮名 *katakana*), like hiragana, is a syllabary, and with minor exceptions, they represent the same set of sounds as hiragana. Their modern usage, however, differs from hiragana. Where hiragana are used to write native Japanese words, katakana are primarily used to write words of foreign origin, called *gairaigo* (外来語 *gairaigo*), to write onomatopoeic words,* to express "scientific" names of plants and animals, or for emphasis—similar to the use of italics to represent foreign words and to express emphasis in English. For example, the Japanese word for *bread* is written パン and is pronounced *pan*. It was borrowed from the Portuguese word *pão*, which is pronounced sort of like *pown*. Katakana are also used to write foreign names. Table 2-20 illustrates the basic and extended katakana syllabary.

Table 2-20. The katakana syllabary

		K	S	T	N	H	M	Y	R	W	G	Z	D	B	P
A	ア	カ	サ	タ	ナ	ハ	マ	ヤ	ラ	ワ	ガ	ザ	ダ	バ	パ
I	イ	キ	シ	チ	ニ	ヒ	ミ		リ	ヰ	ギ	ジ	ヂ	ビ	ピ
U	ウ	ク	ス	ツ	ヌ	フ	ム	ユ	ル		グ	ズ	ヅ	ブ	プ
E	エ	ケ	セ	テ	ネ	ヘ	メ		レ	ヱ	ゲ	ゼ	デ	ベ	ペ
O	オ	コ	ソ	ト	ノ	ホ	モ	ヨ	ロ	ヲ	ゴ	ゾ	ド	ボ	ポ
N	ン														

* *Onomatopoeic* refers to words that serve to describe a sound, such as *buzz* or *hiss* in English. In Japanese, for example, ブクブク (*bukubuku*) represents the sound of a balloon expanding.

The following are some notes to accompany Table 2-20:

- Several katakana have smaller versions, and are as follows (the standard version is in parentheses): ァ (ア), ィ (イ), ゥ (ウ), ェ (エ), ォ (オ), ヵ (カ), ヶ (ケ), ッ (ツ), ャ (ヤ), ュ (ユ), ョ (ヨ), and ヮ (ワ).
- Two katakana, ヰ and ヱ, are no longer commonly used.
- The katakana ヲ is read as *o*, not *wo*.
- The katakana ン is considered an independent syllable, and is pronounced approximately *ng*.

Katakana were derived by extracting a single portion of a whole kanji, and, like hiragana, no longer carry the meaning of the kanji from which they were derived. If you compare several of these characters to some kanji, you may recognize common shapes. Table 2-20 lists the basic katakana characters, along with the kanji from which they were derived.

Table 2-21 enumerates the katakana characters that are included in the JIS X 0208:1997 and JIS X 0213:2004 character set standards. As shown in the table, those in JIS X 0208:1997 are in Row 5, and those in JIS X 0213:2004 are in Plane 1, but spread across Rows 5 through 7.

Table 2-21. Katakana characters in JIS standards

Standard	Row	Characters
JIS X 0208:1997	5	ァアィイゥウェエォオカガキギクグケゲコゴサザシジスズセゼソゾタダチヂッツヅテデトドナニヌネノハバパヒビピフブプヘベペホボポマミムメモャヤュユョヨラリルレロヮワヰヱヲンヴヵヶ
JIS X 0213:2004	5	ガギグゲゴゼゾド
	6	クシストヌハヒフヘホプムラリルレロ
	7	ヴヷヸヹヺ

Katakana, unlike hiragana, have a squared, more rigid feel to them. Structurally speaking, they are quite similar in appearance to kanji, which we discuss later.

The Development of Kana

You already know that kana were derived from kanji, and Table 2-22 provides a complete listing of kana characters, along with the kanji from which they were derived.

Table 2-22. The ideographs from which kana were derived

Katakana	Ideograph			Hiragana
ア	阿		安	あ
イ	伊		以	い
ウ		宇		う
エ	江		衣	え
オ		於		お
カ		加		か
キ		幾		き
ク		久		く
ケ	介		計	け
コ		己		こ
サ	散		左	さ
シ		之		し
ス	須		寸	す
セ		世		せ
ソ		曽		そ
タ	多		太	た
チ	千		知	ち
ツ		川		つ
テ		天		て
ト		止		と
ナ		奈		な
ニ	二		仁	に
ヌ		奴		ぬ
ネ		祢		ね
ノ		乃		の
ハ	八		波	は
ヒ		比		ひ

Table 2-22. The ideographs from which kana were derived

Katakana	Ideograph		Hiragana
フ	不		ふ
ヘ	部		へ
ホ	保		ほ
マ	万	末	ま
ミ	三	美	み
ム	牟	武	む
メ	女		め
モ	毛		も
ヤ	也		や
ユ	由		ゆ
ヨ	與	与	よ
ラ	良		ら
リ	利		り
ル	流	留	る
レ	礼	禮	れ
ロ	呂		ろ
ワ	和		わ
ヰ	井	為	ゐ
ヱ	恵		ゑ
ヲ	乎	遠	を
ン	尓	无	ん

Note how many of the kanji from which katakana and hiragana characters were derived are the same, and how the shapes of several hiragana/katakana pairs are similar. In fact, many katakana are nearly identical to kanji and can usually be distinguished by their smaller size and also by the fact that they are typically found in strings containing other katakana. Table 2-23 shows some examples of this phenomenon.

Table 2-23. Katakana and kanji with similar forms

Katakana	Kanji
エ	工
カ	力
タ	夕
ト	卜
ニ	二
ネ	ネ
ハ	八
ヒ	匕
ム	厶
メ	乂
ロ	口

Hangul

Hangul (한글 *hangeul*) syllables are the characters that are used to express contemporary Korean texts in writing.* Unlike Japanese kana, hangul is not a syllabic script, but rather a script that is composed of elements that represent a pure alphabet and are composed as syllables. How does one make the distinction between an alphabet and syllabary? Each hangul syllable can be easily decomposed into hangul elements, which in turn represent individual sounds (that is, consonants and vowels), not syllables. Hangul elements, which do not carry any meaning, are commonly referred to as *jamo* (자모/字母 *jamo*), meaning "alphabet."†

King Sejong (世宗 *sejong*) of the Yi Dynasty completed the development of what is now referred to as hangul back in the year 1443, and the work was officially announced in 1446.‡ Hangul is considered to be one of the most scientific—or, at least, one of the most well-designed—writing systems due to its extremely regular and predictable structure.

Jamo are typically combined with one or two additional jamo to form a hangul syllable. Table 2-24 lists a handful of hangul syllables, along with the jamo used to build them.

* The word *hangul* was coined sometime around 1910, and means "Korean script."

† Sometimes referred to as *jaso* (자소/字素 *jaso*).

‡ The result of this work was a book entitled 訓民正音 (훈민정음 *hunmin jeongeum*) in which 17 consonants and 11 vowels were announced, rather than tens of thousands of syllables that can be made with them.

Table 2-24. Decomposition of hangul syllables into jamo

Hangul	Reading	Jamo	Transliterated
가	GA	ㄱ plus ㅏ	*g* plus *a*
갈	GAL	ㄱ plus ㅏ plus ㄹ	*g* plus *a* plus *l*
갉	GALG	ㄱ plus ㅏ plus ㄹ plus ㄱ	*g* plus *a* plus *l* plus *g*

There are exactly six ways to combine simple jamo into modern hangul syllables, as illustrated in Figure 2-1, along with examples ("C" stands for consonant, "V" stands for vowel, and the order in which consonants and vowels are read is indicated with numerals).

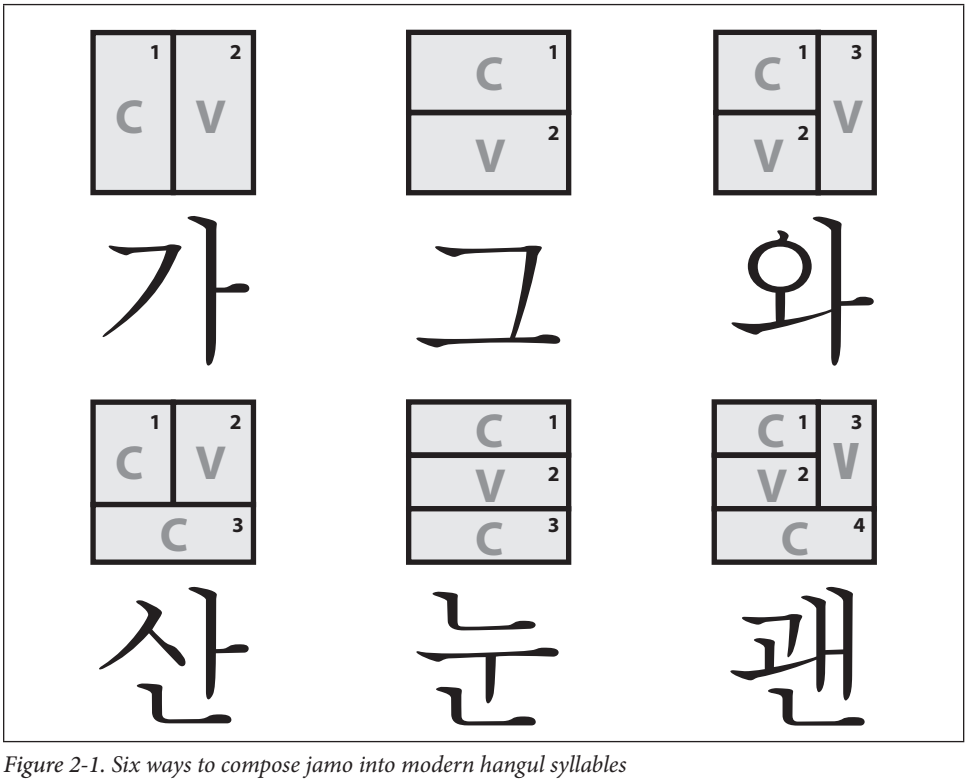


Figure 2-1. Six ways to compose jamo into modern hangul syllables

Korean has special terms for the jamo that are used to construct hangul, depending on where in the syllable they appear:

- *Choseong* (초성/初聲 *choseong*) for the initial sound, usually a consonant
- *Jungseong* (중성/中聲 *jungseong*) for the middle sound, usually a vowel
- *Jongseong* (종성/終聲 *jongseong*) for the final sound, usually a consonant

Table 2-25 provides the complete set of 93 jamo as included in the basic Korean character set standard, specifically KS X 1001:2004. The first 51 of these jamo are considered modern and are used in combination to form modern hangul syllables. The remaining 42 jamo are considered ancient. Also from KS X 1001:2004, the first of its many rows of modern hangul syllables, each of which includes 94 characters, is included in Table 2-25 as well.

Table 2-25. Jamo and hangul syllables in KS X 1001:2004

Row	Characters
4	ᄀ ᄁ ᄂ ᄃ ᄄ ᄅ ᄆ ᄇ ᄈ ᄉ ᄊ ᄋ ᄌ ᄍ ᄎ ᄏ ᄐ ᄑ ᄒ ᅡ ᅢ ᅣ ᅤ ᅥ ᅦ ᅧ ᅨ ᅩ ᅪ ᅫ ᅬ ᅭ ᅮ ᅯ ᅰ ᅱ ᅲ ᅳ ᅴ ᅵ ᆪ ᆬ ᆭ ᆰ ᆱ ᆲ ᆳ ᆴ ᆵ ᆹ ᆻ ᆼ ᆽ ᆾ ᆿ ᇀ ᇁ ᇂ ᇃ ᇄ ᇅ ᇆ ᇇ ᇈ ᇉ ᇊ ᇋ ᇌ ᇍ ᇎ ᇏ ᇐ ᇑ ᇒ ᇓ ᇔ ᇕ ᇖ ᇗ ᇘ ᇙ ᇚ ᇛ ᇜ ᇝ ᇞ ᇟ ᇠ ᇡ ᇢ ᇣ ᇤ ᇥ ᇦ ᇧ ᇨ ᇩ ᇪ ᇫ ᇬ ᇭ ᇮ ᇯ ᇰ ᇱ ᇲ ᇳ ᇴ ᇵ ᇶ ᇷ ᇸ ᇹ ᇺ ᇻ ᇼ ᇽ ᇾ ᇿ ㆍ ㆎ
16	가각간갇갈갉갊감갑값갓갔강갖갗같갚갛개객갠갤갬갭갯갰 갱갸갹갼걀걋걍걔걘걜거걱건걷걸걺검겁것겄겅겆겉겊겋게 겐겔겜겝겟겠겡겨격겪견겯결겸겹겻겼경곁계곈곌곕곗고곡 곤곧골곪곬곯곰곱곳공곶과곽관괄괆

Appendix K provides a complete code table, annotated with readings, for all 2,350 hangul syllables in the KS X 1001:2004 character set standard. This appendix is potentially useful for those who do not yet have a good command of hangul, or who do not have access to a Korean-capable OS.* Also, Wikipedia provides excellent information about the hangul script.†

Ideographs

Ideographs, the single most complex type of character used in CJKV texts, are also called *Chinese characters*, due to their origin in China. Unicode refers to them as *CJK Unified Ideographs* as a way to better distinguish them from other characters, and to emphasize the unified nature of its character set. They are referred to as *hanzi* (汉字/漢字 *hànzì*) in Chinese, as *kanji* (漢字 *kanji*) in Japanese, as *hanja* (한자/漢字 *hanja*) in Korean, and as *chữ Hán*/字漢 in Vietnamese. To grasp the concept of ideographs, one must first understand the magnitude of such a writing system. The 26 characters of the English alphabet (52 characters, if one counts both upper- and lowercase) seem quite limiting compared to the tens of thousands of ideographs in current use by the CJKV locales. It is well documented that the Japanese borrowed the Chinese script over the course of a millennium. What is not well known is that while the Japanese were borrowing ideographs from the

* Given the extent to which modern OSes have matured, globalization-wise, encountering an OS that is not Korean-capable clearly requires taking a step back to earlier OSes.

† *http://en.wikipedia.org/wiki/Hangul*

Chinese, the Chinese were, themselves, adding to the total number of characters in their language by continuing to coin new ideographs.* This means that the Japanese were able, in essence, to capture and freeze a segment of Chinese history every time they borrowed from the Chinese. The same can be said of Korean and Vietnamese, both of whom also borrowed ideographs.

Before we begin discussing the history of ideographs and how they are composed, let's take some time to illustrate some ideographs. Table 2-26 exemplifies the ideographs that are included in the major CJKV character set standards, and covers the first logical row of each plane of each standard. For Unicode, the first 256 ideographs in CJK Unified Ideographs URO (*Unified Repertoire and Ordering*) and CJK Unified Ideographs Extension A are covered.

Table 2-26. Ideographs in various character set standards

Standard	Row	Characters
GB 2312-80	16	啊阿埃挨哎唉哀皑癌蔼矮艾碍爱隘鞍氨安俺 按暗岸胺案肮昂盎凹敖熬翱袄傲奥懊澳芭捌 扒叭吧笆八疤巴拔跋靶把耙坝霸罢爸白柏百 摆佰败拜稗斑班搬扳般颁板版扮拌伴瓣半办 绊邦帮梆榜膀绑棒磅蚌镑傍谤苞胞包褒剥
GB/T 12345-90	16	啊阿埃挨哎唉哀皚癌藹矮艾礙愛隘鞍氨安俺 按暗岸胺案骯昂盎凹敖熬翱襖傲奥懊澳芭捌 扒叭吧笆八疤巴拔跋靶把耙壩霸罷爸白柏百 擺佰敗拜稗斑班搬扳般頒板版扮拌伴瓣半辦 絆邦幫梆榜膀綁棒磅蚌鎊傍謗苞胞包褒剥
CNS 11643-2007 Plane 1	36	一乙丁七乃九了二人儿入八几刀刁力匕十卜 又三下丈上丫丸凡久么也乞于亡兀刃勺千叉 口土士夕大女子孑孓寸小尢尸山川工己已巳 巾干廾弋弓才丑丐不中丰丹之尹予云井互五 亢仁什仃仆仇仍今介仄元允內六兮公冗凶
CNS 11643-2007 Plane 2	1	乂乜凵匚厂万丌乇亍囗屮彳丏冇与丮亓仂仉 仈冘勼卬厹圠夃夬尐巿旡殳毌气爿丱丼仨仜 仩仡仝仚刌刉匟冊圢圣夗夯宁宄尒尻屴屳帄 庀庂忉戉扐氕氶汃氿氻犮犰玊禸肊阞伎优伬 仵伔仱伀价伈伝伂伅伢伓伄仴伒冱刓刑劦

* The Chinese are still coining new ideographs in some locales, especially Hong Kong.

Table 2-26. Ideographs in various character set standards

Standard	Row	Characters
CNS 11643-2007 Plane 3	1	丨丶丿亅丅丄冂冖匸卩厶个亇义凢乆亏亼亾 兦凣刄劜卄夂夊宀巛幺广廴彐彑彡阝𠀋乣乢 亣内仅仏从仌冄円冗冮凨刅办劝勼匀区协厼 卝历厷会双叐収圡屮市弍户戸攴攵无旡生岂 仠伕伌㐲伔伱伒伩侎仝囘同冊写冚切刋㓜
CNS 11643-2007 Plane 4	1	𠂆乀乁乚𡿨丂丩乃几刂卪巜冃冄勺卂卩孑孒 尢屮忄扌犭𠚥丰三仐先冃仌刓㓁乒卬产厸叱 冂壬矢氽氽币开不汇灬亓冈艹辶世丫乇广伯 伨任兂再仒凬凷刉㓝劲劮医厈尾叏叓叽合另 囜囚囙尭压厺尔𡰪户屵机羊祀扎扣甲米禾
CNS 11643-2007 Plane 5	1	𠃌𠃋乙乚𠄌𠂇丩𠃊凵马世屯丮去干屯心少𠂔 㐂仐从仈冭函刂切㓛匌卯厄叒叉叚吃玊屹岂 𢎘朩歹毛爪王亥㐱乩孕孖冋夙甶㓜庀𠣋㔫勉 本夅厍产㔉肉吡另妣孙穴尬叾叫卉纫弘衫忙 戸手扖此卣毋汴汊年夹犳犰而聿节邦附阨
CNS 11643-2007 Plane 6	1	乁卜乂㇉㔾乚二亖亖七𠀌卜工丏旦午丸凡丂 天个亼丹刃勺力可云飞尸光乜亏且丘劢玊凧 禾从𡈼区仃㔹冗穴冋内舟兵出凶瓜刂刃劥芒 廿卬冈冈戏圣叐及収乜囗囘囚史太凬寸不山 巛王五玊巳乆幻廾廾弌号号号号号号礼卜
CNS 11643-2007 Plane 7	1	壼寎韡奭僊僜傔儎儭俚傀僜僚僓傀御僪兟毳 兠圞凜澥劃劘劓劚劇劕劕劉幹寫匲厰愿嗞壼 斋匽嗇嗜嗟喀嗉嗏嘳嗁嗹嘿嘰嗥嘰嘈嗤嘍嚅 嘏嘪嚣嘐嗒嗷嘽嘔嚣嘹圜圞圛鼍墻墧墦墒堤 壖壜嚷齝墨墩覆墲墓墜塈墜壽臺錫蓁橐雞
CNS 11643-2007 Plane 15	1	乙千乣及生么厄冘仈冗凧允凡兊刈匁区匝厄 邓　叫囚个不龙亏　乡屯中乔企　伢伲夙有 冇元安次充凷厄叨　叺扔坔乔夸尬奶存孙龙 尥　町屶岙吕芋　纫处尨杰汉水叉肌辶夲肙 　乪亦宋仪伝关佰佈佳企攸低佬沢罕汾汻

Table 2-26. Ideographs in various character set standards

Standard	Row	Characters
JIS X 0208:1997	16	亜唖娃阿哀愛挨姶逢葵茜穐悪握渥旭葦芦鯵 梓圧斡扱宛姐虻飴絢綾鮎或粟袷安庵按暗案 闇鞍杏以伊位依偉囲夷委威尉惟意慰易椅為 畏異移維緯胃萎衣謂違遺医井亥域育郁磯一 壱溢逸稲茨芋鰯允印咽員因姻引飲淫胤蔭
JIS X 0213:2004 Plane 1	14	俱𠀋㐂丨丯丰亍仡份仿伃伋你佈佉佖佟佪佬 佾侊侔侗侮俉俠倁倂倎倘倧倮偀倻偁傔僌僲 僐僦僧儆儃儋儞儵兊免兕兗㒵冝凃凊凞凢凮 刁㓛刓刕剉剗剡劓勈勉勌勐勖勛勤勰勻匀匇 匜卑卡卣卽厓厝厲吒吧呍咜呫呴呿咈咖咡
JIS X 0213:2004 Plane 2	1	𠂉丂丏丒丩丫丮乀乇么𠂢乑㐆𠂤乚乩亝㐬㐮 亹亻𠆢亼仃仈仐仫仚仱仵伀伖佤伷伾佔佘𠈓 佷佸佺佽侂侅侒侚俦侲侾俅俋俏俒㑪俲倀倐 倓倜倞倢㑨偂偆偎偓偗偣偦偪偰傣傈傒傓傕 傖傜傪𠌫傱傺傻僄僇僳𠎁僎𠍱僔僙僡僩㒒
JIS X 0212-1990	16	丂丄丅丌丒丟丣两丨丫丮丯丰丵乀乁乄乇乑 乚乜乣乨乩乴乵乹乿亍亖亗亝亯亹仃仐仚仛 仠仡仢仨仯仱仳仵份仾仿伀伂伃伈伋伌伒伕 伖众伙伮伱你伳伵伷伹伻伾佀佂佈佉佋佌佒 佔佖佘佟佣佪佬佮佱佷佸佹佺佽佾侁侂侄
KS X 1001:2004	42	伽佳假價加可呵哥嘉嫁家暇架枷柯歌珂痂稼 苛茄街袈訶賈跏軻迦駕刻却各恪慤殼珏脚覺 角閣侃刊墾奸姦干幹懇揀杆柬桿澗癎看磵稈 竿簡肝艮艱諫間乫喝曷渴碣竭葛褐蝎鞨勘坎 堪嵌感憾戡敢柑橄減甘疳監瞰紺邯鑑鑒龕
KS X 1002:2001	55	仮傢咖哿坷宊斝榎檟珈笳耞舸葭謌卻咯埆搉 擱桷偘慳栞榦玕秆茛衎赶迀齦噶楬秸羯蠍鶡 坩埳嵁弇憨撼欿歛泔淦澉矙轗酣鹻韐傋僵壃 忼扛杠橿殭矼穅繈罡羗羫茳豇韁剴匃揩槩玠 磕闓硜賡鏗呿昛秬筥籧胠腒苣莒蕖蘧祛裾

Table 2-26. Ideographs in various character set standards

Standard	Row	Characters
TCVN 6056:1995	42	一丁七万丈三上下不与丐丑且丕世丘丙丞丟 丫中丰串丸丹主丿乂乃久之乍乎乏乖乘乙九 乞也乱乳乾亂了予事二亍于云互五井亙些亟 亡亢交亥亨享京亭人什仁仃仄仆仇今介仍仔 仕他仗付仙仝代令仰件价任份仿企伊伍伎
Unicode—URO	4E	一丁丂七丄丅丆万丈三上下丌不与丏丐丑丒 专且丕世丗丘丙业丛东丝丞丟丠両丢丣两严 並丧丨丩个丫丬中丮丯丰丱串丳临丵丶丷丸 丹为主丼丽举丿乀乁乂乃乄久乆乇么义乊之 乌乍乎乏乐乑乒乓乔乕乖乗乘乙乚乛乜九乞 也习乡乢乣乤乥书乧乨乩乪乫乬乭乮乯买乱 乲乳乴乵乶乷乸乹乺乻乼乽乾乿亀亁亂亃亄 亅了亇予争亊事二亍于亏亐云互亓五井亖亗 亘亙亚些亜亝亞亟亠亡亢亣交亥亦产亨亩亪 享京亭亮亯亰亱亲亳亴亵亶亷亸亹人亻亼亽 亾亿什仁仂仃仄仅仆仇仈仉今介仌仍从仏仐 仑仒仓仔仕他仗付仙仚仛仜仝仞仟仠仡仢代 令以仦仧仨仩仪仫们仭仮仯仰仱仲仳仴仵件 价仸仹仺任仼份仾仿
Unicode—Extension A	34	㐀㐁㐂㐃㐄㐅㐆㐇㐈㐉㐊㐋㐌㐍㐎㐏㐐㐑㐒 㐓㐔㐕㐖㐗㐘㐙㐚㐛㐜㐝㐞㐟㐠㐡㐢㐣㐤㐥 㐦㐧㐨㐩㐪㐫㐬㐭㐮㐯㐰㐱㐲㐳㐴㐵㐶㐷㐸 㐹㐺㐻㐼㐽㐾㐿㑀㑁㑂㑃㑄㑅㑆㑇㑈㑉㑊㑋 㑌㑍㑎㑏㑐㑑㑒㑓㑔㑕㑖㑗㑘㑙㑚㑛㑜㑝㑞 㑟㑠㑡㑢㑣㑤㑥㑦㑧㑨㑩㑪㑫㑬㑭㑮㑯㑰㑱 㑲㑳㑴㑵㑶㑷㑸㑹㑺㑻㑼㑽㑾㑿㒀㒁㒂㒃㒄 㒅㒆㒇㒈㒉㒊㒋㒌㒍㒎㒏㒐㒑㒒㒓㒔㒕㒖㒗 㒘㒙㒚㒛㒜㒝㒞㒟㒠㒡㒢㒣㒤㒥㒦㒧㒨㒩㒪 㒫㒬㒭㒮㒯㒰㒱㒲㒳㒴㒵㒶㒷㒸㒹㒺㒻㒼㒽 㒾㒿㓀㓁㓂㓃㓄㓅㓆㓇㓈㓉㓊㓋㓌㓍㓎㓏㓐 㓑㓒㓓㓔㓕㓖㓗㓘㓙㓚㓛㓜㓝㓞㓟㓠㓡㓢㓣 㓤㓥㓦㓧㓨㓩㓪㓫㓬㓭㓮㓯㓰㓱㓲㓳㓴㓵㓶 㓷㓸㓹㓺㓻㓼㓽㓾㓿

As you can clearly see from Table 2-26, there is quite a wide variety of character set standards available that include ideographs. Chapter 3 will make sense of all these character set standards.

A noteworthy characteristic of ideographs is that they can be quite complex. Believe it or not, they can get much more complex than the sets of 94 or 256 characters just illustrated! Ideographs are composed of radicals and radical-like elements, which can be thought of as building blocks of sorts. Radicals are discussed later in this chapter, in the section "The Structure of Ideographs."

Ideograph Readings

In Japanese, the typical ideograph has at least two readings, and some have more. A reading is simply the pronunciation of a character. For example, the ideograph 生, whose meaning relates to "life," has over 200 readings in Japanese—most of which are used for Japanese given names, which are known for their unusual readings.

Ideograph readings typically come from two sources:

- Language-specific reading
- Borrowed—and usually approximated—reading

The native Japanese reading was how the Japanese pronounced a word before the Chinese influenced their language and writing system. The native Japanese reading is called the *Kun* reading (訓読み *kun yomi*).

The borrowed Chinese reading is the Japanese-language approximation of the original Chinese reading of an ideograph. These borrowed approximate readings are called *On* readings (音読み *on yomi*), *On* being the word for "sound." If a particular ideograph was borrowed more than once, multiple readings can result. Table 2-27 lists several ideographs, along with their respective readings.

Table 2-27. Ideographs and their readings—Japanese

Ideograph	Meaning	On readings	Kun readings
剣	sword	*ken*	*akira, haya, tsurugi, tsutomu*
窓	window	*sō*	*mado*
車	car	*sha*	*kuruma*
万	10,000	*ban, man*	*katsu, kazu, susumu, taka, tsumoru, tsumu, yorozu*
生	life, birth	*sei, shō*	*ari, bu, fu, fuyu, haeru, hayasu, i, ikasu, ikeru, ikiru, iku, ki, mi, nama, nari, nori, o, oki, ou, susumu, taka, ubu, umareru, umu, yo,* and so on
店	store, shop	*ten*	*mise*

So, how does one go about deciding which reading to use? Good question! As you learned earlier, the Japanese mostly borrowed kanji as compounds of two or more kanji, and often use the On reading for such compounds. Conversely, when these same kanji appear in isolation, the Kun reading is often used. Table 2-28 provides some examples of individual kanji and kanji compounds.

Table 2-28. Kanji and kanji compounds—Japanese

Kanji compound	Meaning	Readings
自動車	automobile	*jidōsha*—On readings
車	car	*kuruma*—Kun reading
剣道	Kendo	*kendō*—On readings
剣	sword	*tsurugi*—Kun reading

As with all languages, there are always exceptions to rules! Sometimes you find kanji compounds that use Kun readings for one or all kanji. You may also find kanji in isolation that use On readings. Table 2-29 lists some examples.

Table 2-29. Mixed uses of kanji readings—Japanese

Kanji compound	Meaning	Reading
重箱	nest of boxes	*jūbako*—On plus Kun reading
湯桶	bath ladle	*yutō*—Kun plus On reading
窓口	ticket window	*madoguchi*—Kun plus Kun reading
単	simple, single	*tan*—On reading

Japanese personal names tend to use the Kun readings even though they are in compounds. For example, 藤本 is read *fujimoto* rather than *tōhon*.

The Structure of Ideographs

Ideographs are composed of smaller, primitive units called *radicals*, and other nonradical elements, which are used as building blocks. These elements serve as the most basic units for building ideographs. There are 214 radicals used for indexing ideographs. Several radicals stand alone as single, meaningful ideographs. Table 2-30 provides some examples of radicals, along with several ideographs that can be written with them (examples are taken from Japanese).

Table 2-30. Radicals and ideographs made from them

Radical	Variants	Standalone?	Meaning	Examples
木		Yes	tree	本札朴朶李材条杲林枦栞棚森槝
火	灬	Yes	fire	灯灰灸災炎点無然熊熟熱燃燭爛
水	氵氷	Yes	water	氷永汁江汲沢泉温測港源溢澡濯
辵	辶辶	No	running	辷込辻辺辿迄迅迎近返迚連週還

Note how each radical is placed within ideographs—they are stretched or squeezed so that all of the radicals that constitute an ideograph fit into the general shape of a square. Also note how radicals are positioned within ideographs, specifically on the left, right, top, or bottom.

Radicals and radical-like elements, in turn, are composed of smaller units called *strokes*. A radical can consist of one or more strokes. Sometimes a single stroke is considered a radical. There exists one stroke that is considered a single ideograph: 一, the ideograph that represents the number one. Figure 2-2 shows how a typical ideograph is composed of radicals and strokes.

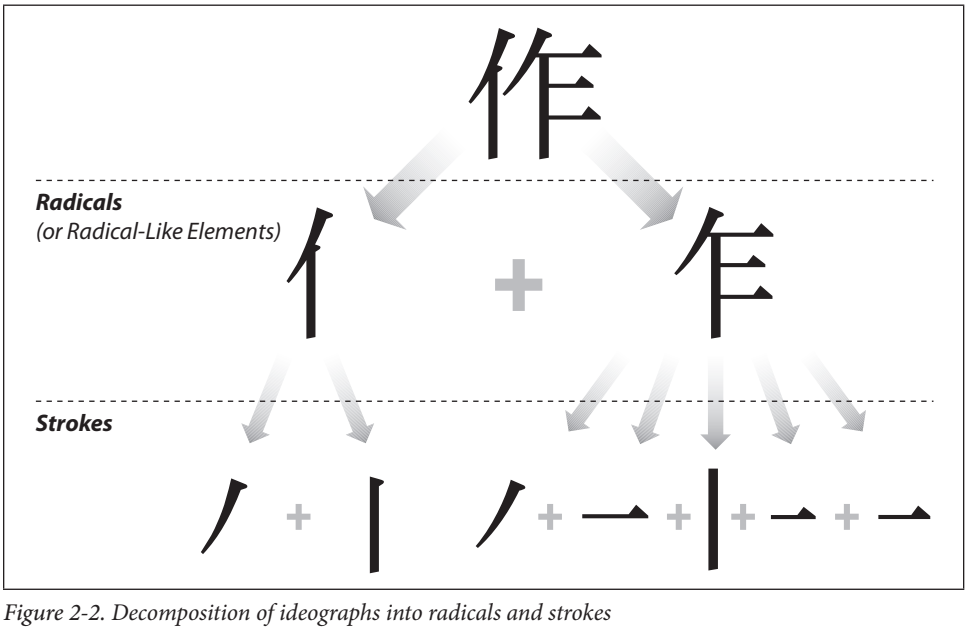


Figure 2-2. Decomposition of ideographs into radicals and strokes

There are many classifications of ideographs, but four are the most common: pictographs, simple ideographs, compound ideographs, and phonetic ideographs. *Pictographs*, the

most basic of the ideographs, are little pictures and usually look much like the object they represent.* Table 2-31 lists examples of pictographs.

Table 2-31. Pictographs

Ideograph	Meaning
日	*sun*
月	*moon*
山	*mountain*
火	*fire*
木	*tree*
車	*car, cart*
口	*mouth, opening*

Whereas pictographs represent concrete objects, *simple ideographs* represent abstract concepts or ideas (as its name suggests), such as numbers and directions.† Table 2-32 lists examples of simple ideographs.

Table 2-32. Simple ideographs

Ideograph	Meaning
上	up
下	down
中	center, middle
一	one
二	two
三	three

Pictographs and simple ideographs can be combined to represent more complex characters and usually reflect the combined meaning of its individual elements. These are called *compound ideographs*.‡ Table 2-33 lists examples of compound ideographs.

* Written as 象形文字 (*xiàngxíng wénzì*) in Chinese, 象形文字 (*shōkei moji*) in Japanese, and 상형문자/象形文字 (*sanghyeong munja*) in Korean.

† Written as 指事文字 (*zhǐshì wénzì*) in Chinese, 指事文字 (*shiji moji*) in Japanese, and 지사문자/指事文字 (*jisa munja*) in Korean.

‡ Written as 会意文字/會意文字 (*huìyì wénzì*) in Chinese, 会意文字 (*kaii moji*) in Japanese, and 회의문자/會意文字 (*hoeui munja*) in Korean.

Table 2-33. Compound ideographs

Ideograph	Components	Meaning
林	木+木	woods
森	木+木+木	forest
明	日+月	clear, bright

Phonetic ideographs account for more than 90% of all ideographs.* They typically have two components: one to indicate reading, and the other to denote etymological meaning. Table 2-34 provides examples that all use the same base reading component, which is the right-side element.

Table 2-34. Phonetic ideographs with common reading component—Japanese

Ideograph	Meaning	Reading	Meaning component	Reading component
銅	copper	*dō*	金 metal	同 *dō*
洞	cave	*dō*	氵 water	同 *dō*
胴	torso	*dō*	肉 organ	同 *dō*
恫	threat	*dō*	忄 heart	同 *dō*

Note that each ideograph uses the 同 component (*dō*) as its reading component. Table 2-35 lists several ideographs that use the same base meaning component.

Table 2-35. Phonetic ideographs with common meaning component—Japanese

Ideograph	Meaning	Reading	Meaning component	Reading component
雰	fog	*fun*	雨 rain	分 *fun*
雲	cloud	*un*	雨 rain	云 *un*
震	shake	*shin*	雨 rain	辰 *shin*
霜	frost	*sō*	雨 rain	相 *sō*

Note that each uses the 雨 ("rain") component for its meaning component, which also serves as the indexing radical. The 雨 component is another example of a radical that can stand alone as a single ideograph.

* Written as 形声文字/形聲文字 (*xíngshēng wénzì*) in Chinese, 形声文字 (*keisei moji*) in Japanese, and 형성문자/形聲文字 (*hyeongseong munja*) in Korean.

Ideographs are subsequently combined with other ideographs as words to form more complex ideas or concepts. These are called *compounds* (熟語 *jukugo* in Japanese) or *ideograph compounds* (漢語 *kango* in Japanese). Table 2-36 lists a few examples. Note that you can decompose words into pieces, each piece being a single ideograph with its own meaning.

Table 2-36. Ideograph compounds

Compound	Meaning	Component ideographs and their meanings
日本	Japan	日 means *sun*, and 本 means *origin—where the sun rises.*
短刀	short sword	短 means *short*, and 刀 means *sword.*
酸素	oxygen	酸 means *acid*, and 素 means *element—the acid element.*
曲線	curve	曲 means *curved*, and 線 means *line—curved line.*
剣道	Kendo	剣 means *sword*, and 道 means *path—the way of the sword.*
自動車	automobile	自 means *self*, 動 means *moving*, and 車 means *car.*
火山	volcano	火 means *fire*, and 山 means *mountain—fire mountain.*

That should give you a sense of how ideographs are constructed and how they are combined with other ideographs to form compounds. But how did they come to be used in Korea and Japan? These and other questions are answered next.

The History of Ideographs

This section provides some brief historical context to explain the development of ideographs and how they came to be used in other regions or cultures, such as Korea, Japan, and Vietnam.

The development of ideographs

Ideographs, believe it or not, share a history similar to that of the Latin alphabet. Both writing systems began thousands of years ago as pictures that encompassed meanings. Whereas the precursors to the Latin alphabet eventually gave up any (nonphonological) semantic association with the characters' shapes, ideographs retained (and further exploited) this feature. Table 2-37 lists several Chinese reference works whose publication dates span a period of approximately 2,000 years.

Table 2-37. The number of ideographs during different periods

Year (AD)	Number of ideographs	Reference work
100	9,353	說文解字
227–239	11,520	聲類
480	18,150	廣雅
543	22,726	玉編
751	26,194	唐韻
1066	31,319	類編
1615	33,179	字彙
1716	47,021	康熙字典
1919	44,908	中華大字典
1969	49,888	中文大辭典
1986	56,000	汉语大字典
1994	85,000	中华字海

Note the nearly ten-fold increase in the number of hanzi over this 2,000-year period. The majority of the ideographs that sprang into existence during this time were phonetic ideographs (see Tables 2-34 and 2-35).

Ideographs in Korea—hanja

One of the earliest cultures to adapt ideographs for their own language was Korea. Although ideographs—called hanja—were extensively used in years past, most Korean writing today is completely done using hangul syllables.

South Korea never abolished hanja entirely, though their use has gradually diminished over time. Keep in mind that the Korean writing system functions well with only hangul syllables and punctuation, which effectively means that hanja are optional. There are some conservative groups within South Korea who would like hanja to be taught more broadly and used throughout society.

The definitive Korean hanja reference is a dictionary entitled 大字源 (대자원 *daejawon*), first published in 1972.

Ideographs in Japan—kanji

There is no evidence to suggest that there was any writing system in place in Japan prior to the introduction of the Chinese script. In fact, it is quite common for writing systems to develop relatively late in the history of languages. A writing system as complex as that used in Chinese is not really an ideal choice for borrowing, but perhaps this was the only writing system from which the Japanese could choose at the time.

The Japanese borrowed ideographs between 222 AD and 1279 AD. During this millennium of borrowing, the Chinese increased their inventory of characters nearly three-fold. Table 2-37 illustrated the number of ideographs that were documented in Chinese at different periods. That table clearly indicated that the Chinese, over a period of about 2,000 years, increased their inventory of characters by roughly a factor of 10 (from 9,353 to 85,000). As you can see, the Japanese were borrowing from the Chinese even while the Chinese were still creating new characters.

The Japanese began borrowing the Chinese script over 16 centuries ago. This massive borrowing took place in three different waves. Several kanji were borrowed repeatedly at different periods, and the reading of each kanji was also borrowed again. This led to different readings for a given kanji depending on which word or words it appeared in, due to dialectal and diachronic differences in China.

The first wave of borrowing took place sometime between 222 and 589 AD by way of Korea, during the Six Dynasties Period in China. Characters borrowed during this period were those used primarily in Buddhist terminology. During this time, the Chinese had between 11,520 and 22,726 hanzi.

The second wave took place between 618 and 907 AD, during the Tang Dynasty in China. Characters borrowed during this period were those used primarily for government and in Confucianism terminology. During this time, the Chinese had between 22,726 and 26,194 hanzi.

The third wave occurred somewhere between 960 and 1279 AD, during the Song Dynasty in China. Characters borrowed during this time were those used in Zen terminology. The Chinese had between 31,319 and 33,179 hanzi by this period.

During all three waves of borrowing, most ideographs were borrowed as compounds of two or more kanji, rather than as isolated characters. It is in this context that you find differences in reading of a particular kanji depending on what word it appears in. For example, the kanji 万, meaning "10,000," can be found in kanji compounds with either the reading *man* or *ban*, such as 万一 (*man* + *ichi*) and 万歳 (*ban* + *zai*—yes, the actual kanji compound for *banzai*!). This (*m*/*b*)*an* alternation would indicate to a trained linguist that these two words were probably borrowed at different periods.

The first two waves of borrowing had the most significant impact on the Japanese lexicon, which accounts for dual On readings for many kanji (*lexicon* simply refers to the individual words that constitute a language). The third wave of borrowing had very little effect on the Japanese lexicon.

I suggest the frontmatter of Jack Halpern's *New Japanese-English Character Dictionary* (Kenkyusha, 1990) as additional reference material on the history and development of the Japanese writing system, more specifically, pages 50a through 60a of that reference. The definitive Japanese kanji reference is a 13-volume dictionary entitled 大漢和辭典 (*dai kanwa jiten*), first published in 1955.

Ideographs in Vietnam—chữ Hán

Vietnam also adopted ideographs for their language, but in a unique way. There are two ways to represent Vietnamese using ideographs. One way is equivalent to Chinese itself (but with approximated readings when pronounced in Vietnamese) and uses characters called *chữ Hán* (genuine ideographs). The other way involves characters that look and feel like ideographs, but were created by the Vietnamese. These are called *chữ Nôm* (𡨸喃). These methods of writing Vietnamese are unique in that they are never used together in the same text: you write using either chữ Hán (Chinese) or chữ Nôm (Vietnamese). More details about chữ Nôm are provided at the end of this chapter.

Both chữ Hán and chữ Nôm were replaced by Quốc ngữ in 1920. Today, chữ Hán and chữ Nôm are still being used—not for the purpose of common or everyday communication, but rather for specialized, religious, or historical purposes.

Ideograph Simplification

Over time, frequently used and complex ideographs tend to simplify. Simplified ideographs are a class of variant. Such simplifications have been different depending on the locale using them. For example, ideographs in their traditional form are still being used in Taiwan. The same holds true for Korea. Also, ideographs in an even more simplified form than that found in Japanese are being used in China and Singapore, although there are some exceptions to this rule. A large number of ideographs are used in an almost identical form in all CJKV locales. Table 2-38 illustrates several ideographs in both traditional and simplified form.

Table 2-38. Traditional and simplified ideographs

Traditional	Simplified—Japan	Simplified—China
廣	広	广
兒	児	儿
兩	両	两
氣	気	气
豐	豊	丰
邊	辺	边
國	国	国
學	学	学
點	点	点
黑	黒	黑

Table 2-38. Traditional and simplified ideographs

Traditional	Simplified—Japan	Simplified—China
佛	仏	佛
骨	骨	骨

Both the simplified and traditional forms of ideographs sometimes coexist within the same character set standard, and some of the pairs from Table 2-38 are such examples—most of them are part of the basic Japanese character set standard, specifically JIS X 0208:1997. You can also see that some simplifications are more extreme than others.

Such simplifications in Japan have led to variants of many characters, and in some character sets both the simplified and traditional forms are included (the examples given earlier are such cases). As an extreme example, let's examine the JIS X 0208:1997 kanji 剣 (Row-Cell 23-85), whose five variant kanji are also encoded within the same character set standard. These variants are listed in Table 2-39 (JIS X 0208:1997 Row-Cell and Unicode scalar values are given).

Table 2-39. Ideograph variants in the same character set

Ideograph	Character code—JIS X 0208:1997	Character code—Unicode
劍	49-88	U+528D
劒	49-89	U+5294
劔	49-90	U+5292
劒	49-91	U+5271
釼	78-63	U+91FC

I should also point out that almost all simplified ideographs have a corresponding traditional form. Some so-called simplified ideographs have been coined as simplified forms, meaning that although they are considered simplified ideographs, there is no corresponding traditional form. Clearly, not all traditional forms have a corresponding simplified form, either because its form is sufficiently simple so as to not require simplification, or its use is not frequent enough to justify simplifcation. Interestingly, a small number of simplified ideographs have more than one traditional form.

Non-Chinese Ideographs

What in the world is a non-Chinese ideograph? Simple. Characters that look, feel, and behave like ideographs, but were not borrowed from China and instead were coined in other regions. The following sections describe this interesting and remarkable phenomenon as it has manifested in Japan, Korea, and Vietnam. Examples are also provided.

Japanese-Made Ideographs—Kokuji

The Japanese have created their own ideographs known as *kokuji* (国字 *kokuji*), literally meaning "national characters," or, more descriptively, "Japanese-made ideographs." Kokuji behave like true ideographs, following the same rules of structure. Specifically, they are composed of radicals, radical-like elements, and strokes, and can be combined with one or more ideographs to form compounds or words. These ideographs were created out of a need for characters not borrowed from China.* Most kokuji are used to represent the names of indigenous Japanese plants and fish. They are also used quite frequently in Japanese place and personal names.

Approximately 200 kokuji have been identified in the basic Japanese character set standard, specifically JIS X 0208:1997. There are even more in the supplemental character sets, specifically JIS X 0212-1990 and JIS X 0213:2004. Table 2-40 provides a few examples of kokuji, and also lists their JIS X 0208:1997 Row-Cell and Unicode scalar values for reference purposes.

Table 2-40. Kokuji examples

Kokuji	Readings	Meanings
鰯 16-83 U+9C2F	*iwashi*	sardine
粂 23-09 U+7C82	*kume*	Used in personal names
込 25-94 U+8FBC	*komu*	(to) move inward
榊 26-71 U+698A	*sakaki*	A species of tree called *sakaki*
働 38-15 U+50CD	*hataraku, dō*[a]	(to) work
峠 38-29 U+5CE0	*tōge*	mountain pass
畑 40-10 U+7551	*hata, hatake*	dry field
枠 47-40 U+67A0	*waku*	frame
凩 49-62 U+51E9	*kogarashi*	cold, wintry wind

a. Considered an On reading.

Additional kokuji were created when the Japanese isolated themselves from the rest of the world for approximately 250 years: from the mid-1600s to the late 1800s. Without direct influence from China, the Japanese resorted to creating their own ideographs as necessary. There is at least one kokuji that was subsequently borrowed by China, specifically 腺 (JIS X 0208:1997 Row-Cell 33-03, which is read *sen*, and means "gland"). In Chinese this

* In fact, some kokuji were even borrowed back by the Chinese as genuine ideographs, which I discuss later.

ideograph is read *xiàn* (GB 2312-80 Row-Cell 47-57). The Unicode scalar value for this ideograph is U+817A.

Seven kokuji have made their way into the standard set of 1,945 kanji called *Jōyō Kanji*, and four are in *Jinmei-yō Kanji* (Chapter 3 provides a full treatment of these and other related character sets). Those in Jōyō Kanji are 込 (25-94), 働 (38-15), 峠 (38-29), 畑 (40-10), 塀 (42-29), 匁 (44-72), and 枠 (47-40). Those in Jinmei-yō Kanji are 笹 (26-91), 凪 (38-68), 柾 (43-79), and 麿 (43-91). Nozomu Ohara (大原望 *ōhara nozomu*) has compiled a list of kokuji, which includes those that are listed in the JIS X 0208:1997 and JIS X 0212-1990 character set standards, along with links to other kokuji-related websites.*

Korean-Made Ideographs—Hanguksik Hanja

Like the Japanese, the Koreans have had the opportunity to create their own ideographs. These are known as *hanguksik hanja* (한국식 한자/韓國式漢字 *hanguksik hanja*). Although you'd expect to find hanguksik hanja only in Korean character set standards, there are approximately 100 hanguksik hanja included in a Chinese character set standard designated GB 12052-89 (you'll understand why after reading about this character set standard in Chapter 3).

Hanguksik hanja—unlike kokuji in Japanese—have many tell-tale signs of their status as non-Chinese ideographs. Table 2-41 lists elements of hanguksik hanja that are used to indicate a final consonant.

Table 2-41. Hanguksik hanja reading elements

Hanguksik hanja element	Reading
乙	L
ㄱ	G
叱	D
ㅇ	NG

Many other hanguksik hanja look and feel like genuine ideographs. It is only after you explore their etymology that you may discover their true Korean origins.

The basic Korean character set standard for use on computers, KS X 1001:2004, includes many hanguksik hanja. The supplemental Korean character set standard, KS X 1002:2001, includes even more hanguksik hanja. Table 2-42 provides some examples of hanguksik hanja, along with their readings and meanings. KS X 1001:2004 Row-Cell and Unicode scalar values are provided for reference purposes.

* *http://homepage2.nifty.com/TAB01645/ohara/*

Table 2-42. Hanguksik hanja examples

Hanguksik hanja	Reading	Meaning
乫 42-65 U+4E6B	갈 *gal*	Used in personal names
畓 51-44 U+7553	답 *dap*	paddy, wet field
乭 52-44 U+4E6D	돌 *dol*	Used in personal and place names
乷 56-37 U+551C	말 *mal*	Used in place names
鐥 64-54 U+9425	선 *seon*	Used in place names
筽 72-04 U+7B7D	오 *o*	Used in place names
岾 79-32 U+5CBE	점 *jeom*	mountain pass[a]

a. Compare this hanguksik hanja with the (Japanese) kokuji 峠 (*tōge*) in Table 2-40. I find it fascinating that Japan and Korea independently coined their own ideograph meaning "mountain pass."

Only one hanguksik hanja, 畓 (답 *dap*), is known to be included in Korea's standard set of 1,800 hanja called *Hanmun Gyoyukyong Gicho Hanja*. This hanguksik hanja is not in the middle school subset of 900 hanja, though.

Vietnamese-Made Ideographs—Chữ Nôm

Unlike Japanese and Korean, in which non-Chinese ideographs are used together with genuine ideographs—a sort of mixing of scripts—Vietnamese has three distinct ways to express its language through writing:

- Latin script (called *Quốc ngữ*)
- Ideographs (called *chữ Hán*)
- Vietnamese-made ideographs (called *chữ Nôm*)

Writing Vietnamese using chữ Hán is considered equivalent to writing in Chinese, not Vietnamese. Using Quốc ngữ or chữ Nôm is considered writing in Vietnamese, not Chinese. For some chữ Nôm characters, there is a corresponding chữ Hán character with the same meaning. Table 2-43 provides a handful of chữ Nôm characters, along with their chữ Hán equivalents (TCVN 5773:1993 and TCVN 6056:1995 Row-Cell codes are provided for chữ Nôm and chữ Hán characters, respectively).

Table 2-43. Chữ Nôm and chữ Hán examples

Chữ Nôm	Reading	Chữ Hán	Reading	Meaning
𠀧 21-47 U+20027	*ba*	三 42-06 U+4E09	*tam*	three
𠓨 29-55 U+219F2	*giữa*	中 42-21 U+4E2D	*trung*	center, middle

Table 2-43. Chữ Nôm and chữ Hán examples

Chữ Nôm	Reading	Chữ Hán	Reading	Meaning
𡨸 34-02 U+21A38	*chữ*	字 50-30 U+5B57	*tự*	character
𣍐 35-77 U+24F93	*trăm*	百 64-02 U+767E	*bá*	hundred

There are times when chữ Hán characters are used in chữ Nôm context (that is, with chữ Nôm characters). Table 2-44 lists two types of chữ Hán characters: those that have different readings depending on context (chữ Nôm versus chữ Hán), and those that have identical readings regardless of context. TCVN 6056:1995 Row-Cell and Unicode scalar values are provided for reference purposes.

Table 2-44. Chữ Hán characters used in chữ Nôm contexts

	Character	Chữ Nôm reading	Chữ Hán reading	Meaning
Unique	主 42-26 U+4E3B	*chúa*	*chủ*	main, primary
	印 45-85 U+5370	*in*	*ấn*	printing
	急 53-14 U+6025	*cấp*	*kíp*	fast, rapid
	所 54-35 U+6240	*thửa*	*sở*	place, location
Identical	文 56-16 U+6587	*văn*	*văn*	sentence
	武 59-22 U+6B66	*vũ*	*vũ*	weapon
	爭 62-44 U+722D	*tranh*	*tranh*	war
	香 76-23 U+9999	*hương*	*hương*	fragrant

Chữ Nôm was the accepted method for writing Vietnamese since the 10th century AD. It was not until the 1920s when chữ Nôm was replaced by Quốc ngữ.

The Vietnamese Nôm Preservation Foundation (會保存遺産喃 *Hội Bảo tồn Di sản Nôm*) was established in 1999 as an effort to preserve this important part of Vietnamese culture, specifically the chữ Nôm characters.*

* *http://nomfoundation.org/*

CHAPTER 3
Character Set Standards

I must first state that achieving a rock-solid understanding of and a deep appreciation for CJKV character set standards—what character classes they include, how many characters they enumerate, how they evolved, which ones are in common use, and so on—forms the foundation on which the remainder of this book is based. Without such a basic understanding, it would be pointless to discuss topics such as encoding methods, input methods, font formats, and typography. This chapter thus represents what I consider to be the *core* or absolutely essential material of this book.

Note that all CJKV character sets can be classified into two basic types, depending on their intended purpose and reason for establishment:

- Noncoded Character Sets—NCSes
- Coded Character Sets—CCSes

For clarification, *noncoded* refers to a character set established without regard to how it would be processed on computer systems, if at all, and *coded* refers to being electronically encoded or computerized. In other words, coded character sets were specifically designed for processing on computer systems. You will soon realize that the characters enumerated in NCSes generally constitute a subset of the characters contained in CCSes, and affect their development. And, to some extent, CCSes can even affect the development of NCSes.

In reading this chapter, I am confident that you will develop a firm understanding about which character classes constitute a given character set, along with information fundamental to dealing with CJKV-related issues. If you discover that you are especially interested in one or more particular CJKV character sets covered in this chapter, either due to a personal interest or to your job requirements, I strongly encourage you to obtain the corresponding character set standard documentation. While this chapter sometimes provides some insights and details not found in the original standard documents, it does not (and, quite frankly, cannot) duplicate or replicate all of the details and information that those documents contain. Still, typical character set standards do not make the most exciting reading material, and they're typically available only in the language used in the

locale for which the character set standard is intended. Much of what is included in their pages, out of necessity, is somewhat mundane. This is simply the nature of standards. To further refine my bluntness, standards can be boring. I have made a concerted effort not to be so in this chapter.

Some character set standards discussed in this chapter may not yet be established—they are in draft form, which means that their designations may change, and in some cases may never be published in final form. Such character sets are indicated by a trailing "X" in the portion of their designation used to specify the year of establishment. As of this writing, affected standards include China's GB/T 13131-2XXX and GB/T 13132-2XXX.

NCS Standards

Long before any CCS standards existed in the CJKV locales (or even before the concept of a CCS standard existed!), several NCS standards were established for pedagogical purposes. These are considered to be the first attempts to limit the number of ideographs in common use.

The NCSes that are described in this chapter include only ideographs. Everyone is expected to learn hiragana and katakana (in Japan) or hangul (in Korea). Only for ideographs, which number in the tens of thousands, is there a need to define a set (and thus, limit the number) of characters that are taught in school or used elsewhere within society.

Chapter 2 provided a brief description and development history of ideographs. If you skipped or missed that chapter and are unfamiliar with ideographs, I strongly suggest going back to read at least that section.

Hanzi in China

The educational system in China requires that students master 3,500 hanzi during their first years of instruction. These hanzi form a subset from a standardized list of 7,000 hanzi defined in 现代汉语通用字表 (*xiàndài hànyǔ tōngyòngzì biǎo*), published on March 25, 1988. We can call this large list *Tōngyòng Hànzì*. Two other hanzi lists further define this 3,500-hanzi subset. The first list, 现代汉语常用字表 (*xiàndài hànyǔ chángyòngzì biǎo*), defines the 2,500 hanzi that are taught during primary school. The second list, 现代汉语次常用字表 (*xiàndài hànyǔ cìchángyòngzì biǎo*), defines an additional 1,000 hanzi that are taught during middle school. We can call these character sets *Chángyòng Hànzì* and *Cìchángyòng Hànzì*. These hanzi lists are commonly abbreviated as 常用字 (*chángyòngzì*) and 次常用字 (*cìchángyòngzì*), respectively, and were published on January 26, 1988. Appendix G provides a complete listing of the 3,500 hanzi defined in 现代汉语常用字表 and 现代汉语次常用字表. The dictionary entitled 汉字写法规范字典 (*hànzì xiěfǎ guīfàn zìdiǎn*) is useful in that it includes both sets of hanzi, and differentiates them through the use of annotations.

In addition, the Chinese government published a document entitled *Simplified Character Table* (简化字总表 *jiǎnhuàzì zǒngbiǎo*) that enumerates 2,249 simplified hanzi (and illustrates the traditional forms from which they were derived—some simplified hanzi were derived from more than one traditional hanzi). This document is divided into three tables, the descriptions of which are listed in Table 3-1.

Table 3-1. Simplified character table contents

Table	Characters	Description
1	350	Independently simplified hanzi
2	146	Simplified components used in other hanzi[a]
3	1,753	Hanzi simplified by using simplified components from "Table 2" of the *Simplified Character Table*

a. Among these, 132 are also used as standalone hanzi.

There has been more than one version of this document, the most recent of which was published in 1986. It is important to note that its development has not been static. Some minor corrections and adjustments have been made over the years, one of which is known to have caused an error in a coded character set, specifically in GB/T 12345-90 (to be covered later in this chapter). As you will learn in this chapter, the propagation of errors from one character set to another—whether coded, noncoded, or both—is something that *can* and *does* occur.

Be aware that there are many hanzi used in China that do not require further simplification—only those that were deemed frequently used and complex were simplified.

Hanzi in Taiwan

The basic set of hanzi in Taiwan is listed in a table called 常用國字標準字體表 (*chángyòng guózì biāozhǔn zìtǐ biǎo*), which enumerates 4,808 hanzi. An additional set of 6,341 hanzi is defined in a table called 次常用國字標準字體表 (*cìchángyòng guózì biāozhǔn zìtǐ biǎo*), 18,480 rare hanzi are defined in a table called 罕用字體表 (*hǎnyòng zìtǐ biǎo*), and 18,609 hanzi variants are defined in a table called 異體國字字表 (*yìtǐ guózì zìbiǎo*). All of these hanzi tables were established by Taiwan's Ministry of Education (教育部 *jiàoyùbù*). Like with the similarly named hanzi lists used in China, the first two of these character sets used in Taiwan can be referred to as *Chángyòng Hànzì* and *Cìchángyòng Hànzì*.

Table 3-2 lists these standards, along with their dates of establishment. These tables, when added together, create a set of 48,238 hanzi.

Table 3-2. Hanzi lists in Taiwan

Standard	Nickname	Date of establishment	Number of hanzi
常用國字標準字體表	甲表 (*jiǎbiǎo*)	September 2, 1982	4,808
次常用國字標準字體表	乙表 (*yǐbiǎo*)	December 20, 1982	6,341

Table 3-2. Hanzi lists in Taiwan

Standard	Nickname	Date of establishment	Number of hanzi
罕用字體表	丙表 (*bǐngbiǎo*)	October 10, 1983	18,480
異體國字字表	*none*	March 29, 1984	18,609

These hanzi lists will become useful when discussing the CNS 11643-2007 and CCCII coded character set standards from Taiwan later in this chapter. Appendix H provides a complete listing of the hanzi that make up the first two lists, specifically Chángyòng Hànzì and Cìchángyòng Hànzì.

Compared to other CJKV locales, Taiwan has clearly established NCSes with the most characters. In fact, the total number of their hanzi is very close to the total number of hanzi in one of the largest CCSes from Taiwan, specifically CNS 11643-2007, which includes 69,134 hanzi in 13 planes.

Kanji in Japan

Noncoded Japanese character sets include *Gakushū Kanji* (formerly *Kyōiku Kanji*)—the 1,006 kanji formally taught during the first six grades in Japanese schools; *Jōyō Kanji* (formerly *Tōyō Kanji*)—the 1,945 kanji designated by the Japanese government as the ones to be used in public documents such as newspapers; *Jinmei-yō Kanji*—the 983 kanji sanctioned by the Japanese government for use in writing personal names; and *NLC Kanji*[*]—the 1,022 kanji beyond Jōyō Kanji that have been deemed useful. The growth and development of these character sets are listed in Table 3-3 (note that some were renamed).

Table 3-3. Evolving kanji lists in Japan

Year	Kyōiku Kanji	Tōyō Kanji	Jinmei-yō Kanji	NLC Kanji
1946		1,850[a]		
1948	881			
1951			92	
1976			120	
1977	996—Gakushū Kanji			
1981		1,945—Jōyō Kanji	166	
1990			284	
1992	1,006[b]			
1997			285	

* NLC stands for National Language Council.

Table 3-3. Evolving kanji lists in Japan

Year	Kyōiku Kanji	Tōyō Kanji	Jinmei-yō Kanji	NLC Kanji
2000				1,022
2004			983[c]	

a. The corresponding glyph table (当用漢字字体表 *tōyō kanji jitai hyō*) was published in 1949, and likewise, the corresponding reading table (当用漢字音訓表 *tōyō kanji onkun hyō*) was published in 1948.

b. This set was established in 1989, but not fully implemented until 1992.

c. One kanji was added in 02/2004, one in 06/2004, three in 07/2004, and 488 in 09/2004. In addition, there were 205 Jōyō Kanji variant forms added, in 09/2004, that were deemed acceptable for use in personal names. It was a busy year.

There is some overlap among these character sets. Gakushū Kanji is a subset of Jōyō Kanji (likewise, Kyōiku Kanji was a subset of Tōyō Kanji).

Table 3-4 shows how you write the names of these character sets in native Japanese orthography, and indicates their meaning.

Table 3-4. The meanings of noncoded Japanese character set standards

Character set	In Japanese	Meaning	Number of kanji
Kyōiku Kanji	教育漢字	Instructional kanji	881
└→ Gakushū Kanji	学習漢字	Educational kanji	1,006
Tōyō Kanji	当用漢字	Common use kanji	1,850
└→ Jōyō Kanji	常用漢字	Everyday use kanji	1,945
Jinmei-yō Kanji	人名用漢字	Personal name use kanji	983
NLC Kanji	表外漢字[a]	Beyond (the Jōyō Kanji) table kanji	1,022

a. This is an abbreviated name. The full name is 常用漢字表以外の漢字 (*jōyō kanji hyō igai-no kanji*).

While Table 3-3 appears to show that the Gakushū Kanji list gained only 10 kanji between 1977 and 1992, the list also experienced some internal shifts. Gakushū Kanji and Kyōiku Kanji can be decomposed into six sets, each corresponding to the grade of school during which they are formally taught. Table 3-5 indicates the six grade levels on the left, along with the number of kanji taught during each one—this is done for Kyōiku Kanji and both versions of Gakushū Kanji.

Table 3-5. The development of Gakushū Kanji

Grade	1958—881 kanji[a]	1977—996 kanji	1992—1,006 kanji
1	46	76	80
2	105	145	160
3	187	195	200
4	205	195	200

Table 3-5. The development of Gakushū Kanji

Grade	1958—881 kanji[a]	1977—996 kanji	1992—1,006 kanji
5	194	195	185
6	144	190	181

a. Kyōiku Kanji was not divided into the six grade levels until 1958.

The general trend shown by Table 3-5 is that more kanji, although not significantly more, are now taught in the earlier grades. The Jōyō Kanji character set is currently under revision and is expected to grow.

Appendix J provides complete listings of the Jōyō Kanji, Gakushū Kanji, Jinmei-yō Kanji, and NLC Kanji character sets.

Hanja in Korea

Korea has defined a list of hanja called *Hanmun Gyoyukyong Gicho Hanja* (한문 교육용 기초 한자/漢文敎育用基礎漢字 *hanmun gyoyukyong gicho hanja*), and enumerates the 1,800 hanja that students are expected to learn during their school years.* The first 900 of these hanja are expected to be learned by students during middle school; the remaining 900 are expected to be learned through high school. These hanja lists were established on August 16, 1972. Forty-four of the 1,800 hanja were replaced on December 30, 2000, which kept the number of hanja steady at 1,800.

Appendix K provides a complete printout of the 1,800 Hanmun Gyoyukyong Gicho Hanja (expanded to accommodate the KS X 1001:2004 character set standard—later in this chapter, you'll learn and appreciate why this expansion is necessary).

The Korean Supreme Court (대법원/大法院 *daebeobwon*) also defined, at various periods, lists of hanja that are considered acceptable for use in writing Korean names—these lists are called *Inmyeong-yong Hanja* (인명용 한자/人名用漢字 *inmyeongyong hanja*). The latest list enumerates 2,964 hanja and was established in July of 1994. Previous versions of this list were established in January and March of 1991.

CCS Standards

Proliferation of computer systems necessitated the creation of coded character set standards. Initially, each vendor (such as IBM, Fujitsu, Hitachi, and so on) established their own corporate standard for use only with their products. But, a computer in isolation is not terribly useful. Interoperability or interchange with other computers became a necessity.

* In 1967, the Korean Newspaper Association defined a set of 2,000 hanja that was referred to as *Sangyong Hanja* (상용 한자/常用漢字 *sangyong hanja*).

The first multiple-byte national coded character set standard among the CJKV locales was established by the Japanese Standards Association (JSA) on January 1, 1978 and was designated JIS C 6226-1978. Without a doubt, the birth of this character set standard sent waves throughout the CJKV locales.

Other CJKV locales, such as Korea and China, inspired by the success of JIS C 6226-1978, followed soon after by imitating the Japanese standard, and in some cases copied more than merely the encoding method or arrangement of characters. It has been claimed, for example, that Taiwan's Big Five character set borrowed many kanji forms from Japan's JIS C 6226-1978 character set.

At the beginning of each subsection, which generally corresponds to each region, I indicate what character sets are the most important. Some character sets come and go, some are useful for historical purposes, but one or two are generally considered more important, either because they are the most commonly used character sets or are mandated by that region's government.

In addition, because of the extreme importance of Unicode today, almost all of the following sections will include a subsection about Unicode compatibility with one or more important standards that are covered. These subsections will indicate what version of Unicode provides full support for their character sets, and also will point out any peculiarities or corner cases that deserve mentioning. This book is written to be part book and part reference material. Although Unicode is thoroughly covered in the latter part of this chapter, it is mentioned early on due to the important nature of Unicode compatibility with respect to each region's character set standards. Some of the information I will provide in the sections about Unicode compatibility is not available elsewhere, because it was discovered through my own work or research or may be cleverly or unintentionally obfuscated to prevent detection.

National Coded Character Set Standards Overview

The CCSes described in this section constitute those maintained by a government or a government-sanctioned organization within a given country and are considered the standard character sets for the locale. In addition, some character set standards form the foundation from which other character set standards are derived, such as international or vendor character set standards. (Note that vendor character set standards are covered in Appendix E.)

Tables 3-6 through 3-12 summarize the national character sets described in this chapter, along with the number and classes of characters enumerated by each. I have decided to use separate tables for each locale because one large table would have been overwhelming. Also note the use of levels in these tables. Some of the early CCSes separated their ideographs into two levels, or sometimes planes. Given the current state of CCSes in use today, noting the first two levels or planes is primarily of historical interest.

Table 3-6. Chinese character set standards—China

Character set	Level 1	Level 2	Additional hanzi	Symbols	Control codes
GB 1988-89[a]				94	34
GB 2312-80	3,755	3,008		682	
GB 6345.1-86	3,755	3,008		814	
GB 8565.2-88	3,755	3,008	636	751	
ISO-IR-165:1992	3,755	3,008	775	905	
GB/T 12345-90[b]	3,755	3,008	103	843	
GB 7589-87	7,237				
GB/T 13131-2XXX	7,237				
GB 7590-87	7,039				
GB/T 13132-2XXX	7,039				
GBK	3,755	3,008	14,240	883	
GB 18030-2000	3,755	3,008	20,770	894	
GB 18030-2005	3,755	3,008	63,481[c]	6,184[d]	

a. Also known as GB-Roman.

b. Although the number of Level 1 and 2 hanzi suggests that the content is identical to GB 2312-80, GB/T 12345-90 includes traditional forms at the same code point as simplified forms.

c. GB 18030-2005 acknowledges for the first time Unicode's CJK Unified Ideographs Extension B, and includes glyphs for its 42,711 characters.

d. GB 18030-2005 acknowledges for the first time six regional scripts: Korean, Mongolian, Tai Le, Tibetan, Uyghur, and Yi. For these six regional scripts, 5,290 glyphs are printed in the GB 18030-2005 manual proper, and are thus included in this figure.

Table 3-7. Chinese character set standards—Taiwan

Character set	Level 1	Level 2	Additional hanzi	Symbols	Control codes
Big Five	5,401	7,652		441	
Big Five Plus	5,401	7,652	7,619	913	
CNS 5205-1989[a]				94	34
CNS 11643-1986	5,401	7,650	13,488[b]	684	
CNS 11643-1992	5,401	7,650	34,976[c]	684	
CNS 11643-2007	5,401	7,650	56,083[d]	1,605	
CCCII[e]	75,684				

a. Also known as CNS-Roman.

b. Planes 14 and 15.

c. Planes 3 through 7.

d. Planes 3 through 7, and 10 through 15.

e. The "Level 1" figure represents the total number of characters.

Table 3-8. Chinese character set standards—Hong Kong

Character set	Base character set	Additional hanzi	Other characters
Hong Kong GCCS	Big Five	3,049	0
Hong Kong SCS-1999	Big Five	4,261	441
Hong Kong SCS-2001	Big Five	4,377	441
Hong Kong SCS-2004	Big Five	4,500	441
Hong Kong SCS-2008	Big Five	4,568	441

Table 3-9. Japanese character set standards

Character set	Level 1	Level 2	Additional kanji	Symbols	Control codes
JIS X 0201-1997[a]				157[b]	34
JIS C 6226-1978	2,965	3,384		453	
JIS X 0208-1983	2,965	3,384	4[c]	524	
JIS X 0208-1990	2,965	3,384	6[c]	524	
JIS X 0208:1997	2,965	3,384	6[c]	524	
JIS X 0212-1990	5,801			266	
JIS X 0213:2000	1,249[d]	2,436[e]		659	
JIS X 0213:2004	1,249[d]	2,436[e]	10[f]	659	

a. Part of this standard includes JIS-Roman.

b. This figure includes 94 JIS-Roman characters plus 63 half-width katakana characters.

c. These additional kanji are considered part of JIS Level 2, because they immediately follow its 3,384 kanji. It is thus common to see in other reference works the figures 3,388 or 3,390 as the number of JIS Level 2 kanji, depending on its vintage (1983 versus 1990 or 1997).

d. JIS Level 3.

e. JIS Level 4.

f. These 10 additional kanji are considered part of JIS Level 3, because they are all in Plane 1. It is thus common to see in other reference works the figures 1,249 or 1,259 as the number of JIS Level 3 kanji, depending on its vintage (2000 versus 2004).

Table 3-10. Korean character set standards

Character set	Country	Hangul	Hanja	Symbols	Control codes
KS X 1003:1993[a]	South Korea			94	34
KS X 1001:1992	South Korea	2,350	4,888	986	
KS X 1001:1998	South Korea	2,350	4,888	988	
KS X 1001:2002	South Korea	2,350	4,888	989	
KS X 1001:2004	South Korea	2,350	4,888	989	
KS X 1002:1991	South Korea	3,605[b]	2,856	1,188	
KS X 1002:2001	South Korea	3,605[b]	2,856	1,188	
KPS 9566-97	North Korea	2,679	4,653	927	

Table 3-10. Korean character set standards

Character set	Country	Hangul	Hanja	Symbols	Control codes
KPS 10721-2000[c]	North Korea		19,469		
GB 12052-89	China	5,203[d]	94	682	

a. Also known as KS-Roman.

b. These 3,605 hangul are split into two levels, enumerating 1,930 and 1,675 characters, respectively. The second set of hangul (1,675 characters) are considered to be ancient hangul.

c. The exact number of characters in this standard is unknown, but what is known is that 19,469 of its hanja map to Unicode.

d. These 5,203 hangul are split into three levels, enumerating 2,068, 1,356, and 1,779 characters each.

Table 3-11. Vietnamese character set standards

Character set	Ideographs	Symbols	Control codes
TCVN 5712:1993[a]		233[b]	34
TCVN 5773:1993	2,357		
TCVN 6056:1995	3,311		

a. Also known as TCVN-Roman.

b. This figure includes 94 ASCII characters plus 139 additional (mostly accented) characters, 5 of which are combining marks.

Table 3-12. Other national character set standards

Character set	Country	Total characters	Control codes
ASCII	USA	94	34
ANSI Z39.64-1989	USA	15,686	

What a list of standards, huh? Of course, for Table 3-12, there are many more character set standards than those listed—only those that relate to this book are shown. In any case, after you have carefully read this chapter, Tables 3-6 through 3-12 will no longer seem overwhelming. They are also useful for general reference purposes, so be sure to dog-ear or bookmark these pages.

The national standards that are based on ISO 10646—specifically GB 13000.1-93, CNS 14649-1:2002, CNS 14649-2:2003, JIS X 0221-1995, JIS X 0221-1:2001, JIS X 0221:2007, and KS X 1005-1:1995—are covered in the section entitled "International Character Set Standards." The terms *Level 1* and *Level 2* have not yet been described. They simply refer to the two such groups of ideographs usually defined within each CJKV character set standard. Level 1 typically contains frequently used ideographs, whereas Level 2 contains less–frequently used ideographs. Some character sets, such as JIS X 0212-1990 and CNS 11643-2007, contain only a single block of ideographs or consist of multiple planes.

ASCII

Most readers of this book are familiar with the ASCII character set, so it is a good place to begin our discussion of coded character set standards and will serve as a common point of reference.

The ASCII character set is covered in this book because it is quite often mixed with CJKV characters within text. Note, however, that the ASCII character set standard is not specific to any CJKV locale.

ASCII stands for *American Standard Code for Information Interchange*. The ASCII character set standard is described in the standard designated ANSI X3.4-1986;* it is the U.S. version and at the same time the International Reference Version (IRV) of ISO 646:1991,† which defines the framework for related national standards.

The ASCII character set is composed of 128 characters, 94 of which are considered printable. There are also 34 other characters, which include a space character and many control characters, such as Tab, Escape, Shift-in, and so on, which are defined in ISO 6429:1992, entitled *Information Technology—Control Functions for Coded Character Sets*. The control codes are technically not part of ASCII nor ISO 646:1991. Table 3-13 lists the 94 printable ASCII characters.

Table 3-13. The ASCII character set

Character class	Characters
Lowercase Latin	abcdefghijklmnopqrstuvwxyz
Uppercase Latin	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Numerals	0123456789
Symbols	!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

Most of these printable characters are also used in EBCDIC, one of the encoding methods covered in Chapter 4. The binary nature of computers allows these 128 characters to be represented using 7 bits, but because computers evolved through processing information in 8-bit segments (a typical *byte*), these 128 ASCII characters are usually represented by 8-bit units in which the 8th bit (also known as the highest-order bit) is set to `0` (zero). Other character sets often incorporate the characters of the ASCII character set, sometimes with minor locale-specific adjustments.

* ANSI is short for *American National Standards Institute*; an earlier version of this standard was designated ANSI X3.4-1977.

† ISO is short for *International Organization for Standardization*; an earlier version of this standard was designated ISO 646:1983.

ASCII Variations

There are, as of this writing, 15 extensions of the ASCII character sets, all approved by and published through ISO. These character sets contain the ASCII character set as their common base, plus additional characters. Extended ASCII character sets are used to represent other writing systems, such as Arabic, Cyrillic, Greek, Hebrew, and Thai. There is also an extensive collection of additional Latin characters, which are usually additional symbols and accented versions of other Latin characters.

Eight-bit representations can handle 128 more characters than 7-bit representations—the reality is that they handle only up to 94 or 96 additional characters. The documents ISO 8859 Parts 1 through 16 (*Information Processing—8-Bit Single-Byte Coded Graphic Character Sets*) describe character sets that can be encoded in the additional 128 positions when an 8-bit representation is used.* Table 3-14 lists the contents of each of the 15 parts of ISO 8859, indicating what languages are supported by each.

Table 3-14. The 15 parts of ISO 8859

Part	Year	Contents	Languages
1	1998	Latin alphabet No. 1	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish
2	1999	Latin alphabet No. 2	Albanian, Czech, English, German, Hungarian, Polish, Romanian, Serbo-Croatian, Slovak, Slovene
3	1999	Latin alphabet No. 3	Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish
4	1998	Latin alphabet No. 4	Danish, English, Estonian, Finnish, German, Greenlandic, Lappish, Latvian, Lithuanian, Swedish, Norwegian
5	1999	Latin/Cyrillic alphabet	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo-Croatian, Ukrainian
6	1999	Latin/Arabic alphabet	Arabic
7	2003	Latin/Greek alphabet	Greek
8	1999	Latin/Hebrew alphabet	Hebrew
9	1999	Latin alphabet No. 5	Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish
10	1998	Latin alphabet No. 6	Danish, English, Estonian, Finnish, German, Greenlandic, Lappish, Latvian, Lithuanian, Swedish, Norwegian
11	2001	Latin/Thai alphabet	Thai
13	1998	Latin alphabet No. 7	Baltic Rim
14	1998	Latin alphabet No. 8	Celtic

* Note that Part 12 does not exist, at least as of this writing. It was reserved for Indic scripts, and at this point will never be used due to Unicode.

Table 3-14. The 15 parts of ISO 8859

Part	Year	Contents	Languages
15	1999	Latin alphabet No. 9	Part 1 revision
16	2001	Latin alphabet No. 10	Albanian, Croatian, Finnish, French, German, Hungarian, Irish Gaelic, Italian, Polish, Romanian, Slovenian

Table 3-15 lists the 95 additional non-ASCII characters from ISO 8859-1:1998 (also known as ISO Latin-1 or ISO-8859-1). Appendix M provides a complete ISO 8859-1:1998 code table.

Table 3-15. ISO 8859-1:1998 character samples

Character class	Characters
Lowercase Latin	àáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
Uppercase Latin	ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞß
Symbols	¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶•¸¹º»¼½¾¿×÷

These characters, as you can probably guess, are not that useful when working with CJKV text. In fact, many of them are not found in CJKV CCSes. This table simply illustrates the types of characters available in the ISO 8859 series. Note again that these additional ASCII character sets require a full 8 bits per character for encoding because they contain far more than 128 characters.

CJKV-Roman

The Chinese, Japanese, Koreans, and Vietnamese have developed their own variants of the ASCII character set, known as GB-Roman (from GB 1988-89), CNS-Roman (from CNS 5205-1989), JIS-Roman (from JIS X 0201-1997), KS-Roman (from KS X 1003:1993), and TCVN-Roman (from TCVN 5712:1993), respectively. Or, these can be collectively referred to as CJKV-Roman. These character sets, like ASCII, consist of 94 printable characters, but there are some minor differences.* The characters that differ are indicated in Table 3-16.

Table 3-16. Special CJKV-Roman characters

Character code	ASCII[a]	GB-Roman	CNS-Roman[b]	JIS-Roman	KS-Roman
0x24	$ (dollar)	¥ (yuan)	$	$	$
0x5C	\ (backslash)	\	\	¥ (yen)	₩ (won)

* But one, specifically TCVN 5712:1993, contains more than these 94 characters.

Table 3-16. Special CJKV-Roman characters

Character code	ASCII[a]	GB-Roman	CNS-Roman[b]	JIS-Roman	KS-Roman
0x7E	~ (tilde)[c]	‾ (overline)	‾	‾	‾

a. TCVN-Roman is identical to ASCII as far as these three characters are concerned.

b. CNS-Roman is ambiguous with regard to glyphs. The glyphs shown in this column were made consistent with the other CJKV-Roman character sets.

c. The vertical positioning of the tilde may vary depending on the implementation.

Because the difference between ASCII and the CJKV-Roman character sets is minor, they are usually treated as the same throughout this book. You will also find that most terminal software supports only one of these character sets. This means that terminals that support only JIS-Roman display the ASCII backslash as the JIS-Roman "yen" currency symbol. For systems that require the backslash, such as MS-DOS for indicating directory hierarchy, the yen symbol is used instead. Stranger yet, Perl programs displayed on a terminal that supports GB-Roman would have variables prefixed with "yuan" currency symbols (instead of the customary "dollar" currency symbol). You will also find that most CJKV software supports CJKV-Roman instead of ASCII. It is possible that the computer supports both ASCII and CJKV-Roman. Changing the display from CJKV-Roman to ASCII (and vice versa), though, may be as simple as changing the display font. You will learn in the next chapter that this is because ASCII and CJKV-Roman almost always occupy the same encoding space, which can actually lead to code conversion problems when dealing with Unicode.

It is important to realize that character set standards do not prescribe the widths of characters—it is simply customary to associate characters with specific widths, usually half- and full-width.

The standard designated GB 1988-89, entitled *Information Processing—7-Bit Coded Character Set for Information Interchange* (信息处理—信息交换用七位编码字符集 *xìnxī chǔlǐ—xìnxī jiāohuàn yòng qīwèi biānmǎ zìfújí*), and established on July 1, 1990, contains the definition of the GB-Roman character set.* This manual is virtually identical to ISO 646:1991, except that it is written in Chinese.

The standard designated CNS 5205-1989, entitled *Information Processing—7-Bit Coded Character Set for Information Interchange* (資訊處理及交換用七數元碼字元集 *zīxùn chǔlǐ jí jiāohuàn yòng qīshùyuán mǎzìyuánjí*), contains the definition of the CNS-Roman character set.† This manual is virtually identical to ISO 646:1991, except that it is written in Chinese.

The standard designated JIS X 0201-1997, entitled *7-Bit and 8-Bit Coded Character Sets for Information Interchange* (7 ビット及び 8 ビットの情報交換用符号化文字集合

* The original version of this standard was designated GB 1988-80.

† Earlier versions of this standard were dated 1980, 1981, and 1983.

nana-bitto oyobi hachi-bitto no jōhō kōkan yō fugōka moji shūgō), and established on January 20, 1997, provides the definition for the JIS-Roman character set.* Like GB 1988-89, this manual is virtually identical to ISO 646:1991, except that it is written in Japanese and defines the extensions for half-width katakana.

The standard designated KS X 1003:1993, entitled *Code for Information Interchange* (정보 교환용 부호 (로마 문자) *jeongbo gyohwanyong buho (roma munja)*), and established on January 6, 1993, contains the definition of the KS-Roman character set.† Like GB 1988-89, this manual is identical to ISO 646:1991, except that it is written in Korean.

The standard designated TCVN 5712:1993, *Công Nghệ Thông Tin—Bộ Mã Chuẩn 8-Bit Kí Tự Việt Dùng Trong Trao Đổi Thông Tin* (*Information Technology—Vietnamese 8-Bit Standard Coded Character Set for Information Interchange*), established on May 12, 1993, contains the definition of the TCVN-Roman character set. TCVN-Roman contains the basic 94 ASCII characters, plus up to 139 additional characters, most of which are adorned with diacritic marks (and represent all possible Quốc ngữ characters). Five of these 139 additional characters are combining marks that indicate tone.

Common CJKV-Roman issues

The glyph differences that are shown in Table 3-16 have caused more problems than one could imagine, especially when Unicode is involved. The best example, which is also the most commonly encountered problem with regard to this issue, is how 0x5C is handled in Japanese. According to JIS-Roman, 0x5C is the "yen" currency symbol. For contexts that require the use of the "backslash" symbol, which is the corresponding glyph for ASCII at the same code point, 0x5C, it is expected that the "yen" currency symbol be used as the glyph. The semantics of the "yen" currency and "backslash" symbols couldn't be more different. What is being interpreted in such contexts is the code point, not the glyph. In a non-Unicode context, there are no issues, except for the lack of a code point that displays a "backslash" glyph. Some vendor extensions of JIS X 0208:1997, such as the Shift-JIS encoding for Apple's Mac OS, encodes the "backslash" symbol at 0x80.

In a Unicode context, however, the "yen" currency symbol is more properly encoded at U+00A5, which corresponds to ISO 8859-1:1998 0xA5. The "backslash" character is encoded at U+005C.

Because there are legacy issues with which developers must grapple, some Unicode-based font implementations may continue to map both U+005C and U+00A5 to the "yen" currency symbol. Unfortunately there is no easy solution to this seemingly complex problem other than knowing that it exists.

One possible way to work around this issue, when one must explicitly use one of these glyphs but not the other, is to use the Unicode code points that have no such legacy

* JIS X 0201-1997 was previously designated JIS X 0201-1976 (which itself was reaffirmed in 1984 and in 1989).

† This standard was previously designated KS C 5636-1993. The original version, KS C 5636-1989, was established on April 22, 1989.

issues, such as the full-width "yen" currency and "backslash" symbols, which are encoded at U+FFE5 and U+FF3C, respectively. Of course, if everything is done using Unicode, there is no ambiguity.

Chinese Character Set Standards—China

As you learned earlier in this chapter, Japan was the first to develop and implement a multiple-byte national character set. The other major CJKV locales—China, Taiwan, and Korea—soon followed by developing their own. This section describes the character set standards established by China, or more specifically, the People's Republic of China, or PRC (中华人民共和国 *zhōnghuá rénmín gònghé guó*).

All Chinese character set standards begin with the designator GB, which stands for "Guo Biao" (国标 *guóbiāo*), which itself is short for "Guojia Biaozhun" (国家标准 *guójiā biāozhǔn*) and means "National Standard." Some GB standards have a "/T" tacked onto the "GB" to form "GB/T." The "T" here stands for "Tui" (推 *tuī*), which is short for "Tuijian" (推荐 *tuījiàn*) and means "recommended" (as opposed to "forced" or "mandatory"). Contrary to popular belief, the "T" does *not* stand for "traditional" (as in "traditional hanzi").

The two most important Chinese character set standards are GB 2312-80 and GB 18030-2005. The former forms the foundation for all Chinese character sets that followed, and the latter's support has been declared mandatory by the Chinese government. While reading the following sections, please pay special attention to these two character set standards.

GB 2312-80—where it all began

This character set standard, established on May 1, 1981 by the People's Republic of China (PRC), enumerates 7,445 characters. Its official title is *Code of Chinese Graphic Character Set for Information Interchange Primary Set* (信息交换用汉字编码字符集—基本集 *xìnxī jiāohuàn yòng hànzì qīwèi biānmǎ zìfújí—jīběnjí*). Table 3-17 lists how characters are allocated to each row.

Table 3-17. The GB 2312-80 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	72	Numerals 1–20 with period, parenthesized numerals 1–20, encircled numerals 1–10, parenthesized hanzi numerals 1–20, uppercase Roman numerals 1–12
3	94	Full-width GB 1988-89 (GB-Roman; equivalent to ASCII)
4	83	Hiragana
5	86	Katakana
6	48	Upper- and lowercase Greek alphabet
7	66	Upper- and lowercase Cyrillic alphabet

Table 3-17. The GB 2312-80 character set

Row	Characters	Content
8	63	26 full-width Pinyin characters, 37 zhuyin (bopomofo) characters
9	76	Full-width line-drawing elements
10–15	0	Unassigned
16–55	3,755	Level 1 hanzi (last is 55-89)
56–87	3,008	Level 2 hanzi (last is 87-94)
88–94	0	Unassigned

Level 1 hanzi (第一级汉字 *dìyījí hànzì*) are arranged by reading. *Level 2 hanzi* (第二级汉字 *dìèrjí hànzì*) are arranged by radical, and then by total number of strokes. To give you a feel for the GB 2312-80 character set, Table 3-18 briefly illustrates the types of characters in GB 2312-80.

Table 3-18. GB 2312-80 character samples

Character class	Sample characters		
Miscellaneous symbols	、。·ˉˇ¨〃々—	…	□■△▲※→←↑↓〓
Annotated numerals	1. 2. 3. 4. 5. 6. 7. 8. 9. 10.	…	Ⅲ Ⅳ Ⅴ Ⅵ Ⅶ Ⅷ Ⅸ Ⅹ Ⅺ Ⅻ
Full-width GB-Roman	! " ＃￥%＆′ （） *	…	u v w x y z ｛｜｝ ¯
Hiragana	ぁあぃいぅうぇえぉお	…	りるれろゎわゐゑをん
Katakana	ァアィイゥウェエォオ	…	ロヮワヰヱヲンヴカヶ
Greek characters	Α Β Γ Δ Ε Ζ Η Θ Ι Κ	…	ο π ρ σ τ υ φ χ ψ ω
Cyrillic characters	А Б В Г Д Е Ё Ж З И	…	ц ч ш щ ъ ы ь э ю я
Full-width Pinyin	ā á ǎ à ē é ě è ī í	…	ǔ ù ü ê ɑ ḿ ń ň ǹ g
Zhuyin (bopomofo)	ㄅㄆㄇㄈㄉㄊㄋㄌㄍㄎ	…	ㄠㄡㄢㄣㄤㄥㄦㄧㄨㄩ
Line-drawing elements	——｜｜------┆┆……	…	┴┬┤├┼┼┼┼┼
Level 1 hanzi	啊阿埃挨哎唉哀皑癌蔼	…	尊遵昨左佐柞做作坐座
Level 2 hanzi	亍丌兀丐廿卅丕亘丞鬲	…	黲黯黝黜黠黟黢黩黧黥

Encoding methods that cover GB 2312-80—and its extensions, described shortly—include ISO-2022-CN, ISO-2022-CN-EXT, and EUC-CN. GBK, GB 18030, and the various encoding forms of Unicode are compatible with GB 2312-80 in that they are able to encode all of its characters.

CJKV font developers should be aware that early printings of the GB 2312-80 manual had the code points of two uppercase Cyrillic characters (in row 7) swapped. Table 3-19 illustrates the incorrect and correct order of these characters in GB 2312-80. Note the different ordering of the two uppercase Cyrillic characters Ф (Row-Cell 07-22) and Х (Row-Cell 07-23), both of which have been underlined.

Table 3-19. Uppercase Cyrillic character ordering in GB 2312-80

	Character sequence
Incorrect	А Б В Г Д Е Ё Ж З И Й К Л М Н О П Р С Т У Х Ф Ц Ч Ш Щ Ъ Ы Ь Э Ю Я
Correct	А Б В Г Д Е Ё Ж З И Й К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я

I have encountered at least one Chinese type foundry whose font data propagates the character-ordering error illustrated in Table 3-19.

There are three common extensions to GB 2312-80, one of which was used to issue two corrections. Table 3-20 illustrates the number of characters in GB 2312-80 and its three extensions.

Table 3-20. GB 2312-80 and its three extensions

Character set	Characters	Characters added	Number of corrections
GB 2312-80	7,445		
GB 6345.1-86	7,577	132	2
GB 8565.2-88	8,150	705	
ISO-IR-165:1992	8,443	998	

These minor extensions to the GB 2312-80 character set standard are described in the following sections.

GB 6345.1-86—corrections and extensions to GB 2312-80

Corrections for and additions to GB 2312-80 have been issued through a separate character set standard that is designated GB 6345.1-86, which was established on December 1, 1986. This standard is entitled *32 × 32 Dot Matrix Font Set of Chinese Ideograms for Information Interchange* (信息交换用汉字 32×32 点阵字模集 *xìnxī jiāohuàn yòng hànzì 32×32 diǎnzhèn zìmújí*), and it resulted in 132 additional characters for a new total of 7,577 characters (6,763 hanzi plus 814 non-hanzi). Table 3-21 highlights the additional characters for GB 2312-80 specified by GB 6345.1-86.

Table 3-21. The GB 6345.1-86 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	72	Numerals 1–20 with period, parenthesized numerals 1–20, encircled numerals 1–10, parenthesized hanzi numerals 1–20, uppercase Roman numerals 1–12
3	94	Full-width GB 1988-89 (GB-Roman; equivalent to ASCII)
4	83	Hiragana
5	86	Katakana
6	48	Upper- and lowercase Greek alphabet

Table 3-21. The GB 6345.1-86 character set

Row	Characters	Content
7	66	Upper- and lowercase Cyrillic alphabet
8	69	32 full-width Pinyin characters, 37 zhuyin (bopomofo) characters
9	76	Full-width line-drawing elements
10	94	Half-width GB 1988-89 (GB-Roman; equivalent to ASCII)
11	32	Half-width Pinyin characters
12–15	0	Unassigned
16–55	3,755	Level 1 hanzi (last is 55-89)
56–87	3,008	Level 2 hanzi (last is 87-94)
88–94	0	Unassigned

While Table 3-21 clearly shows what characters were added to GB 2312-80, it does not list the corrections. Table 3-22 shows the two corrections to GB 2312-80 mandated by GB 6345.1-86.

Table 3-22. GB 6345.1-86 corrections

Row-Cell	GB 2312-80	GB 6345.1-86
03-71	g	g
79-81	鍾	锺

The GB 2312-80 character form for Row-Cell 79-81 happens to be the same as that found in the GB/T 12345-90 standard—that is, the traditional form, and at the same code point. GB/T 12345-90 is described shortly. This error is still found in recent publications that list all GB 2312-80 hanzi, so evidently information about this correction is not yet widely known.

GB 8565.2-88—another extension to GB 2312-80

The GB 8565.2-88 standard, established on July 1, 1988, defines additions to the GB 2312-80 character set. This standard is entitled *Information Processing—Coded Character Sets for Text Communication—Part 2: Graphic Characters* (信息处理—文本通信用编码字符集—第二部分—图形字符集 *xìnxī chǔlǐ—wénběn tōngxìn yòng biānmǎ zìfújí—dì'èr bùfen—túxíng zìfújí*). These additions, however, are independent from those specified by GB 6345.1-86. The number of additional characters totals 705, bringing the total number of characters to 8,150 (7,399 hanzi plus 751 non-hanzi).

Table 3-23 provides a listing of characters in GB 8565.2-88, and those above and beyond GB 2312-80 are highlighted.

Table 3-23. The GB 8565.2-88 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	72	Numerals 1–20 with period, parenthesized numerals 1–20, encircled numerals 1–10, parenthesized hanzi numerals 1–20, uppercase Roman numerals 1–12
3	94	Full-width GB 1988-89 (GB-Roman; equivalent to ASCII)
4	83	Hiragana
5	86	Katakana
6	48	Upper- and lowercase Greek alphabet
7	66	Upper- and lowercase Cyrillic alphabet
8	63	26 full-width Pinyin characters, 37 zhuyin (bopomofo) characters
9	76	Full-width line-drawing elements
10–12	0	Unassigned
13	50	Hanzi from GB 7589-87 (last is 13-50)
14	92	Hanzi from GB 7590-87 (last is 14-92)
15	93	69 non-hanzi plus 24 hanzi (last is 15-93)
16–55	3,755	Level 1 hanzi (last is 55-89)
56–87	3,008	Level 2 hanzi (last is 87-94)
88–89	0	Unassigned
90–94	470	Hanzi from GB 7589-87 (last is 94-94)

Note how GB 8565.2-88 does not include the additions specified by GB 6345.1-86. But, it does include its corrections as shown in Table 3-22.

ISO-IR-165:1992—yet another extension to GB 2312-80

ISO-IR-165:1992, also known as the *Consultative Committee on International Telephone and Telegraph* (CCITT) Chinese Set, enumerates 8,443 characters.* It is based on the GB 2312-80 character set, and it includes all modifications and additions specified in GB 6345.1-86 and GB 8565.2-88. That is, it contains 7,445 characters from GB 2312-80, 132 added due to GB 6345.1-86, 705 added due to GB 8565.2-88, plus 161 added by ISO-IR-165:1992.

Table 3-24 provides a listing of characters in ISO-IR-165:1992, and those rows that have content above and beyond GB 2312-80 are highlighted.

* ISO-IR-165:1992 is short for *ISO International Registry #165*, established on July 13, 1992.

Table 3-24. The ISO-IR-165:1992 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	72	Numerals 1–20 with period, parenthesized numerals 1–20, encircled numerals 1–10, parenthesized hanzi numerals 1–20, uppercase Roman numerals 1–12
3	94	Full-width GB 1988-89 (GB-Roman; equivalent to ASCII)
4	83	Hiragana
5	86	Katakana
6	70	48 upper- and lowercase Greek alphabet, 22 background (shading) characters
7	66	Upper- and lowercase Cyrillic alphabet
8	69	32 full-width Pinyin characters, 37 zhuyin (bopomofo) characters
9	76	Full-width line-drawing elements
10	94	Half-width GB 1988-89 (GB-Roman; equivalent to ASCII)
11	32	Half-width Pinyin characters
12	94	94 hanzi (last is 12-94)
13	94	50 hanzi from GB 7589-87 plus 44 hanzi (last is 13-94)
14	92	Hanzi from GB 7590-87 (last is 14-92)
15	94	69 non-hanzi plus 25 hanzi (last is 15-94)
16–55	3,755	Level 1 hanzi (last is 55-89)
56–87	3,008	Level 2 hanzi (last is 87-94)
88–89	0	Unassigned
90–94	470	Hanzi from GB 7589-87 (last is 94-94)

ISO-IR-165:1992 is, as you can see, a superset of GB 2312-80 and all previous extensions thereof.

GB/T 12345-90—the traditional analog of GB 2312-80

This character set standard, established on December 1, 1990 by the People's Republic of China, enumerates 7,709 characters (6,866 hanzi plus 843 non-hanzi). Its official name is *Code of Chinese Ideogram Set for Information Interchange Supplementary Set* (信息交换用汉字编码字符集—辅助集 *xìnxī jiāohuàn yòng hànzì biānmǎ zìfújí—fǔzhùjí*). Table 3-25 lists how characters are allocated to each row. Note the similarities to GB 2312-80, and that the GB 6345.1-86 additions are included.

Table 3-25. The GB/T 12345-90 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	72	Numerals 1–20 with period, parenthesized numerals 1–20, encircled numerals 1–10, parenthesized hanzi numerals 1–20, uppercase Roman numerals 1–12
3	94	Full-width GB 1988-89 (GB-Roman; equivalent to ASCII)
4	83	Hiragana
5	86	Katakana
6	77	48 upper- and lowercase Greek alphabet, 29 vertical-use characters
7	66	Upper- and lowercase Cyrillic alphabet
8	69	32 full-width Pinyin characters, 37 zhuyin (bopomofo) characters
9	76	Full-width line-drawing elements
10	94	Half-width GB 1988-89 (GB-Roman; equivalent to ASCII)
11	32	Half-width Pinyin characters
12–15	0	Unassigned
16–55	3,755	Level 1 hanzi (last is 55-89)
56–87	3,008	Level 2 hanzi (last is 87-94)
88–89	103	Additional hanzi (last is 89-09)
90–94	0	Unassigned

As was the case with GB 2312-80, Level 1 hanzi are arranged by reading, and Level 2 hanzi are arranged by radical, and then by number of remaining strokes. The 103 additional hanzi are arranged by the order in which their counterparts from Levels 1 and 2 hanzi appear. Table 3-26 briefly illustrates the types of characters in GB/T 12345-90.

Table 3-26. GB/T 12345-90 character samples

Character class	Sample characters
Miscellaneous symbols	、。·ˉˇ¨〃々— … □■△▲※→←↑↓〓
Annotated numerals	1. 2. 3. 4. 5. 6. 7. 8. 9. 10. … Ⅲ Ⅳ Ⅴ Ⅵ Ⅶ Ⅷ Ⅸ Ⅹ Ⅺ Ⅻ
Full-width GB-Roman	！＂＃￥％＆＇（）＊ … ｕｖｗｘｙｚ｛｜｝￣
Hiragana	ぁあぃいぅうぇえぉお … りるれろゎわゐゑをん
Katakana	アアィイウウェエォオ … ロワワヰヱヲンヴカケ
Greek characters	Α Β Γ Δ Ε Ζ Η Θ Ι Κ … ο π ρ σ τ υ φ χ ψ ω
Vertical-use characters	，。、：；！？ ︵ ︶ ﹇ … 〖〗 ﹄ ﹃ ﹁ ﹂ ｜ … ｜ ｛
Cyrillic characters	А Б В Г Д Е Ё Ж З И … ц ч ш щ ъ ы ь э ю я
Full-width Pinyin	ā á ǎ à ē é ě è ī í … ǔ ù ü ê ɑ ḿ ń ň ǹ ɡ
Zhuyin (bopomofo)	ㄅ ㄆ ㄇ ㄈ ㄉ ㄊ ㄋ ㄌ ㄍ ㄎ … ㄠ ㄡ ㄢ ㄣ ㄤ ㄥ ㄦ ㄧ ㄨ ㄩ

Table 3-26. GB/T 12345-90 character samples

Character class	Sample characters		
Line-drawing elements	─━│┃┄┅┆┇┈	…	┴┵┶┷┸┹┺┻┼┽
Half-width GB-Roman	!"#¥%&′()*	…	uvwxyz{|}‾
Half-width Pinyin	āáǎàēéěèīí	…	ǚǜüêɑḿńňǹɡ
Level 1 hanzi	啊阿埃挨哎唉哀皚癌藹	…	尊遵昨左佐柞做作坐座
Level 2 hanzi	亍丌兀丐廿卅丕亘丞鬲	…	黲黯齡齙齬鼴鼷鼽鼾齄
Additional hanzi	襬闆錶彆蔔纔厂冲丑齣	…	髒症隻只緻製种硃筑准

Compare Levels 1 and 2 hanzi in Table 3-26 with those for GB 2312-80 in Table 3-18, and note how the same hanzi are used, but that a handful are in the traditional form. In fact, there are 2,180 traditional hanzi forms in GB/T 12345-90 when compared to GB 2312-80, most of which are replacements for simplified hanzi.

The 2,180 hanzi that are used to transform GB 2312-80 into GB/T 12345-90 can be divided into the two classes, as indicated in Table 3-27.

Table 3-27. GB/T 12345-90 characters not in GB 2312-80

Characters	Character class
2,118	Traditional hanzi replacements—rows 16 through 87
62	Additional hanzi—scattered throughout rows 88 and 89

In addition to the replacements and additions in Table 3-27, 41 hanzi from GB 2312-80 rows 16 through 87 are scattered throughout GB/T 12345-90 rows 88 and 89, and four pairs of hanzi between Levels 1 and 2 hanzi were swapped. Appendix G provides more details about the four pairs of swapped hanzi and the mappings for hanzi in rows 88 and 89—it also includes a long and complete listing of the 2,118 traditional hanzi replacements, which is something that even the GB/T 12345-90 does not provide.

Like other character set standards, GB/T 12345-90 is not without errors. Chinese type foundries should take note that the GB/T 12345-90 manual has at least two (but, unfortunately, generally not known) printing errors, as indicated in Table 3-28.

Table 3-28. GB/T 12345-90 corrections

Original	Corrected	Row-Cell	Original in Unicode	Original in GB 18030
隷	隸	33-05	U+96B7	EB 5F
鳧	鳬	57-76	U+9CE7	F8 44

In addition, there is often some misunderstanding of the scope and content of the GB/T 12345-90 character set standard. Some printouts of the GB/T 12345-90 character set use slightly different glyphs from the official standard. One specific instance of GB/T 12345-90 provided to The Unicode Consortium used 22 different glyphs, each of which has a

different Unicode code point. This causes lots of confusion. Table 3-29 lists these characters, along with their (incorrect) Unicode mappings and GB 18030 cross-references. For all 22 of these characters, their glyphs in GB/T 12345-90 are intended to be identical to those in GB 2312-80.

Table 3-29. Incorrect mappings between GB/T 12345-90 and Unicode

Correct	GB 2312-80 and GB/T 12345-90	Incorrect	Unicode	GB 18030
叠	21-94	疊	U+758A	AF 42
换	27-27	換	U+63DB	93 51
唤	27-29	喚	U+559A	86 BE
痪	27-30	瘓	U+7613	AF 88
焕	27-32	煥	U+7165	9F A8
涣	27-33	渙	U+6E19	9C 6F
晋	29-90	晉	U+6649	95 78
静	30-18	靜	U+975C	EC 6F
净	30-27	凈	U+51C8	83 F4
栖	38-60	棲	U+68F2	97 AB
弃	38-90	棄	U+68C4	97 89
潜	39-17	潛	U+6F5B	9D 93
挣	53-85	掙	U+6399	92 EA
睁	53-86	睜	U+775C	B1 A0
狰	53-88	猙	U+7319	AA 62
争	53-89	爭	U+722D	A0 8E
伫	56-89	佇	U+4F47	81 D0
陧	58-77	隉	U+9689	EA 9F
奂	59-28	奐	U+5950	8A 4A
峥	65-31	崢	U+5D22	8D 98
戬	74-15	戩	U+6229	91 EC
筝	83-61	箏	U+7B8F	B9 7E

In summary, GB/T 12345-90 is the traditional analog of GB 2312-80. Because of this relationship, we can say that the scope of GB/T 12345-90 is to include all traditional forms of hanzi in GB 2312-80. This brings us to one last error that is in GB/T 12345-90. There is one hanzi in GB/T 12345-90, 囉 (88-51), which actually should not be included because its corresponding simplified form, 啰 (U+5570), is not in GB 2312-80! This hanzi is in both GB 7589-87 (22-51) and GB 8565.2-88 (15-93). The reason why the hanzi 囉 was included in GB/T 12345-90 is due to an error in the 1956 draft version of 简化字总表 (*jiǎnhuàzì zǒngbiǎo*; later corrected in the 1964 version), whereby the two hanzi 羅 and 囉 were mistakenly labeled as traditional forms of the simplified hanzi 罗 (34-62 in GB2312-80)—see

Table 3-1 at the beginning of this chapter. Only the hanzi 羅 is the true traditional form of the simplified hanzi 罗.

In the next section, you learn that there are two more Chinese character set standards, GB 7589-87 and GB 7590-87, and that both of them, like GB 2312-80, have traditional analogs. Their traditional analogs, GB/T 13131-2XXX and GB/T 13132-2XXX, have not yet been published, and probably never will be.

Other Chinese character set standards

There are many other character set standards developed by China, each of which is commonly referred to as a GB standard. All of these GB standards share several common characteristics:

- For every GB standard that includes simplified hanzi, there is a corresponding GB standard that replaces simplified forms by their government–sanctioned traditional forms. GB 2312-80 and GB/T 12345-90, which you read about earlier, represent one such pair of character set standards.
- Every GB standard is also referred to by a numeric designation, with the most basic character set being zero (that is, "GB0" for GB 2312-80).

Table 3-30 lists the relevant GB character set standards in a way that indicates their relationship with one another, along with their assigned numeric designation. Note how simplified character sets are indicated by even-numbered designations, and traditional character sets by odd.

Table 3-30. GB character set standards—simplified and traditional

Simplified character set	Hanzi	Traditional character set	Additional hanzi
GB 2312-80—GB0	6,763	GB/T 12345-90—GB1	103[a]
GB 7589-87—GB2	7,237	GB/T 13131-2XXX—GB3	
GB 7590-87—GB4	7,039	GB/T 13132-2XXX—GB5	

a. These 103 additional hanzi occupy all of row 88 (94 hanzi) and the first part of row 89 (9 hanzi).

An oddball character set standard in this regard is GB 8565.2-88, which is sometimes referred to as GB8.

The hanzi in GB 7589-87 and GB 7590-87 (this also applies, of course, to their traditional analogs, specifically GB/T 13131-2XXX and GB/T 13132-2XXX) are ordered by radical, and then by total number of strokes, and then begin allocating characters at row 16. GB 7589-87 was established on December 1, 1987 and is entitled *Code of Chinese Ideograms Set for Information Interchange—the Second Supplementary Set* (信息交换用汉字编码字符集—第二辅助集 *xìnxī jiāohuàn yòng hànzì biānmǎ zìfújí—dì'èr fǔzhùjí*). GB 7590-87 was established on the same date and is entitled *Code of Chinese Ideograms Set for Information Interchange—the Fourth Supplementary Set* (信息交换用汉字编码字符集—第四辅

助集 *xìnxī jiāohuàn yòng hànzì biānmǎ zìfújí—dìsì fǔzhùjí*). Tables 3-31 and 3-32 list the character allocation for GB 7589-87 and GB 7590-87, respectively.

Table 3-31. The GB 7589-87 character set

Row	Characters	Content
0–15	0	Unassigned
16–92	7,237	Hanzi (last is 92-93)

Table 3-32. The GB 7590-87 character set

Row	Characters	Content
0–15	0	Unassigned
16–90	7,039	Hanzi (last is 90-83)

It is interesting to note that all the hanzi specified in GB 7589-87 and GB 7590-87 are handwritten. Needless to say, fonts that support these character set standards are scarce.

Note that not all hanzi in the simplified character set standards are replaced by a corresponding traditional hanzi. In the case of the GB 2312-80 and GB/T 12345-90 pair, 2,180 additional hanzi are needed to transform GB 2312-80 into GB/T 12345-90. The majority are simple one-to-one replacements, but some are hanzi that swap code points or split into two or more separate hanzi (some simplified hanzi were derived from two or more traditional hanzi).

Appendix G provides complete GB 2312-80 and GB/T 12345-90 code tables. An inadequate supply of fonts precluded the inclusion of code tables for the other GB character set standards that were mentioned thus far. This volume also includes a reading index for Level 1 hanzi and a radical index for Level 2 hanzi. But note that the GB 2312-80 standard itself, as a printed manual, includes many useful indexes.

GBK—extended GB 2312-80

Another well-known GB character set is very closely aligned to ISO 10646-1:1993 (equivalent to Unicode version 1.1) and is designated GB 13000.1-93. It is, for all practical purposes, the Chinese translation of ISO 10646-1:1993. What is interesting about GB 13000.1-93 is the Chinese-specific subset called GBK. GBK, known as the *Chinese Internal Code Specification* (汉字内码扩展规范 *hànzì nèimǎ kuòzhǎn guīfàn*), is simply an extension to GB 2312-80 that accommodates the remaining ideographs in ISO 10646-1:1993 (GB 13000.1-93). From a character-allocation point of view, GBK is composed of the following characters:

- GB 2312-80 base (with some corrections/additions specified in GB 6345.1-86)
- Non-hanzi from GB/T 12345-90

- 14,240 additional hanzi
- 166 additional symbols

Ten of the 29 non-hanzi specific to GB/T 12345-90 that are mostly vertical variants, along with the half-width characters, are not included in GBK. Lowercase Roman numerals 1 through 10 have been added. GBK is logically arranged in five parts, as listed in Table 3-33.

Table 3-33. The five parts of GBK

Part	Characters	Content
GBK/1	717	GB 2312-80 and GB/T 12345-90 non-hanzi
GBK/2	6,763	GB 2312-80 hanzi
GBK/3	6,080	Hanzi from ISO 10646-1:1993
GBK/4	8,160	8,059 hanzi from ISO 10646-1:1993 plus 101 additional hanzi
GBK/5	166	Non-hanzi from Big Five and other characters

The number of hanzi in GBK/2 through GBK/4 is 21,003, which is 101 more than are found in the CJK Unified Ideographs block of ISO 10646-1:1993. The 101 additional hanzi in GBK/4 account for this difference.

From a compatibility point of view, there is comfort in knowing that every character in GB 2312-80 is at the *same* code point in GBK. Chapter 4 provides more details about GBK, including its encoding specifications.

The Simplified Chinese version of Microsoft Windows 95 and later uses GBK; this character set is also known as Microsoft Code Page 936.

GB 18030-2005—further extending GB 2312-80 to encompass all of Unicode

The latest and greatest character set standard established by China is designated GB 18030-2005. Its official title is *Information Technology—Chinese Coded Character Set* (信息技术 中文编码字符集 *xìnxī jìshù zhōngwén biānmǎ zìfújí*). Its initial version was designated GB 18030-2000. What makes GB 18030-2005 important is the fact that its support has been declared mandatory by the Chinese government for products sold within China. An organization called CESI (*China Electronics Standardization Institute*; 中国电子技术标准化研究所 *zhōngguó diànzǐ jìshù biāozhǔn huà yánjiū suǒ*) publishes GB 18030 certification guidelines, and performs GB 18030 certification testing.*

Some of the details in this section are truthfully a description of GB 18030-2000, but because the 2005 revision needs to be emphasized, such details are included here. Additional details about GB 18030-2000 can be found in its own section, which immediately follows this one.

* *http://www.cesi.cn/* or *http://www.cesi.com.cn/*

GB 18030-2005, simply put, is a superset of all Chinese character set standards described thus far, and it is also code-point–compatible with Unicode. Just as GBK is an extended version of GB 2312-80, GB 18030-2005 is an extended version of GBK. The code-point–compatible nature of GB 18030-2005 shall be covered in depth in Chapter 4.

So, what changed between GBK and GB 18030-2005 in terms of additional characters? Ignoring the additional characters printed in the GB 18030-2005—for the six regional scripts (to be discussed later in this section) and the tens of thousands of ideographs in CJK Unified Ideographs Extension B—Table 3-34 lists the characters added to GBK to create GB 18030-2000.

Table 3-34. Characters specific to GB 18030-2000—not present in GBK

Number of characters	Description	Encoding
10	Lowercase Roman numerals 1–10	<A2 A1> through <A2 AA>
1	Euro currency symbol	<A2 E3>
6,530	CJK Unified Ideographs Extension A	<81 39 EE 39> through <82 35 87 38>

Interestingly, like GBK, GB 18030 is divided into parts—that is clearly the best way to think about the structure of this relatively large character set. Table 3-35 lists the five parts of GB 18030's two-byte region, along with the number of characters in each. Please compare this with the contents of Table 3-33.

Table 3-35. The five parts of GB 18030's two-byte region

Part	Characters	Content
1	728[a]	GBK/1 plus lowercase Roman numerals 1–10 and the euro currency symbol
2	6,763	GB 2312-80 hanzi
3	6,080	Hanzi from ISO 10646-1:1993
4	8,160	8,059 hanzi from ISO 10646-1:1993 plus 101 additional hanzi
5	166	Non-hanzi from Big Five and other characters

a. Table 2 of GB 18030 (on page 8 of GB 18030-2000, and on page 5 of GB 18030-2005) states that Part 1 contains 718 characters, but if you actually count them, the total number of characters is 728. I am convinced that the 10-character discrepancy is due to an oversight of 10 of the 29 non-hanzi that are specific to GB/T 12345-90. These 10 characters are not included in GBK and could be easily overlooked.

You will learn later in this chapter that CJK Unified Ideographs Extension A contains 6,582 characters, all of which are ideographs. However, GB 18030 seems to include only 6,530 of these ideographs. So, what happened to 52 of its ideographs? Why would GB 18030 exclude them? Put simply, these 52 ideographs were already included in GBK, specifically in GBK/4, as part of the 101 additional hanzi in that region. Table 3-36 shows how these 52 characters are mapped, from GB 18030 code points to Unicode CJK Unified Ideographs Extension A code points.

Table 3-36. Fifty-two special-case mappings for CJK Unified Ideographs Extension A

Hanzi	GB 18030	Unicode
偬	FE 55	U+3473
佁	FE 56	U+3447
唲	FE 5A	U+359E
嚈	FE 5B	U+361A
嘪	FE 5C	U+360E
惾	FE 5F	U+396E
恰	FE 60	U+3918
拻	FE 62	U+39CF
挸	FE 63	U+39DF
攓	FE 64	U+3A73
扱	FE 65	U+39D0
柍	FE 68	U+3B4E
殒	FE 69	U+3C6E
汰	FE 6A	U+3CE0
瞜	FE 6F	U+4056
穇	FE 70	U+415F
绀	FE 72	U+4337
耀	FE 77	U+43B1
耬	FE 78	U+43AC
胯	FE 7A	U+43DD
劳	FE 7B	U+44D6
褵	FE 7C	U+4661
褹	FE 7D	U+464C
讶	FE 80	U+4723
谾	FE 81	U+4729
賰	FE 82	U+477C
赕	FE 83	U+478D
鎘	FE 85	U+4947
钑	FE 86	U+497A
铓	FE 87	U+497D
鏐	FE 88	U+4982
镸	FE 89	U+4983
镾	FE 8A	U+4985
钁	FE 8B	U+4986

Table 3-36. Fifty-two special-case mappings for CJK Unified Ideographs Extension A

Hanzi	GB 18030	Unicode
㘚	FE 8C	U+499F
㘝	FE 8D	U+499B
㘎	FE 8E	U+49B7
㘍	FE 8F	U+49B6
㙡	FE 92	U+4CA3
㙴	FE 93	U+4C9F
㙺	FE 94	U+4CA0
㙍	FE 95	U+4CA1
㙡	FE 96	U+4C77
㙦	FE 97	U+4CA2
㚃	FE 98	U+4D13
㚆	FE 99	U+4D14
㚗	FE 9A	U+4D15
㚘	FE 9B	U+4D16
㚙	FE 9C	U+4D17
㚪	FE 9D	U+4D18
㚫	FE 9E	U+4D19
奚	FE 9F	U+4DAE

Note that while the Unicode values generally are in ascending order, there are a small number of exceptions.

One major concern with GB 18030 has been *Private Use Area* (PUA) usage with regard to Unicode mappings. Although PUA mappings for 24 characters are still printed in the GB 18030-2005 standard proper, it is important to note that as long as Unicode version 4.1 or greater is used, all 24 of these characters can be safely mapped or otherwise handled without resorting to PUA code points. This means that GB 18030-2005 can be represented in its entirety using Unicode, without the use of PUA code points. This is important to realize, and it is a very good thing. Table 3-37 lists these 24 mappings, showing their PUA and non-PUA (Unicode version 4.1) mappings.

Table 3-37. PUA code points in GB 18030-2005 versus Unicode version 4.1

Character	GB 18030-2005	PUA	Unicode 4.1 (Non-PUA)
’	A6 D9	U+E78D	U+FE10
。	A6 DA	U+E78E	U+FE12
、	A6 DB	U+E78F	U+FE11

Table 3-37. PUA code points in GB 18030-2005 versus Unicode version 4.1

Character	GB 18030-2005	PUA	Unicode 4.1 (Non-PUA)
：	A6 DC	U+E790	U+FE13
；	A6 DD	U+E791	U+FE14
！	A6 DE	U+E792	U+FE15
？	A6 DF	U+E793	U+FE16
﹁	A6 EC	U+E794	U+FE17
﹂	A6 ED	U+E795	U+FE18
⋮	A6 F3	U+E796	U+FE19
𠀇	FE 51	U+E816	U+20087
𠂉	FE 52	U+E817	U+20089
𠃌	FE 53	U+E818	U+200CC
龴	FE 59	U+E81E	U+9FB4
龵	FE 61	U+E826	U+9FB5
龶	FE 66	U+E82B	U+9FB6
龷	FE 67	U+E82C	U+9FB7
龺	FE 6C	U+E831	U+215D7
龻	FE 6D	U+E832	U+9FB8
龼	FE 76	U+E83B	U+2298F
龽	FE 7E	U+E843	U+9FB9
龾	FE 90	U+E854	U+9FBA
龿	FE 91	U+E855	U+241FE
鿀	FE A0	U+E864	U+9FBB

These 24 PUA mappings are printed in the GB 18030-2005 standard proper because it was prepared in 2003, prior to Unicode version 4.1. GB 18030-2005 is obviously dated in 2005, but was printed in 2006. Considering the code-point–compatible nature of GB 18030-2005 with Unicode, mapping these 24 characters to Unicode version 4.1 code points (meaning to non-PUA code points) is clearly the prudent thing to do.

Note that 6 of the 24 Unicode 4.1 mappings are non-BMP, specifically into CJK Unified Ideographs Extension B, which is in Plane 2. These have been highlighted in Table 3-37. It is because of these six characters and their appropriate mappings that beyond-BMP support is necessary for GB 18030 compliance in the context of Unicode version 4.1 or greater.

GB 18030-2005 silently corrected four prototypical glyphs used for printing the standard. GB standards have a demonstrated history of making silent corrections, or corrections set forth through new standards. Table 3-38 lists the four prototypical glyph corrections that were implemented in GB 18030-2005.

Table 3-38. GB 18030-2005 prototypical glyph corrections

Hanzi (Correct)	GB 18030	Unicode	Glyph error description
堺	82 30 AD 37	U+3665	The 13th stroke was missing.
癵	82 32 A1 39	U+3FD0	The 10th stroke was missing.
鯖	82 34 E6 31	U+4C6A	The left-side component was simplified, but should be traditional.
鸛	82 34 F4 32	U+4CFD	The right-side component was simplified, but should be traditional.

The main focus of the 2005 revision of the GB 18030 standard is clearly in two areas, which are best described as follows:

- Acknowledgment of CJK Unified Ideographs Extension B—42,711 hanzi
- Acknowledgment of the six regional scripts: Korean, Mongolian, Tai Le, Tibetan, Uyghur, and Yi

All 42,711 characters of CJK Unified Ideographs Extension B are printed in the GB 18030-2005 manual, on pages 240 through 443. The characters for the six regional scripts are printed in the standard as well. All of these newly acknowledged characters are supported through the use of GB 18030's four-byte region.

GB 18030 compliance guidelines

Because GB 18030 support has been mandated by the Chinese government for software sold in that region, knowing what constitutes GB 18030 compliance thus becomes critical to any business that deals with encoded text.

Given GB 18030's close relationship with Unicode, combined with the fact that Unicode is supported by today's OSes, the simplest way to support GB 18030 is through Unicode. In other words, as long as your application correctly handles arbitrary Unicode text, chances are it is already GB 18030–compliant. Or, at least, the amount of additional work that is necessary for GB 18030 compliance should be minimal, possibly even trivial.

As stated earlier in this section, an organization called CESI performs GB 18030 certification testing. Because CESI is physically located in China, and because the testers are Chinese, I strongly suggest to anyone outside of China who plans to have his software tested for GB 18030 compliance that he partner with a company located in China to facilitate the testing and certification process. I suggest this for a couple of reasons. First, there is an obvious language barrier for software companies that do not have presence in China, and the Chinese companies help to break this barrier. Second, the Chinese companies typically have experience working with CESI, specifically related to GB 18030 certification.

There are two areas of GB 18030-2005 for which certification may seem problematic or otherwise nontrivial:

- Supporting the six regional scripts
- Supporting CJK Unified Ideographs Extension B

For both of these areas, support is not necessary in order to achieve certification, at least at the time of this writing. While it is necessary for the code points for these characters to be correctly processed, such as converting to and from Unicode and GB 18030's native encoding, correctly rending the characters is not necessary.

Finally, if your software has specific areas that may be lacking in terms of GB 18030 support, it is best to identify those areas in a "ReadMe" file or other documentation that is submitted to CESI. I want to point out that CESI would rather know about deficient areas beforehand, rather than discover them through testing.

GB 18030-2000

The original version of GB 18030, designated GB 18030-2000, was published on March 17, 2000. Its title is more complex than its 2005 revision: *Information Technology—Chinese Ideograms Code Character Set for Information Interchange—Extension for the Basic Set* (信息技术 信息交换用汉字编码字符集 基本集的扩充 *xìnxī jìshù xìnxī jiāohuàn yòng hànzì biānmǎ zìfújí jīběnjí de kuòchōng*).

Table 3-39 lists the only character whose encoding was changed for the 2005 revision.

Table 3-39. GB 18030 encoding changes

Character	GB 18030	Unicode (2000)	Unicode (2005)
𠃌	A8 BC	U+E7C7 (PUA)	U+1E3F

Effectively, the change is from PUA to non-PUA, which is a very good thing, and similar treatment is expected for 24 additional characters (see Table 3-37).

Unicode compatibility with GB standards

The three most important GB standards with regard to Unicode compatibility are GB 2312-80, GBK, and GB 18030. Full support for GB 2312-80 is in the earliest versions of Unicode. GBK made use of some PUA code points. The 2000 and 2005 versions of GB 18030 specify PUA code points to some extent, but as long as Unicode version 4.1 or greater is used, PUA usage is not necessary. But, as I pointed out earlier in this chapter, in lieu of PUA usage, six GB 18030 hanzi require Plane 2 support because their non-PUA mappings are in CJK Unified Ideographs Extension B.

Given the code-point compatibility between GB 18030 and Unicode, which is at the encoding level, the likelihood of continued compatibility between these standards is very high. This is a good thing.

Chinese Character Set Standards—Taiwan

Taiwan (臺灣 *táiwān*), or more officially, the Republic of China or ROC (中華民國 *zhōnghuá mínguó*), represents another standards-producing Chinese locale. Taiwan does

not use simplified ideographs, and typically uses a larger number of characters than all other CJKV locales combined.

Big Five

Big Five (大五 *dàwǔ*) is the most widely implemented character set standard used in Taiwan and was established on May 1st, 1984 by the Institute for Information Industry of Taiwan through the publishing of *Computer Chinese Glyph and Character Code Mapping Table* (電腦用中文字型與字碼對照表 *diànnǎoyòng zhōngwén zìxíngyù zìmǎ duìzhào biǎo*), Technical Report (技術通報 *jìshù tōngbào*) C-26. Its name refers to the five companies that collaborated in its development.

Unlike the other CJKV character set standards, Big Five's character space is set in a disjoint 94×157 matrix, for a maximum capacity of 14,758 cells. The Big Five character set standard specifies 13,494 standard characters (13,053 hanzi plus 441 non-hanzi), but some vendor-specific implementations often have a larger repertoire.

I feel compelled to warn you that Big Five is not a national standard, but is used much more widely than the national character set standard for Taiwan, specifically CNS 11643-2007, described next. In other words, Big Five has become a *de facto* standard for Taiwan. Table 3-40 lists the character allocation of the Big Five character set.

Table 3-40. The Big Five character set

Row	Characters	Content
1	157	Two abbreviations, 155 miscellaneous symbols
2	157	Nine hanzi for measurements, 9 abbreviations, 21 full-width line-drawing elements, numerals 0–9, uppercase Roman numerals 1–10, Chinese numerals 1–12, upper- and lowercase Latin characters (except for w–z), 38 miscellaneous symbols
3	127	Lowercase Latin characters w–z, 48 upper- and lowercase Greek characters, 37 zhuyin (bopomofo) characters, 5 tone marks, 33 abbreviations for control characters
4–38	5,401	Level 1 hanzi (last is 38-63)
39–40	0	Unassigned
41–89	7,652	Level 2 hanzi (last is 89-116)[a]
90–94	0	Unassigned

a. CNS 11643-2007, discussed shortly, has only 7,650 characters in Level 2 hanzi—Big Five has two duplicate hanzi that the designers of CNS 11643-2007 decided not to include.

The hanzi in each of the two levels are arranged by increasing total number of strokes, and then by radical (the inverse of the ordering criteria used for Level 2 of both GB 2312-80 and JIS X 0208:1997; their ideographs are ordered by radical then by increasing number of strokes).

Table 3-41 illustrates examples for each of the character classes that compose the Big Five character set.

Table 3-41. Big Five character samples

Character class	Sample characters
Miscellaneous symbols	，、。．•；：？！ … ㎜㎝㎞㏎㎡㎎㎏㏄°
Hanzi for measurements	瓧瓩瓱瓰瓸瓼瓫甅糎
Line-drawing elements	▁▂▃▄▅▆▇█ ▏ … ╞╪╡◢◣◥◤╱╲╳
Numerals	０１２３４５６７８９
Roman numerals	ⅠⅡⅢⅣⅤⅥⅦⅧⅨⅩ … ⅰⅱⅲⅳⅴⅵⅶⅷⅸⅹ
Short forms for numerals	〡〢〣〤〥〦〧〨〩十卄卅
Latin	ＡＢＣＤＥＦＧＨＩＪ … ｑｒｓｔｕｖｗｘｙｚ
Greek	ΑΒΓΔΕΖΗΘΙΚ … οπρστυφχψω
Zhuyin (bopomofo)	ㄅㄆㄇㄈㄉㄊㄋㄌㄍㄎ … ㄠㄡㄢㄣㄤㄥㄦㄧㄨㄩ
Tone marks[a]	˙ ˊˇˋ
Control characters	NULL TC1(SOH) TC2(STX) TC3(ETX) TC4(EOT) TC5(ENQ) TC6(ACK) BELL FE0(BS) FE1(HT) … TC10(ETB) CAN EM SUB ESC IS4(FS) IS3(GS) IS2(RS) IS1(US) DEL
Level 1 hanzi	一乙丁七乃九了二人儿 … 黷豔鑿鸚爨驪鬱鸛鸞籲
Level 2 hanzi	乂乜凵匚厂万丌乇亍囗 … 癵驫鱺鸝灩灪麤齾齉龘

a. Note that the second tone mark has no form—it is blank.

Table 3-42 illustrates the two hanzi that are duplicately encoded in Big Five. This duplication is a result of an error in design.

Table 3-42. Duplicate hanzi in Big Five

Hanzi	Code points	Unicode
兀	<A4 61> and <C9 4A>	U+5140 and U+FA0C
嗀	<DC D1> and <DD FC>	U+55C0 and U+FA0D

CNS 11643-2007, to be discussed soon, corrects this error by eliminating the second instance of each of these two hanzi, specifically <C9 4A> and <DD FC>. Big Five <A4 61> and <C9 4A> are the same as CNS 11643-2007 Plane 1's 36-34, and Big Five <DC D1> and <DD FC> are the same as CNS 11643-2007 Plane 2's 33-86.

The authors of *An Introduction to Chinese, Japanese and Korean Computing* (World Scientific Publishing, 1989), Jack Huang (黃克東 *huáng kèdōng*) and Timothy Huang (黃大一 *huáng dàyī*), claim that the designers of the Big Five character set simply copied many kanji from JIS C 6226-1978. Many of the same ideographs are used in Chinese, Japanese, and Korean, but there are often subtle character-form differences between them. The Big Five character set contains many non-Chinese forms of hanzi (one could go so far as to say that they are actually *kanji*, not *hanzi*!), and no attempt was ever made to remedy this problem (although CNS 11643-2007 can be considered a cure). Appendix H provides a complete Big Five code table.

Big Five Plus

An extension to Big Five known as *Big Five Plus* (or *Big5+* as a shortened form) was developed by a number of companies in close collaboration and cooperation. Although the specification for Big Five Plus is finalized, it has yet to be fully implemented in any OS. It has, however, been implemented in TwinBridge Chinese Partner, beginning with version 4.98.*

Big Five Plus includes a total of 21,585 characters (and 2,355 user-defined characters), which come from three distinct sources, indicated as follows:

- 13,463 total characters, consisting of the Big Five character set
- 4,670 total characters, consisting of CNS 11643-2007 Plane 3 hanzi in Unicode (3,875), CNS 11643-2007 Plane 1 characters (88), the 214th radical (missing from CNS 11643-2007), Japanese kana (177), ETen full-width line-drawing elements (34), hanzi "zero" (〇), Unicode "shape" characters (5), and CNS 11643-2007 Plane 4 hanzi in Unicode (489)
- 3,452 total characters, consisting of additional CNS 11643-2007 Plane 4 hanzi in Unicode (402), CNS 11643-2007 Plane 5 hanzi in Unicode (61), CNS 11643-2007 Plane 6 hanzi in Unicode (29), CNS 11643-2007 Plane 7 hanzi in Unicode (16), CNS 11643-2007 Plane 15 hanzi in Unicode (152), additional hanzi in Unicode (247), PRC simplified hanzi (2,105), and Japanese kanji and Korean hanja (440)

The end result of Big Five Plus is a character set comparable to GBK in terms of including all of Unicode's first block of 20,902 CJK Unified Ideographs, which are referred to as the *Unified Repertoire and Ordering* (URO). In fact, their encoding definitions, at a high level, are the same. Table 3-43 shows how these 21,585 Big Five Plus characters, along with user-defined characters, are grouped in terms of encoding ranges.

Table 3-43. The Big Five Plus character set

Encoding	Characters	Content
A4 40–C6 7E	5,401	Big Five Level 1
C9 40–F9 FE	7,693	Big Five Level 2 and 41 ETen characters from row 0xF9
A1 40–A3 FE	471	Big Five non-hanzi
C6 A1–C8 FE	408	ETen characters from rows 0xC6 through 0xC8
81 80–FE A0	4,158	Hanzi
81 40–83 FE	471	Hanzi and hanzi variants
8E 40–A0 FE	2,983	Hanzi, simplified hanzi, kanji, and hanja
FA 40–FE FE	785	User-defined characters
84 40–8D FE	1,570	User-defined characters

* *http://www.twinbridge.com/*

For the sake of compatibility with Big Five, Big Five Plus still includes the two duplicately encoded hanzi that are listed in Table 3-42. Given the extent to which Unicode is used in today's software, the likelihood of Big Five Plus becoming widely used is relatively low.

CNS 11643-2007

The roots of CNS 11643-2007, by far the largest legacy CJKV character set standard in current use, can be traced back to Big Five. For this reason, I often consider CNS 11643-2007 to be a corrected and supplemented version of Big Five, and you will soon understand why.

CNS 11643-2007, established on August 9, 2007, enumerates a staggering 69,134 hanzi, and 70,939 total characters when its 1,605 non-hanzi are factored in.* CNS 11643-2007 is entitled *Chinese Standard Interchange Code* (中文標準交換碼 *zhōngwén biāozhǔn jiāohuànmǎ*). CNS stands for *Chinese National Standard* (中國國家標準 *zhōngguó guójiā biāozhǔn*).

As was the case with the Big Five character set, all of the hanzi in CNS 11643-2007 planes (字面 *zì miàn*) are ordered by total number of strokes, then by radical.† Table 3-44 details the characters allocated to the 13 occupied planes of CNS 11643-2007, specifically Planes 1 through 7 and 10 through 15.

Table 3-44. The CNS 11643-2007 character set

Plane	Row	Characters	Content
1	1–6	443	Symbols
	7–18	1,127	213 classical radicals, extended zhuyin, kana, Church Latin characters
	19–33	0	Unassigned
	34	35	Graphic representations of control characters, Euro, ideographic zero
	35	0	Unassigned
	36–93	5,401	Hanzi (last is 93-43)
	94	0	Unassigned
2	1–82	7,650	Hanzi (last is 82-36)
	83–94	0	Unassigned
3	1–66	6,148	Hanzi (last is 66-38)
	67	0	Unassigned
	68–71	128	Hanzi (68-40 through 71-10)
	72–94	0	Unassigned

* *http://www.cns11643.gov.tw/*

† The only exceptions to this basic rule are CNS 11643-2007 Planes 7 and 11, which are actually a continuation of Planes 6 and 10, respectively.

Table 3-44. The CNS 11643-2007 character set

Plane	Row	Characters	Content
4	1–78	7,298	Hanzi (last is 78-60)
	79–94	0	Unassigned
5	1–92	8,603	Hanzi (last is 92-49)
	93–94	0	Unassigned
6	1–68	6,388	Hanzi (last is 68-90)
	69–94	0	Unassigned
7	1–70	6,539	Hanzi (last is 70-53)
	71–94	0	Unassigned
10	1–94	8,836	Hanzi (last is 94-94)
11	1–40	3,698	Hanzi (last is 40-32)
	41–94	0	Unassigned
12	1–65	443	Hanzi (last is 65-54)
	66–94	0	Unassigned
13	1–82	763	Hanzi (last is 82-78)
	83–94	0	Unassigned
14	1–43	408	Hanzi (last is 43-73)
	44–94	0	Unassigned
15	1–77	6,831	Hanzi (last is 77-25)
	78–94	0	Unassigned

Ideograph devotees will no doubt wonder why there are only 213 of the 214 classical radicals in CNS 11643-2007 Plane 1. Good question! First off, the missing radical is the 34th one, specifically 夂, which is very similar in shape to the 35th radical, specifically 夂. A conscious decision was made by the Ministry of Education to drop the 34th radical due to its similarity—in appearance—with the 35th one. The designers of CNS 11643 were simply following the guidelines set forth by the standard list of 4,808 hanzi, published by the Ministry of Education. This missing 34th radical is available elsewhere in CNS 11643-2007, specifically in Plane 3 at Row-Cell 01-25. Also consider that 210 of these 213 classical radicals are identical to hanzi found in Planes 1 through 3. 167 of them map to Plane 1 hanzi, 20 map to Plane 2 hanzi, and 23 map to Plane 3 hanzi. Three are found only in the radical block of Plane 1.

Planes 3 and 12 through 15 deserve special mention or attention, because their hanzi are not in contiguous blocks. In other words, there are gaps. For Plane 3, only the 128 hanzi in rows 68 through 71 are not contiguous, meaning that there are gaps within that small block. The primary block of hanzi in Plane 3, meaning rows 1 through 66, are contiguous. Planes 12 through 14 are very sparse, meaning that each row contains very few characters. Plane 15 has gaps, but very few when compared to Planes 12 through 14. Plane

15 originated in CNS 11643-1986 and was established in 1990. The section about CNS 11643-1986 will provide additional details about Plane 15 and how it changed for the 2007 version of the standard.

While the 13 planes of CNS 11643-2007 are fairly autonomous, meaning that they are considered separate collections of characters when it comes to ordering them by total number of strokes, Planes 6 and 7 actually form a continuous set of characters, as do Planes 10 and 11. Also of potential interest is that Plane 6 contains hanzi with 1 to 14 total strokes, and Plane 7 contains those with 15 or more total strokes. The stroke-count index in Appendix H illustrates this point well.

Table 3-45 illustrates the character classes that make up the CNS 11643-2007 character set. Because the rows containing symbols are not broken up—across rows—by character class, they are lumped together in Table 3-45. You can reference Table 3-41, which provides sample Big Five characters, to see a more complete breakdown of CNS 11643-2007's symbol class. Only the hanzi in the first seven planes are shown.

Table 3-45. CNS 11643-2007 character samples—Planes 1 through 7

Character class	Sample characters
Symbols	，、。．‧；：？！ … ⅰ ⅱ ⅲ ⅳ ⅴ ⅵ ⅶ ⅷ ⅸ ⅹ
Classical radicals	一丨丶丿乙亅二亠人儿 … 黽鼎鼓鼠鼻齊齒龍龜龠
Control characters	NULL TC_1(SOH) TC_2(STX) TC_3(ETX) TC_4(EOT) TC_5(ENQ) TC_6(ACK) BELL FE_0(BS) FE_1(HT) … TC_{10}(ETB) CAN EM SUB ESC IS_4(FS) IS_3(GS) IS_2(RS) IS_1(US) DEL
Plane 1 hanzi	一乙丁七乃九了二人儿 … 黷豔鑿鸚爨驪鬱鸛鸞籲
Plane 2 hanzi	乂乜凵匚厂万丌乇亍囗 … 驫鱺鸝灩灪爩麤齾齉龘
Plane 3 hanzi	丨丶丿亅丅丄冂冖匚卩 … 轤鸎黷[illegible]鱻龓鸐靉䨻靐
Plane 4 hanzi	𠂆乀乁乚𡿨丂丩乃几刂 … 軅鬞龖龝纍䰟鷃齾鱻靆
Plane 5 hanzi	ㄣㄥ乙ㄑ[illegible]ナㄐ[illegible]凵马 … 鸛虉黵欗𨟺龞豔齺黌龖
Plane 6 hanzi	乁卜乂丩亾乚二亖亖七 … 鳥鳥鴕鴕麨鼎黝黜黽鼓
Plane 7 hanzi	壴寡奪奠僊僕僱僟僸僼 … 龝虊灩鱻讐靁鬱斖靈飆

Pages 501 through 524 of the CNS 11643-2007 standard is a list of 639 hanzi whose glyphs changed, in subtle ways, from the CNS 11643-1992 standard. These, of course, cover only Planes 1 through 7. The prototypical glyphs shown in the CNS 11643-1992 standard were rather coarse bitmaps, and those in the CNS 11643-2007 standard appear to be generated from outlines.

Encoding methods for CNS 11643-2007 include ISO-2022-CN, ISO-2022-CN-EXT, and and EUC-TW. The encoding forms of Unicode are able to encode most of its characters. Big Five can be considered an alternate encoding for CNS 11643-2007 Planes 1 and 2.

In terms of overall structure, CNS 11643-2007 is composed of 80 planes, each of which is a 94×94 matrix capable of encoding up to 8,836 characters. The original 1986 version,

which will be covered in a later section, was composed of only 16 such planes. Its 1992 instance, described next, encodes characters only in its first seven planes.

CNS 11643-1992

CNS 11643-1992, established on May 21, 1992, enumerated a still-staggering 48,027 hanzi, and 48,711 total characters when its 684 non-hanzi are included. CNS 11643-1992 was entitled *Chinese Standard Interchange Code* (中文標準交換碼 *zhōngwén biāozhǔn jiāohuànmǎ*), which is the same as CNS 11643-2007. Unlike CNS 11643-2007, its 1992 version includes characters only in its first seven planes. Table 3-46 lists the characters that are included in CNS 11643-1992's seven planes.

Table 3-46. The CNS 11643-1992 character set

Plane	Row	Characters	Content
1	1–6	438	Symbols
	7–9	213	213 classical radicals
	10–33	0	Unassigned
	34	33	Graphic representations of control characters
	35	0	Unassigned
	36–93	5,401	Hanzi (last is 93-43)
	94	0	Unassigned
2	1–82	7,650	Hanzi (last is 82-36)
	83–94	0	Unassigned
3	1–66	6,148	Hanzi (last is 66-38)
	67–94	0	Unassigned
4	1–78	7,298	Hanzi (last is 78-60)
	79–94	0	Unassigned
5	1–92	8,603	Hanzi (last is 92-49)
	93–94	0	Unassigned
6	1–68	6,388	Hanzi (last is 68-90)
	69–94	0	Unassigned
7	1–70	6,539	Hanzi (last is 70-53)
	71–94	0	Unassigned

Although Planes 2, 4, 5, 6, and 7 are identical to CNS 11643-2007, note the subtle difference in Planes 1 and 3 when comparing Tables 3-44 and 3-46.

There were some known errors in CNS 11643-1992, all of which are errors in calculating the total number of strokes. These errors are provided on page 318 of the CNS 11643-1992 manual. Table 3-47 shows how many such errors are indicated in the CNS 11643-1992 manual.

Table 3-47. Stroke-count errors in CNS 11643-1992

Plane number	Number of errors
1	1
2	6
3	20

Appendix H provides CNS 11643-1992 code tables for all of its seven planes. The same appendix also provides their stroke-count indexes.

CNS 11643-1986

The original version of the CNS 11643-2007 standard was established on August 4, 1986, and enumerated far fewer characters. Its title was also different: *Standard Interchange Code for Generally Used Chinese Characters* (通用漢字標準交換碼 *tōngyòng hànzì biāozhǔn jiāohuànmǎ*). There were only two planes with assigned characters: Planes 1 and 2. Plane 14 was published in June of 1988 as 通用漢字標準交換碼—使用者加字區交換碼 (*tōngyòng hànzì biāozhǔn jiāohuànmǎ—shǐyòngzhě jiāzìqū jiāohuànmǎ*).* Plane 15—also known as 戶政用字 *hùzhèngyòngzì*—was published in June or July of 1990. Table 3-48 shows the character allocation for CNS 11643-1986, including Planes 14 and 15.

Table 3-48. The CNS 11643-1986 character set

Plane	Row	Characters	Content
1	1–6	438	Symbols
	7–9	213	213 classical radicals
	10–33	0	Unassigned
	34	33	Graphic representations of control characters
	35	0	Unassigned
	36–93	5,401	Hanzi (last is 93-43)
	94	0	Unassigned
2	1–82	7,650	Hanzi (last is 82-36)
	83–94	0	Unassigned
14	1–68	6,319	Hanzi (last is 68-21)
	69–94	0	Unassigned
15	1–77	7,169	Hanzi (last is 77-25)
	78–94	0	Unassigned

* Sometimes CNS 11643-1986 Plane 14 is referred to as Plane E. Why is this? The letter "E" is the hexadecimal equivalent of decimal 14. Likewise, CNS 11643-1986 Plane 15 is often referred to as Plane F for the same reason.

So, what happened to Planes 14 and 15 when CNS 11643-1992 was established? The hanzi in CNS 11643-1986 Plane 14 were divided into multiple planes of CNS 11643-1992, and for a good reason: Plane 14 was composed of two independent collections of hanzi, both of which were independently ordered by total number of strokes. The first 6,148 hanzi in this plane (01-01 through 66-38) became CNS 11643-1992 Plane 3; the remaining 171 hanzi (66-39 through 68-21) were scattered throughout CNS 11643-1992 Plane 4, along with thousands of additional hanzi. Exactly where within CNS 11643-1992 Plane 4 these 171 hanzi were scattered is documented in a table on page 317 of the CNS 11643-1992 standard, and this same information is provided in Appendix H.

Plane 15 never became part of CNS 11643-1992 proper, but 338 of its hanzi are among those in CNS 11643-1992 Planes 4 through 7. When CNS 11643-2007 was established, Plane 15 was restored, and 6,831 of its original 7,169 hanzi are included in their original code points. This is precisely the reason why CNS 11643-2007 Plane 15 includes gaps.

Big Five versus CNS 11643-2007

Although the Big Five and CNS 11643-2007 character sets share many qualities, they are, in fact, different character sets. The following is an enumeration of some facts to consider:

- CNS 11643-2007 Plane 1 enumerates 5,401 hanzi, as does Big Five.
- CNS 11643-2007 Plane 2 enumerates 7,650 hanzi, but Big Five has two additional hanzi—both duplicately encoded.
- CNS 11643-2007 includes several additional planes of hanzi. Big Five has only two levels.
- Big Five does not enumerate the 213 classical radicals of CNS 11643-2007. Remember that 187 of these 213 classical radicals are identical to hanzi found in CNS 11643-2007 Planes 1 and 2, which correspond to Big Five Levels 1 and 2.
- A handful of character forms are different, such as CNS 11643-2007 Plane 1's 01-26 through 01-29 compared to Big Five's <A1 59> through <A1 5C>.
- A handful of characters are in a slightly different order, due to corrected stroke counts in CNS 11643-2007: two non-hanzi and six hanzi instances between Big Five Level 1 hanzi and CNS 11643-2007 Plane 1, and 17 hanzi instances between Big Five Level 2 hanzi and CNS 11643-2007 Plane 2.
- Big Five had become a *de facto* standard due to its long-standing use on Mac OS and Windows.

Consider these facts well when comparing and contrasting these two character sets from Taiwan. The mere fact that Big Five has become a *de facto* standard is often the best reason to adopt its use.

Table 3-49 lists the two non-hanzi and six hanzi that are in a different order in Big Five Level 1 and CNS 11643-2007 Plane 1.

Table 3-49. Different ordering—Big Five Level 1 versus CNS 11643-2007 Plane 1

Character	Big Five Level 1	CNS 11643-2007 Plane 1
←	A1 F6	02-56
→	A1 F7	02-55
耄	AC FE	55-51
銬	BE 52	75-48
薦	C2 CB	85-21
羅	C3 B9	88-69
繳	C3 BA	88-68
嚨	C4 56	88-13

Table 3-50 lists the 17 hanzi that are in a different order in Big Five Level 2 and CNS 11643-2007 Plane 2.

Table 3-50. Different ordering—Big Five Level 2 versus CNS 11643-2007 Plane 2

Hanzi	Big Five Level 2	CNS 11643-2007 Plane 2
刓	C9 BE	01-44
攷	CA F7	02-45
筘	D6 CC	30-67
莖	D7 7A	31-74
筍	DA DF	23-79
銷	EB F1	53-43
鴐	EC DE	55-02
鏐	EE EB	68-15
磿	F0 56	61-84
銻	F0 CB	58-08
鑋	F1 6B	71-65
鱉	F2 68	73-20
豐	F4 B5	70-45
鑽	F6 63	74-43
鬮	F9 C4	81-70
鸙	F9 C5	82-20
爨	F9 C6	82-32

I urge you to compare the code points in Tables 3-49 and 3-50 with the complete Big Five and CNS 11643-2007 code tables in Appendix H to verify that their ordering is

indeed different. Although these characters are ordered differently in these two character set standards established in Taiwan, this trivia is effectively rendered moot, or perhaps flattened, when these characters are encoded according to Unicode.

CCCII

One of the most well thought-out character set standards from Taiwan is known as CCCII (*Chinese Character Code for Information Interchange*; 中文資訊交換碼 *zhōngwén zīxùn jiāohuànmǎ*), which was developed by the Chinese Character Analysis Group (CCAG; 國字整理小組 *guózì zhěnglǐ xiǎozǔ*) in Taiwan. Its first version was published in 1980, followed by substantial revisions in 1982 and 1987.

CCCII is structured as 16 layers, each of which is composed of up to six consecutive 94×94 planes (there are a total of 94 planes). This results in a 94×94×94 cube for encoding characters. Each layer is allocated for a particular class of character. Table 3-51 lists what character classes are allocated to what layers.

Table 3-51. The structure of CCCII

Layer	Planes	Content
1	1–6	Non-hanzi and hanzi
2	7–12	Simplified hanzi (as used in China)
3–12	13–72	Variant forms of hanzi in Layer 1
13	73–78	Japanese kana and kanji
14	79–84	Korean jamo, hangul, and hanja
15	85–90	Reserved
16	91–94	Other characters

The hanzi in CCCII are arranged according to radical, and then by total number of strokes, in ascending order, of course. Table 3-52 illustrates the contents of CCCII Layer 1.

Table 3-52. The structure of CCCII Layer 1

Plane	Row	Characters	Content
1	1	0	Reserved for control codes
1	2–3		
1	4–10	0	Unassigned
1	11	35	Chinese punctuation
1	12–14	214	Classical radicals
1	15	78	Chinese numerals and phonetic symbols (zhuyin)
1	16–67	4,808	Most frequently used hanzi
1–3	68–64[a]	17,032	Next most frequently used hanzi

Table 3-52. The structure of CCCII Layer 1

Plane	Row	Characters	Content
3–6	65–5[b]	20,583	Other hanzi
6	6–94	0	Unassigned

a. This range spans Plane 1, row 68 through Plane 3, row 64.

b. This range spans Plane 3, row 65 through Plane 6, row 5.

CCCII Layer 1 thus provides the basic (but very large) set of hanzi. The remaining layers are used for variant forms of characters found in Layer 1. The relationship between the layers is very important to understand. Table 3-53 illustrates the relationship between variant forms in CCCII. CNS 11643-2007 references are provided for the sake of comparison.

Table 3-53. The relationship between CCCII layers

Hanzi	Layer	Plane	Row-Cell	Status	CNS 11643-2007
來	1	1	17-44	Standard form	1-43-84
来	2	7	17-44	Simplified form	4-04-38
俫	3	13	17-44	Variant form	3-15-47
徠	4	19	17-44	Variant form	1-58-26

Note how the four hanzi in Table 3-53 all share the same Row-Cell value, and differ only in which layer they exist (although the plane numbers appear to differ, they are all the first plane within each layer). This mechanism provides a very convenient and logical method to access simplified or otherwise variant forms of hanzi. The same cannot be said of CNS 11643-2007.

The latest nondraft version of CCCII, dated February of 1987, defines a total of 53,940 characters. A subsequent revision may have 75,684 characters (44,167 orthographics plus 31,517 variants). Professor Chang, one of the primary CCCII contributors, sadly passed away in 1997, and he left behind some unfinished work, including the finalization of these 75,684 characters. Professor Ching-Chun Hsieh (謝清俊 *xiè qīngjùn*) and other researchers are working to complete the next CCCII revision. Table 3-54 details the history of CCCII.

Table 3-54. The history of CCCII

Year	Characters	Description
1980	4,808	4,808 most frequently used hanzi
1982	17,032	17,032 next most frequently used hanzi—first revision
1985	33,357	Combined 1980 and 1982 sets plus revision
1985	11,517	11,517 additional variants

Table 3-54. The history of CCCII

Year	Characters	Description
1987	53,940	Volume III—combined and revision
1989	75,684	First variant revision draft

ANSI Z39.64-1989 (entitled *East Asian Character Code For Bibliographic Use*, or EACC as an abbreviated form) is a derivative work of CCCII that contains a total of 15,686 characters. Some consider EACC to effectively be an historical "snapshot" of CCCII, but it is actually a fairly important precursor to the development of Unicode, and it is still extensively used for bibliographic purposes.

While the structure of CCCII is something to be truly admired in that it establishes relationships between characters, such as simplified and other variants, contemporary font technologies—such as OpenType, which is covered in Chapter 6—provide the same level of glyph substitution functionality at a level beyond encoding.

Unicode compatibility with Big Five and CNS standards

The most important characters in Big Five and CNS 11643 are included in Unicode. All of Big Five Levels 1 and 2, along with CNS 11643-1986 Planes 1 and 2, are included in the very early versions of Unicode. A small number of their non-hanzi do not yet map directly to Unicode.

Table 3-42 listed the two hanzi in Big Five that are duplicately encoded. In order to maintain round-trip conversion capability, these two duplicate hanzi are mapped to Unicode's CJK Compatibility Ideographs block.

All seven planes of CNS 11643-1992 have almost complete coverage in Unicode's CJK Unified Ideographs, specifically the URO, Extension A, and Extension B. A small number of edge cases are not yet mapped to Unicode.

Chinese Character Set Standards—Hong Kong

Hong Kong (香港 *xiānggǎng*), considered a part of China since 1997 as a *Special Administrative Region* (SAR), uses many hanzi that are specific to its locale. Of the two most common Chinese character set standards in use today, China's GB 2312-80 and Taiwan's Big Five, Hong Kong has standardized on Big Five. But, Big Five was not sufficient for their needs. Several companies, such as DynaComware and Monotype Imaging, have developed their own—and conflicting—Hong Kong extensions for Big Five. These vendor-specific Hong Kong extensions are covered in Appendix E.

The Hong Kong character set standards described in this section are distinct from other CJKV character set standards in that the characters are not ordered in any meaningful way, such as by reading, by indexing radical, or by number of total or remaining strokes. In addition, their logical encoding "rows" exhibit difficult-to-explain gaps.

Hong Kong SCS-2008

Hong Kong SCS (*Hong Kong Supplementary Character Set*, sometimes further abbreviated as *HKSCS*) is an extension to Big Five, and a subset of Unicode, that currently includes 5,009 characters, 4,568 of which are hanzi. Its current version is referred to as Hong Kong SCS-2008 (*Hong Kong Supplementary Character Set, 2008 Revision*).*

The first version of Hong Kong SCS was published in September of 1999, as Hong Kong SCS-1999, and included 4,702 characters, 4,261 of which were hanzi. Its 2001, 2004, and 2008 revisions added only hanzi: 116, 123, and 68, respectively.

Table 3-55 details the number of characters in each version of Hong Kong SCS, along with the number of hanzi that map to CJK Unified Ideographs Extensions B and C, meaning that they include Unicode mappings that are beyond the BMP.

Table 3-55. The history of Hong Kong SCS

Year	Hanzi	Other characters	Total characters	Extension B	Extension C
1999	4,261	441	4,702	1,623	0
2001	4,377	441	4,818	1,652	0
2004	4,500	441	4,941	1,693	0
2008	4,568	441	5,009	1,712	1

Of the mainstream character sets, Hong Kong SCS has the greatest number of mappings to CJK Unified Ideographs Extension B, meaning non-BMP characters. This effectively means that BMP-only Unicode support is insufficient for handling Hong Kong SCS, regardless of its version.

Hong Kong SCS has a process in place for adding new hanzi. In addition, effective March 31, 2008, the principles for the inclusion of new hanzi stipulate that the proposed hanzi already be in Unicode. Given that Unicode is clearly the preferred way in which characters are encoded in today's OSes and applications, this stipulation is a good thing. Any new characters will thus be in Unicode, and will not be assigned a Big Five code point.

Appendix I provides a complete code table for the Hong Kong SCS-2008 character set, specifically the characters that are beyond Big Five.

Hong Kong GCCS

In 1994, as a precursor to Hong Kong SCS-1999, Hong Kong's Special Administrative Region (SAR) Government published a set of 3,049 hanzi that were above and beyond those available in Big Five. This character set is called Hong Kong GCCS (*Hong Kong Government Chinese Character Set*). Tze-loi Yeung's dictionary entitled 標準中文輸入碼大字典 (*biāozhǔn zhōngwén shūrùmǎ dà zìdiǎn*, literally meaning "Big Dictionary of Standard

* *http://www.ogcio.gov.hk/ccli/eng/hkscs/introduction.html*

Chinese Input Codes"; Juxian Guan, 1996) provides full coverage of both Big Five (13,053 hanzi) plus this set of 3,049 hanzi published by the Hong Kong government.

Some implementations of Hong Kong GCCS also include an additional set of 145 hanzi specified by Hong Kong's Department of Judiciary, which are encoded in rows 0x8A (132 hanzi) and 0x8B (13 hanzi) of Big Five encoding. A small number of these 145 hanzi are still not in Unicode.

Hong Kong SCS-2008 versus Hong Kong GCCS

For the most part, Hong Kong SCS-2008 (and the original Hong Kong SCS-1999) is a superset of Hong Kong GCCS. However, Hong Kong SCS-1999 explicitly excludes 106 hanzi found in Hong Kong GCCS, either because they were unified with hanzi in Big Five proper or in Hong Kong SCS, or because their sources could not be verified. And, a small number of hanzi were changed in terms of their prototypical glyphs.

Table 3-56 lists 84 of the 106 Hong Kong GCCS hanzi that were excluded from Hong Kong SCS, specifically those that were unified. The hanzi that is shown is rendered according to Hong Kong SCS. Also provided in this table are the corresponding CNS 11643-2007 and Unicode code points. Note that some of these hanzi map to CNS 11643-2007 Plane 3, which serves as an indicator that they are outside the definition of Big Five proper, at least when unification is ignored.

Table 3-56. Eighty-four Hong Kong GCCS hanzi excluded from Hong Kong SCS

Hanzi	Hong Kong GCCS	Hong Kong SCS	CNS 11643-2007	Unicode
箸	8E 69	BA E6	1-74-15	U+7BB8
篾	8E 6F	ED CA	2-62-23	U+7C06
糎	8E 7E	A2 61	1-03-03	U+7CCE
緒	8E AB	BA FC	1-74-37	U+7DD2
縝	8E B4	BF A6	1-81-78	U+7E1D
者	8E CD	AA CC	1-47-15	U+8005
耨	8E D0	BF AE	1-81-86	U+8028
菁	8F 57	B5 D7	1-65-61	U+83C1
蒨	8F 69	E3 C8	2-45-47	U+84A8
莅	8F 6E	DB 79	2-31-62	U+840F
覦	8F CB	BF CC	1-82-22	U+89A6
覩	8F CC	A0 D4	3-46-68	U+89A9
起	8F FE	B0 5F	1-56-36	U+8D77
都	90 6D	B3 A3	1-61-71	U+90FD
銹	90 7A	F9 D7	3-47-48	U+92B9
靜	90 DC	C0 52	1-82-91	U+975C

Table 3-56. Eighty-four Hong Kong GCCS hanzi excluded from Hong Kong SCS

Hanzi	Hong Kong GCCS	Hong Kong SCS	CNS 11643-2007	Unicode
響	90 F1	C5 54	1-91-32	U+97FF
靈	91 BF	F1 E3	2-69-15	U+9F16
蕚	92 44	92 42	3-46-44	U+8503
尅	92 AF	A2 59	1-02-89	U+5159
尩	92 B0	A2 5A	1-02-90	U+515B
尭	92 B1	A2 5C	1-02-92	U+515D
尰	92 B2	A2 5B	1-02-91	U+515E
鍮	92 C8	A0 5F	3-54-65	U+936E
瑮	92 D1	E6 AB	2-50-19	U+7479
涅	94 47	D2 56	2-16-25	U+6D67
禛	94 CA	E6 D0	2-50-56	U+799B
邢	95 D9	CA 52	2-02-82	U+9097
龑	96 44	9C E4	3-57-47	U+975D
瀞	96 ED	96 EE	3-58-50	U+701E
嬨	96 FC	E9 59	2-54-67	U+5B28
爁	9B 76	EF F9	2-66-07	U+7201
麤	9B 78	C5 F7	1-92-67	U+77D7
類	9B 7B	F5 E8	2-75-86	U+7E87
駖	9B C6	E8 CD	2-53-86	U+99D6
釔	9B DE	D0 C0	2-13-65	U+91D4
愾	9B EC	FD 64	3-21-89	U+60DE
澶	9B F6	BF 47	1-81-17	U+6FB6
輶	9C 42	EB C9	2-58-85	U+8F36
侻	9C 53	CD E7	2-09-09	U+4FBB
營	9C 62	C0 E7	1-84-18	U+71DF
鄄	9C 68	DC 52	2-32-87	U+9104
鷰	9C 6B	F8 6D	2-79-91	U+9DF0
荷	9C 77	DB 5D	2-31-34	U+83CF
尐	9C BC	C9 5C	2-01-28	U+5C10
秣	9C BD	AF B0	1-55-19	U+79E3
婧	9C D0	D4 D1	2-20-52	U+5A67
鞏	9D 57	E0 7C	2-40-04	U+8F0B
筑	9D 5A	B5 AE	1-65-20	U+7B51
拐	9D C4	A9 E4	1-45-70	U+62D0

Table 3-56. Eighty-four Hong Kong GCCS hanzi excluded from Hong Kong SCS

Hanzi	Hong Kong GCCS	Hong Kong SCS	CNS 11643-2007	Unicode
恢	9E A9	AB EC	1-49-16	U+6062
痹	9E EF	DE CD	2-37-19	U+75F9
汊	9E FD	C9 FC	2-02-61	U+6C4A
鬮	9F 60	F9 C4	2-81-70	U+9B2E
鼗	9F 66	91 BE	3-60-16	U+9F17
僭	9F CB	B9 B0	1-71-86	U+50ED
弍	9F D8	93 61	3-01-68	U+5F0C
蠯	A0 63	8F B6	3-59-11	U+880F
拎	A0 77	A9 F0	1-45-82	U+62CE
瑨	A0 D5	94 7A	3-40-02	U+7468
煢	A0 DF	DE 72	2-36-56	U+7162
牐	A0 E4	94 55	3-34-60	U+7250
倩	FA 5F	AD C5	1-52-08	U+5029
偽	FA 66	B0 B0	1-56-83	U+507D
包	FA BD	A5 5D	1-37-93	U+5305
廿	FA C5	A2 CD	1-04-31	U+5344
卿	FA D5	AD EB	1-52-46	U+537F
嘅	FB 48	9D EF	3-38-22	U+5605
婷	FB B8	B4 40	1-62-69	U+5A77
幵	FB F3	C9 DB	2-02-27	U+5E75
廐	FB F9	9D FB	3-38-64	U+5ED0
彘	FC 4F	D8 F4	2-27-56	U+5F58
悤	FC 6C	A0 DC	3-21-82	U+60A4
撐	FC B9	BC B5	1-76-93	U+6490
晴	FC E2	B4 B8	1-63-61	U+6674
杞	FC F1	A7 FB	1-42-61	U+675E
汴	FD B7	CB 58	2-04-56	U+6C9C
渝	FD B8	B4 FC	1-64-35	U+6E1D
港	FD BB	B4 E4	1-64-11	U+6E2F
煮	FD F1	B5 4E	1-64-52	U+716E
猪	FE 52	99 75	3-29-09	U+732A
瑜	FE 6F	B7 EC	1-69-20	U+745C
廷	FE AA	A2 60	1-03-02	U+74E9
耆	FE DD	CF F1	2-12-51	U+7809

Table 3-57 lists the 22 Hong Kong GCCS hanzi that could not be verified, and were thus excluded from Hong Kong SCS.

Table 3-57. Twenty-two unverifiable Hong Kong GCCS hanzi

Hanzi	Hong Kong GCCS
豅	9E AC
鬱	9E C4
蹕	9E F4
纆	9F 4E
嗯	9F AD
酗	9F B1
薛	9F C0
乺	9F C8
贗	9F DA
澞	9F E6
詈	9F EA
鎯	9F EF
衱	A0 54
熵	A0 57
恚	A0 5A
癵	A0 62
蚙	A0 72
嗷	A0 A5
礬	A0 AD
糨	A0 AF
亂	A0 D3
憔	A0 E1

The fact that these 22 Hong Kong GCCS hanzi could not be verified and were thus excluded from Hong Kong SCS does not mean that they do not exist. At some point, they may become verified and included in a future version of Hong Kong SCS, or may become part of Unicode through another source.

Unicode compatibility with Hong Kong standards

The entity that is charged with the development of Hong Kong SCS has stated that all future character submissions must be in Unicode, or will be submitted to the Ideographic Rapporteur Group (IRG). In terms of Unicode compatibility, this is a good thing, and means that Hong Kong will take an active role in submitting new characters to Unicode, as appropriate, as they are discovered.

Chinese Character Set Standards—Singapore

Singapore (新加坡 *xīnjiāpō*) does not have its own character set standard, and simply uses China's GB 2312-80 as its character set. However, 226 of the 6,582 ideographs in CJK Unified Ideographs Extension A, introduced in Unicode version 3.0, map to a Singapore source. The source is not named and is simply referred to as *Singapore characters*. To what extent these 226 characters are *ad-hoc*, or codified by a Singapore national standard, is unknown, at least to me. My suspicion is that they are *ad-hoc* simply for the apparent lack of any Singapore national standard.

Japanese Character Set Standards

Six CCSes are widely used in Japan. These character sets are ASCII, JIS-Roman, half-width katakana, JIS X 0208:1997 (and its predecessors), JIS X 0212-1990, and JIS X 0213:2004. ASCII and JIS-Roman were already discussed. JIS-Roman and half-width katakana* are described in JIS X 0201-1997. The most common of these CCSes is JIS X 0208:1997, which includes JIS Levels 1 and 2. The sixth standard, JIS X 0213:2004, defines JIS Levels 3 and 4.

This section includes a description of the latest character set standards established by Japan. The two most common Japanese character set standards are JIS X 0208:1997 and JIS X 0213:2004. JIS X 0221:2007, which is directly aligned with and equivalent to ISO 10646:2003 (with Amendments 1 and 2), is described in the section "International Character Set Standards," found later in this chapter.

Half-width katakana

Japan's first attempted to adapt their writing system to computer systems through the creation of half-width katakana. This formed a limited set of characters that could be easily encoded on early computer systems because they could be displayed in the same space as typical ASCII/JIS-Roman characters.† This collection of 63 half-width katakana characters is defined in the document JIS X 0201-1997, and consists of the basic katakana characters, along with enough punctuation marks and symbols to write Japanese text—but the result is not very readable.‡ Table 3-58 illustrates all the characters found in this relatively small character set.

* *http://www.ryukyu.ad.jp/~shin/jdoc/hankaku-kana.html*

† Also known as *hankaku* (半角 *hankaku*) katakana. Furthermore, some folks refer to these as *half-wit katakana*, either as a result of a typo, or for humorous purposes to colorfully describe their apparent lack of typographic usefulness.

‡ One could therefore argue that katakana is somewhat of a write-only writing system when used to completely express Japanese.

Table 3-58. The half-width katakana character set

Character class	Characters
Katakana	ヲアイウエオカキクケコサシスセソタチツテトナニヌネノハヒフヘホマミムメモヤユヨラリルレロワン
Small katakana	ｧｨｩｪｫｬｭｮｯ
Symbols	｡｢｣､･ﾞﾟｰ

Sometimes a half-width space character is considered part of this character set, encoded at `0xA0`, which brings the total to 64 characters.

Half-width katakana occupy half the display width of the equivalent full-width katakana found in JIS X 0208:1997 (described in the following section). The katakana characters enumerated in JIS X 0208:1997 are known as full-width characters.* Full-width, in this case, translates to roughly a square space, meaning that the width and the height of the character are the same. Half-width characters have the same height as full-width characters, but occupy half their width.

The dakuten- and handakuten-annotated counterparts of katakana are not included in the half-width katakana character set. The dakuten (゛) and the handakuten (゜) are used to create additional katakana characters. The dakuten and handakuten are treated as separate characters in the half-width katakana character set. Table 3-59 illustrates the relationship between half- and full-width katakana characters, and how dakuten and handakuten marks are treated as separate characters.

Table 3-59. Dakuten versus handakuten and full- versus half-width

Character class	ka	ga—dakuten	ha	pa—handakuten
Full-width	カカカカカ	ガガガガガ	ハハハハハ	パパパパパ
Half-width	ｶｶｶｶｶ	ｶﾞｶﾞｶﾞｶﾞｶﾞ	ﾊﾊﾊﾊﾊ	ﾊﾟﾊﾟﾊﾟﾊﾟﾊﾟ

When the ASCII/JIS-Roman and half-width katakana character set standards are combined into a single collection of characters, this newly formed character set is often referred to as ANK, short for *Alphabet, Numerals, and Katakana*.

JIS X 0208:1997—formerly JIS X 0208-1990

The first attempt by the Japanese to create a coded character set standard that better represented their written language bore fruit in 1978 with the establishment of JIS C 6226-1978. The work that eventually became JIS C 6226-1978 actually began as early as 1969. JIS C 6226-1978 represented the very first *national* coded character set standard to include ideographs, and is also significant in that it broke the *one-byte-equals-one-character*

* Also known as *zenkaku* (全角 *zenkaku*).

barrier. JIS C 6226-1978 went through three revisions to eventually become JIS X 0208:1997 on January 20, 1997.

The official title of the JIS X 0208:1997 standard is *7-Bit and 8-Bit Double Byte Coded Kanji Sets for Information Interchange* (7 ビット及び 8 ビットの 2 バイト情報交換用符号化漢字集合 *nana bitto oyobi hachi bitto no ni baito jōhō kōkan yō fugōka kanji shūgō*). The current version of this standard, JIS X 0208:1997, is considered the most basic Japanese coded character set. This character set standard enumerates 6,879 characters, most of which are kanji. The character space is arranged in a 94×94 matrix. Rows 1 through 8 are reserved for non-kanji, rows 9 through 15 are unassigned, rows 16 through 84 are reserved for kanji, and rows 85 through 94 are unassigned. Table 3-60 provides a much more detailed description of the characters allocated to each row (note that character allocation is identical to that of JIS X 0208-1990, but older versions are slightly different).

Table 3-60. The JIS X 0208:1997 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	53	Miscellaneous symbols
3	62	Numerals 0–9, upper- and lowercase Latin alphabet[a]
4	83	Hiragana
5	86	Katakana
6	48	Upper- and lowercase Greek alphabet
7	66	Upper- and lowercase Cyrillic alphabet
8	32	Full-width line-drawing elements
9–15	0	Unassigned
16–47	2,965	JIS Level 1 kanji (last is 47-51)
48–83	3,384	JIS Level 2 kanji (last is 83-94)
84	6	Additional kanji (last is 84-06)[b]
85–94	0	Unassigned

a. Usually implemented as full-width characters.

b. The 6 kanji in row 84 are usually considered part of JIS Level 2 kanji, so the total number of kanji that one would see for JIS Level 2 is 3,390, which includes row 84.

There are 6,355 kanji in this character set. The kanji are broken into two distinct sections. The first section is called *JIS Level 1 kanji* (JIS 第一水準漢字 *JIS daiichi suijun kanji*), and the kanji within it are arranged by On (old Chinese) reading.* The second section of kanji,

* Some kanji do not have an On reading. In these cases, they are arranged by their Kun (Japanese) reading. Also, there is one instance of an incorrectly ordered kanji in JIS X 0208:1997 Level 1 kanji. The kanji 馨 (*kaori*; 19-30) falls between 浬 (*kairi*; 19-29) and 蛙 (*kaeru*; 19-31), but this reading should come after that of 蛙 (19-31). This means that the sequence 浬馨蛙 should have been 浬蛙馨.

called *JIS Level 2 kanji* (JIS 第二水準漢字 *JIS daini suijun kanji*), are arranged by radical, and then by total number of strokes.* The six additional kanji in row 84 are arranged by radical, and then by number of strokes, like JIS Level 2 kanji. JIS Levels 1 and 2 kanji are mutually exclusive—each level contains no kanji found in the other. Together they constitute a set of 6,355 unique kanji.†

A complete code table for the characters that constitute JIS X 0208:1997 can be found in Appendix J, along with a reading index for JIS Level 1 kanji and a radical index for JIS Level 2 kanji.

Table 3-61 provides a graphic representation for the first and last characters from each of the character classes (note that the complete set for numerals and additional kanji is provided).

Table 3-61. JIS X 0208:1997 character samples

Character class	Sample characters
Miscellaneous symbols	、。，．・：；？！ … ∬Å‰♯♭♪†‡¶○
Numerals	0 1 2 3 4 5 6 7 8 9
Latin	A B C D E F G H I J … q r s t u v w x y z
Hiragana	ぁあぃいぅうぇえぉお … りるれろゎわゐゑをん
Katakana	ァアィイゥウェエォオ … ロヮワヰヱヲンヴヵヶ
Greek	Α Β Γ Δ Ε Ζ Η Θ Ι Κ … ο π ρ σ τ υ φ χ ψ ω
Cyrillic	А Б В Г Д Е Ё Ж З И … ц ч ш щ ъ ы ь э ю я
Line-drawing elements	─ │ ┌ ┐ └ ┘ ├ ┤ ┬ ┴ … ┣ ┳ ┫ ┻ ╋
JIS Level 1 kanji	亜唖娃阿哀愛挨姶逢葵 … 亙亘鰐詫藁蕨椀湾碗腕
JIS Level 2 kanji	弌丐丕个丱丶丼丿乂乖 … 齦齧齬齟齠齡齲齶龕龜龠
Additional kanji	堯槇遙瑤凜熙

Symbols include punctuation marks, mathematical symbols, and various types of parentheses. Numerals and Latin characters are what one would normally find in the ASCII character set (less the ASCII/JIS-Roman symbols, which are scattered throughout row 1)—these are full-width, not half-width. The hiragana and katakana characters, too, are full-width, not half-width. Cyrillic and Greek characters are included, perhaps because Japanese technical works include occasional Russian or Greek words. The full-width line-drawing elements are used for building charts on a per-character basis—not terribly useful in this day and age of applications with built-in support for graphics and tables.

* Actually, the ordering is based on the order of entries in the kanji dictionary entitled 新字源 (*shinjigen*). Compare 鬮 (79-72) and 澗 (79-73), whose indexing radicals are 門 and 氵 (水), respectively. Their 新字源 (1994 edition) index numbers are 8831 and 8832, respectively.

† This is true only if you count character variants as separate entities.

This character set standard was first established on January 1, 1978 as JIS C 6226-1978, modified for the first time on September 1, 1983 as JIS X 0208-1983, modified again on September 1, 1990 as JIS X 0208-1990, and finally became JIS X 0208:1997 on January 20, 1997.* It is widely implemented on a variety of platforms. Encoding methods for JIS X 0208:1997 include ISO-2022-JP, EUC-JP, and Shift-JIS. The encoding forms of Unicode can also be used to represent these characters. These encodings are covered in Chapter 4.

JIS X 0208:1997, although it did not add any characters to the character set, does offer some significant improvements to the standard itself, described as follows:

- Explicitly describes ISO-2022-JP and Shift-JIS encodings. Previous installments of this standard, specifically the JIS C 6226 and JIS X 0208 series, included no such specifications.
- More clearly defines the kanji unification rules and principles, and applies them to the standard.
- Provides an extremely thorough treatment of kanji variant forms using these well-established unification principles.

You may sometimes encounter systems and documentation that are based on earlier versions of JIS X 0208:1997, the most likely of which is JIS X 0208-1983. That standard was originally known as JIS C 6226-1983. On March 1, 1987, JSA decided to rename many JIS standards from a "C" to an "X" designation (don't ask me why). JIS C 6226-1983, with no substantive changes, was renamed to JIS X 0208-1983. Table 3-62 illustrates this evolution of the JIS X 0208 series.

Table 3-62. The evolution of JIS X 0208

Year	Designation	Status
1978	JIS C 6226-1978	Establishment
1983	JIS C 6226-1983	Update
1987	JIS X 0208-1983	Designation change
1990	JIS X 0208-1990	Update
1997	JIS X 0208:1997	Update, but no change in number of characters

Since its conception in 1978, this character set standard has experienced a slight increase in the total number of characters. Table 3-63 lists how characters are allocated to each row in the 1978, 1983, and 1990 (same as 1997) versions.

* It is common practice to review standards every five years or so, and reissue them, if necessary.

Table 3-63. Comparing different versions of JIS X 0208

Row	1978	1983	1990	Content
1	94	94	94	Miscellaneous symbols
2	14	53	53	Miscellaneous symbols
3	62	62	62	Numerals 0–9, upper- and lowercase Latin alphabet
4	83	83	83	Hiragana
5	86	86	86	Katakana
6	48	48	48	Upper- and lowercase Greek alphabet
7	66	66	66	Upper- and lowercase Cyrillic alphabet
8	0	32	32	Full-width line-drawing elements
9–15	0	0	0	Unassigned
16–47	2,965	2,965	2,965	JIS Level 1 kanji (last is 47-51)
48–83	3,384	3,384	3,384	JIS Level 2 kanji (last is 83-94)
84	0	4	6	Additional kanji
85–94	0	0	0	Unassigned

More detailed information about the differences between JIS C 6226-1978, JIS X 0208-1983, JIS X 0208-1990, and JIS X 0208:1997 can be found in Appendix J.

Some of the similarities between JIS X 0208:1997 and GB 2312-80 (Chinese) are quite close. First, note how the allocation of rows 4 through 7 (hiragana, katakana, Greek, and Cyrillic characters) is identical in both character sets. Also, rows 1 through 15 are reserved for nonideographs. And finally, the ideographs are divided into two levels, with the first level being the most frequently used and arranged by reading, and the second level being more rarely used and arranged by radical, and then by total number of strokes.

JIS X 0212-1990—a supplemental character set

A supplemental Japanese character set standard, JIS X 0212-1990, was established by JISC on October 1, 1990, and specified 6,067 characters (5,801 kanji plus 266 non-kanji). These characters are in addition to those found in JIS X 0208:1997, but like that character set standard, JIS X 0212-1990 is also composed of a 94×94 character space. Also like JIS X 0208:1997, rows 1 through 15 are reserved for non-kanji, rows 9 through 15 are unassigned, rows 16 through 84 are reserved for kanji, and rows 85 through 94 are unassigned.

The official title of JIS X 0212-1990 is *Code of the Supplementary Japanese Graphic Character Set for Information Interchange* (情報交換用漢字符号—補助漢字 *jōhō kōkan yō kanji fugō—hojo kanji*). Table 3-64 lists how characters are allocated to each of its 94 rows.

Table 3-64. The JIS X 0212-1990 character set

Row	Characters	Content
1	0	Unassigned
2	21	Diacritics and miscellaneous symbols
3–5	0	Unassigned
6	21	Greek characters with diacritics
7	26	Eastern European characters
8	0	Unassigned
9–11	198	Miscellaneous alphabetic characters
12–15	0	Unassigned
16–77	5,801	Supplemental kanji (last is 77-67)
78–94	0	Unassigned

The 5,801 kanji are arranged by radical, and then total number of strokes (like JIS Level 2 kanji of JIS X 0208:1997). When these data are merged with JIS X 0208:1997, you see that there are now 12,156 unique standard kanji, and 12,946 total characters. However, very few software systems can use these 6,067 supplemental characters. The latest version of GNU Emacs, now a powerful multilingual text editor, is an example of a program that supports the encoding of the JIS X 0212-1990 character set standard. The Unix/Linux *kterm* terminal console program has supported JIS X 0212-1990 for years. Much of the difficulty in supporting JIS X 0212-1990 is the poor availability of fonts that include its characters.

Table 3-65 illustrates the first and last 10 characters in each class of characters listed previously.

Table 3-65. JIS X 0212-1990 character samples

Character class	Sample characters
Miscellaneous symbols	˘ ˇ ¸ ˙ ˝ ¯ ˛ ° ˜ ˈ … ¡ ¦ ¿ º ª ©®™ ¤№
Greek	Ά Έ Ή Ί Ϊ Ό Ύ Ϋ Ώ ά … ή ί ΰ ς ό ύ ϋ ύ ώ
Eastern European	Ђ Ѓ Є S I Ї J Љ Њ Ћ … s i ї j љ њ ћ ќ ў џ
Alphabetic	Æ Đ Ħ Ĳ Ł Ŀ Ŋ Ø Œ Ŧ … ů ǔ ű ŵ ý ÿ ŷ ź ž ż
Supplemental kanji	丐 丄 丅 丌 丒 丟 乹 两 丨 丫 … 龔 龕 龔 龖 龗 龝 龡 龤 龥

A complete code table for the characters that make up JIS X 0212-1990 can be found in Appendix J, as can a radical index for the 5,801 kanji of JIS X 0212-1990.

At one point, JISC considered the possibility of adding four katakana characters to JIS X 0212-1990, as listed in Table 3-66.

Table 3-66. Four characters that could have been added to JIS X 0212-1990

Katakana	Reading
ヷ	va
ヸ	vi
ヹ	ve
ヺ	vo

These four characters, although rarely used, are employed for writing foreign words, and can already be found in at least one vendor character set standard—specifically Apple's Mac OS-J, which is covered in Appendix E. Encoding space for these four characters has been allocated in JIS X 0212-1990—if they had been accepted for inclusion into this character set standard, they would have been encoded in row 5, beginning at cell 87 (in other words, from 05-87 through 05-90).* Table 3-67 illustrates these possible changes.

Table 3-67. Proposed change to the JIS X 0212-1990 character set

Row	Characters	Content
1	0	Unassigned
2	21	Diacritics and miscellaneous symbols
3–4	0	Unassigned
5	4	Katakana
6	21	Greek characters with diacritics
7	26	Eastern European characters
8	0	Unassigned
9–11	198	Miscellaneous alphabetic characters
12–15	0	Unassigned
16–77	5,801	Supplemental kanji (last is 77-67)
78–94	0	Unassigned

It would seem natural that JIS X 0213:2000 would incorporate these four additional katakana characters. Well, it did, but not at 05-87 through 05-90, but rather at 07-82 through 07-85, immediately following the lowercase Cyrillic characters. Of course, these four characters are included in Unicode, from U+30F7 through U+30FA.

Incorporating the 6,067 characters of JIS X 0212-1990 into ISO-2022-JP encoding was trivial: a new two-byte character escape sequence (explained in Chapter 4) for this new

* There is significance to this starting point. If one were to overlay the non-kanji portions of JIS X 0208:1997 and JIS X 0212-1990 (that is, rows 1 through 16), there would be zero instances of characters in one character set occupying the code point of characters in the other. The katakana in JIS X 0208:1997 end at 05-86, so using 05-87 as the starting code point for these four additional katakana seems like a logical thing to do.

character set was registered. ISO-2022-JP-2 encoding was subsequently born. It is not possible to encode JIS X 0212-1990 in Shift-JIS encoding because there is not enough encoding space left to accommodate its characters. EUC-JP encoding does not suffer from this problem of limited encoding space, and in Chapter 4 you will learn how JIS X 0212-1990 is supported by EUC-JP encoding. In the end, though, what matters most today is how the characters are encoded according to Unicode. The importance of Unicode compatibility is especially true for JIS X 0213:2004, which is described in the next section.

JIS X 0213:2004

JSA has developed a new character set standard, whose designation is JIS X 0213:2000, and defines JIS Levels 3 and 4.* Its title is *7-bit and 8-bit double byte coded extended KANJI sets for information interchange* (7 ビット及び 8 ビットの 2 バイト情報交換用符号化拡張漢字集合 *nana bitto oyobi hachi bitto no ni baito jōhō kōkan yō fugōka kakuchō kanji shūgō*). JIS Level 3 contains 1,249 kanji, JIS contains 2,436 kanji, and there are also 659 symbols—for a grand total of 4,344 characters.

This standard was revised in 2004, and in terms of additional characters, there are only 10, all of which are kanji. These became part of JIS Level 3 by virtue of being in Plane 1, which now has 1,259 kanji. The total number of characters in the standard is now 4,354.

When JIS X 0213:2004 is combined with JIS X 0208:1997 and encoded according to Shift-JIS encoding, all but 47 of its code points are used. That is, out of the possible 11,280 Shift-JIS code points, 11,233 are filled by JIS X 0208:1997 and JIS X 0213:2004 characters.†

Many JIS X 0212-1990 non-kanji and kanji are included in JIS X 0213:2004, which effectively means that JIS X 0212-1990 may no longer be maintained by JSA. However, the fact that JIS X 0212-1990 has complete coverage in Unicode from its earliest versions means that there is a legacy issue that JSA must accept. This basically means that JIS X 0212-1990 will live on for many years to come.

The best way to think of JIS X 0213:2004 is as an add-on or supplement to JIS X 0208:1997, or as a superset. In fact, that is how it is intended to be used. Tables 3-68 and 3-69 list the characters, on a per-row basis, for JIS X 0208:1997 and JIS X 0213:2004 when combined into a larger set of characters.

Table 3-68. The JIS X 0213:2004 character set, Plane 1—combined with JIS X 0208

Row	X 0208	X 0213	Content
1	94	0	Miscellaneous symbols
2	53	41	Miscellaneous symbols

* Some people (mistakenly) referred to JIS X 0212-1990 as *JIS Level 3 kanji*, but the establishment of JIS X 0213:2000 has set the record straight once and for all.

† Interestingly, Shift-JIS encoding for JIS X 0213 is Informative, not Normative. The official way in which JIS X 0213 is encoded is Unicode.

Table 3-68. The JIS X 0213:2004 character set, Plane 1—combined with JIS X 0208

Row	X 0208	X 0213	Content
3	62	32	Numerals 0–9, upper- and lowercase Latin alphabet, miscellaneous symbols
4	83	8	Hiragana
5	86	8	Katakana
6	48	46	Upper- and lowercase Greek alphabet, katakana, miscellaneous symbols
7	66	28	Upper- and lowercase Cyrillic alphabet, miscellaneous symbols
8	32	52	Full-width line-drawing elements, miscellaneous symbols
9–10	0	188	Additional Latin characters
11	0	94	IPA
12	0	85	Annotated
13	0	77	NEC Row 13
14	0	94	JIS Level 3 kanji (04-01 added in 2004)
15	0	94	JIS Level 3 kanji (15-94 added in 2004)
16–46	2,914	0	JIS Level 1 kanji
47	51	43	JIS Level 1 kanji (47-01 through 47-51), JIS Level 3 kanji (47-52 through 47-94; 47-52 and 47-94 added in 2004)
48–83	3,384	0	JIS Level 2 kanji (last is 83-94)
84	6	88	Additional JIS X 0208 kanji (84-01 through 84-06), JIS Level 3 kanji (84-07 through 84-94; 84-07 added in 2004)
85–94	0	940	JIS Level 3 kanji (last is 94-94; 94-90 through 94-94 added in 2004)

Table 3-69. The JIS X 0213:2004 character set, Plane 2

Row	Characters	Content
1	94	JIS Level 4 kanji
2	0	Unassigned
3–5	282	JIS Level 4 kanji
6–7	0	Unassigned
8	94	JIS Level 4 kanji
9–11	0	Unassigned
12–15	376	JIS Level 4 kanji
16–77	0	Unassigned
78–94	1,590	JIS Level 4 kanji (last is 94-86)

Note how the unassigned rows in JIS X 0213:2004 Plane 2 correspond to the rows in JIS X 0212-1990 that have characters assigned. This is so that the EUC-JP implementation of

JIS X 0213:2004 does not trample on JIS X 0212-1990, effectively allowing the standards to coexist.

Table 3-70 lists the 10 kanji that were added to JIS Level 3 in JIS X 0213:2004.

Table 3-70. Ten kanji added to JIS Level 3 in 2004

Kanji	Row-Cell	Related kanji	Row-Cell
俱	14-01	俱	22-70
剝	15-94	剥	39-77
叱	47-52	叱	28-24
呑	47-94	呑	38-61
噓	84-07	噓	17-19
姸	94-90	姸	53-11
屛	94-91	屛	54-02
幷	94-92	并	54-85
痩	94-93	痩	33-73
繋	94-94	繋	23-50

Relationships among Japanese character set standards

You already read about the slight difference between the ASCII and JIS-Roman character sets. With only one exception, the character set standards JIS X 0208:1997 and JIS X 0212-1990 contain no characters found in the other—together they are designed to form a larger collection of characters (12,156 kanji plus 790 non-kanji). These two duplicate characters are illustrated in Table 3-71.

Table 3-71. Duplicate characters in JIS X 0208:1997 and JIS X 0212-1990

Standard	Kanji	Row-Cell	Unicode
JIS X 0208:1997	〆	01-26	U+3006
JIS X 0212-1990	〆	16-17	U+4E44

The characters are the same, have identical meanings, and are used in the same contexts—it is really only a character-classification difference. In JIS X 0208:1997, this character is treated as a non-kanji (in a row of symbols), but in JIS X 0212-1990, it is treated as a full-fledged kanji. This character, in both instances, is read *shime*, and means "deadline," "(to) sum up," or "seal," depending on context.

The internal structures of JIS X 0208:1997 and JIS X 0212-1990 share several unique qualities, the most notable being that they are both composed of a 94×94 character space for a maximum number of 8,836 characters. Thus, they both occupy the same character space. Furthermore, the non-kanji characters of both standards are allocated to rows 1 through

15, and the kanji characters are allocated to rows 16 through 84 (that is not to say that all those rows are currently filled, but rather that they have been allocated for those character classes). Chapter 4 discusses how computer systems can distinguish between these two character sets using different encoding methods.

Another interesting aspect of these character set standards is how the non-kanji are arranged so that if one superimposed one set onto the other, there would be absolutely no overlap of assigned character positions. This would make it possible to merge rows 1 through 15 of both standards with no assigned character positions overlapping. In fact, the four katakana characters that may eventually be added to JIS X 0212-1990 are positioned in such a way that they would appear immediately after the katakana assigned to JIS X 0208:1997.*

There is one last tidbit of information to mention about the relationship between these two character set standards. There are 28 kanji in JIS X 0212-1990 that were in JIS C 6226-1978, but were replaced with different glyphs in JIS X 0208-1983. In essence, 28 kanji that were lost during the transition from JIS C 6226-1978 to JIS X 0208-1983 were restored in JIS X 0212-1990. Table 3-72 lists these 28 kanji pairs.

Table 3-72. Twenty-eight JIS C 6226-1978 kanji in JIS X 0212-1990

JIS C 6226-1978		JIS X 0208:1997		JIS X 0212-1990	
俠	22-02	俠	22-02	俠	17-34
啞	16-02	唖	16-02	啞	21-64
噛	19-90	噛	19-90	噛	22-58
嚢	39-25	嚢	39-25	嚢	22-76
塡	37-22	填	37-22	塡	24-20
屢	28-40	屡	28-40	屢	26-90
掻	33-63	掻	33-63	掻	32-43
摑	36-47	掴	36-47	摑	32-59
攅	58-25	攅	58-25	攅	33-34
潑	40-14	溌	40-14	潑	40-53
瀆	38-34	涜	38-34	瀆	41-12
焔	17-75	焔	17-75	焔	41-79
痩	33-73	痩	33-73	痩	45-87
禱	37-88	祷	37-88	禱	48-80
繍	29-11	繍	29-11	繍	52-55
繋	23-50	繋	23-50	繋	52-58

* This would be difficult to implement because numerous vendors have effectively filled these open rows of JIS X 0208:1997.

Table 3-72. Twenty-eight JIS C 6226-1978 kanji in JIS X 0212-1990

JIS C 6226-1978		JIS X 0208:1997		JIS X 0212-1990	
萊	45-73	莱	45-73	萊	56-39
蔣	30-53	蒋	30-53	蔣	57-22
蠟	47-25	蜡	47-25	蠟	59-88
軀	22-77	躯	22-77	軀	64-52
醬	30-63	酱	30-63	醬	66-83
醱	40-16	酽	40-16	醱	66-87

Unicode compatibility with JIS standards

JIS X 0208:1997 and JIS X 0212-1990 are compatible with the earliest versions of Unicode, specifically version 1.0.1 and greater. JIS X 0213:2004 is compatible with Unicode version 3.2 and greater. In other words, all three JIS standards are fully supported in the context of Unicode version 3.2.

Interestingly, a small number of characters in JIS X 0213:2004 (25 to be exact) cannot be represented in Unicode using a single code point, and instead require a sequence of two Unicode code points. Table 3-73 lists these 25 characters, along with their singular JIS X 0213:2004 Row-Cell values (they are all in Plane 1) and the sequence of two code points necessary to represent them in Unicode.

Table 3-73. Special Unicode handling of 25 JIS X 0213:2004 characters

Character	Row-Cell	Unicode sequence
が	04-87	<U+304B, U+309A>
ぎ	04-88	<U+304D, U+309A>
ぐ	04-89	<U+304F, U+309A>
げ	04-90	<U+3051, U+309A>
ご	04-91	<U+3053, U+309A>
ガ	05-87	<U+30AB, U+309A>
ギ	05-88	<U+30AD, U+309A>
グ	05-89	<U+30AF, U+309A>
ゲ	05-90	<U+30B1, U+309A>
ゴ	05-91	<U+30B3, U+309A>
ゼ	05-92	<U+30BB, U+309A>
ヅ	05-93	<U+30C4, U+309A>
ド	05-94	<U+30C8, U+309A>
プ	06-88	<U+31F7, U+309A>

Table 3-73. Special Unicode handling of 25 JIS X 0213:2004 characters

Character	Row-Cell	Unicode sequence
æ̀	11-36	<U+00E6, U+0300>
ɔ̀	11-40	<U+0254, U+0300>
ɔ́	11-41	<U+0254, U+0301>
ʌ̀	11-42	<U+028C, U+0300>
ʌ́	11-43	<U+028C, U+0301>
ə̀	11-44	<U+0259, U+0300>
ə́	11-45	<U+0259, U+0301>
ɚ̀	11-46	<U+025A, U+0300>
ɚ́	11-47	<U+025A, U+0301>
˩˥	11-69	<U+02E9, U+02E5>
˥˩	11-70	<U+02E5, U+02E9>

In addition, 303 kanji in JIS X 0213:2004 are mapped to CJK Unified Ideographs Extension B, which necessitates beyond-BMP support. JIS X 0213:2000 had only 302 kanji that mapped beyond the BMP, and one of the 10 kanji added during the 2004 revision effectively became the 303rd kanji to map beyond the BMP.

Also of significance is that 82 kanji of JIS X 0213:2004 map to Unicode's CJK Compatibility Ideographs block in the BMP.

Korean Character Set Standards

Korean character set standards have been developed by South Korea, North Korea, and China, and some of them demonstrate some very unique attributes, such as the following:

- Contain thousands of hangul syllables
- Hanja (ideographs) with multiple readings are encoded more than once

In essence, hangul are treated as though they were ideographs as far as character-allocation is concerned. This is quite natural, because hangul play an important role in the Korean writing system.

KS X 1001:2004

The most commonly used Korean character set standard, specified in the document KS X 1001:2004 and entitled *Code for Information Interchange (Hangeul and Hanja)* (정보 교환용 부호계 (한글 및 한자) *jeongbo gyohwanyong buhogye (hangeul mich hanja)*),

enumerates 8,227 characters.* This standard was established on December 28, 2004 by the Korean Standards Association (also known as the Korean Bureau of Standards) of South Korea (Republic of Korea or ROK; 대한민국/大韓民國 *daehan minguk*).

The KS X 1001:2004 standard contains 4,888 hanja and 2,350 hangul syllables, both arranged by reading. The older KS X 1001:1992 standard contained 986 symbols, but KS X 1001:2004 now includes 989 symbols. The euro currency symbol and registered trademark symbol were added in KS X 1001:1998 at Row-Cell positions 02-70 and 02-71, respectively. KS X 1001:2002 added only the new Korean postal code symbol, written ㉾, at Row-Cell position 02-72 (its Unicode code point is U+327E). Table 3-74 lists the characters that constitute KS X 1001:2004.

Table 3-74. The KS X 1001:2004 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2[a]	72	6 abbreviations, 66 miscellaneous symbols
3	94	Full-width KS X 1003:1993 (KS-Roman; equivalent to ASCII)
4	94	Jamo (hangul elements)
5	68	Upper- and lowercase Roman numerals 1–10, 48 upper- and lowercase Greek alphabet
6	68	Full-width line-drawing elements
7	79	Abbreviations
8	91	13 alphabetic characters, 28 encircled jamo and hangul, encircled lowercase Latin characters, encircled numerals 1–15, 9 fractions
9	94	16 alphabetic characters, 28 parenthesized jamo and hangul, parenthesized lowercase Latin characters, parenthesized numerals 1–15, 5 superscripts, 4 subscripts
10	83	Hiragana
11	86	Katakana
12	66	Upper- and lowercase Cyrillic alphabet
13–15	0	Unassigned
16–40	2,350	Hangul syllables (last is 40-94)
41	0	Unassigned
42–93	4,888	Hanja (last is 93-94)
94	0	Unassigned

a. KS X 1001:1992 included only 69 characters in this row, and KS X 1001:1998 included only 71 characters in this row.

Due to the multiple readings for some hanja, 268 of the 4,888 hanja in KS X 1001:2004 are genuine duplicate characters—most of these duplicate characters are single instances of repeated characters, meaning that there are two instances of the same character in the

* Previously designated KS C 5601-1992

character set, but a small number of hanja are repeated more than once! This effectively means that there are 4,620 unique hanja in KS X 1001:2004, not 4,888. Table 3-75 provides three example hanja from KS X 1001:2004, each repeated a different number of times. Their corresponding Unicode code points are also provided.

Table 3-75. Repeated hanja in KS X 1001:2004—examples

Hanja	Row-Cell	Unicode
賈	42-25, 45-47	U+8CC8, U+F903
龜	47-47, 48-02, 48-24	U+9F9C, U+F907, U+F908
樂	49-66, 53-05, 68-37, 72-89	U+F914, U+F95C, U+6A02, U+F9BF

KS X 1001:2004 is the only CJKV character set that multiply encodes ideographs due to multiple readings. Big Five includes two duplicate hanzi, but that was due to an error *in* design, not *by* design.

Appendix K provides a complete KS X 1001:2004 code table, along with a complete listing of its 268 duplicate hanja. The same appendix provides reading indexes for the hangul and hanja in KS X 1001:2004. Table 3-76 illustrates the many character classes in KS X 1001:2004.

Table 3-76. KS X 1001:2004 character samples

Character class	Sample characters
Miscellaneous symbols	、。·‥…¨〃―∥ … ♪♬㉿㈜№㏇™㏂㏘℡
Full-width KS-Roman	！＂＃＄％＆＇（） … ｕｖｗｘｙｚ｛｜｝￣
Jamo	ㄱㄲㄳㄴㄵㄶㄷㄸㄹㄺ … ㆅㆆㆇㆈㆉㆊㆋㆌㆍㆎ
Roman numerals	ⅰⅱⅲⅳⅴⅵⅶⅷⅸⅹ … ⅠⅡⅢⅣⅤⅥⅦⅧⅨⅩ
Greek	ΑΒΓΔΕΖΗΘΙΚ … οπρστυφχψω
Line-drawing elements	─│┌┐┘└├┬┤┴ … ╀╁╃╄╅╆╇╈╉╊
Latin ligatures	㎕㎖㎗ℓ㎘㏄㎣㎤㎥㎦ … ㎪㎫㎬㏝㏐㏓㏃㏉㏜㏆
Alphabetic characters	ÆÐªĦĲĿŁØŒº … ĸŀłøœßþŧŋŉ
Encircled jamo/hangul	㉠㉡㉢㉣㉤㉥㉦㉧㉨㉩ … ㉲㉳㉴㉵㉶㉷㉸㉹㉺㉻
Annotated Latin/numerals	ⓐⓑⓒⓓⓔⓕⓖⓗⓘⓙ … ⑥⑦⑧⑨⑩⑪⑫⑬⑭⑮
Fractions	½⅓⅔¼¾⅛⅜⅝⅞
Parenthesized jamo/hangul	㈀㈁㈂㈃㈄㈅㈆㈇㈈㈉ … ㈒㈓㈔㈕㈖㈗㈘㈙㈚㈛
Parenthesized Latin/numerals	⒜⒝⒞⒟⒠⒡⒢⒣⒤⒥ … ⑹⑺⑻⑼⑽⑾⑿⒀⒁⒂
Superscripts and subscripts	^{1} ^{2} ^{3} ^{4} ^{n} $_{1}$ $_{2}$ $_{3}$ $_{4}$
Hiragana	ぁあぃいぅうぇえぉお … りるれろゎわゐゑをん
Katakana	ァアィイゥウェエォオ … ロヮワヰヱヲンヴヵヶ
Cyrillic	АБВГДЕЁЖЗИ … цчшщъыьэюя

Table 3-76. KS X 1001:2004 character samples

Character class	Sample characters
Hangul syllables	가각간갇갈갉갊감갑값 … 휩흉히힉힌힐힘힙힛힝
Hanja	伽佳假價加可呵哥嘉嫁 … 晞曦熙熹熺犧禧稀羲詰

The hanja specified in KS X 1001:2004 are considered to be in the traditional form. Some examples of simplified versus traditional ideographs are listed in Chapter 2.

Encoding methods that support the KS X 1001:2004 character set include ISO-2022-KR, EUC-KR, Johab, and Unified Hangul Code (UHC). Of course, the encoding forms of Unicode also support these characters.

This Korean character set standard is similar to JIS X 0208:1997 and GB 2312-80 in that it contains the same set of hiragana, katakana, Greek, and Cyrillic characters (but in different rows). And, although hangul are not considered the same as hanja, they do begin at row 16, like the ideographs in JIS X 0208:1997 and friends.

Earlier versions of this standard were designated KS C 5601-1987 and KS C 5601-1989 (the latter was established on April 22, 1989)—their character set being identical. The differences between versions are the annexes and their contents.

Historically speaking, there was a standard designated KS C 5601-1982, but it enumerated only the 51 basic jamo in a one-byte, 7- and 8-bit encoding. This information is still part of the KS X 1001 series in the form of an annex (Annex 4 of KS X 1001:2004).

In the very early days of Korean information processing, there were character set standards known as KS C 5619-1982 and KIPS (*Korean Information Processing System*). KS C 5619-1982 enumerated only 51 jamo (modern jamo), 1,316 hangul syllables, and 1,672 hanja. KIPS, on the other hand, enumerated 2,058 hangul syllables and 2,392 hanja. Both of these standards were rendered obsolete by KS C 5601-1987, which is currently designated KS X 1001:2004.

KS X 1001:2004—an alternate plane

Annex 3 of the KS X 1001:2004 standard describes an extension whereby all possible modern hangul syllables, 11,172 of them built up using the basic set of 51 jamo, are encoded.* This alternate plane of KS X 1001:2004, which also enumerates the same set of symbols and hanja (but at different code points), is known as *Johab* (조합/組合 *johap*), which means "combining." The standard plane of KS X 1001:2004—the one that enumerates only 2,350 hangul—is known as *Wansung* (완성/完成 *wanseong*), which means "precomposing."

* Encoding all 11,172 possible modern hangul syllables is almost like encoding all possible three-letter words in English—while all combinations are possible, only a fraction represent real words.

Hangul can be composed of two or three jamo (some jamo are considered compound). Johab uses 19 initial jamo (consonants), 21 medial jamo (vowels), and 27 final jamo (consonants; there are 28 when you include the "fill" character for hangul containing only 2 jamo). Multiplying these numbers (19×21×28) results in the figure 11,172, which matches the total number of modern hangul syllables in Johab.

Johab is best explained in the context of encoding methods, so any further discussion is deferred until Chapter 4.

KS X 1002:2001

South Korea (Republic of Korea) developed an extended character set designated KS X 1002:2001, entitled *Extension Code for Information Interchange* (정보 교환용 부호 확장 세트 *jeongbo gyohwanyong buho hwakjang seteu*).* This character set, originally established on December 31, 1991, and revised on December 6, 2001, provides an additional 3,605 hangul syllables (all of which, by the way, are covered by Johab, Unified Hangul Code, and Unicode version 2.0; you will learn about these later in the book), 2,856 hanja (ordered by reading), and 1,188 other characters, for a total of 7,649 characters. These 2,856 hanja are listed in Appendix K. Table 3-77 lists the characters in KS X 1002:2001.

Table 3-77. The KS X 1002:2001 character set

Row	Characters	Content
1–7	613	Lower- and uppercase Latin characters with diacritics
8–10	273	Lower- and uppercase Greek characters with diacritics
11–13	275	Miscellaneous symbols
14	27	Compound jamo
15	0	Unassigned
16–36	1,930	Hangul syllables (last is 36-50)
37–54	1,675	Yesgeulja (last is 54-77)[a]
55–85	2,856	Hanja (last is 85-36)
86–94	0	Unassigned

a. Written 옛글자 (*yesgeulja*), meaning "old hangul."

It is interesting to note that the 2,856 hanja enumerated by this standard are handwritten in the official KS X 1002:2001 manual (as was the case in China's GB 7589-87 and GB

* Previously designated KS C 5657-1991

7590-87). I have encountered the following errors and inconsistencies during my perusal of the KS X 1002:2001 manual:

- Page 2 of the manual states that rows 1 through 7 enumerate 615 characters, but I counted only 613. Page 19 of the standard seems to include 2 duplicate characters at Row-Cell values 01-23 (<21 37>) and 01-90 (<21 7A>): "X" and "TM."
- Page 2 of the manual also states that rows 37 through 54 contain 1,677 hangul, but I counted only 1,675.

I have not heard of a revised version of KS X 1002:2001, so I assume that these errors and inconsistencies are still present in the standard, as well as its handwritten hanja. Although this character set standard is not encoded according to any legacy encoding, its hanja did serve as one of the many sources for Unicode's URO.

KPS 9566-97

North Korea (officially, Democratic People's Republic of Korea, or DPRK; 조선 민주 주의 인민 공화국/朝鮮民主主義人民共和國 *joseon minju juui inmin gonghwaguk*) developed their own character set standard in April of 1997 that enumerates hangul syllables and hanja, designated KPS 9566-97 and entitled *DPRK Standard Korean Graphic Character Set for Information Interchange*.* It is similar to South Korea's KS X 1001:2004 in many respects, but also different in several ways. This standard enumerates a total of 8,259 characters. Table 3-78 lists the characters that make up KPS 9566-97.

Table 3-78. The KPS 9566-97 character set

Row	Characters	Content
1	83	55 punctuation symbols plus 28 vertical variants
2	94	Miscellaneous symbols
3	62	Numerals 0–9, upper- and lowercase Latin alphabet
4	71	65 jamo (hangul elements), 6 hangul
5	66	Upper- and lowercase Cyrillic alphabet
6	68	48 upper- and lowercase Greek alphabet, upper- and lowercase Roman numerals 1–10
7	88	Encircled numerals 1–30, 28 encircled jamo and hangul, 10 superscripts 0–9, 10 subscripts 0–9, 10 fractions
8	94	Unit symbols and Latin ligatures
9	68	Full-width line-drawing elements
10	83	Hiragana
11	86	Katakana
12	64	Miscellaneous symbols

* KPS 9566-97 is also known as ISO-IR-202:1998.

Table 3-78. The KPS 9566-97 character set

Row	Characters	Content
13–15	0	Unassigned
16–44	2,679	Hangul (last is 44-47)
45–94	4,653	Hanja (last is 94-47)

The designers of KPS 9566-97 appear to have been inspired by KS X 1001:2004, but chose not to multiply encode hanja with multiple readings. They also decided to directly encode vertical variants in row 1, which has a total of 28 vertical-use characters.

Interestingly, row 4 includes six multiply encoded hangul (not jamo). 04-72 through 04-74 are the three hangul 김일성 (*gim il seong*), which represent the name of the former leader of DPRK, *Kim Il Sung*. 04-75 through 04-77 are the three hangul 김정일 (*gim cheong il*), which represent the name of the current leader of DPRK, *Kim Jong Il* (Kim Il Sung's son). This was clearly done as a tribute to the past and present leaders of DPRK.

The user-defined region of KPS 9566-97 includes a total of 188 code points and is comprised of all of row 15 (94 code points), the last half of row 44 (47 code points), and the last half of row 94 (47 code points).

There is one particular hangul in KPS 9566-97 that is not in KS X 1001:2004 and is used to render the name of a famous Korean literary work. The hangul 똠 (*ttom*), as used in the book entitled 똠방각하 (*ttom bang gak ha*), is encoded as KPS 9566-97 38-02, Johab <99 B1>, and Unicode U+B620. This represents a classic example that illustrates why KS X 1001:2004's 2,350 hangul syllables are not sufficient.

KPS 10721-2000—additional hanja

North Korea developed a second character set standard, designated KPS 10721-2000, that includes at least 19,469 hanja. I am unable to obtain additional details of this character set, and the only information that I have managed to unearth are the mappings to Unicode up through version 5.0. It is entitled *Code of the supplementary Korean Hanja Set for Information Interchange*.

KS X 1001:2004 versus KPS 9566-97

Although KS X 1001:2004 and KPS 9566-97 are developed in different Korean-speaking locales, it is useful to explicitly draw attention to their differences and similarities. In terms of similarities, both contain roughly the same number of hangul and hanja.

The differences between these standards are perhaps more interesting, and can be enumerated as follows:

- Whereas KS X 1001:2004 multiply-encodes hanja with multiple readings, KPS 9566-97 does not.
- KPS 9566-97 includes 28 vertical variants, but KS X 1001:2004 includes none.

- KPS 9566-97 multiply-encodes four hangul, to represent the names of North Korea's past and present leaders at separate and unique code points.

Although these two standards exhibit differences that seem to preclude interoperability, as long as they are expressed in Unicode, both standards can be completely expressed on today's OSes.

GB 12052-89

What is a GB standard doing in a section about Korean character sets? Consider this. There is a rather large Korean population in China (there have been border disputes with China for thousands of years; many Koreans escaped Japanese colonization, during the period 1910–1945, by moving to Manchuria, which is a northeastern part of China), and they need a character set standard for communicating with each other using hangul.

This character set standard, designated GB 12052-89 and entitled *Korean Character Coded Character Set for Information Interchange* (信息交换用朝鲜文字编码字符集 *xìnxī jiāohuàn yòng cháoxiān wénzì biānmǎ zìfújí*), is a Korean character set standard established by China on July 1, 1990, and enumerates a total of 5,979 characters. GB 12052-89 has no relationship nor compatibility with Korea's KS X 1001:2004. Table 3-79 lists the characters in GB 12052-89.

Table 3-79. The GB 12052-89 character set

Row	Characters	Content
1	94	Miscellaneous symbols
2	72	Numerals 1–20 with period, parenthesized numerals 1–20, encircled numerals 1–10, parenthesized hanzi numerals 1–20, uppercase Roman numerals 1–12
3	94	Full-width GB 1988-89 (GB-Roman; equivalent to ASCII)[a]
4	83	Hiragana
5	86	Katakana
6	48	Upper- and lowercase Greek alphabet
7	66	Upper- and lowercase Cyrillic alphabet
8	63	26 full-width Pinyin characters, 37 zhuyin (bopomofo) characters
9	76	Full-width line-drawing elements
10–15	0	Unassigned
16–37	2,068	Level 1 hangul, Part 1 (last is 37-94)
38–52	1,356	Level 1 hangul, Part 2 (last is 52-40)
53–72	1,873	Level 2 hangul (71-88 is unassigned; last is 72-88)[b]
73–94	0	Unassigned

a. The first 1,779 of these characters are hangul (53-01 through 71-87), and the remainder are 94 hanja (71-89 through 72-88).

Rows 1 through 9 look a lot like GB 2312-80, huh? Well, they're identical, except for 03-04, which is a "dollar" currency symbol ($) instead of GB 2312-80's "yuan" currency symbol (¥).

I have noted the following errors and inconsistencies during my ventures into the GB 12052-89 manual:

- Page 1 of the manual correctly states that a total of 5,979 characters are enumerated (682 symbols plus 5,297 hangul and hanja). However, page 3 of the manual states that rows 53 through 72 enumerate 1,876 characters, but I counted only 1,873—rows 53 through 71 enumerate 1,779 hangul, and rows 71 and 72 enumerate 94 hanja.

I have not heard of a revised version of GB 12052-89, so I can only assume that these errors and inconsistencies are still present in the standard.

Unicode compatibility with KS and KPS standards

The most interesting aspects of the KS and KPS standards, in terms of Unicode compatibility, is the extent to which their hanja map to its various CJK Unified Ideographs and CJK Compatibility Ideographs blocks.

The 4,888 hanja in KS X 1001:2004 map as follows: 4,620 map to the *Unified Repertoire and Ordering* (URO, which represent the initial block of CJK Unified Ideographs), and the remaining 268 map to the CJK Compatibility Ideographs block in the BMP. All 2,856 hanja of KS X 1002:2001 also map to the URO.

The 4,653 hanja in KPS 9566-97 map as follows: 4,652 map to the URO, and the single remaining hanja maps to CJK Unified Ideographs Extension A.

Although all of the hanja in KPS 10721-2000 do not yet map to Unicode, 10,358 map to the URO, 3,187 map to CJK Unified Ideographs Extension A, 5,767 map to CJK Unified Ideographs Extension B, 107 map to the CJK Compatibility Ideographs in the BMP, and 50 map to the CJK Compatibility Ideographs in Plane 2.

All of the modern hangul that are encoded in the KS and KPS standards are clearly in Unicode, as of version 2.0, given its complete coverage of 11,172 hangul syllables.

Vietnamese Character Set Standards

Although not widely known, Vietnam (Việt Nam) is one of the locales that has developed character set standards enumerating thousands of Chinese and Chinese-like characters. They have thus far established two character set standards that enumerate ideographs. All Vietnamese standard designations begin with TCVN, which stands for *Tiêu Chuẩn Việt Nam* (meaning "Vietnamese Standard").

Both standards covered in this section enumerate only Chinese and Chinese-like characters (chữ Hán and chữ Nôm) and are encoded in a single 94×94 matrix as two separate levels. Table 3-80 illustrates the allocation of characters by row.

Table 3-80. The TCVN 5773:1993 and TCVN 6056:1995 character sets

Row	Characters	Content
1–15	0	Unassigned; reserved for symbols
16–41	2,357	TCVN 5773:1993 (last is 41-07)
42–77	3,311	TCVN 6056:1995 (last is 77-21)
78–94	0	Unassigned

Note how both character sets are included in one superset—they are more or less treated as separate levels.

TCVN 5773:1993

Vietnam's very first character set that enumerates ideographs is designated TCVN 5773:1993, *Công Nghệ Thông Tin—Bộ Mã Chuẩn 16-Bit Chữ Nôm Dùng Trong Trao Đổi Thông Tin* (*Information Technology—Nom 16-Bit Standard Code Set for Information Interchange*), and was established on December 31, 1993.

TCVN 5773:1993 enumerates 2,357 ideographs, most of which are considered to be "Nom proper" (chữ Nôm) characters. That is, they are characters that appear to be of Chinese origin, but are in fact of Vietnamese origin. Think of them as Vietnamese-made ideographs.* The ordering of the characters is by radical, and then by total number of strokes. Approximately 600 are considered to be of genuine Chinese origin (chữ Hán).

TCVN 5773:1993 provides mappings to ISO 10646—equivalent for these purposes to Unicode—for every character, as follows:

- 587 characters are included in the standard set of 20,902 ideographs in Unicode
- 1,770 characters map to the PUA region of ISO 10646, beginning at U+A000 and ending at U+A6E9

TCVN 5773:1993 distinguishes these two types of mappings *implicitly* by the reference ISO 10646 code points. This standard also *explicitly* distinguishes these two types of mappings by using one of two encoding prefixes:

- U+ (for example, U+4EC9) refers to mappings into ISO 10646's standard set of 20,902 ideographs—the 587 characters just noted in the preceding list.
- V+ (for example, V+A000) refers to mappings into ISO 10646's PUA region—the 1,770 characters just noted in the preceding list.

* *Chữ Nôm* are different from Japanese *kokuji* in that they are never used in texts that are written using characters strictly of Chinese origin. See Chapter 2 for more details.

TCVN 6056:1995

A second Vietnamese character set standard, designated TCVN 6056:1995, *Công nghệ thông tin—Bộ Mã Chuẩn 16-Bit Chữ Nôm Dùng Trong Trao Đổi Thông Tin—Chữ Nôm Hán* (*Information Technology—Nom 16-Bit Standard Code for Information Interchange—Han Nom Character*), enumerates an additional 3,311 ideographs. The original TCVN 6056:1995 enumerated 3,349 characters, but it has since been revised. Thirty-eight duplicate characters—duplicates of characters in both TCVN 5773:1993 and TCVN 6056:1995 itself—have been removed from the character set.

Whereas TCVN 5773:1993 included both Chinese (chữ Hán) and "Nom proper" (chữ Nôm) characters, this standard includes only chữ Hán (characters of true Chinese origin). Appendix L provides a complete TCVN 6056:1995 code table.

As was the case with TCVN 5773:1993, these characters are ordered by radical, and then by total number of strokes, and references to ISO 10646 code points are provided for reference. All 3,311 of these characters map to ISO 10646 BMP code points. Remember that TCVN 5773:1993 mapped most of its characters into ISO 10646's PUA region.

Unicode compatibility with TCVN standards

While all 3,311 ideographs in TCVN 6056:1995 are included in Unicode, all of which are in the URO, only 2,246 of the 2,357 ideographs in TCVN 5773:1993 are in Unicode, up through version 5.1.

International Character Set Standards

Many organizations, corporations, and researchers have been actively involved in the development of international character set standards in an attempt to encode most of the world's written languages in a single repertoire of characters. One early effort was Xerox's XCCS (*Xerox Character Code Standard*). CCCII can also be considered such an attempt. In any case, these character sets should be of interest to you because they include tens of thousands of ideographs and thousands of hangul syllables. The only standards that will be covered in this section are ISO 10646 and Unicode, and any national standard that is based on them.

Table 3-81 lists the international character set standards covered in this section, all of which are based on different versions of Unicode.

Table 3-81. International character set standards

Standard	Ideographs	Hangul	Other characters[a]	PUA	Total
Unicode version 1.0—1991	0	2,350	4,746	5,632	7,096
Unicode version 1.0.1—1992	20,902	2,350	5,042	6,144	28,294
Unicode version 1.1—1993	20,902	6,656[b]	6,610	6,400	34,168
Unicode version 2.0—1996	20,902	11,172	6,811	6,400	38,885

Table 3-81. International character set standards

Standard	Ideographs	Hangul	Other characters[a]	PUA	Total
Unicode version 2.1—1998	20,902	11,172	6,813	6,400	38,887
Unicode version 3.0—1999	27,484	11,172	10,538	137,468	49,194
Unicode version 3.1—2001	70,195	11,172	12,773	137,468	94,140
Unicode version 3.2—2002	70,195	11,172	13,789	137,468	95,156
Unicode version 4.0—2003	70,195	11,172	15,015	137,468	96,382
Unicode version 4.1—2005	70,217	11,172	16,266	137,468	97,655
Unicode version 5.0—2006	70,217	11,172	17,635	137,468	99,024
Unicode version 5.1—2008	70,225	11,172	19,251	137,468	100,648
ISO 10646-1:1993[c]	Equivalent to Unicode version 1.1				
ISO 10646-1:2000	Equivalent to Unicode version 3.0[d]				
ISO 10646-2:2001	Part of Unicode version 3.1 (Supplementary Planes)[d]				
ISO 10646:2003[e]	Equivalent to Unicode version 4.0				
GB 13000.1-93	Equivalent to ISO 10646-1:1993				
CNS 14649-1:2002	Equivalent to ISO 10646-1:2000				
CNS 14649-2:2003	Equivalent to ISO 10646-2:2001				
JIS X 0221-1995	Equivalent to ISO 10646-1:1993				
JIS X 0221-1:2001	Equivalent to ISO 10646-1:2000				
JIS X 0221:2007	Equivalent to ISO 10646:2003 with Amendments 1 and 2 (Unicode version 5.0)				
KS X 1005-1:1995	Equivalent to ISO 10646-1:1993 with Amendments 1 through 7 (Unicode version 2.0)				

a. These figures include CJK Compatibility Ideographs.

b. This figure is composed of 2,350 basic hangul (from KS X 1001:2004), 1,930 Supplemental Hangul A (from KS X 1002:2001), and 2,376 Supplemental Hangul B (source unknown to this author).

c. ISO 10646-1:1993 with Amendments 1 through 7 is equivalent to Unicode version 2.0. Amendment 8 makes it equivalent to Unicode version 2.1.

d. ISO 10646-1:2000 with two characters from Amendment 1 and ISO 10646-2:2001 are equivalent to Unicode version 3.1. ISO 10646-1:2000 with Amendment 1 and ISO 10646-2:2001 are equivalent to Unicode version 3.2.

e. ISO 10646:2003 with Amendment 1 is equivalent to Unicode version 4.1. ISO 10646:2003 with Amendments 1 and 2, along with four Singhi characters from Amendment 3, is equivalent to Unicode version 5.0.

It is useful to take note that Unicode version 1.0 never became a national or international standard—it is obsolete.

Unicode and ISO 10646

The International Organization for Standardization (ISO) and The Unicode Consortium have jointly developed—and continue to jointly develop—a multilingual character set designed to combine the majority of the world's writing systems and character set standards into a significantly larger repertoire of characters.

ISO designated their standard *ISO 10646*, and The Unicode Consortium named their standard *Unicode*. These standards started out separately, were wildly different, and wisely merged their work into what can be treated as a single standard. Both of these standards can be represented by the same encoding forms, specifically UTF-8, UTF-16, and UTF-32. All three will be covered in detail in Chapter 4, along with variants thereof. Both standards began with characters encoded only in the area that is called the *Basic Multilingual Plane* (BMP). It is called the BMP because ISO 10646-1:1993 was composed of groups of planes. The first characters to be encoded outside the BMP were in Unicode version 3.1, which is equivalent to ISO 10646-1:2000 and ISO 10646-2:2001.

Unicode approached the CJKV problem by attempting to unify all ideographs from the many CJKV national character set standards into a single set of ideographs. This effort became incorrectly known as *Han Unification*. "Han" comes from the Chinese reading of the ideograph 漢—in Chinese, Korean, and Vietnamese it is read *han*, and in Japanese it is read *kan*. Note that this effort does not represent *true* or *genuine* unification of ideographs due to the *Source Separation Rule* (to be explained shortly) and other factors. So, why is Han Unification not an appropriate way to describe the process that took place to create this character set?

If we consider the evolutionary processes that have affected ideographs over the course of history, we see that they have diversified into a number of locale-specific variations. That is, they were adapted by several non-Chinese cultures, such as Japan, Korea, and Vietnam. With any borrowing, whether it is from the lexicon (that is, words) or the orthography (that is, the writing system), there is a certain amount of change that is almost always guaranteed to take place over time. As Joe Becker so succinctly stated, diversification and variation are the real historical processes that actually took place, but the so-called Han Unification is not a real process. Rather, it has been called diversification and variation as seen through a mirror.

The real goal of those who compiled the ideographs in Unicode was to simply provide coverage for the major CJKV character set standards existing at the time, so that *every* ideograph in these standards would have an equivalent code point in Unicode. This goal effectively provides users and developers with two very important and real benefits:

- A much larger repertoire of characters than found in other CJKV character set standards
- Compatibility with the characters in existing CJKV character set standards—this is perhaps more important than you think

Unicode, when encoded according to the original UCS-2 encoding, provides only 65,536 16-bit code points. When the first edition of this book was written, at the end of 1998, 38,887 of these code points had been assigned characters for Unicode version 2.1, 38,885 for version 2.0, 34,168 for version 1.1, 28,294 for version 1.0.1, and a mere 7,096 for version 1.0. The character-encoding space for Unicode is now set in 17 planes, each of which is a 256×256 matrix.

For a historical perspective, Table 3-82 details, by version, how many ideographs, radicals, and strokes are included in Unicode. Those that are lightly shaded have been accepted as part of ISO 10646:2003 Amendment 5, but as of this writing are not yet included in Unicode. Similarly, those that are shaded darker have also been accepted, but as part of Amendment 6, and also are not yet included in Unicode.

Table 3-82. Unicode ideographs, radicals, and strokes

Version	Number of characters added	Code point ranges	Total number of characters
CJK Unified Ideographs			
1.1	20,902	U+4E00–U+9FA5 (URO)	20,902
3.0	6,582	U+3400–U+4DB5 (Extension A)	27,484
3.1	42,711	U+20000–U+2A6D6 (Extension B)	70,195
4.1	22	U+9FA6–U+9FBB	70,217
5.1	8	U+9FBC–U+9FC3	70,225
	3	U+9FC4–U+9FC6	70,228
	4,149	U+2A700–U+2B734 (Extension C)	74,377
	5	U+9FC7–U+9FCB	74,382
CJK Compatibility Ideographs[a]			
1.1	302	U+F900–U+FA2D	302
3.1	542	U+2F800–U+2FA1D	844
3.2	59	U+FA30–U+FA6A	903
4.1	106	U+FA70–U+FAD9	1,009
	3	U+FA6B–U+FA6D	1,012
CJK Radicals Supplement			
3.0	115	U+2E80–U+2E99, U+2E9B–U+2EF3	115
Kangxi Radicals			
3.0	214	U+2F00–U+2FD5	214
CJK Strokes			
4.1	16	U+31C0–U+31CF	16
5.1	20	U+31D0–U+31E3	36

a. Note that the following 12 code points within the CJK Compatibility Ideographs block are actually CJK Unified Ideographs: U+FA0E, U+FA0F, U+FA11, U+FA13, U+FA14, U+FA1F, U+FA21, U+FA23, U+FA24, and U+FA27 through U+FA29.

Unicode version 5.1, issued a decade after version 2.1, pushes the total number of assigned characters over the 100,000 mark for the first time. To see what code points have been newly assigned for each version of Unicode, the *DerivedAge.txt* file that is provided on the Unicode website is very useful.*

As mentioned earlier, the original block of CJK Unified Ideographs, consisting of 20,902 ideographs, are the result of merging many character set standards into one larger repertoire through the process of Han Unification. This block of CJK Unified Ideographs is referred as the URO (*Unified Repertoire and Ordering*). According to The Unicode Standard, version 2.0, most of the ideographs contained in the character set standards listed in Table 3-83 have been included.

Table 3-83. Source character sets of the URO

Character set standard	Country	Ideographs
ANSI Z39.64-1989 (EACC)	USA	13,481
Xerox Chinese	USA	9,776
GB 2312-80	China	6,763
GB/T 12345-90	China	2,180
GB 7589-87	China	4,835
GB 7590-87	China	2,842
General Use Characters for Modern Chinese[a]	China	41
GB 8565.2-88	China	290
GB 12052-89	China	94
PRC Telegraph Code	China	≈8,000
Big Five[b]	Taiwan	13,053
CCCII, Level 1	Taiwan	4,808
CNS 11643-1986 Plane 1	Taiwan	5,401
CNS 11643-1986 Plane 2	Taiwan	7,650
CNS 11643-1986 Plane 14[c]	Taiwan	4,198
Taiwan Telegraph Code	Taiwan	9,040
JIS X 0208-1990 (equivalent to JIS X 0208:1997)[d]	Japan	6,356
JIS X 0212-1990	Japan	5,801
KS X 1001:1992 (equivalent to KS X 1001:2004)	Korea	4,620

* *http://www.unicode.org/Public/UNIDATA/DerivedAge.txt*

Table 3-83. Source character sets of the URO

Character set standard	Country	Ideographs
KS X 1002:1991 (equivalent to KS X 1002:2001)	Korea	2,856
TCVN 6056:1995	Vietnam	3,311

a. Better known as 现代汉语通用字表 (*xiàndài hànyǔ tōngyòngzì biǎo*).

b. Two of these hanzi are duplicately encoded, and are mapped into Unicode's CJK Compatibility Ideographs block.

c. CNS 11643-1986 Plane 14 contains 6,319 hanzi.

d. There are normally considered to be 6,355 kanji in JIS X 0208:1997. The extra character, 仝 (01-24), is in its non-kanji region.

Table 3-83 lists approximately 121,000 ideographs in total, but when they were merged according to the rules and principles of Han Unification, they became 20,902 unique characters. These ideographs are subsequently arranged by radical, followed by the number of additional strokes (this is similar to the ordering of GB 2312-80 Level 2 and JIS X 0208:1997 Level 2). Redundant characters—considered to be duplicate characters across the source character sets—among the approximately 121,000 ideographs that are represented in Table 3-83 were effectively removed to form the final set of 20,902.

The two ideographs in Table 3-84 have similar structure—similar enough, in fact, to justify unifying them. However, they have completely unrelated etymologies and meanings, so merging did not take place. In other words, they are *noncognate*.

Table 3-84. Nonunified ideographs—example

Ideograph	Meaning	Unicode
土	earth	U+571F
士	scholar, knight	U+58EB

As illustrated in Table 3-84, the relative lengths of strokes in ideographs can change their meaning. However, ideographs such as those in Table 3-85 have been unified because their difference lies in their glyphs, which can be thought of as a simple typeface style issue. In other words, they are *cognate* and share the same abstract shape.

Table 3-85. Unified ideographs—example

Ideograph	Meaning	Source character set	Row-Cell	Unicode
父	father	JIS X 0208-1983	41-67	U+7236
父	father	JIS X 0208-1990	41-67	U+7236

Note how these are microscopic variations of the same structure, and that they have the same meaning. They also share the same encoding, but come from different versions or vintages of the JIS X 0208 character set standard. The earlier versions of JIS X 0208-1990 are not considered part of the Japanese sources for Unicode—only JIS X 0208-1990 (now designated JIS X 0208:1997) and JIS X 0212-1990 were used.

The glyphs for ideographs can be compared using a three-dimensional model. The X-axis (semantic) separates characters by their meaning. The Y-axis (abstract shape) separates a character on the X-axis into its abstract shapes. Traditional and simplified forms of a particular ideograph fall into the same X-axis position, but have different positions along the Y-axis. The Z-axis (typeface) separates a character into glyph differences. Only Z-axis differences were merged or unified in Unicode. Table 3-85 provided an example of a Z-axis difference. Glyph differences are usually found when using different typefaces to produce the same character. The same character in different languages may also appear differently because of locale differences that have resulted from diversification. Figure 3-1 illustrates the three-axis model used for comparing the glyphs for ideographs.

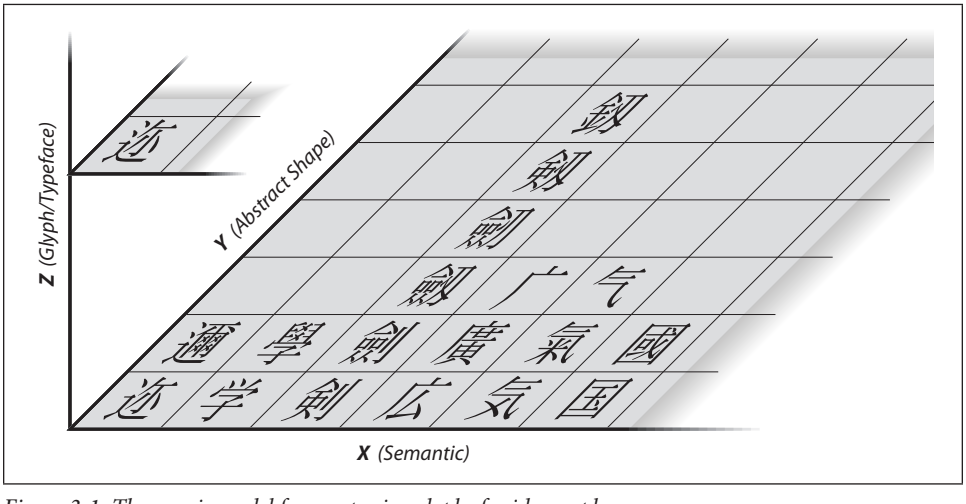


Figure 3-1. Three-axis model for comparing glyphs for ideographs

Unfortunately, early standards were inconsistent in their encoding models for ideographs, and some still are, resulting in separately encoded Z-axis variants.

There were four sets of national standards from which Unicode's first CJK Unified Ideographs block, the URO, was derived, specifically character sets from China, Taiwan, Japan, and Korea. For example, there were two Japanese character sets in the Japanese source set: JIS X 0208-1990 (now JIS X 0208:1997) and JIS X 0212-1990. Unification of two characters cannot take place if they have unique encoded positions within a single source set. Table 3-86 lists the kanji 剣 (meaning "sword," and read *ken* or *tsurugi*) and its five variants—all of these characters have unique code points in JIS X 0208:1997 and are thus not unified.* This effectively guarantees that round-trip conversion is possible. This is also why the two duplicately encoded hanzi of Big Five and the 268 multiply encoded hanja of KS X 1001:2004 are in Unicode's first CJK Compatibility Ideographs block

* Another variant of 剣 is 劍 (*jiàn*), but that is outside the context of this Japanese-specific discussion.

(version 1.1 in Table 3-82). JIS X 0208:1997 and JIS X 0212-1990 contain many kanji that could potentially be unified, and the kanji in Table 3-86 represent but a single collective example.

Table 3-86. Six kanji from JIS X 0208:1997 that are not unified

Kanji	JIS X 0208:1997	Unicode
剣	23-85	U+5263
劍	49-88	U+528D
劒	49-89	U+5294
劔	49-90	U+5292
劒	49-91	U+5271
釼	78-63	U+91FC

Each of these six ideographs has a unique code point, both in the source character set, specifically JIS X 0208:1997, and in Unicode. This is by design.

The result of all this effort was a collection of ideographs whose ordering can be considered culturally neutral. How ideographs are ordered by radical is often locale-specific, and devising an ordering that would please all locales that use ideographs was a very difficult task. We could call this *pancultural*. The CJK Joint Research Group (CJK-JRG), an *ad-hoc* committee of ISO/IEC JTC1/SC2/WG2 (*Joint Technical Committee 1, Sub Committee 2, Working Group 2*), selected four ideograph dictionaries, reflecting ideograph usage in CJKV locales:

- Common traditional—康熙字典 (*kāngxī zìdiǎn*)
- Japan—大漢和辭典 (*dai kanwa jiten*)
- China—汉语大字典 (*hànyǔ dà zìdiǎn*)
- Korea—大字源 (*daejawon*)

These four dictionaries were subsequently checked, in the order just given, for each ideograph. If the first dictionary checked did not include the ideograph, the next dictionary was checked, and so on until each of the 20,902 ideographs was found. Not a simple task. Truthfully, it's quite daunting.

The CJK-JRG is now known as the *Ideographic Rapporteur Group* (IRG), and continues to operate under the auspices of WG2.* The IRG meets twice a year, for a full week of sometimes heated discussions. Much of what the IRG does takes place between meetings, and the meetings are used to make important decisions or to otherwise perform work that requires the presence of its national body members.

* *http://www.cs.cuhk.edu.hk/~irg/*

For additional information regarding the Unicode character set or Han Unification, please refer to the book entitled *The Unicode Standard, Version 5.0* or to The Unicode Consortium's website.* The ISO 10646:2003, GB 13000.1-93, JIS X 0221:2007, and KS X 1005-1:1995 manuals are also available and contain similar information. The most inexpensive of these four standards is GB 13000.1-93, followed by KS X 1005-1:1995, but they are also the most difficult to obtain, at least where I live. Among them, the Unicode book is the easiest to obtain because the latest version is made available in PDF on The Unicode Consortium's website.†

Annex S—unification rules and principles

Annex S, introduced in ISO 10646-1:2000, details and describes the unification rules and principles that are used to compile the CJK Unified Ideograph and CJK Compatibility Ideographs blocks. Annex S is informative and entitled "Procedure for the unification and arrangement of CJK Ideographs." It serves as the guide for determining whether a new character is to be unified with an existing character or treated as a separate character. Table 3-87 details one way to think about the basic principles used by Annex S for ideograph pairs or triplets that are considered cognate.

Table 3-87. To unify or not to unify…

	Actual shape	
	Exact match	**Different**
Same abstract shape	Unify	Unify
Different abstract shape		Do Not Unify

Of course, what is considered abstract shape, or whether the character pairs (or triplets, in some cases) being considered are cognate, are sometimes not easily understood and require further research. Still, the Annex S principles hold true and are genuinely useful.

Extending Unicode beyond the BMP—UTF-16 encoding

The well-established Surrogates Area, U+D800 through U+DFFF, provides a method for extending Unicode to accommodate 1,048,576 additional code points. This is better known as UTF-16 encoding, which is described in Chapter 4. The following is a brief description of how the Surrogates Area functions:

- The High Surrogates, U+D800 through U+DBFF, represent the first element in a surrogate pair
- The Low Surrogates, U+DC00 through U+DFFF, represent the second element in a surrogate pair

* *http://www.unicode.org/*

† *http://www.unicode.org/standard/standard.html*

For example, U+D800 plus U+DC00 and U+DBFF plus U+DFFF represent the first and last characters in the Surrogates area, respectively.

In essence, characters encoded in the Surrogates Area are represented by four bytes (or two 16-bit values). 2,048 code points within the BMP are effectively sacrificed to create an additional 1,048,576 code points. That's not a bad trade-off!

One benefit of this technique may not be immediately obvious: one always knows whether a particular 16-bit unit represents the first or second element of a surrogate pair, by virtue of each element using a different encoding range. This scheme allows software processing to be simpler than if the two ranges overlap.

131,068 of the code points in the Surrogates Area are reserved as PUA code points. This effectively means that Unicode now provides 137,468 PUA code points (6,400 plus 131,068). More information about PUA code points is presented in the following section.

Some people mistakenly use the word *Surrogates* to refer to any character or code point outside of the BMP. I need to clearly point out that the direct UTF-8 and UTF-32 representations of the High and Low Surrogates are illegal and invalid in those encoding forms, and that in the UTF-8 and UTF-32 encoding forms, non-BMP code points are not given any special treatment and are directly encoded in those encoding forms. In other words, Surrogates are specific to UTF-16 encoding. If one wants to refer to non-BMP code points in an encoding-independent fashion, using the word *Surrogates* is not the right way and is misleading. Simply using *non-BMP* or referring to specific planes beyond the BMP, such as Plane 2, is the preferred and correct way in which to refer to characters or code points that are encoded beyond the BMP.

Private Use Area

As mentioned in the previous section, Unicode includes 137,468 Private Use Area (PUA) code points. These are encoded in three separate ranges, as specified in Table 3-88.

Table 3-88. Unicode Private Use Area ranges

Code point range	Number of characters	Plane
U+E000 through U+F8FF	6,400	0 —BMP
U+F0000 through U+FFFFD	65,534	15
U+100000 through U+10FFFD	65,534	16

One would normally think that Planes 15 and 16 would provide 65,536 PUA code points each, but the last two code points in every plane have *noncharacter* status, and thus cannot be used for any purpose whatsoever. This means that U+FFFFE, U+FFFFF, U+10FFFE, and U+10FFFF are explicitly excluded from the PUA.

The use of PUA code points should be avoided at all costs, because their interpretation, in terms of character properties, and their interaction with legacy character set standards (in other words, interoperability) cannot be guaranteed.

Planes and encoding forms

Although Chapter 4 will cover encodings in detail, I will take the opportunity to present here the structure of Unicode, along with a very brief introduction to its encoding forms, because drawing a comparison between its structure, from a character set point of view, and its encoding forms can be intriguing and enlightening.

Unicode is composed of 17 planes. The *Basic Multilingual Plane* (BMP) represents the first plane, and because everything in the computer world begins at zero, it is also known as Plane 0. The remaining 16 planes are referred to as the *Supplementary Planes*, and are referred to by number, specifically Planes 1 through 16.

Unicode is encoded according to three basic encoding forms, called UTF-8, UTF-16, and UTF-32.

Table 3-89 lists each of the 17 planes of Unicode, along with their encoding-independent ranges indicated through the use of Unicode scalar values.

Table 3-89. Unicode's 17 planes and their ranges

Plane	Name	Scalar value range
0	BMP (Basic Multilingual Plane)	U+0000–U+FFFF
1	SMP (Supplementary Multilingual Plane)	U+10000–U+1FFFF
2	SIP (Supplementary Ideographic Plane)	U+20000–U+2FFFF
3		U+30000–U+3FFFF
4		U+40000–U+4FFFF
5		U+50000–U+5FFFF
6		U+60000–U+6FFFF
7		U+70000–U+7FFFF
8		U+80000–U+8FFFF
9		U+90000–U+9FFFF
10		U+A0000–U+AFFFF
11		U+B0000–U+BFFFF
12		U+C0000–U+CFFFF
13		U+D0000–U+DFFFF
14	SSP (Supplementary Special-purpose Plane)	U+E0000–U+EFFFF
15	SPUA-A or PUP (Supplementary Private Use Area-A or Private Use Plane)	U+F0000–U+FFFFF
16	SPUA-B or PUP (Supplementary Private Use Area-B or Private Use Plane)	U+100000–U+10FFFF

Next, Table 3-90 presents the same 17 planes of Unicode with the Unicode scalar value ranges, along with the encoding ranges as specified by more specific UTF-8, UTF-16, and UTF-32 encoding forms.*

Table 3-90. Unicode's 17 planes and their encoding ranges

Plane	Scalar value range	UTF-8	UTF-16	UTF-32
0	U+0000–U+FFFF	00–EF BF BF[a]	0000–FFFF[b]	00000000–0000FFFF
1	U+10000–U+1FFFF	F0 90 80 80–F0 9F BF BF	D800 DC00–D83F DFFF	00010000–0001FFFF
2	U+20000–U+2FFFF	F0 A0 80 80–F0 AF BF BF	D840 DC00–D87F DFFF	00020000–0002FFFF
3	U+30000–U+3FFFF	F0 B0 80 80–F0 BF BF BF	D880 DC00–D8BF DFFF	00030000–0003FFFF
4	U+40000–U+4FFFF	F1 80 80 80–F1 8F BF BF	D8C0 DC00–D8FF DFFF	00040000–0004FFFF
5	U+50000–U+5FFFF	F1 90 80 80–F1 9F BF BF	D900 DC00–D93F DFFF	00050000–0005FFFF
6	U+60000–U+6FFFF	F1 A0 80 80–F1 AF BF BF	D940 DC00–D97F DFFF	00060000–0006FFFF
7	U+70000–U+7FFFF	F1 B0 80 80–F1 BF BF BF	D980 DC00–D9BF DFFF	00070000–0007FFFF
8	U+80000–U+8FFFF	F2 80 80 80–F2 8F BF BF	D9C0 DC00–D9FF DFFF	00080000–0008FFFF
9	U+90000–U+9FFFF	F2 90 80 80–F2 9F BF BF	DA00 DC00–DA3F DFFF	00090000–0009FFFF
10	U+A0000–U+AFFFF	F2 A0 80 80–F2 AF BF BF	DA40 DC00–DA7F DFFF	000A0000–000AFFFF
11	U+B0000–U+BFFFF	F2 B0 80 80–F2 BF BF BF	DA80 DC00–DABF DFFF	000B0000–000BFFFF
12	U+C0000–U+CFFFF	F3 80 80 80–F3 8F BF BF	DAC0 DC00–DAFF DFFF	000C0000–000CFFFF
13	U+D0000–U+DFFFF	F3 90 80 80–F3 9F BF BF	DB00 DC00–DB3F DFFF	000D0000–000DFFFF
14	U+E0000–U+EFFFF	F3 A0 80 80–F3 AF BF BF	DB40 DC00–DB7F DFFF	000E0000–000EFFFF
15	U+F0000–U+FFFFF	F3 B0 80 80–F3 BF BF BF	DB80 DC00–DBBF DFFF	000F0000–000FFFFF
16	U+100000–U+10FFFF	F4 80 80 80–F4 8F BF BF	DBC0 DC00–DBFF DFFF	00100000–0010FFFF

a. The complete one-, two-, and three-byte encoding ranges are <00>–<7F>, <C2 80>–<DF BF>, and <E0 A0 80>–<EF BF BF>.

b. U+D800 through U+DFFF are excluded from this range because they represent the High and Low Surrogates, which are used to encode the 16 Supplementary Planes.

The big take-away from Table 3-90 is that as soon as one goes beyond the BMP, every character must be represented by four effective bytes, regardless of the encoding form, whether it is four bytes (UTF-8), two 16-bit units (UTF-16), or a single 32-bit unit (UTF-32). In addition, the most human-readable Unicode encoding form is clearly UTF-32, which simply looks like a zero-padded version of the Unicode scalar values that use the U+XXXX notation.

Of course, there is much more to the Unicode encoding forms than is captured in this table, or even in this chapter, and Chapter 4 will provide the details that you need.

* Note that this table does not address the important issue of byte order, which affects the UTF-16 and UTF-32 encoding forms. The values are in big-endian byte order. Technically, the UTF-16 column specifies the UTF-16BE (*UTF-16 Big-Endian*) encoding form, and likewise, the UTF-32 column specifies the UTF-32BE (*UTF-32 Big-Endian*) encoding form.

Kangxi Radicals, CJK Radicals Supplement, and CJK Strokes

Characters that represent radicals and strokes, the building blocks of ideographs, are included in Unicode. The Kangxi Radicals block, U+2F00 through U+2FD5, includes characters that represent the complete set of 214 classical radicals as used by the vast majority of ideograph dictionaries. All of these characters have corresponding CJK Unified Ideograph code points. For example, it is obvious that the Kangxi Radical U+2F00 (⼀) should be visually identical to CJK Unified Ideograph U+4E00 (一), and under almost all conditions should be rendered the same. In fact, for what I consider to be well-designed fonts, both code points are rendered using the same underlying glyph, meaning that both code points map to the same glyph.

115 radical variants, consisting primarily of those for Simplified Chinese, are in the CJK Radicals Supplement block, encoded from U+2E80 through U+2EF3 (note that U+2E9A is unassigned). This collection of radical variants appears to be somewhat *ad-hoc*, simply because it is.

Sixteen strokes are currently encoded in the CJK Strokes region, encoded from U+31C0 through U+31CF. These were added in Unicode version 4.1. Twenty additional strokes have been recently approved and added to Unicode, encoded from U+31D0 through U+31E3 and included in version 5.1. That means that 36 primitive strokes are now available in Unicode.

CJK Unified Ideographs URO—Unified Repertoire and Ordering

Unicode's original block of 20,902 ideographs is referred to as the URO, which stands for *Unified Repertoire and Ordering*. Its range is U+4E00 through U+9FA5. Note that additional CJK Unified Ideographs have been appended to this block, and as of version 5.0, U+9FA6 through U+9FBB (22 code points) have been assigned. These 22 CJK Unified Ideographs were introduced in version 4.1 for the purpose of more completely supporting GB 18030-2000 and Hong Kong SCS-2004. An additional 16 ideographs were more recently appended to this block, encoded from U+9FBC through U+9FCB. Until U+9FFF is assigned, we can expect additional characters to be added to this seemingly small region, as long as the number of characters to be added is relatively small.

CJK Unified Ideographs Extension A

The IRG has compiled an additional set of ideographs, 6,582 total, that became part of Unicode as of version 3.0. This second block of ideographs is called *CJK Unified Ideographs Extension A*.* These additional 6,582 characters are encoded in the region left vacant from the 6,656 hangul that were in Unicode version 1.1, specifically U+3400 through U+4DB5.

Some preliminary mapping data was made available for this set of 6,582 characters, and Table 3-91 lists CJKV character set standards, along with the number of mappings. (The

* The original proposal included 6,585 ideographs, but three were found to be duplicates of characters among the CJK Compatibility Ideographs and were rightfully excised.

number of mappings are according to material dated from February 25, 1997, when there were 6,584 characters in this set, so consider the numbers to be approximate figures.)

Table 3-91. Mappings for Unicode's CJK Unified Ideographs Extension A

Character set standard	Number of mappings
GB/T 13131-2XXX	2,391
GB/T 13132-2XXX	1,226
General Use Characters for Modern Chinese[a]	120
Singapore characters	226
CNS 11643-1992 Plane 3	2,178
CNS 11643-1992 Plane 4	2,912
CNS 11643-1992 Plane 5	392
CNS 11643-1992 Plane 6	194
CNS 11643-1992 Plane 7	133
CNS 11643-1986 Plane 15	71
Unified Japanese IT Vendors Contemporary Ideographs, 1993	660
PKS C 5700-2 1994[b]	1,834
TCVN 5773:1993	128

a. Better known as 现代汉语通用字表 (*xiàndài hànyǔ tōngyòngzì biǎo*).

b. This could be a draft of Part 2 of KS X 1005-1:1995.

This set of 6,582 ideographs represents the last large repertoire of ideographs to be added to Unicode's BMP. Any further large repertoires of ideographs must be assigned to code points outside of the BMP. As the next section describes, Unicode version 3.1 introduced *CJK Unified Ideographs Extension B*, which is encoded outside the BMP, specifically in Plane 2.

CJK Unified Ideographs Extension B

Unicode version 3.1 introduced *CJK Unified Ideographs Extension B*, which contains 42,711 characters. Due to the staggering number of characters, they were necessarily placed outside the BMP. They are in Plane 2. The first character is encoded at U+20000, and the block continues to U+2A6D6. Also, 542 additional CJK Compatibility Ideographs were added and are also in Plane 2, encoded from U+2F800 through U+2FA1D.

Who makes use of Extension B characters? Several character sets map into Extension B, and in order for an implementation to be compliant, without resorting to PUA code points, it must be supported, but not necessarily in its entirety. Table 3-92 lists common locale-specific character sets, along with the number of Extension B mappings that they use.

Table 3-92. Locale-specific character sets that map to Extension B

Character set	Number of Extension B mappings
GB 18030-2005	6 or 42,711[a]
CNS 11643-1992	30,713
Hong Kong SCS-2008	1,712
JIS X 0213:2004	303
KPS 10721-2000	5,767
TCVN 5773:1993	1,515

a. The number of mappings depends on whether the glyphs printed in the 2000 or 2005 manual are used as the basis. There are a mere six for the former and a staggering 42,711 for the latter.

CJK Unified Ideographs Extensions C and D

The IRG has been working on further blocks of CJK Unified Ideographs. *CJK Unified Ideographs Extension C* was declared final in April of 2008, and its characters are now encoded in Plane 2, immediately following Extension B. Its final form includes 4,149 characters, encoded from U+2A700 through U+2B734.

CJK Unified Ideographs Extension D is still in process. And, at some point, *CJK Unified Ideographs Extension E* will become a reality. These and future extensions come about as the result of IRG efforts. National bodies submit new ideographs, which are prioritized and categorized. Those that are determined to be new CJK Unified Ideographs eventually make their way into an extension. Those that are unified with an existing CJK Unified Ideograph are thus rejected.

IICore

Ideographic International Core (IICore), defined by the IRG in 2005, is a subset of the CJK Unified Ideographs in Unicode that have been deemed to be frequently used across locales, and are useful for implementations that require a minimum number of characters. IICore includes 9,810 CJK Ideographs, and are identified as a separate field in the Unihan Database. These ideographs are spread throughout the BMP and Plane 2. Forty-two of them are in Extension A, 62 are in Extension B, and the remaining 9,706 are in the URO.

CJK Compatibility Ideographs

There are currently two blocks of CJK Compatibility Ideographs in Unicode, one of which is in the BMP, and the other is outside the BMP, specifically in Plane 2. Twelve of the characters in the BMP's CJK Compatibility Ideographs block are considered to be CJK Unified Ideographs. They are as follows: U+FA0E, U+FA0F, U+FA11, U+FA13, U+FA14, U+FA1F, U+FA21, U+FA23, U+FA24, and U+FA27 through U+FA29. Some CJK Compatibility Ideographs are genuine duplicate characters according to their source character sets, such as those from Big Five and KS X 1001:2004.

To what extent are CJK Compatibility Ideographs treated differently than CJK Unified Ideographs? Most importantly, *Normalization* can be applied to CJK Compatibility Ideographs. Normalization is a well-established process whereby the representation of characters is made uniform. For example, accented characters can be represented by a single code point or by multiple code points, such as its Base Character followed by one or more separate characters that represent accents or other adornments. Furthermore, in the case of multiple accents, the ordering of the accents is made uniform through the application of Normalization.

There are different levels or types of Normalization. In the context of CJK Compatibility Ideographs, Normalization simply means that a CJK Compatibility Ideograph may be converted to its *Canonical Equivalent*, which is the corresponding CJK Unified Ideograph, and it is done at the sole discretion of the application.

If the distinction of a CJK Compatibility Ideograph is important for your needs, I strongly advise a representation that preserves the distinction. Today's software interoperates, and because one cannot control the extent to which software applies Normalization, the probability of Normalization being applied is very high and, more significantly, may not be under your direct control. In other words, any attempt to prevent Normalization from taking place is futile. I claim that Normalization can be prevented only in completely closed environments. But then again, such closed environments are not very interesting. Software interaction is what makes the current generation of applications useful, and this claim is likely to become stronger as the years pass.

I have observed that among the hundreds of CJK Compatibility Ideographs in Unicode, at least through version 5.0, that none of them map to any character set standard developed in China. In other words, of the seven primary sources or locales—China, Taiwan, Hong Kong, Japan, South Korea, North Korea, and Vietnam—all but China have mappings in the CJK Compatibility Ideograph blocks. What does this mean? Nothing. It is merely an observation of the data and nothing more.

Normalization and Canonical Equivalents

As stated in the previous section, the process called Normalization can be applied to CJK Compatibility Ideographs, and the result is that they are converted into their Canonical Equivalents. Normalization effectively flattens differences when it is possible to represent the same *character* in Unicode in multiple ways. This obviously applies to accented characters. Table 3-93 provides some examples of CJK Compatibility Ideographs and their Canonical Equivalents.

Table 3-93. CJK Compatibility Ideographs and their Canonical Equivalents—examples

CJK Compatibility Ideograph	Canonical Equivalent	Locale	Comments
U+F907 龜	U+9F9C 龜	Korea	Identical
U+F907 龜	U+9F9C 龜	Hong Kong	Different in Hong Kong SCS-2008

Table 3-93. CJK Compatibility Ideographs and their Canonical Equivalents—examples

CJK Compatibility Ideograph	Canonical Equivalent	Locale	Comments
U+F908 龜	U+9F9C 龜	Korea	Identical
U+F914 樂	U+6A02 樂	Korea	Identical
U+F95C 樂	U+6A02 樂	Korea	Identical
U+F9BF 樂	U+6A02 樂	Korea	Identical
U+F9D0 類	U+985E 類	Korea	Identical
U+F9D0 類	U+985E 類	Japan	Different in JIS X 0213:2004
U+FA0C 兀	U+5140 兀	Taiwan	Identical
U+FA0D 殼	U+55C0 殼	Taiwan	Identical
U+FA47 漢	U+6F22 漢	Japan	Different in JIS X 0213:2004
U+FA66 辶	U+8FB6 辶	Japan	Different in JIS X 0213:2004

Note that for some locales and for some code points, the application of Normalization effectively removes distinctions.

Interestingly, dakuten- and handakuten-adorned kana are also subject to Normalization, as is the distinction between half- and full-width katakana. Those adornments are considered to be accents in the context of Normalization.

There are four types or levels of Normalization, as described in Table 3-94. The Normalization that is applied to CJK Compatibility Ideographs takes place regardless of the Normalization type. However, the Kangxi Radicals and CJK Radicals Supplement blocks are subject to Normalization, but only for the NFKD and NFKC types.

Table 3-94. The four Normalization types

Normalization type	Full name	Description
NFD	Normalization Form D	Canonical Decomposition
NFC	Normalization Form C	Canonical Decomposition, followed by Canonical Composition
NFKD	Normalization Form KD	Compatibility Decomposition
NFKC	Normalization Form KC	Compatibility Decomposition, followed by Canonical Composition

Table 3-95 uses katakana ガ (pronounced *ga*) in both full- and half-width forms, along with the CJK Compatibility Ideograph 漢 (U+FA47) and Kangxi Radical 一 (U+2F00), to exemplify the differences between the four types of Normalization and how they are applied to typical Japanese text.

Table 3-95. The four types of Normalization—Japanese examples

Character	Unicode	NFD	NFC	NFKD	NFKC
ガ	U+30AC	<U+30AB, U+3099>	U+30AC	<U+30AB, U+3099>	U+30AC
ガ (カ + ゛)	<U+30AB, U+3099>	<U+30AB, U+3099>	U+30AC	<U+30AB, U+3099>	U+30AC
ｶﾞ (ｶ + ﾞ)	<U+FF76, U+FF9E>	<U+FF76, U+FF9E>	<U+FF76, U+FF9E>	<U+30AB, U+3099>	U+30AC
漢	U+FA47	U+6F22	U+6F22	U+6F22	U+6F22
㆒	U+2F00	U+2F00	U+2F00	U+4E00	U+4E00

Korean hangul are also subject to Normalization, and NFD and NFKD result in decomposition into their component jamo. Compound jamo, however, do not further decompose. Table 3-96 provides examples of Normalization of Korean text, using the hangul syllable 갃 (pronounced *gaks*) and the hanja 樂 (pronounced *nak*, *rak*, *ak*, or *yo*) as examples.

Table 3-96. The four types of Normalization—Korean examples

Character	Unicode	NFD	NFC	NFKD	NFKC
갃	U+AC03	<U+1100, U+1161, U+11AA>	U+AC03	<U+1100, U+1161, U+11AA>	U+AC03
樂	U+F914	U+6A02	U+6A02	U+6A02	U+6A02
樂	U+F95C	U+6A02	U+6A02	U+6A02	U+6A02
樂	U+F9BF	U+6A02	U+6A02	U+6A02	U+6A02

More details about how Normalization of hangul syllables is handled, including some useful historical information, can be found online.*

The complexity of Normalization is clearly beyond the scope of this book, and I encourage you to explore Unicode resources if what is presented here does not satisfy your needs.†

Other CJK-related characters

U+3000 through U+33FF is a region for encoding additional CJKV-related characters, such as CJKV-specific punctuation, hiragana, katakana, zhuyin, jamo, kanbun, CJK strokes, CJK-specific annotated forms, katakana ligatures, and so on.

There is one code point within this region that deserves some historical mention. U+332C is a katakana ligature meaning "parts" (パーツ *pātsu*). The original intent of the submission was to represent the Thai currency symbol "Baht" (see U+0E3F), which is expressed in Japanese by バーツ (*bātsu*). Unfortunately, this パ (*pa*) versus バ (*ba*) error was present in Unicode for so long that a correction was neither viable nor practical, and would likely cause compatibility issues in some environments. To some extent, U+332C can be considered a "phantom" character.

* *http://www.i18nl10n.com/korean/jamo.html*

† *http://www.unicode.org/charts/normalization/*

Ideographic Variation Sequences—the encoding of otherwise unified glyphs

Unicode version 4.0 introduced 240 Variation Selectors (VSes), encoded in Plane 14, from U+E0100 through U+E01EF. These are named VS17 through VS256. If you're wondering about VS1 through VS16, they are encoded in the BMP, from U+FE00 through U+FE0F. Only VS17 through VS256 are used within the context of Ideographic Variation Sequences (IVSes).

Put simply, an IVS is a sequence of two Unicode characters, specifically a Base Character followed by a VS (meaning VS17 through VS256). A Base Character is any valid CJK Unified Ideograph, including Extensions A and B, but excluding CJK Compatibility Ideographs (with the exception of the 12 that are considered CJK Unified Ideographs, specifically U+FA0E, U+FA0F, U+FA11, U+FA13, U+FA14, U+FA1F, U+FA21, U+FA23, U+FA24, and U+FA27 through U+FA29), CJK Radicals Supplement (U+2E80 through U+2EF3), and Kangxi Radicals (U+2F00 through U+2FD5).

Furthermore, an IVS must be registered, to guarantee uniqueness and interoperability. The registration process is described in *Unicode Technical Standard* (UTS) #37, entitled *Ideographic Variation Database*.* The registration process involves a 90-day Public Review Issue period, and also recommends that the glyphs assigned to each Base Character adhere to Annex S unification principles.

The 14,665 ideographs found in the Adobe-Japan1-6 character collection (Adobe Systems' Japanese glyph set, which will be covered in depth when we reach Chapter 6) are the first glyphs for which IVSes were assigned. Its registration process started in mid-December of 2006, it went through two 90-day Public Review Issue periods (PRI 98 and PRI 108), and final form was declared on December 14, 2007.† Of the 14,665 ideographs in Adobe-Japan1-6, 14,647 now have IVSes assigned and registered, and all but 20 remain without an IVS and are categorized as follows:

- One was found in Extension C at U+2A9E6 (CID+14145). Its IVS can be registered now that Extension C has been declared final.
- Nineteen were submitted to Unicode as new characters. Any determined to fall under Annex S unification principles, meaning that they are to be treated as variants of existing CJK Unified Ideographs, shall be registered as IVSes.

In terms of text processing, IVSes represent a new challenge for applications and OSes. Using UTF-16 encoding as an example, the Base Character component requires 16 bits (BMP) or two 16-bit units (Plane 2), and the Variation Selector component requires two 16-bit units (Plane 14). This means that 48 or 64 effective bits are required to represent an IVS, depending on whether the Base Character is encoded in the BMP or in Plane 2.

* *http://www.unicode.org/reports/tr37/*

† *http://www.unicode.org/ivd/*

Ideographic Description Characters and Ideographic Description Sequences

The 12 Ideographic Description Characters (IDCs), encoded from U+2FF0 through U+2FFB, are used to visually describe the structure of ideographs by enumerating their components and arrangement.* Ideographic Description Sequences (IDSes) are useful for determining whether two characters are the same, or even visually similar. In other words, one practical application of IDSes is to detect potential duplicate characters and to prevent them from being introduced into Unicode. It should be noted that Taichi Kawabata (川幡太一 *kawabata taichi*) has developed and manages extensive and extremely useful IDS databases.† The first attempt to create an IDS database for all Unicode ideographs was done by Kyoto University's *CHaracter Information Service Environment* (CHISE) project.‡

Given that there are locale-specific character form differences, multiple IDSes can be defined for single code points. In fact any new ideographs to be proposed for Unicode must include IDSes. Given the large number of characters in Unicode's CJK Unified Ideographs blocks, the IDS requirement for new submissions is a very prudent measure.

Table 3-97 lists the 12 IDCs that are in Unicode, their Unicode code points, and examples of their usage in single-operator, non-recursive IDSes.

Table 3-97. Ideographic Description Characters and example Ideographic Description Sequences

IDC	Unicode	IDS examples
⿰	U+2FF0	瑠＝⿰王留
⿱	U+2FF1	美＝⿱羊大
⿲	U+2FF2	粥＝⿲弓米弓
⿳	U+2FF3	壱＝⿳士𠂉匕
⿴	U+2FF4	圏＝⿴口巻
⿵	U+2FF5	閏＝⿵門王
⿶	U+2FF6	凶＝⿶凵メ
⿷	U+2FF7	医＝⿷匚矢
⿸	U+2FF8	厭＝⿸厂猒
⿹	U+2FF9	気＝⿹气メ
⿺	U+2FFA	越＝⿺走戉
⿻	U+2FFB	衍＝⿻行氵

* The use of U+2FFB (⿻) is strongly discouraged. However, it seems that some cases clearly require its use.

† *http://kanji-database.sourceforge.net/*

‡ *http://kanji.zinbun.kyoto-u.ac.jp/projects/chise/ids/*

Each IDC must be followed by two or three elements (the exact number depends on the IDC). In addition, many characters require multiple IDCs, which results in IDSes with a hierarchical structure. Table 3-98 illustrates examples of such IDSes. The parentheses that are used in the "IDS Hierarchy" column serve to better illustrate the parsing and are not considered part of an IDS.

Table 3-98. Hierarchical Ideographic Description Sequences—examples

Ideograph	IDS	IDS hierarchy
隱	⿰阝⿱爫⿰彐心	⿰阝(⿱爫⿰彐心)
穎	⿰禾⿱匕禾頁	⿰禾(⿱匕禾)頁
慧	⿱⿱⿰丰丰彐心	⿱(⿱(⿰丰丰)彐)心
鬱	⿱⿰木缶木冖⿰鬯彡	⿱(⿰木缶木)冖(⿰鬯彡)
蠹	⿱⿱士冖⿱石⿰虫虫	⿱(⿱士冖)(⿱石(⿰虫虫))

The ordering of IDS elements does not necessarily represent the correct stroke-order sequence for the ideograph, even ignoring those cases when the stroke-order of an ideograph is locale-dependent.

In terms of statistics, the IDC that is used the most frequently as the first IDC in an IDS is U+2FF0 (⿰), which describes a left-right or side-by-side arrangement of elements. Of the 70K or so ideographs in Unicode, approximately 50K use this as the first IDC. The next most frequently used IDC is U+2FF1 (⿱), which describes a top-bottom arrangement; approximately 20K of Unicode's ideographs use this as the first IDC in their IDSes.

In terms of their history, IDCs were originally included in the GBK character set, in which they were encoded from <A9 8A> through <A9 95>. Temporary Unicode mappings were assigned, from U+E7E8 through U+E7F3, and then Unicode version 3.0 subsequently added them from U+2FF0 through U+2FFB. The GB 18030-2000 character set standard specifies the Unicode version 3.0 mappings for these characters.

Character Description Language

Tom Bishop of Wenlin Institute invented what is now known as the *Character Description Language* (CDL), which is best described as an XML application for visually describing ideographs. Between him and Richard Cook of UC Berkeley, CDL descriptors for tens of thousands of ideographs, covering Unicode's CJK Unified Ideographs blocks, have been created.

The CDL was originally conceived as a way to synthesize the glyphs for a font to be used in Wenlin Institute's own application called *Wenlin*.* In addition to its ability to fully describe any ideograph in a visual manner, comparable to an IDS, its 128×128 design space is

* *http://www.wenlin.com/*

sufficient to render subtle details, and also has the ability to capture the writing order of each stroke, along with the writing direction of the stroke.

CDL descriptors can be nested, meaning that a CDL descriptor can refer to another CDL descriptor for one or more of its components. The CDL also has a mechanism for specifying variant forms.

More information about the CDL can be found on Wenlin Institute's website.*

CJKV locale-specific character form differences

The result of the processes that took place when developing Unicode, from a practical point of view, is a set of 20,902 partially unified ideographs. This has certain implications for those who feel that they can build a single typeface that will satisfy the character-form criteria for all CJKV locales.

Table 3-99 illustrates several ideographs contained in Unicode, along with example representations in four of the CJKV locales.

Table 3-99. CJKV character form differences

Unicode code point	China	Taiwan	Japan	Korea
U+4E00	一	一	一	一
U+4E0E	与	与	与	与
U+5224	判	判	判	判
U+5668	器	器	器	器
U+5B57	字	字	字	字
U+6D77	海	海	海	海
U+9038	逸	逸	逸	逸
U+9AA8	骨	骨	骨	骨

Note how U+4E00 can use the same character form (glyph) across all four CJKV locales, but that the others are slightly different across locales, though some forms are shared by more than one locale. This is not a criticism of Unicode, but rather the reality one must deal with when building products based on Unicode that are designed to cover more than one CJKV locale. In terms of font products, they are referred to as *Pan-CJKV* fonts.

* *http://www.wenlin.com/cdl/*

If you look carefully at the glyphs in Table 3-99, there are clear cases of locale-specific differences that would persist regardless of typeface design, such as 骨 versus 骨. The top element appears to be simply mirrored, but the former glyph is considered to be one stroke less, thus simplified.

Unicode versus vendor character sets

Prior to Unicode effectively becoming the default or *de facto* way in which text is handled in modern OSes and applications, vendor character sets were quite common. A vendor character set is typically built on top of a national character set. For example, there are at least a dozen vendor character sets that have been built on top of Japan's JIS X 0208 character set standard. What made vendor character sets problematic is that they often conflicted with one another.

Interestingly, and for the greater good, Unicode has all but made the need for vendor character sets go away. The fact that characters are added to Unicode on a somewhat regular basis, thanks to the process that is in place to do so, means that vendors who would otherwise resort to defining their own character set extensions can now find the necessary characters somewhere within Unicode. If they cannot find the characters that they need, they can take the necessary steps to propose them as new characters.

GB 13000.1-93

The Chinese translation of ISO 10646-1:1993 is designated GB 13000.1-93 (信息技术—通用多八位编码字符集 (UCS)—第一部分: 体系结构与基本多文种平面 *xìnxī jìshù—tōngyòng duōbāwèi biānmǎ zìfùjí (UCS)—dìyī bùfen: tǐxì jiégòu yǔ jīběn duōwénzhǒng píngmiàn*), and established on August 1, 1994. GB 13000.1-93 appears to be a verbatim translation. As indicated in Table 3-81, the GB 13000.1-93 standard is aligned with Unicode version 1.1.

Given GB 18030-2000 and its 2005 update, along with the extent to which they are closely aligned to Unicode and ISO 10646, the need to update GB 13000.1-93, specifically to keep it aligned with the latest versions of Unicode and ISO 10646, has clearly diminished.

CNS 14649-1:2002 and CNS 14649-2:2003

Like China, Taiwan has also developed its own national standard that is effectively a translation of ISO 10646. These translations are designated CNS 14649-1:2002 and CNS 14649-2:2003, and are entitled 資訊技術—廣用多八位元編碼字元集 (UCS)—第 1 部: 架構及基本多語文字面 (*zīxùn jìshù—guǎngyòngduō bā wèiyuán biānmǎ zìyuánjí (UCS)—dìyībù: jiàgòu jí jīběn duōyǔ wénzìmiàn*) and 資訊技術—廣用多八位元編碼字元集 (UCS)—第 2 部: 輔助字面 (*zīxùn jìshù—guǎngyòngduō bā wèiyuán biānmǎ zìyuánjí (UCS)—dìèrbù: fǔzhù zìmiàn*), respectively. Due to the versions of the two ISO 10646 parts to which they are aligned, CNS 14649-1:2002 and CNS 14649-2:2003 are aligned with Unicode versions 3.0 and 3.1, respectively.

JIS X 0221:2007

The first Japanese translation of the ISO 10646-1:1993 standard was designated JIS X 0221-1995 (国際符号化文字集合 (UCS)—第 1 部：体系及び基本多言語面 *kokusai fugōka moji shūgō (UCS)—daiichibu: taikei oyobi kihon tagengomen*), and was established on January 1, 1995. JIS X 0221-1995 contained additional sections, on pages 799–1027, that listed Japanese-specific information. The most interesting of these are the following:

- A table that provides the Chinese 康熙字典 (*kāngxī zìdiǎn*) and Japanese 大漢和辭典 (*dai kanwa jiten*) index numbers for all 20,902 ideographs
- A section that describes the Japanese subsets

The Japanese subsets as discussed in JIS X 0221-1995 are listed in Table 3-100. The subsets are described in terms of seven parts.

Table 3-100. JIS X 0221-1995's Japanese subrepertoires

Subrepertoire	Characters	Description
Basic Japanese	6,884	JIS X 0208:1997, JIS X 0201-1997
Japanese Non-ideographic Supplement	1,913	JIS X 0212-1990 non-kanji plus other non-kanji
Japanese Ideographic Supplement 1	918	JIS X 0212-1990 kanji
Japanese Ideographic Supplement 2	4,883	Remainder of JIS X 0212-1990
Japanese Ideographic Supplement 3	8,745	Remainder of ideographs
Full-width Alphanumeric	94	For compatibility
Half-width katakana	63	For compatibility

JIS X 0221 was updated in 2001 to bring it into alignment with ISO 10646-1:2000, and it was redesignated JIS X 0221-1:2000. It was updated again in 2007. As indicated in Table 3-81, JIS X 0221-1995 is aligned with ISO 10646-1:1993, and JIS X 0221-1:2001 is aligned with ISO 10646-1:2000.

KS X 1005-1:1995

The Korean translation of ISO 10646-1:1993, designated KS X 1005-1:1995 (국제 문자 부호계 (UCS) 제 1 부 : 구조 및 기본 다국어 평면 *gukje munja buhogye (UCS) je 1 bu: gujo mich gibon dagukeo pyeongmyeon*), was established on December 7, 1995.* While the frontmatter is translated into Korean, the annexes are left untranslated. As indicated in Table 3-81, KS X 1005-1:1995 is aligned with Unicode version 2.0.

* Previously designated KS C 5700-1995

Character Set Standard Oddities

While the contents of character set standards are, for the most part, assumed* to be error-free by software developers and users, many of them do exhibit some interesting characteristics that can only be described as oddities. For example, some character set standards contain duplicate characters, some include characters that do not or should not exist (although one could argue that such characters now exist by virtue of being in a character set standard), some do not contain some characters that should have been included (because they are needed to complete character pairs, to form a known word, or to provide the corresponding traditional form of a simplified character), and some became endowed with fictitious extensions. The following sections detail these oddities and draw examples from common character set standards.

Duplicate Characters

The most common character set oddity is duplicate characters. In some cases, duplicate characters are intentional. Take, for instance, the hanja in KS X 1001:2004 that are ordered according to their readings. For those hanja that have been classified with multiple readings, multiple instances of the hanja have been encoded in that standard. For reasons of compatibility, Unicode needs to propagate these duplicate characters, which it does through the use of its CJK Compatibility Ideographs blocks.

Speaking of CJK Compatibility Ideographs, not all of them are genuine duplicate characters. Many of them are intended to be different from their Canonical Equivalents, sometimes in seemingly subtle ways, yet fall within the scope of the Annex S unification principles.

The most intriguing cases of duplicate characters are clearly those that are not intentional. Character sets are designed by humans, and being the imperfect creatures that we are, things that we create are prone to having errors. The greater the number of characters with which one is dealing, the greater the potential for the introduction of errors, to include duplicate characters. Thus, as the number of ideographs in Unicode grows, the greater the potential for introducing duplicate characters. Thankfully, methods have been proposed to help prevent duplicate characters from being introduced.

From my own experience, the committees that are charged with compiling new or revised national standards can take advantage of Unicode by mapping their characters to its latest version, which is one way to detect duplicate characters. This method of detecting duplicate characters is performed at the character code level. The introduction of IDSes is yet another way to detect duplicate characters, and is best applied for new characters being proposed to Unicode. Due to the nature of IDSes, this method is visual and can result in false positives. Clearly, the use of IDSes eases the process by which duplicate characters can be detected, in order to prevent them from being introduced into Unicode.

* In case you were not aware, the word *assume* is an acronym.

Phantom Ideographs

While the ideographs that are included in noncoded character set standards have been carefully accounted for during the development process, there are documented cases of coded character sets that include ideographs whose origins cannot be determined. These are called phantom ideographs, written as 幽霊漢字 (*yūrei kanji*) in Japanese.

During the development of JIS X 0208:1997, a team of researchers lead by Kohji Shibano (芝野耕司 *shibano kōji*) attempted to account for every kanji included in the standard in terms of sources and references. While some marginal cases were discovered during this lengthy process, which involved sifting through countless name records, at least one phantom character was identified. It is 弼, which is JIS X 0208:1997 55-27. The description of the team's findings starts on page 291 of the standard document.

Interestingly, any character set standard that is in any way based on JIS X 0208:1997 (such as vendor extensions thereof, and even Unicode) will inherit this phantom character. The corresponding Unicode code point for 弼 is U+5F41. The same can be said about other coded character set standards that happen to include phantom characters.

Incomplete Ideograph Pairs

Several studies of the GB 2312-80 character set standard have been conducted since its establishment, and some of them point out the fact that for several two-hanzi words (or ideograph pairs), only one of the component hanzi is in GB 2312-80. The most remarkable aspect of this phenomenon is that these hanzi usually appear together—almost never with other hanzi or in isolation. These pairs of hanzi are called 连绵字 or 联绵字 (both transliterated *liánmiánzì*) in Chinese.

One such study, conducted by Dejin Wang (王德进 *wáng déjìn*) and Sheying Zhang (张社英 *zhāng shèyīng*) in 1989, identified six such cases in which one of the component hanzi is in GB 7589-87 (four cases), GB 7590-87 (one case), or GB 13000.1-93 (one case).* Table 3-101 provides two examples of two-hanzi compounds that contain one hanzi not present in GB 2312-80.

Table 3-101. Incomplete hanzi pairs—examples

Hanzi	Reading	Meaning	GB code points	Unicode code points
鹠鹩	*xiǔliú*	owlet	GB 2312-80 80-28 and GB 7589-87 62-11	U+9E3A and U+9DB9
欷歔	*xīxū*	to sob, sigh	GB 2312-80 76-04 and GB 7590-87 48-24	U+6B37 and U+6B54

* The study appeared in a paper entitled "Amendments on the GB 2312-80" (关于修改GB 2312-80的几点意见 *guānyú xiūgǎi GB 2312-80 de jǐdiǎn yìjian*), published in *Proceedings Papers of International Symposium on Standardization for Chinese Information Processing* (中文信息处理标准化国际研讨会论文集 *zhōngwén xìnxī chǔlǐ biāozhǔnhuà guójì yántǎohùi lùnwénjí*), 1989, also known as *SCIP 89*.

Since GB 18030 is so important today, because it is the most important GB standard to date, it should be pointed out that GB 7589-87 62-11 and Unicode U+9DB9 correspond to GB 18030 <FA 56>, and that GB 7590-87 48-24 and Unicode U+6B54 correspond to GB 18030 <9A 5B>.

Given the relative size of Unicode and GB 18030, the occurence of incomplete ideograph pairs has clearly diminished, but given the open-ended nature of the writing system, one clearly cannot claim that this phenomenon has been eliminated.

Simplified Ideographs Without a Traditional Form

Although it is obvious that all traditional ideograph forms need not have a corresponding simplified form—either because their form is already considered simple enough, or because its use is relatively rare (only commonly used ideographs tend to develop simplified forms)—there are instances of simplified ideographs that lack a corresponding traditional form. I must point out that while it is generally the case that simplified forms have a corresponding traditional form, it is not an absolute requirement, because it is possible to coin new characters that use simplified components.

In general, the lack of a corresponding traditional forms can be considered a limitation of the character set, and is indeed a character set oddity. As a real-world example of this phenomenon, consider U+4724 (讠), which lacks a corresponding traditional form, at least in the context of Unicode version 5.0 and earlier. Interestingly, this did not remain the case for long, because its corresponding traditional form was proposed and subsequently accepted for inclusion in Unicode at code point U+9FC1 (訁) and is included in Unicode version 5.1.

Fictitious Character Set Extensions

When the 20,902 ideographs in ISO 10646-1:1993 (Unicode version 1.1) were compiled from numerous national standards, the respective national-standard–developing organizations submitted character set materials to the CJK-JRG (*CJK Joint Research Group*, now called the IRG, which is an abbreviation for *Ideographic Rapporteur Group*) for inclusion. However, in order to ensure that certain ideographs became part of the final set of 20,902 characters, at least two national standards included fictitious extensions. That is, ideographs that are not part of the standard, and never likely to be, were added. Affected character sets include China's GB/T 12345-90 and Taiwan's CNS 11643-1986 Plane 14 (bear in mind that CNS 11643-1992 was not yet published at that time).

For GB/T 12345-90, any code point beyond Row-Cell 89-09 should be questioned and is likely to fall into this area. For CNS 11643-1986 Plane 14, any code point beyond Row-Cell 68-21 should be in doubt. When dealing with CNS 11643-1992 Plane 3 (identical to the first 6,148 hanzi in CNS 11643-1986 Plane 14; the remaining 171 hanzi became scattered throughout CNS 11643-1992 Plane 4), any code point beyond Row-Cell 66-38 may be problematic. Well, to be perfectly honest, they *will* be problematic.

An alternative set of Unicode mapping tables have been developed by Koichi Yasuoka (安岡孝一 *yasuoka kōichi*), which do not include these fictitious character set extensions.*

Seemingly Missing Characters

As pointed out earlier in this chapter, Chinese and Korean character sets include the complete set of Japanese kana, specifically hiragana and katakana. While this seems to indicate that Chinese and Korean fonts that are based on these character sets can render arbitrary Japanese text that includes hiragana and katakana, this is not the case. As an example of a katakana character that is missing from most Chinese and Korean character sets that otherwise suggest that they include all Japanese kana characters, take for instance the Japanese long vowel mark, ー, which is JIS X 0208:1997 01-28 and maps to Unicode U+30FC. This character is absolutely necessary to properly render arbitrary katakana text. To less of an extent, JIS X 0208:1997 01-19 through 01-22, which correspond to Unicode U+30FD, U+30FE, U+309D, and U+309E, respectively, are also necessary to render hiragana and katakana text.

Some Chinese character sets have recognized these omissions and have since added these characters. GBK, which is an extended version of GB 2312-80, added the Japanese long vowel mark at <96 C0>. GB 18030-2000 and its 2005 revision, due to their GBK heritage, inherit the same character at the same code point.

CJK Unified Ideographs with No Source

The ideographs that are included in Unicode almost always have a source or a source glyph, meaning that the character is included because it maps to another standard, which is a character set standard, or in some cases, a dictionary. A very small number of CJK Unified Ideographs Extension B characters have no source glyph, specifically the following three code points: U+221EC, U+22FDD, and U+24FB9. Given the importance of associating or linking an ideograph with one or more sources, especially in the context of CJK Unified Ideographs, it makes one wonder how these ideogaphs were able to get into the standard. Still, it is pointless to argue about them, and the only concern is how to properly design their glyphs, whether it is for one or more locales.

Vertical Variants

CJKV text can be set vertically, which can cause some characters to have slightly different forms or different relative positions. So, some character sets have chosen to include vertical forms as separate characters, meaning they are at separate code points from their horizontal counterparts. Some character sets do not directly encode vertical variants, and instead expect some higher-level functionality, such as a glyph substitution feature triggered by vertical writing, to handle vertical forms. The OpenType 'vert' GSUB (*Glyph*

* *http://kanji.zinbun.kyoto-u.ac.jp/~yasuoka/CJK.html*

SUBstitution) feature is an example of a well-established glyph substitution implementation that is used by a wide variety of applications that support vertical writing.

Although the JIS standards themselves do not directly encode vertical variants, some vendor character sets that are based on them, in particular JIS X 0208:1997, do so.

Vertical forms of characters intended for horizontal writing, when compared to their parent (horizontal) forms, may rotate 90° clockwise, rotate 90° clockwise then flip or be mirrored, change their form completely, or reposition. Parentheses and other bracket-like characters, including arrows, rotate 90° clockwise. The wave dash (U+301C) and Japanese long vowel mark (U+30FC) are examples of characters that rotate then flip or are mirrored. Periods and commas, along the small kana that are specific to Japanese, reposition to the upper-right corner.

Interestingly, Unicode directly encodes some, but not all, vertical variants by virtue of the fact that some national character set standards encode them. This is merely for code-point compatibility. The corresponding Unicode code points and code point ranges are U+FE10 through U+FE19, U+FE30 through U+FE44, U+FE47, and U+FE48.

Some characters deserve special mention because they are used strictly for vertical writing, and thus do not have a corresponding horizontal form. Some examples, used in Japan, include U+3033 through U+3035 and U+303B. In fact, U+3033 and U+3034 are intended to be combined or fused with U+3035.

Continued discussions of horizontal versus vertical forms, to include a complete correspondence table showing their forms, can be found in Chapter 7, in which typography is covered in detail.

Noncoded Versus Coded Character Sets

Noncoded character sets relate to coded ones in a variety of ways, depending on the locale. Ideally, all NCSes should be mapped to CCSes. As NCSes change or grow, CCSes are necessarily affected. This section illustrates how CCSes attempt to follow and keep pace with NCSes, using the situations in China, Taiwan, Japan, and Korea as examples. You will also learn how some NCSes exert influence, sometimes quite strongly, over the development of new and existing CCSes.

China

All 2,500 Chángyòng Hànzì are included in GB 2312-80. All but five are in Level 1 hanzi. For the 1,000 Cìchángyòng Hànzì, 998 are in GB 2312-80. 880 are in Level 1 hanzi, 118 are in Level 2 hanzi, and 2 are in GB 7589-87. Table 3-102 indicates how these two hanzi can be mapped.

Table 3-102. Two Cìchángyòng Hànzì not in GB 2312-80

Hanzi	GB 7589-87	GB 8565.2-88	GB/T 12345-90	GBK and GB 18030	Unicode
啰	22-51	15-93	n/a[a]	<86 AA>	U+5570
瞭	58-43	93-47	88-49	<B2 74>	U+77AD

a. Oddly enough, its traditional form, 囉, is at 88-51. Neither GB 2312-80 nor GB/T 12345-90 contain the simplified form.

The remaining 3,500 hanzi of the Tōngyòng Hànzì list are distributed as follows: 3,095 are in GB 2312-80 (380 are in Level 1 hanzi, and the remaining 2,715 are in Level 2 hanzi), 404 are in GB 7589-87, and 1 is in GB 7590-87.

Appendix G includes the Chángyòng Hànzì and Cìchángyòng Hànzì hanzi lists broken down by the character sets, and the hanzi levels that support them.

Taiwan

The list of 4,808 hanzi, which can be called Taiwan's Chángyòng Hànzì, was used as the basis for Big Five Level 1, CNS 11643-2007 (and -1992 and -1986) Plane 1, and CCCII. The additional hanzi lists, such as their Cìchángyòng Hànzì, were used to define the remainder of the above coded character sets.

Japan

All of Tōyō Kanji were included in JIS Level 1 kanji of JIS C 6226-1978. When Jōyō Kanji was introduced in 1981, the additional 95 kanji and subsequent glyph changes forced the creation of JIS X 0208-1983 (first called JIS C 6226-1983, and then changed to the new designation in 1987)—those extra 95 characters had to be made part of JIS Level 1 kanji (22 simplified and traditional kanji pairs exchanged code points between JIS Levels 1 and 2 kanji). Appendix J lists the 95 kanji that were added to Tōyō Kanji in 1981 in order to become Jōyō Kanji.

The kanji specified in Jinmei-yō Kanji, on the other hand, could appear in either JIS Levels 1 or 2 kanji, so that is why four kanji were appended to JIS X 0208-1983, and two to JIS X 0208-1990. Table 3-103 lists the four kanji appended to JIS Level 2 kanji in 1983 in order to create JIS X 0208-1983.

Table 3-103. Four kanji appended to JIS X 0208-1983

Kanji	JIS X 0208-1983	Unicode
堯	84-01	U+582F
槇	84-02	U+69C7
遙	84-03	U+9059
瑤	84-04	U+7464

Table 3-104 lists the two kanji that were appended to JIS Level 2 kanji in 1990 in order to create JIS X 0208-1990.

Table 3-104. Two kanji appended to JIS X 0208-1990

Kanji	JIS X 0208-1990	Unicode
凜	84-05	U+51DC
熙	84-06	U+7199

There is no direct relationship between Gakushū Kanji and CCSes, but Gakushū Kanji is a subset of Jōyō Kanji.* Thus, Gakushū Kanji is primarily discussed in the context of NCSes.

The NLC Kanji set, consisting of 1,022 kanji, along with the current set of 983 Jinmei-yō Kanji, are the primary reasons for the changes made to the JIS X 0213:2004 standard—specifically the prototypical glyph changes for 168 of its kanji, along with 10 additional kanji. This again demonstrates that NCSes continue to be developed, and that they exert influence, sometime heavily, on CCSes.

Another potentially interesting aspect of NCSes when compared to their coded counterparts is that they tend to be under stricter glyph control. In other words, a commissioned group of experts spend a significant amount of time and effort to determine the correct form of kanji, typically for those that are more frequently used. If one compares the number of kanji in the NCSes with those in the CCSes, one sees that the potential for future change, in terms of prototypical glyph changes in CCSes, exists. Table 3-105 lists the number of unique kanji in the current Japanese noncoded and coded character sets.

Table 3-105. The number of kanji in Japanese noncoded and coded character sets

Character set type	Character sets	Number of unique kanji
NCS	Jōyō Kanji + Jinmei-yō Kanji + NLC Kanji	3,506[a]
CCS	JIS X 0208 + JIS X 0212 + JIS X 0213	13,109[b]

a. The Jinmei-yō Kanji and NLC Kanji sets significantly overlap. Of the 983 kanji in the former, and the 1,022 kanji in the latter, 444 kanji are common. This means that there are 1,561 unique kanji when the two sets are combined.

b. The JIS X 0213 and JIS X 0212 character sets significantly overlap. Of the 3,695 kanji in the former, and the 5,801 kanji in the latter, 3,058 kanji are common. This means that there are 6,753 unique kanji when the two sets are combined.

Table 3-105 draws attention to the fact that barely over 25% of the kanji in the JIS standards are under a stricter degree of glyph control. What does this mean? Simply that there is potential for additional prototypical glyph changes, but at the same time, the chance of this happening is relatively low. To quote FBI Special Agent Fox Mulder, *it is not outside the realm of extreme possibilities*.

* But, this means that Gakushū Kanji is a subset of JIS Level 1 kanji.

Figure 3-2 illustrates how earlier versions of the Japanese NCSes relate to each other and to JIS Levels 1 and 2 kanji, defined in the JIS standards (specifically JIS X 0208:1997).

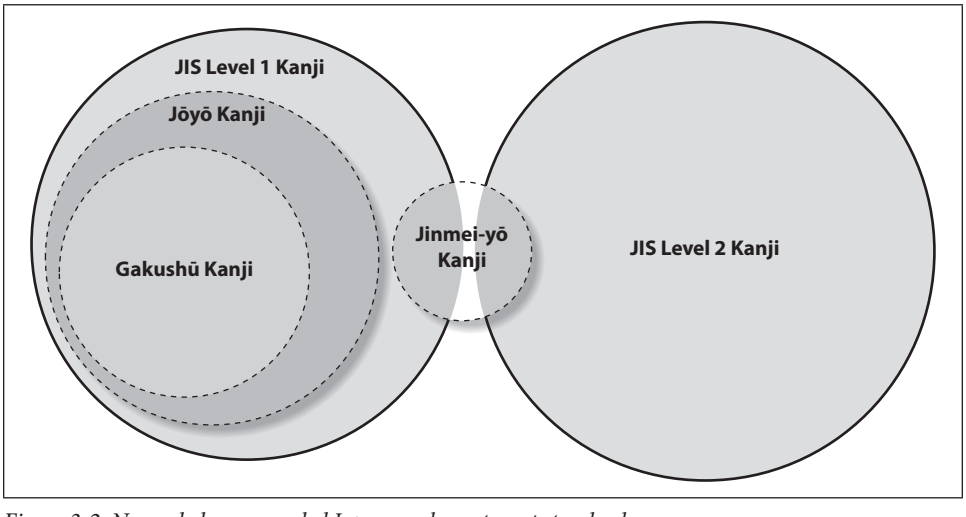


Figure 3-2. Noncoded versus coded Japanese character set standards

Korea

The 1,800 hanja enumerated in Korea's Hanmun Gyoyukyong Gicho Hanja form a subset of the 4,888 hanja in KS X 1001:2004. But, because 268 of the hanja in KS X 1001:2004 are the result of multiple encoding due to multiple readings, these 1,800 Hanmun Gyoyukyong Gicho Hanja need to be represented by more than 1,800 hanja in KS X 1001:2004. For the same reason, the middle school subset containing 900 hanja are represented by more than 900 hanja in KS X 1001:2004.

Information Interchange and Professional Publishing

Besides the distinction between noncoded and coded character set standards, there are two types of character sets in use today: those for information interchange, and those for professional and commercial publishing. The goals and intended uses of these character sets differ. The following sections detail their characteristics and the ways in which they are different from one another.

Character Sets for Information Interchange

The majority of character set standards in use today were originally designed for information interchange. Well, after all, look at some of their titles. The title of JIS X 0208:1997 is *7-bit and 8-bit Double Byte Coded Kanji Sets for Information Interchange*, and

the title of ASCII (officially designated ANSI X3.4-1986) is *Coded Character Set—7-bit American National Standard Code for Information Interchange*. Information interchange can be defined as the process of moving data from one hardware or software configuration to another with little or no loss of information. Of course, no loss of information is better than little loss.

Unicode, although it provides a vast number of characters, was still designed with information interchange in mind—building a superset based on national standards, which were designed for information interchange, still results in a character set for information interchange. This is not a bad thing. In fact, it is a very good thing. Unicode effectively levels the playing field in terms of how characters are represented, and also serves to simplify and ease software development.

The vast majority of today's CJKV fonts include glyphs for characters from character set standards designed for information interchange. While the glyphs that these fonts provide are sufficient for most users' needs, they are far from ideal for professional or commercial publishing.

Character Sets for Professional and Commercial Publishing

The ASCII character set, as was discussed earlier in this chapter, consists of 94 printable characters. ISO 8859-1:1998, the most common extension to ASCII, adds another 95. But these two character sets are still missing characters critical for professional or commercial publishing, such as smart (or "curly") quotes, the various dashes (such as the en dash and em dash), and so on. Even the most basic English-language publication requires characters not found in ISO 8859-1:1998. Most of today's non-CJKV fonts, such as Type 1, TrueType, and OpenType, include glyphs for professional publishing. Their glyph set specifications evolved out of the needs of the professional and commercial publisher, and, at the same time, still provide the basic set of characters (in other words, ASCII) for nonprofessional users.

The same is true for the CJKV locales. JIS X 0208:1997, for example, is missing characters that are necessary for professional publishing. In Japan, the following character classes and features are necessary in character sets intended for professional or commercial publishing purposes:

- Proportional Latin and corresponding italic design for Latin text
- Macron-adorned vowels—such as ā, ī, ū, ē, and ō—for transliterated Japanese text
- Kanji variants (simplified, traditional, and other forms)
- Additional kanji, including variant forms
- Additional punctuation and symbols
- Alternate metrics (half-width symbols/punctuation, proportional kana, and so on)

More information regarding these character classes and typographic features will be presented in Chapter 7.

Vendors that provide professional publishing systems commonly build or supply fonts based on character sets for information interchange, but heavily supplement such fonts with glyphs for the character classes just listed, usually at unique code points. Why do these character sets include characters for information interchange? Most documents that are eventually published originate on systems that process character sets for information interchange, so that is why character sets for professional publishing must use those characters as their base.

Examples of character sets for professional publishing include Fujitsu's JEF, Morisawa's MOR-CODE II, and Shaken's SK78. The major disadvantage of these and other character sets for professional publishing is that they are usually restricted to proprietary hardware or software, and thus require a major investment.

Future Trends and Predictions

There are two specific issues that I'd like to cover that represent future trends and are effectively predictions based on observations I have made over the years. The shorter-term issue is today's trend to use of *emoji*, which are character-like pictures that convey meanings, including subtle emotional states. These are broadly used in Japan and are associated with text messages sent to and from mobile devices. The longer-term issue is a prediction that the locale-specific distinctions of ideographs, such as those shown in Table 3-99, may some day go away.

Emoji

Mobile phones have become extremely popular in Japan and other regions. In Japan, mobile phones are referred to as 携帯電話 (*keitai denwa*). Their capabilities have become so sophisticated that many people use them in lieu of a computer. In addition to being able to use them for voice communication, meaning as a phone or for leaving or listening to voice messages, they also serve to send and receive text messages. Given the necessary brevity of text messages, the Japanese culture came up with a new class of character called *emoji*, which is written 絵文字 (*emoji*) in Japanese.* These characters are often called *emoticons*, though most emoji are not true emoticons. Of course, many simple emoticons have been used, and continue to be used, such as the infamous *smiley* that is composed of three ASCII characters, and necessarily rotated 90° counter-clockwise: :-).

The primary issue with emoji is that they lack standardization. There are three sets of emoji being used in Japan, each of which is associated with a different mobile phone provider: *i-mode* by NTT DoCoMo, *EZweb* by KDDI au, and *SoftBank Mobile* (formerly Vodafone). Google also supports emoji as a fourth set. Table 3-106 lists a small number of

* *http://en.wikipedia.org/wiki/Emoji*

emoji, along with the various codes associated with them, according to these four emoji sets. The codes are Shift-JIS (user-defined region), Unicode (PUA) scalar values, and simple index numbers. Although these examples do not represent widely used emoji, they are ones that happen to have Unicode code points, as the table indicates.

Table 3-106. Emoji examples

Emoji	Meaning	Unicode	i-mode	EZweb	SoftBank Mobile	Google
	sunny	U+2600	F89F/U+E63E 1	F660/U+E488 44	F98B/U+E04A 74	U+FE000
	cloudy	U+2601	F8A0/U+E63F 2	F665/U+E48D 107	F98A/U+E049 73	U+FE001
	rain	U+2614	F8A1/U+E640 3	F664/U+E48C 95	F98C/U+E04B 75	U+FE002
	snow	U+26C4	F8A2/U+E641 4	F65D/U+E485 191	F989/U+E048 72	U+FE003
	hot springs	U+2668	F99C/U+E6F7 147	F695/U+E4BC 224	F763/U+E123 125	U+FE7FA

Clearly, there is a problem. There are two standardization issues that need to be solved. First, these four sets of emoji are not completely compatible with one another, meaning that when a text message containing emoji is sent from a cell phone from one carrier to another, the proper treatment of emoji cannot be guaranteed. Second, these emoji are currently being exchanged through the use of Unicode PUA or Shift-JIS user-defined code points, both of which are considered bad practice in today's world of open systems.

In terms of their visual representation, emoji are almost always colored, often with more than one color. Some are even animated. Given the limitations of today's font formats, the multicolored and animated emoji are effectively graphics with character-like attributes. They can be represented through various means, but what is important is the way in which they are conveyed in an "information interchange" environment.

Finally, I need to point out that emoji are not specific to Japan and are used throughout the world. The first step to solving this issue is to first acknowledge that emoji are characters, and to treat them as such in terms of standardization. Any standardization efforts affecting their status as characters obviously involves Unicode.

Genuine Ideograph Unification

Table 3-99 illustrated that there are sometimes subtle, but necessary, glyph differences that must be taken into account when representing ideographs across CJKV locales. These glyph differences are simply an artifact of the conventions used in those regions. The ideographs themselves represent the same meaning, regardless of these subtle differences.

To some extent, the notion of *Simplified Chinese* is anything but simple, and instead made the world a more complex place. As a font developer, it is interesting to note that typical Simplified Chinese fonts include more glyphs than those for Traditional Chinese, Japanese, or Korean. Simplified Chinese served to increase the repertoire of ideographs, because the original, or traditional, forms must be maintained. And, in terms of complexity,

the relationship between simplified and traditional forms, which is not always a one-to-one relationship, must also be maintained.

I predict that within 25 years there will be an initiative to unify or normalize ideographs across all locales that use them. To a great extent, Unicode and the way in which its CJK Unified Ideograph blocks capture character-level distinctions are serving as the spark for such an effort.

In other words, I predict that at some point many years into the future, a single glyph per CJK Unified Ideograph code point will become acceptable for all CJKV locales. After all, the Web and other globalization activities are serving as a catalyst to make the world a smaller place, and thus forcing or enabling cultures to interact to a greater extent than previously possible. When cultures interact, they exert influence on each other. Writing systems are easily influenced, especially if there is something to gain through their unification.

Bear in mind that until this takes place, it is prudent to preserve locale-specific glyph distinctions, and single fonts that are intended to support multiple locales must include more than one glyph for many Unicode code points, along with a mechanism for switching glyphs based on locale.

Advice to Developers

A difficult question often posed by developers is, "What character sets should my product support?" The answer to that seemingly all-important question really hinges on at least three factors, posed as the following questions:

- What specific CJKV locale or locales need to be supported by the product?
- To what extent is the developer embracing or supporting Unicode?
- On what specific platform or platforms does the application run?

If your product is designed for a single CJKV locale (which really assumes two locales: the specific CJKV locale plus non-CJKV support, meaning ASCII or the locale-specific equivalent at a minimum), you need to decide which character set to support. As will be explained, this is independent of supporting Unicode. If at all possible, you should consider supporting the supplemental character sets—such as the additional planes of CNS 11643-2007 for Taiwan, JIS X 0213:2004 for Japan, KS X 1002:2001 for Korea, and so on—especially if there is a demand from your customers. In the case of China, the use of GB 18030-2005 has been mandated by their government. But, at a bare minimum, the basic character sets should be implemented or otherwise supported for each CJKV locale that is supported. Table 3-107 lists eight major CJKV locales, along with their most commonly used and additionally recommended CCSes.

Table 3-107. The most common and recommended CJKV character sets

Locale	Common character set	Recommended character set
China	GB 2312-80	GB 18030-2005
Singapore	GB 2312-80	GB 2312-80
Taiwan	Big Five or CNS 11643-2007 Planes 1 and 2[a]	Additional planes of CNS 11643-2007
Hong Kong	Hong Kong SCS-2008	same
Japan	JIS X 0208:1997	JIS X 0213:2004[b]
South Korea	KS X 1001:2004	same
North Korea	KPS 9566-97	same
Vietnam	TCVN 5773:1993 and TCVN 6056:1995	same

a. Which one you choose depends on the underlying operating system.

b. Its definition includes all of the characters in JIS X 0208:1997, and is thus a superset thereof.

One should be aware that supporting CCSes for Chinese locales, meaning China, Taiwan, and Hong Kong, can sometimes lead to confusion or unexpected complexities. Some products attempt to support both major Chinese locales—China and Taiwan—in a single product (such was the case with Apple's Chinese Language Kit), but that is not an absolute requirement. In fact, it is sometimes beneficial to build separate products for each of these Chinese locales, for political or other reasons.

Bear in mind that things change over time. What is current at the time of this writing may be outdated five years later. The footnotes that are scattered throughout this book are references to resources that are likely to provide up-to-date information, such as new or updated character set standards.

The Importance of Unicode

But what about Unicode—and to some extent, the international and national CCSes based on it, specifically ISO 10646:2003, GB 13000.1-93, CNS 14649-1:2002, CNS 14649-2:2003, JIS X 0221:2007, and KS X 1005-1:1995? Unicode *is* here to stay and is being used to implement *real* software products. In fact, Unicode has become the preferred method by which characters are represented in today's software. I cannot stress enough the importance of embracing Unicode.

Some seemingly locale-specific character sets are best implemented through Unicode, or are intended to be implemented only through it. JIS X 0213:2004 is an example of a Japanese CCS that is intended to be implemented only through the use of Unicode. GB 18030-2005 is an example of a Chinese CCS that is best implemented through Unicode. Regardless, the best way in which to develop software today is by embracing Unicode.

But, when embracing Unicode in terms of a character set, and when implementing only one CJKV locale, what specific code points should be used? That's easy. You map the locale-specific CCS that you support to the corresponding Unicode code points. For each

locale, the work of determining which Unicode characters are the most important to support has been done by virtue of the CCSes that have been established.

Plan 9 (experimental) by AT&T Bell Laboratories, Solaris by Sun Microsystems, and Windows NT by Microsoft were OSes that represented very early adopters of Unicode. Virtually all of today's OSes, such as Linux, FreeBSD, Windows 2000/XP/Vista, and Apple's Mac OS X, fully support Unicode, including characters beyond the BMP. The Java, .NET, JavaScript, and ActionScript programming languages support Unicode as their primitive "char" type and for their "String" objects. Although these environments support Unicode in a more or less native way, developers can still take full advantage of Unicode's benefits by processing Unicode internally using virtually any programming language that supports vectors of 16-bit unsigned integers (or equivalent) as long as UTF-16 is the preferred encoding. UTF-32 encoding obviously requires 32-bit support.

It is safe to state that unless you have a very good reason not to support Unicode in your product, which is highly unlikely, you should. In fact, you are effectively crippling your product by not supporting Unicode. Several Unicode-enabled applications that have been shipping for years are excellent examples of Unicode's success (and also serve as examples of success by embracing Unicode). Early versions of JustSystems' *Ichitaro* (一太郎 *ichitarō*) Japanese word processor, for example, processed Unicode internally, yet worked as though it were still processing Shift-JIS encoding internally. The following is a small sampling of *early* software developed using Unicode in one way or another (a complete list of *current* software is now obviously very large):

- Internet Explorer
- Netscape Communicator
- Microsoft Excel
- Microsoft Word
- Oracle
- Sybase SQL Server

The Unicode Consortium maintains a list of Unicode-enabled products, which I encourage you to explore.*

Those developers who need some assistance in supporting Unicode (or are developing on a platform that does not yet support Unicode, which is a rare thing today) should consider the many Unicode-enabling programming environments or libraries that are now available. *International Components for Unicode* (ICU) is an example of a highly respected and widely used Unicode-enabling library.† See the section entitled "Other Programming Environments" in Chapter 9 for more details.

* *http://www.unicode.org/onlinedat/products.html*

† *http://www.icu-project.org/*

What follows is a bit of history that may be of interest, and that makes an important point. When JIS C 6226-1978 was first introduced in Japan in 1978, it was not met with general acceptance from the industry. The problem was that it did not include all the characters found in the major Japanese vendor character set standards already in place at the time. JIS C 6226-1978 eventually did become the standard in Japanese industry, and is now generally accepted. The Unicode Consortium, on the other hand, made sure that all of the characters from the major national standards were included as part of Unicode.* Unicode, like most other character sets, is constantly growing and evolving. Perhaps more so than other character sets given its almost universal acceptance.

* This also means that Unicode has inherited any and all errors that are in the national standards from which it was designed. With this comes compatibility with such standards, which clearly outweighs all other considerations.

CHAPTER 4
Encoding Methods

In this chapter you will learn how the CCSes presented in Chapter 3 are encoded for use on computer systems. The discussions in this chapter apply only to CCSes. That is, they do not apply to NCSes, such as Japan's Jōyō Kanji, Gakushū Kanji, Jinmei-yō Kanji, and their predecessors. To recap what you learned early on in this book, encoding is simply the mapping or assignment of a numeric value to a character.

Now that we are beginning to focus on encoding methods, you should expect to acquire a complete understanding of a large number of encoding methods, and more importantly, how they relate to and interoperate with each other. One important goal of this book will then have been achieved. If you happen to absorb other information, that is also a good thing. Otherwise, please treat much of this chapter as reference material, and consult it on an as-needed basis, being sure to bookmark or dog-ear what you feel are key pages.

Factor in that the first edition of this book was published at the end of 1998. At that time, Unicode was not yet universally recognized, accepted, or implemented. Now it is safe to state that Unicode's encoding forms have become universally recognized and accepted as the preferred encoding for text data. But, because interoperability with non-Unicode encodings continues to play an important role in today's software, you will still learn about legacy encoding methods, such as ISO-2022, EUC, Big Five, and Shift-JIS, which are still commonly used, and are expected to be for many more years. These are referred to as the *legacy encoding methods* because the word "legacy" also conveys a sense of time, as in lots of it. Because of the importance of Unicode for today's software, its encoding forms will be covered first, in order to lay the foundation for describing how legacy encoding methods compare to them and to each other.

In the first pages of this chapter, you will learn about three basic Unicode encoding forms, two of which have variations that differ only in their byte order. These are the most important encoding forms used today, and are expected to remain so for years to come. In other words, their use, although already widespread today, is ever-increasing. It is important to understand how these Unicode encoding forms work, and perhaps even more importantly, how they relate to one another.

In essence, you will learn about the following Unicode encoding forms and how they are related to each other:

- UTF-16 (UTF-16BE and UTF-16LE)
- UTF-32 (UTF-32BE and UTF-32LE)
- UTF-8

Because the UTF-16 and UTF-32 encoding forms are not byte-based, and because byte order plays a critical role in the interpretation of their code units, they typically require a beginning-of-file marker known as the *Byte Order Mark* (BOM) or must be specified by a higher-level protocol. In the absence of these, UTF-16 and UTF-32 are assumed to use big-endian byte order. There are variant forms of the UTF-16 and UTF-32 encoding forms that explicitly indicate byte order, and thus do not require a BOM to be present, specifically UTF-16BE (UTF-16 Big-Endian) and UTF-16LE (UTF-16 Little-Endian) for the UTF-16 encoding form, and likewise, UTF-32BE and UTF-32LE for the UTF-32 encoding form. You will also learn why it is incredibly useful to use a BOM for UTF-8–encoded text, although one is not required according to its specification.

In terms of legacy encodings, there are two encoding methods that are common to nearly every CJKV character set, with the exception of the GB 18030-2005, Big Five, and Hong Kong SCS-2008 character sets:

- ISO-2022
- EUC (Extended Unix Code)*

As you will learn later in this chapter, the exact definition of these two very basic encoding methods depends greatly on the locale. In other words, there are locale-specific instances of these encodings, and they will be described in this chapter, at least the ones that are CJKV-related.

There are also a number of locale-specific legacy encoding methods that are still in use today, such as the following, with the locales that they support being indicated after the dash:

- GBK and GB 18030—China
- Big Five—Taiwan and Hong Kong
- Big Five Plus—Taiwan
- Shift-JIS—Japan
- Johab—Korea

* You will soon learn that although the "U" in EUC stands for "Unix," this encoding is commonly used on other platforms, such as Mac OS and Windows.

In terms of structure and mechanics, the encoding methods described in this chapter fall into one of four possible categories:

- Modal
- Nonmodal
- Fixed-length
- Hybrid (fixed-length and nonmodal)

Modal encoding methods require escape sequences or other special characters for the specific purpose of switching between character sets or between different versions of the same character set; this sometimes involves switching between one- and two-byte modes. Modal encoding methods additionally use what I would refer to as a two-stage encoding process. The first stage is the mode switching that is initiated by the escape sequence or mode-switching characters. The second stage is the handling of the actual bytes that represent the characters. Modal encoding methods typically use seven-bit bytes. The best example of modal encoding is ISO-2022 encoding. UTF-7 encoding, although its use is deprecated, is also considered a modal encoding.

Nonmodal encoding methods, on the other hand, make use of the numeric values of the bytes themselves in order to decide when to switch between one- and two-byte modes. And, sometimes the number of bytes per character is three or four. Nevertheless, the principle is the same, specifically that the values of the bytes themselves drive the switching. These encoding methods typically make liberal use of eight-bit bytes (that is, the eighth bit of the byte is turned on or enabled) and are typically variable-length. Examples include the various locale-specific instances of EUC encoding, along with GBK, GB 18030, Big Five, Big Five Plus, Shift-JIS, Johab, and UTF-8 encodings. Nonmodal encodings typically use less space—in terms of the number of bytes required per character—than modal and fixed-length encoding methods.

Fixed-length encoding methods use the same number of bits or bytes to represent all the characters in a character set. There is no switching between one- and two-byte modes. This type of encoding method simplifies text-intensive operations, such as searching, indexing, and sorting of text, but can waste storage space or memory. Efficiency and ease-of-use are reasons why fixed-length encodings are often used for the purpose of internal processing. Examples of fixed-length encoding methods include ASCII and the UTF-32 encoding form.

UTF-16 is an example of an encoding form that can be considered *hybrid*, in the sense that although it is based on fixed-length, 16-bit code units, it also has a nonmodal component that pairs these 16-bit units as an extension mechanism to encode 1,048,576 additional code points. UTF-16 is one of the most widely used encoding forms today, so understanding how its encoding works and how to interoperate with the other Unicode encoding forms is important.

Table 4-1 summarizes all of the CJKV encoding methods that are covered in this chapter, indicating whether they are Unicode or legacy (non-Unicode), their encoding type,

the specific locales that they serve (if any) and the number of bytes or bits that they use. The three obsolete and deprecated Unicode encodings are also included for the sake of completeness.

Table 4-1. CJKV encodings

	Encoding	Encoding type	Locale	Number of bytes/bits
Unicode encoding forms	UTF-7[a]	Modal	n/a	Variable
	UTF-8	Nonmodal	n/a	One- through four-byte
	UCS-2[a]	Fixed-length	n/a	16-bit
	UTF-16	Hybrid	n/a	One or two 16-bit units
	UTF-16BE	Hybrid	n/a	One or two 16-bit units
	UTF-16LE	Hybrid	n/a	One or two 16-bit units
	UCS-4[a]	Fixed-length	n/a	32-bit
	UTF-32	Fixed-length	n/a	32-bit
	UTF-32BE	Fixed-length	n/a	32-bit
	UTF-32LE	Fixed-length	n/a	32-bit
Legacy encoding methods	**Locale-independent**			
	ASCII	Fixed-length	Multiple	One-byte
	EBCDIC/EBCDIK	Fixed-length	Multiple	One-byte
	ISO-2022	Modal	Multiple	One- and two-byte
	EUC	Nonmodal	Multiple	One- through four-byte
	Locale-specific			
	GBK	Nonmodal	China	One- and two-byte
	GB 18030	Nonmodal	China	One-, two-, and four-byte
	Big Five	Nonmodal	Taiwan and Hong Kong	One- and two-byte
	Big Five Plus	Nonmodal	Taiwan	One- and two-byte
	Shift-JIS	Nonmodal	Japan	One- and two-byte
	Johab	Nonmodal	Korea	One- and two-byte

a. Obsolete and deprecated

Be aware that not all of the encodings described in this chapter have been fully implemented—they have been defined by appropriate agencies, corporations, or committees so that their implementation, once it begins, is simplified. To some extent, the introduction and universal acceptance of the Unicode encodings has effectively stalled the development of other legacy encodings. The only legacy encoding to have been introduced since the first edition of this book was published (exactly 10 years ago) is GB 18030 encoding.

Table 4-2 enumerates all of the legacy encoding methods that are covered in this chapter, along with the specific CCSes from Chapter 3 that they support. The ISO-2022 and EUC encodings are clearly the richest, because they support the largest number of CCSes.

Table 4-2. Legacy encoding methods and supported CCSes

Encoding	Supported character sets
ASCII	ASCII, GB-Roman, CNS-Roman, JIS-Roman, KS-Roman
Extended ASCII	ASCII, GB-Roman, CNS-Roman, JIS-Roman, half-width katakana, KS-Roman, TCVN-Roman
EBCDIC/EBCDIK	ASCII, GB-Roman, CNS-Roman, JIS-Roman, half-width katakana, KS-Roman, TCVN-Roman
ISO-2022	ASCII, GB-Roman, CNS-Roman, JIS-Roman, half-width katakana, KS-Roman, TCVN-Roman,[a] GB 2312-80, GB/T 12345-90,[a] CNS 11643-2007, JIS X 0208:1997, JIS X 0212-1990, JIS X 0213:2004, KS X 1001:2004, KS X 1002:2001,[a] KPS 9566-97, TCVN 5773:1993,[a] TCVN 6056:1995[a]
EUC	ASCII, GB-Roman, CNS-Roman, JIS-Roman, half-width katakana, KS-Roman, TCVN-Roman, GB 2312-80, GB/T 12345-90, CNS 11643-2007, JIS X 0208:1997, JIS X 0212-1990, JIS X 0213:2004, KS X 1001:2004, KS X 1002:2001, KPS 9566-97, TCVN 5773:1993, TCVN 6056:1995
GBK	ASCII, GB-Roman, GB 2312-80, GB/T 12345-90,[b] GBK
GB 18030	ASCII, GB-Roman, GB 2312-80, GB/T 12345-90,[b] GBK, GB 18030-2005, Unicode
Big Five	ASCII, CNS-Roman, Big Five, Hong Kong GCCS, Hong Kong SCS-2008
Big Five Plus	ASCII, CNS-Roman, Big Five, Big Five Plus
Shift-JIS	ASCII, JIS-Roman, half-width katakana, JIS X 0208:1997
Johab	ASCII, KS-Roman, KS X 1001:2004

a. No escape sequence has been registered for GB/T 12345-90 (and several other character set standards), but its design fits nicely into the ISO-2022 model.

b. All the characters specific to GB/T 12345-90, meaning its 2,180 traditional hanzi, are included in GBK and GB 18030 encoding, but at different code points than in EUC-CN encoding.

As I mentioned at the very beginning of this chapter, because of the important role that Unicode plays in the software development efforts of today, their encodings are covered first. For those who wish to explore other encoding possibilities related to Unicode, I encourage you to explore Wenlin Institute's page that describes UCS-G, UCS-E, and UCS-∞ encodings.*

Unicode Encoding Methods

Prior to the introduction of Unicode, typical CJKV encoding methods supported thousands or tens of thousands of code points. Unicode changed all this, somewhat by storm. The full definition of Unicode supports up to 1,112,064 code points. And, as you learned in Chapter 3, Unicode version 5.1 assigns characters to over 100,000 of these code points. These figures are certainly staggering, but they should be far from overwhelming.

* *http://wenlin.com/ucs-x.htm*

Before I describe each Unicode encoding form, I feel that it is useful to introduce some new concepts that will make these encodings forms easier to understand. The sections that follow draw attention to special characters or properties of Unicode. Knowing when these special characters should be used, and understanding the properties, will help to guide your way through the rest of this chapter.

Special Unicode Characters

Before we dive into full descriptions and explanations of the various Unicode encoding forms, I first want to draw your attention to five special characters in Unicode that are worth mentioning in the context of this book, all of which are listed in Table 4-3, along with their representations in the Unicode encoding forms that are covered in this chapter.

Table 4-3. Special Unicode characters

Character name	Unicode	UTF-32BE	UTF-32LE	UTF-16BE	UTF-16LE	UTF-8
Ideographic Space	U+3000	00 00 30 00	00 30 00 00	30 00	00 30	E3 80 80
Geta Mark	U+3013	00 00 30 13	13 30 00 00	30 13	13 30	E3 80 93
Ideographic Variation Indicator	U+303E	00 00 30 3E	3E 30 00 00	30 3E	3E 30	E3 80 BE
Byte Order Mark (BOM)	U+FEFF	00 00 FE FF	FF FE 00 00	FE FF	FF FE	EF BB BF
Replacement Character	U+FFFD	00 00 FF FD	FD FF 00 00	FF FD	FD FF	EF BF BD

The *Ideographic Space* (U+3000) character is special in the sense that it easily confused with the *Em Space* (U+2003). In terms of rendering, both characters will almost always appear the same, and one would be hard-pressed to distinguish them visually. The difference is one of context and semantics. CJKV character sets typically include what is referred to as a full-width space. CJKV character sets also include ideographs, and the full-width space is used with them to maintain a grid-like text composition. This full-width space corresponds to the Ideographic Space character. The Em Space is intended to be used in non-CJKV contexts.

The *Geta Mark* (〓; U+3013; see JIS X 0208:1997 02-14) may be used to indicate an ideograph that is not available in the selected font for proper rendering. "Geta" (下駄 *geta*) is a Japanese word that refers to a type of wooden shoe whose sole appears as two heavy horizontal bars, and the Geta Mark character looks remarkably similar.

The *Ideographic Variation Indicator* (〾; U+303E) is used to indicate that the intended ideograph is a variant form of the ideograph that immediately follows it. The use of this character simply means that the document author was not able to find an exact match for the intended ideograph. It serves as a flag to indicate that the ideograph that was really intended is not available in Unicode, but may otherwise be available in a font.

The *Byte Order Mark* (U+FEFF) has (or rather, had) multiple semantics. When it occurs as the first character in a file, its purpose is to explicitly indicate the byte order (thus, its

appropriate name). When it occurs in all other contexts (that is, buried within a file, stream, buffer, or string), it is used as a *Zero-Width No-Break Space* (ZWNBSP). However, its use as a ZWNBSP is deprecated, and *Word Joiner* (U+2060) should be used instead. The BOM is necessary only for the UTF-16 and UTF-32 encoding forms, but it is also useful for the UTF-8 encoding form in that it serves as an indicator that the data is intended to be Unicode, and not ASCII or a legacy encoding method that happens to include ASCII as a subset.

The *Replacement Character* (�; U+FFFD) is generically used for characters that cannot be represented in Unicode, or for representing invalid or illegal input, such as invalid UTF-8 byte sequences, unpaired UTF-16 High or Low Surrogates, and so on.

Some of these special characters will be referred to throughout this chapter, and perhaps elsewhere in this book. I know from personal experience that being aware of these characters and their proper usage is extremely helpful.

Unicode Scalar Values

When Unicode characters are referenced outside the context of a specific encoding form, Unicode scalar values are used. This is a notation that serves to explicitly identify Unicode characters and unambiguously distinguishes them from other characters, and from each other. This notation also supports sequences of Unicode code points.

Unicode scalar values are represented by "U+" followed by four, five, or six hexadecimal digits. Zero-padding is used for values that would otherwise be represented by one to three hexadecimal digits. The very first and last characters in Unicode are thus represented as U+0000 and U+10FFFF, respectively. Sequences are separated by a comma and enclosed in less-than and greater-than symbols. For example, the registered *Ideographic Variation Sequence* (IVS) for the glyph 劍 (Adobe-Japan1 CID+14106) is represented by <U+528D, U+E0101> as a Unicode sequence.

Byte Order Issues

There are two classes of computer architectures in terms of the order in which data is interpreted beyond the single-byte level: little- and big-endian. At one point, machines that ran the DOS, Windows, and VMS Vax OSes used processors that interpret data in little-endian byte order, and those that ran Mac OS, Mac OS X, and Unix OSes used processors that interpret data in big-endian byte order. Thanks to the latest Mac OS X machines that use Intel processors, and Linux systems that run on machines that also use Intel processors, the byte order distinction based on the OS has effectively become blurred. I suggest that you refer to the section entitled "What Are Little- and Big-Endian?" in Chapter 1.

As long as you are dealing with encodings that are byte-driven, such as Shift-JIS encoding for Japanese that is covered later in this chapter or the UTF-8 encoding form that is covered in this section, byte order makes absolutely no difference. The data is simply treated

as an ordered sequence of bytes. For encoding forms that make use of code units that leap beyond the single byte, such as UTF-16 and UTF-32, byte order is absolutely critical.

The Unicode encodings that are affected by this byte order issue can (and should) make use of the important BOM, covered in the previous section, in order to explicitly indicate the byte order of the file. Incorrect interpretation of byte order can lead to some fairly "amusing" results. For example, consider U+4E00, the very first ideograph in Unicode's *CJK Unified Ideographs URO*. Its big-endian byte order is <4E 00>, and if it were to be reversed to become <00 4E>, it would be treated as though it were the uppercase Latin character "N" (U+004E or ASCII 0x4E). This is not what I would consider to be desired behavior….

BMP Versus Non-BMP

Unicode was originally designed to be represented using only 16-bit code units. This is equivalent to and fully contained in what is now referred to as Plane 0 or the BMP. The now obsolete and deprecated UCS-2 encoding represented a pure 16-bit form of Unicode. Unicode version 2.0 introduced the UTF-16 encoding form, which is UCS-2 encoding with an extension mechanism that allows 16 additional planes of 65,536 code points each to be encoded. The code points for these 16 additional planes, because they are outside or beyond the BMP, are considered to be non-BMP code points.

Some early implementors of Unicode built a 16-bit environment in order to support UCS-2 encoding. When Unicode version 3.1 introduced the very first non-BMP characters, the fate of UCS-2 encoding was effectively sealed. Although non-BMP characters became theoretically possible as of Unicode version 2.0, non-BMP characters instantly became a reality as of version 3.1. Today, there are tens of thousands of characters in Planes 1, 2, and 14, all of which are outside the BMP.

Readers of this book simply need to know one thing: one cannot claim to support Unicode if non-BMP characters are not supported. There are no excuses. There are only solutions.

Unicode Encoding Forms

UTF is an abbreviation for *Unicode* (or *UCS*[*]) *Transformation Format*, and refers to a series of well-established encoding forms developed for Unicode and ISO 10646. There are three basic UTF encoding forms, designated UTF-8, UTF-16, and UTF-32. The decimal portion of their designators refer to the smallest code unit implemented by each encoding form, in terms of number of bits. The UTF-8 encoding form, for example, uses eight-bit code units, meaning bytes. The UTF-16 and UTF-32 encoding forms thus use 16- and 32-bit code units, respectively.

Table 4-4 provides an overview of the UTF encoding forms, indicating their important and noteworthy characteristics. Please take some time to study this table, and consider

[*] UCS stands for *Universal Character Set*.

marking the page, because this table can serve as a useful guide or overview of these all-important encoding forms.

Table 4-4. Unicode encoding form overview

Characteristic	UTF-8	UTF-16	UTF-16BE	UTF-16LE	UTF-32	UTF-32BE	UTF-32LE
Code unit bits	8	16	16	16	32	32	32
Byte order	n/a	<BOM>	BE	LE	<BOM>	BE	LE
Bytes (BMP)	1–3	2	2	2	4	4	4
Bytes (>BMP)	4	4	4	4	4	4	4
U+0000	00	0000	00 00	00 00	00000000	00 00 00 00	00 00 00 00
U+10FFFF	F4 8F BF BF	DBFF DFFF	DB FF DF FF	FF DB FF DF	0010FFFF	00 10 FF FF	FF FF 10 00

The U+0000 and U+10FFFF code points in Table 4-4, which represent the first and last code points of Unicode, are presented in big-endian byte order for the UTF-16 and UTF-32 columns, despite the fact that these encoding forms require the BOM to be present.* These encoding forms are to be interpreted in big-endian byte order if the BOM is missing or is otherwise not present, which effectively means that big-endian is their default byte order. Also note that regardless of which encoding form is used, non-BMP characters require four effective bytes for representation.

One of the most significant characteristics of the UTF encoding forms is that all of them encode the same set of 1,112,064 code points. In other words, they encode 17 planes of 65,536 code points each, whereby the 2,048 Surrogates in Plane 0 (the BMP) are excluded from this figure because they serve only as the extension mechanism for the UTF-16 encoding form. This is important, because these encodings are 100% interoperable. This is why these UTF encoding forms have become so successful. Interoperability is everything.

The UTF-32 encoding form

I consider UTF-32 to be the most fundamental of the Unicode encoding forms due to its full-form nature, and due to the fact that its form (at least, in big-endian byte order) appears to be an eight-digit zero-extended Unicode scalar value without the tell-tale "U+" prefix. For example, U+20000, the very first ideograph in *CJK Unified Ideographs Extension B*, is represented as <00 02 00 00> according to the UTF-32 encoding form in big-endian byte order. For this reason, describing the UTF-32 encoding form first seems to be both practical and appropriate.

UTF-32 is a pure subset of the original (and now obsolete and deprecated) UCS-4 encoding, and specifically supports only the BMP and 16 additional planes, 1 through 16. For

* Interestingly, if the BOM is missing, UTF-16 and UTF-32 encodings are assumed to use big-endian byte order.

historical reasons, UCS-4 encoding is described later in this chapter. Because it is useful to draw a comparison between UCS-4 and UTF-32 encodings, specifically that the UTF-32 encoding form is a pure subset of UCS-4 encoding, Table 4-5 provides the specifications for both encoding forms.

Table 4-5. UCS-4 and UTF-32 encoding form specifications

Encoding form	Decimal	Hexadecimal
UCS-4		
First byte range	0–127	00–7F
Second byte range	0–255	00–FF
Third byte range	0–255	00–FF
Fourth byte range	0–255	00–FF
UTF-32		
First byte	0	00
Second byte range	0–16	00–10
Third byte range	0–255	00–FF
Fourth byte range	0–255	00–FF

UTF-32 is a fixed-length encoding form that uses 32-bit code units to represent any Unicode character. Every character thus uses the same amount of storage space, specifically 32 bits. To some extent, the UTF-32 encoding form may seem wasteful in that the last Unicode character, U+10FFFF, requires only 21 of the 32 bits to represent it. This character's binary bit array is as follows:

```
00000000000010000111111111111111111
```

Note how only 21 bits are used, and that it is zero-extended to 32 bits. Given that 21-bit—or even 24-bit, if we round up to an effective 3-byte representation—code units are not directly supported on today's computers, and the likelihood of this happening is relatively low, allocating 32 bits to support a 21-bit encoding is quite appropriate.

To start our exploration of the UTF encoding forms, beginning with the UTF-32 encoding form, I will use the string 吉野家* as the example, which is composed of three ideographs, the first of which is non-BMP, specifically encoded in Plane 2, and more specifically in the CJK Unified Ideographs Extension B block. The BMP versus non-BMP distinction is useful, especially as this example is expanded to cover the other Unicode encoding forms, covered later in this chapter. Table 4-6 lists the Unicode scalar values for this example string, along with their UTF-32 representations in big- and little-endian byte order.

* These three ideographs are pronounced *yoshinoya* in Japanese, and represent the name of a famous Japanese noodle soup restaurant chain. Its first kanji is somewhat rare, and not found in any current JIS standard. The JIS X 0208:1997 kanji 吉 (U+5409) is usually used instead of this somewhat rare variant form.

Table 4-6. UTF encoding form example—UTF-32 encoding form

Encoding form	String		
Characters	吉	野	家
Scalar value	U+20BB7	U+91CE	U+5BB6
UTF-32BE	00 02 0B B7	00 00 91 CE	00 00 5B B6
UTF-32LE	B7 0B 02 00	CE 91 00 00	B6 5B 00 00

When the UTF-32 encoding form is used or otherwise specified, a BOM is necessary to indicate byte order. There are two variations of the UTF-32 encoding form that do not require a BOM because they explicitly indicate the byte order. The UTF-32LE encoding form is equivalent to the UTF-32 encoding form in little-endian byte order. Likewise, the UTF-32BE encoding form is equivalent to the UTF-32 encoding form in big-endian byte order. According to Unicode, if the UTF-32 encoding form is specified, and if no BOM is present, its byte order is to be interpreted as big-endian.

The UTF-16 encoding form

The UTF-16 encoding form is an extension of the original (and now obsolete and deprecated) UCS-2 encoding, described elsewhere in this chapter. The extension mechanism that is used by the UTF-16 encoding form effectively makes it a hybrid encoding, because a portion of its encoding definition is fixed-length, and its extension mechanism is non-modal in nature.

The UTF-16 encoding form was originally defined in Appendix C.3 of *The Unicode Standard, Version 2.0* (Addison-Wesley, 1996), and in Amendment 1 of ISO 10646-1:1993.* Unicode version 2.0 added the Surrogates that allows for expansion beyond the 16-bit code space of the original UCS-2 encoding form. In essence, UTF-16 continues to encode the BMP according to UCS-2 encoding, but also allows the 16 additional planes, which were previously accessible through UCS-4 encoding, to be encoded. You get the compactness of UCS-2 encoding for the most widely used characters, but also have access to the most useful subset of UCS-4 encoding.

The UTF-16 encoding form, which is fundamentally quite simple in design and implementation, works as follows:

- Two blocks of 1,024 code points, referred to as High and Low Surrogates (this is Unicode terminology; ISO terminology is different), are set aside in the BMP (in other words, in Plane 0).
- Combinations of a single High Surrogate followed by a single Low Surrogate, which are referred to as Surrogate Pairs, are used to address 2^{20} (1024×1024 or 1,048,576) additional code points outside the BMP, which correspond to the code points for

* Originally referred to as *UCS-2E* or *Extended UCS-2 encoding*.

Unicode Planes 1 (U+10000 through U+1FFFF) through 16 (U+100000 through U+10FFFF).

In other words, Plane 0 (BMP) characters are represented by a single 16-bit code unit, and Plane 1 through 16 characters are represented by two 16-bit code units. As of Unicode version 5.1, there are characters assigned to only Planes 0, 1, 2, and 14. Unicode allocates 917,504 additional non-PUA code points beyond the BMP, specifically in Planes 1 through 14, and 131,072 additional PUA code points, specifically in Planes 15 and 16.

Software that processes the UTF-16 encoding form internally must deal with issues similar to those of legacy encoding methods that use variable-length encodings. However, UTF-16 should not be problematic, because Unicode is a single character set, and Surrogates do not overlap at all, so it is always clear what you're dealing with. Table 4-7 provides the specifications for the UTF-16 encoding form.

Table 4-7. UTF-16 encoding form specifications

	Encoding	Decimal	Hexadecimal
BMP	First byte range	0–215, 224–255	00–D7, E0–FF
	Second byte range	0–255	00–FF
Non-BMP	**High Surrogates**		
	First byte range	216–219	D8–DB
	Second byte range	0–255	00–FF
	Low Surrogates		
	First byte range	220–223	DC–DF
	Second byte range	0–255	00–FF

Interestingly, in the early days of the UTF-16 encoding form, there were varying levels of UTF-16 support. Given the extent to which UTF-16 is supported today, it seems somewhat silly to consider different levels of UTF-16 encoding form support. In any case, Table 4-8 lists the four levels of support for the UTF-16 encoding form, which were useful at some point in the past.

Table 4-8. Four levels of UTF-16 support

UTF-16 support level	Meaning
UCS-2 Only	No interpretation of Surrogate Pairs, and no pair integrity
Weak	Interpretation of Surrogate Pairs, but no pair integrity
Aware	No interpretation of Surrogate Pairs, but pair integrity
Strong	Interpretation of Surrogate Pairs, and pair integrity

Preserving pair integrity simply means that Surrogate Pairs are treated as a single unit when it comes to processing operations, such as deleting or inserting characters.

Interpreting Surrogate Pairs simply means that Surrogate Pairs are treated as UCS-4 (now UTF-32) characters. Of course, what is expected of today's software is the "Strong" UTF-16 support level.

Let's continue our exploration of the UTF encodings, adding UTF-16 to the representations for the 吉野家* example string. The BMP versus non-BMP distinction is especially useful for UTF-16 encoding, because Surrogates are necessary and used for non-BMP characters. Table 4-9 lists the Unicode scalar values for this string, along with their UTF-32 and UTF-16 representations in big- and little-endian byte order.

Table 4-9. UTF encoding form example—UTF-32 and UTF-16 encoding forms

Encoding form	String		
Characters	吉	野	家
Scalar value	U+20BB7	U+91CE	U+5BB6
UTF-32BE	00 02 0B B7	00 00 91 CE	00 00 5B B6
UTF-32LE	B7 0B 02 00	CE 91 00 00	B6 5B 00 00
UTF-16BE	D8 42 DF B7	91 CE	5B B6
UTF-16LE	42 D8 B7 DF	CE 91	B6 5B

Table 4-10 provides the complete encoding ranges for Unicode Planes 1 through 16, as encoded according to UTF-32BE and UTF-16BE encodings, the latter of which makes use of the complete Surrogates region.

Table 4-10. Mapping Unicode Planes 1 through 16 to UTF-32 and UTF-16 encoding forms

Plane	Unicode ranges	UTF-32BE encoding ranges	UTF-16BE encoding ranges
1	U+10000–U+1FFFF	00 01 00 00–00 01 FF FF	D8 00 DC 00–D8 3F DF FF
2	U+20000–U+2FFFF	00 02 00 00–00 02 FF FF	D8 40 DC 00–D8 7F DF FF
3	U+30000–U+3FFFF	00 03 00 00–00 03 FF FF	D8 80 DC 00–D8 BF DF FF
4	U+40000–U+4FFFF	00 04 00 00–00 04 FF FF	D8 C0 DC 00–D8 FF DF FF
5	U+50000–U+5FFFF	00 05 00 00–00 05 FF FF	D9 00 DC 00–D9 3F DF FF
6	U+60000–U+6FFFF	00 06 00 00–00 06 FF FF	D9 40 DC 00–D9 7F DF FF
7	U+70000–U+7FFFF	00 07 00 00–00 07 FF FF	D9 80 DC 00–D9 BF DF FF
8	U+80000–U+8FFFF	00 08 00 00–00 08 FF FF	D9 C0 DC 00–D9 FF DF FF
9	U+90000–U+9FFFF	00 09 00 00–00 09 FF FF	DA 00 DC 00–DA 3F DF FF
10	U+A0000–U+AFFFF	00 0A 00 00–00 0A FF FF	DA 40 DC 00–DA 7F DF FF
11	U+B0000–U+BFFFF	00 0B 00 00–00 0B FF FF	DA 80 DC 00–DA BF DF FF

* *http://www.yoshinoya.com/*

Table 4-10. Mapping Unicode Planes 1 through 16 to UTF-32 and UTF-16 encoding forms

Plane	Unicode ranges	UTF-32BE encoding ranges	UTF-16BE encoding ranges
12	U+C0000–U+CFFFF	00 0C 00 00–00 0C FF FF	DA C0 DC 00–DA FF DF FF
13	U+D0000–U+DFFFF	00 0D 00 00–00 0D FF FF	DB 00 DC 00–DB 3F DF FF
14	U+E0000–U+EFFFF	00 0E 00 00–00 0E FF FF	DB 40 DC 00–DB 7F DF FF
15	U+F0000–U+FFFFF	00 0F 00 00–00 0F FF FF	DB 80 DC 00–DB BF DF FF
16	U+100000–U+10FFFF	00 10 00 00–00 10 FF FF	DB C0 DC 00–DB FF DF FF

Likewise with UTF-32, when the UTF-16 encoding form is used or otherwise specified, a BOM is necessary to indicate byte order and is assumed to use big-endian byte order if the BOM is not present. There are two variations of the UTF-16 encoding form that do not require a BOM because they explicitly indicate the byte order. The UTF-16LE encoding form is equivalent to the UTF-16 encoding form in little-endian byte order. Likewise, the UTF-16BE encoding form is equivalent to the UTF-16 encoding form in big-endian byte order. According to Unicode, if the UTF-16 encoding form is specified, and if no BOM is present, its byte order is to be interpreted as big-endian.

The UTF-8 encoding form

The UTF-8 encoding form was developed as a way to represent Unicode text as a stream of 1 or more 8-bit code units (also known as bytes or octets) rather than as larger 16- or 32-bit code units.* In addition, it was also a goal that a stream of bytes did not contain zeros (`0x00`) so that they could be handled by standard C string APIs, which terminate strings with `0x00`. UTF-8 is therefore an eight-bit, variable-length, nonmodal encoding form. The UTF-8 encoding form represents Unicode characters through the use of a mixed one- through four-byte encoding. BMP characters are represented using one, two, or three bytes. Non-BMP characters are represented using four bytes. UTF-8 is especially suitable for transferring text between different systems because it avoids all byte order issues. The UTF-8 encoding form is fully described and defined in François Yergeau's RFC 3629 (obsoletes RFC 2279), *UTF-8, a transformation format of ISO 10646*.†

To best understand the UTF-8 encoding form, we need to explore its origins and its full definition. It was originally designed to interoperate with the now obsolete and deprecated UCS-2 and UCS-4 encodings, which were pure fixed-length 16- and 32-bit encodings, respectively. Table 4-11 lists the six different UTF-8 representations as per-code unit bit arrays, along with the corresponding UCS-2 and UCS-4 encoding ranges in big-endian byte order.

* UTF-8 was once known as UTF-2, FSS-UTF (*File System Safe UTF*), or UTF-FSS.

† *http://www.ietf.org/rfc/rfc3629.txt*

Table 4-11. UCS-2/UCS-4 encoding ranges and UTF-8 bit arrays

Encoding	Encoding range	UTF-8 bit arrays
	00 00–00 7F	0xxxxxxx
UCS-2	00 80–07 FF	110xxxxx 10xxxxxx
	08 00–FF FF	1110xxxx 10xxxxxx 10xxxxxx
	00 01 00 00–00 1F FF FF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
UCS-4	00 20 00 00–03 FF FF FF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
	04 00 00 00–7F FF FF FF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

For all but the ASCII-compatible range (U+0000 through U+007F), the number of first-byte high-order bits set to 1 indicates the number of bytes for the character's byte sequence. This sequence, of two or more bits set to 1, is followed by exactly one bit set to 0. For example, 11110xxx indicates that the byte length must be 4 because there are four initial 1s. The second and subsequent bytes all use 10 as their high-order bits.

In order to make the encoding ranges a bit clearer to grasp, Table 4-12 lists the specific encoding ranges that correspond to the bit arrays listed in Table 4-11. All 2,147,483,648 UCS-4 code points can be represented using this full definition of UTF-8 encoding; note how each encoding range excludes a range that can encode all encoding ranges with less number of bytes—table footnotes explicitly indicate the excluded encoding ranges. Note how the second and subsequent bytes share the same encoding range, specifically 0x80 through 0xBF.

Table 4-12. UTF-8 encoding form specifications—full definition

Encoding range	Decimal	Hexadecimal
00 00–00 7F (128 code points)		
Byte range	0–127	00–7F
00 80–07 FF (1,920 code points)[a]		
First byte range	192–223	C0–DF
Second byte range	128–191	80–BF
08 00–FF FF (63,488 code points)[b]		
First byte range	224–239	E0–EF
Second and third byte range	128–191	80–BF
00 01 00 00–00 1F FF FF (2,031,616 code points)[c]		
First byte range	240–247	F0–F7
Second through fourth byte range	128–191	80–BF
00 20 00 00–03 FF FF FF (65,011,712 code points)[d]		
First byte range	248–251	F8–FB
Second through fifth byte range	128–191	80–BF

Table 4-12. UTF-8 encoding form specifications—full definition

Encoding range	Decimal	Hexadecimal
04 00 00 00–7F FF FF FF (2,080,374,784 code points)[e]		
First byte range	252–253	FC–FD
Second through sixth byte range	128–191	80–BF

a. <C0 80> through <C1 BF> (128 code points) are not available and are invalid, because the one-byte range handles them.

b. <E0 80 80> through <E0 9F BF> (2,048 code points) are not available and are invalid, because the one- and two-byte ranges handle them.

c. <F0 80 80 80> through <F0 8F BF BF> (65,536 code points) are not available and are invalid, because the one- through three-byte ranges handle them.

d. <F8 80 80 80 80> through <F8 87 80 80 80> (2,097,152 code points) are not available and are invalid, because the one- through four-byte ranges handle them.

e. <FC 80 80 80 80 80> through <FC 83 80 80 80 80> (67,108,864 code points) are not available and are invalid, because the one- through five-byte ranges handle them.

Table 4-13 details the Unicode definition of the UTF-8 encoding form, which is a small subset of its original full definition. Because the Unicode definition of the UTF-8 encoding form is designed to interoperate with all Unicode code points, Unicode scalar values are used to represent the Unicode ranges.

Table 4-13. UTF-8 encoding form specifications—Unicode definition

Encoding range	Decimal	Hexadecimal
U+0000–U+007F (128 code points)		
Byte range	0–127	00–7F
U+0080–U+07FF (1,920 code points)[a]		
First byte range	192–223	C0–DF
Second byte range	128–191	80–BF
U+0800–U+FFFF (63,488 code points)[b]		
First byte range	224–239	E0–EF
Second and third byte range	128–191	80–BF
U+10000–U+10FFFF (1,048,576 code points)[c]		
First byte range	240–244	F0–F4
Second through fourth byte range	128–191	80–BF

a. <C0 80> through <C1 BF> (128 code points) are not available and are invalid, because the one-byte range handles them.

b. <E0 80 80> through <E0 9F BF> (2,048 code points) are not available and are invalid, because the one- and two-byte ranges handle them.

c. <F0 80 80 80> through <F0 8F BF BF> (65,536 code points) are not available and are invalid, because the one- through three-byte ranges handle them. Also, when the first byte is 0xF4, the highest possible value for the second byte is 0x8F, not 0xBF. Thus, U+10FFFF, which is the highest possible code point, is represented as <F4 8F BF BF>.

When applying the UTF-8 encoding form to UTF-16–encoded text, special care must be taken when dealing with the 2,048 Surrogate code points that form the 1,048,576

Surrogate Pairs. When blindly converting this range (U+D800 through U+DFFF) to UTF-8 encoding, it becomes <ED A0 80> through <ED BF BF>. The correct method for dealing with a UTF-16 Surrogate Pair when converting to the UTF-8 encoding form is to first convert them into their corresponding UTF-32 code points, and then to convert the resulting UTF-32 code point into UTF-8 encoding, which always results in a four-byte sequence.

There are many attractive and genuinely useful characteristics of the UTF-8 encoding form, such as the following:

- The UTF-8 encoding form does not break any of the legacy C string-manipulation APIs that expect strings to be terminated by a null byte, and additionally preserves the meaning of 0x2F (slash), which serves as the Unix path separator.
- Any valid ASCII-encoded string is a valid UTF-8–encoded string—that is what I would call backward compatibility! However, because many legacy encodings can also make the same claim, this introduces ambiguity with regard to distinguishing UTF-8 encoding from the many legacy encodings that share this property.
- It is possible to probe anywhere within a UTF-8 file and still be able to determine the location of character boundaries.
- UTF-8 encoding, by definition, can have absolutely no instances of the bytes 0xFE or 0xFF, so that there is no risk of confusion with the BOM as used in the UTF-16 and UTF-32 encoding forms. A BOM, encoded according to the UTF-8 encoding form, specifically <EF BB BF>, is allowed in UTF-8, though not required. The next section provides examples for when doing so results in real-world benefits.

Our exploration of the UTF encoding forms further continues by adding UTF-8 to the representations for the 吉野家* example string. The BMP versus non-BMP distinction is also useful for UTF-8 encoding, because four bytes are necessary only for non-BMP characters. BMP characters are represented using one, two, or three bytes. CJK Unified Ideographs in the BMP, specifically the URO and Extension A, are represented using three bytes. Table 4-14 lists the Unicode scalar values for this string, along with their UTF-32, UTF-16, and UTF-8 representations in big- and little-endian byte order, as appropriate.

Table 4-14. UTF encoding form example—UTF-32, UTF-16, and UTF-8 encoding forms

Encoding form	String		
Characters	吉	野	家
Scalar value	U+20BB7	U+91CE	U+5BB6
UTF-32BE	00 02 0B B7	00 00 91 CE	00 00 5B B6
UTF-32LE	B7 0B 02 00	CE 91 00 00	B6 5B 00 00

* *http://www.yoshinoyausa.com/*

Table 4-14. UTF encoding form example—UTF-32, UTF-16, and UTF-8 encoding forms

Encoding form	String		
UTF-16BE	D8 42 DF B7	91 CE	5B B6
UTF-16LE	42 D8 B7 DF	CE 91	B6 5B
UTF-8	F0 A0 AE B7	E9 87 8E	E5 AE B6

UTF-8 is the default encoding form for XML, and the most common encoding for today's web pages. Its backward compatibility with ASCII clearly made the UTF-8 encoding form very attractive and appealing for this purpose. The Java programming language uses Unicode internally—initially with UCS-2 encoding, and now with the UTF-16 encoding form—through the use of type *char*. But, it also provides support for the UTF-8 encoding form for data import and export purposes. As an historical example, Plan 9, which is briefly described in Chapter 10, is an experimental OS that uses the UTF-8 encoding form to represent Unicode text.

The UTF-8 encoding form and the BOM

Although the BOM (U+FEFF) is not necessary for UTF-8–encoded text, because its eight-bit code units are not affected by big- and little-endian byte order issues, using it at the beginning of files, streams, buffers, or strings is useful in that doing so explicitly identifies the encoding as UTF-8 and distinguishes it from other encodings that may share some of UTF-8 encoding's attributes, such as its ASCII subset.

The BOM is represented as <EF BB BF> in UTF-8 encoding. Considering the ASCII subset of UTF-8 encoding, and that a large number of legacy encodings use the same ASCII subset, the possibility of confusion seems relatively high, in terms of identifying the encoding of arbitrary data. Naturally, including the BOM in UTF-8–encoded data eliminates any confusion and unambiguously identifies the encoding form as UTF-8.

This means that software developers should write software that allows UTF-8 data to include the BOM, but the same software should not mandate it. Whether your software includes the BOM in UTF-8 data that it generates is a separate consideration. Doing so seems to be a prudent measure in contexts or situations in which ambiguity may exist. But, keep in mind that some people feel that including a UTF-8–encoded BOM is a nuisance at best, and very undesireable at worst.

UTF encoding form interoperability

As mentioned earlier in this chapter, all UTF encodings can represent the same set of 1,112,064 code points. While this figure appears to be staggering, and it is, there is 100% interoperability between these encodings.

The only special handling that is necessary is for the UTF-16 Surrogates. I have found that the best way to handle UTF-16 Surrogates is to map each Surrogate Pair to its corresponding UTF-32 code point. If the destination encoding is UTF-32, then the work is

done. If the destination encoding is UTF-8, then the well-established UTF-32 to UTF-8 conversion algorithm is used.

Numeric Character References—human-readable Unicode encoding

Of the Unicode encoding forms, only UTF-32 can be considered somewhat human-readable in that its big-endian instance is simply a zero-padded, eight-digit version of a Unicode *scalar value*. Unicode scalar values are what are typically published in reference materials that support Unicode, such as dictionaries and related publications. The use of little-endian byte order naturally disturbs this human-readability property.

HTML and XML support what are known as Numeric Character References (NCRs), which is a special ASCII-based syntax that is used to reliably store and display any Unicode character using a human-readable form that is directly based on Unicode scalar values. NCRs have their heritage from SGML, from which HTML and XML were derived.

An NCR that specifies a Unicode character is made up of a prefix and a suffix that surrounds a four-, five-, or six-digit hexadecimal value that is reminiscent of a Unicode scalar value without its "U+" prefix. Decimal values can be used, but that detracts from one of the primary advantages of NCRs, specifically their human-readability. The prefix is &#x (<U+0026, U+0023, U+0078>), and the suffix is ; (U+003B). If decimal values are used, the prefix becomes the one-character-shorter &# (<U+0026, U+0023>) form. Zero-padding is optional, for both decimal and hexadecimal notations, but in my experience, zero-extending to at least four hexadecimal digits improves human-readability.

We finish our exploration of the Unicode encodings by adding NCRs to the representations for the 吉野家* example string. Table 4-15 lists the Unicode scalar values for this string, along with the values' UTF-32, UTF-16, and UTF-8 representations in big- and little-endian byte order, as appropriate, and their NCRs.

Table 4-15. UTF encoding example—UTF-32, UTF-16, and UTF-8 encoding forms plus NCRs

Encoding form	String		
Characters	吉	野	家
Scalar value	U+20BB7	U+91CE	U+5BB6
UTF-32BE	00 02 0B B7	00 00 91 CE	00 00 5B B6
UTF-32LE	B7 0B 02 00	CE 91 00 00	B6 5B 00 00
UTF-16BE	D8 42 DF B7	91 CE	5B B6
UTF-16LE	42 D8 B7 DF	CE 91	B6 5B
UTF-8	F0 A0 AE B7	E9 87 8E	E5 AE B6
NCR	&#x20BB7;	&#x91CE;	&#x5BB6;

* *http://en.wikipedia.org/wiki/Yoshinoya*

NCRs are useful for environments that ultimately produce XML or HTML files. The characters represented by NCRs are explicitly identified as Unicode characters, yet benefit from the strength and persistence made possible by an ASCII-based notation. NCRs are useful to represent hard-to-input characters, or characters whose underlying representation may be ambiguous when displayed in binary form. Take for instances the two Unicode characters "A" (U+0041) and "Ａ" (U+FF21). Both represent the same uppercase Latin letter; however, the former is proportional, but sometimes implemented using half-width metrics, and the latter is explicitly full-width. When it is critical to unambiguously use one character over another, in the context of web documents, using `&#xFF21;` to represent the full-width uppercase Latin letter "A" has great value. Interestingly, differentiating Latin "A" (U+0041), Cyrillic "А" (U+0410), and Greek "Α" (U+0391) is an even greater feat, and NCRs become especially useful when differentiating such characters becomes critical.

NCRs do, however, require up to 10 bytes for representation, so their advantages in the contexts in which they can be used come at the expense of increased string size.

There are other NCR-like notations, often referred to as *escaping*, whose use should be pointed out or described, such the `\u`*`XXXX`* notation as used by C/C++, Java, and JavaScript for specifying Unicode characters, the use of `\`*`XXXX`* for CSS, and the use of `\x{`*`XXXX`*`}` to do the same using Perl. These programming language notations also make use of Unicode scalar values, meaning that their use can improve the readability of source code that uses them. NCRs, however, are directly related to the Web, and as already stated, are immediately usable in HTML and XML files. Chapter 12 will revisit the use of NCRs in the context of authoring web documents. Richard Ishida's "Unicode Code Converter" is an excellent way to explore NCRs and related notations.*

Obsolete and Deprecated Unicode Encoding Forms

Three of the original Unicode encodings, specifically UTF-7, UCS-2, and UCS-4, are now obsolete, and their use is considered deprecated. These encodings became obsolete for a variety of reasons. UTF-7 and UCS-2 encodings became obsolete because they were closely tied to BMP-only implementations. UTF-7 was also a source of a lot of security issues. As soon as characters were encoded outside the BMP, the usefulness of these encodings significantly diminished. As of this writing, there are characters encoded in Planes 1, 2, and 14. UCS-4 encoding became obsolete because it contains vastly more code points than in Unicode. The UTF-32 encoding form effectively took its place and is a pure subset of UCS-4 encoding in that both encodings represent the BMP and Planes 1 through 16 in the same way.

Interestingly, the UTF-8 encoding form did not become obsolete, but was instead redefined such that it became fully compatible with the UTF-16 and UTF-32 encoding forms. Its five- and six-byte representations were eliminated for the sake of Unicode compatibility. In fact, a portion of its four-byte representation was also eliminated.

* *http://rishida.net/scripts/uniview/conversion.php*

The UCS-2 and UCS-4 encoding forms

ISO 10646-1:1993 defined two basic encoding forms. The first is the 32-bit form—actually, a 31-bit form, because only 31 bits are used—that is referred to as UCS-4 encoding. The second is the 16-bit form, referred to as UCS-2 encoding. Note that the second form is identical to the initial 16-bit encoding used for Unicode, to encode all of the characters in what is now referred to as the BMP. A 16-bit representation can encode up to 65,536 code points. A complete 32-bit representation, on the other hand, can encode up to 4,294,967,296 code points.*

These encodings are fixed-length, and thus use the same number of bytes to represent each character. All 16 or 32 bits (actually, 31 bits) are used for representing characters. ASCII control characters—0x00 through 0x1B and 0x7F—whose use are otherwise forbidden in other encodings, are completely valid for encoding printable characters in the context of UCS-2 and UCS-4 encodings.† The same statement is naturally true for the UTF-16 and UTF-32 encoding forms. Table 4-16 details the encoding specifications for UCS-2 and UCS-4 encodings.

Table 4-16. UCS-2 and UCS-4 encoding form specifications

Encoding form	Decimal	Hexadecimal
UCS-2		
High byte range	0–255	00–FF
Low byte range	0–255	00–FF
UCS-4		
First byte range	0–127	00–7F
Second byte range	0–255	00–FF
Third byte range	0–255	00–FF
Fourth byte range	0–255	00–FF

Because Unicode and ISO 10646:2003 allocate their entire encoding space for characters, it makes no sense to waste space by including a figure showing either their encoding space or how it compares to that of other encoding methods.

UCS-2 encoding is severely limited in that it can represent only the characters in the BMP. Non-BMP characters simply cannot be encoded. In fact, UCS-2 encoding and the BMP portion of the UTF-16 encoding form are 100% compatible with one another and are represented in the same way. UCS-2 encoding can be referred to as BMP-only UTF-16 encoding.

* Considering that UCS-4 is really a 31-bit encoding, there are a "mere" 2,147,483,648 code points available.

† Some control characters, such as 0x09 (tab), 0x0A (newline), and 0x0D (carriage return), are commonly used in almost every textfile.

Likewise, the UTF-32 representation for Unicode characters is identical to the UCS-4 representation. This is due to the fact that the UTF-32 encoding form is a pure subset of UCS-4 encoding, covering only Planes 0 through 16.

Table 4-17 illustrates how the five characters My 河豚* are encoded according to UCS-2 and UCS-4 encodings, in big- and little-endian byte order. An example string that is different from that used for the UTF encoding form examples was selected because the first character in the UTF encoding form example string is outside the BMP, and thus cannot be represented in UCS-2 encoding.

Table 4-17. UCS-2 and UCS-4 encoding form example

Encoding form	String				
Characters	M	y		河	豚
UCS-2 big-endian	00 4D	00 79	00 20	6C B3	8C 5A
UCS-2 little-endian	4D 00	79 00	20 00	B3 6C	5A 8C
UCS-4 big-endian	00 00 00 4D	00 00 00 79	00 00 00 20	00 00 6C B3	00 00 8C 5A
UCS-4 little-endian	4D 00 00 00	79 00 00 00	20 00 00 00	B3 6C 00 00	5A 8C 00 00

It is useful to note how every character in UCS-4 encoding is represented by four bytes, and is otherwise equivalent to UCS-2 encoding, but is prefixed or suffixed with <00 00>, depending on byte order. This prefixing or suffixing of <00 00> can be referred to as "zero-extending to 32 bits."

It is important to be aware that all characters have the same encoding length whether they are encoded according to UCS-2 or UCS-4 encoding. From a computer-processing point of view, they are treated the same for certain processing operations, such as searching. Variable-length encodings pose difficulties for searching and other processing operations, but at the same time this does not make the operations impossible. Because non-BMP characters became a reality as of Unicode version 3.1, UCS-2 and UCS-4 encodings effectively became obsolete, and their use is necessarily deprecated. Now, the UTF-16 and UTF-32 encoding forms should be used in lieu of UCS-2 and UCS-4 encodings, respectively. Transitioning to the UTF-16 or UTF-32 encoding form is a trivial matter considering the compatibility between these encodings. For example, any valid UCS-2 code point—except, of course, for the 2,048 Surrogates—is a valid UTF-16 code point.

The main point for this section is to be aware of UCS-2 and UCS-4 encodings for historical purposes, but that the UTF encoding forms should be used instead. In fact, it would clearly be a mistake to implement software based on UCS-2 or UCS-4 encoding.

* *http://en.wikipedia.org/wiki/Fugu*

The UTF-7 encoding form

The UTF-7 encoding form, which is designed as a human-unreadable but mail-safe transformation of Unicode, is remarkably similar to the Base64 transformation, which is described near the end of this chapter. In fact, UTF-7 uses the same set of Base64 characters except for the "pad" character (0x3D, "="). The pad character is not necessary, because the UTF-7 encoding form does not require padding for incomplete three-byte segments. The UTF-7 encoding form is also different from Base64 transformation in that its Base64-like transformation is not applied to an entire file, stream, buffer, or string, but to specific characters only. UTF-7 encoding is described in Deborah Goldsmith and Mark Davis' RFC 2152 (obsoletes RFC 1642), *UTF-7: A Mail-Safe Transformation Format of Unicode*.*

Table 4-18 lists the 64 Base64 characters that are used by the UTF-7 encoding form, along with the ASCII characters that represent the 6-bit values to which they correspond, and their hexadecimal equivalents.

Table 4-18. The Base64 character set

Value—Decimal	Bit array	Base64 character—ASCII	Hexadecimal
0	000000	A	41
1	000001	B	42
2	000010	C	43
3	000011	D	44
4	000100	E	45
5	000101	F	46
6	000110	G	47
7	000111	H	48
8	001000	I	49
9	001001	J	4A
10	001010	K	4B
11	001011	L	4C
12	001100	M	4D
13	001101	N	4E
14	001110	O	4F
15	001111	P	50
16	010000	Q	51
17	010001	R	52
18	010010	S	53

* *http://www.ietf.org/rfc/rfc2152.txt*

Table 4-18. The Base64 character set

Value—Decimal	Bit array	Base64 character—ASCII	Hexadecimal
19	010011	T	54
20	010100	U	55
21	010101	V	56
22	010110	W	57
23	010111	X	58
24	011000	Y	59
25	011001	Z	5A
26	011010	a	61
27	011011	b	62
28	011100	c	63
29	011101	d	64
30	011110	e	65
31	011111	f	66
32	100000	g	67
33	100001	h	68
34	100010	i	69
35	100011	j	6A
36	100100	k	6B
37	100101	l	6C
38	100110	m	6D
39	100111	n	6E
40	101000	o	6F
41	101001	p	70
42	101010	q	71
43	101011	r	72
44	101100	s	73
45	101101	t	74
46	101110	u	75
47	101111	v	76
48	110000	w	77
49	110001	x	78
50	110010	y	79
51	110011	z	7A
52	110100	0	30

Table 4-18. The Base64 character set

Value—Decimal	Bit array	Base64 character—ASCII	Hexadecimal
53	110101	1	31
54	110110	2	32
55	110111	3	33
56	111000	4	34
57	111001	5	35
58	111010	6	36
59	111011	7	37
60	111100	8	38
61	111101	9	39
62	111110	+	2B
63	111111	/	2F

The actual binary representation for the values in Table 4-18 correspond to bit arrays of the same values. For example, the bit array for Value 0 is "000000" (0x00), and the bit array for Value 63 is "111111" (0x3F).

Because UTF-7 does not blindly apply Base64 transformation to an entire file, stream, buffer, or string, there are some characters—which form a subset of the ASCII character set—that represent themselves according to ASCII encoding. Table 4-19 lists the characters that can be directly encoded according to ASCII encoding.

Table 4-19. UTF-7 directly encoded characters

Characters	UCS-2/UTF-16BE encoding form	UTF-7 encoding form
'	00 27	27
(	00 28	28
)	00 29	29
,	00 2C	2C
-	00 2D	2D
.	00 2E	2E
/	00 2F	2F
0–9	00 30–00 39	30–39
:	00 3A	3A
?	00 3F	3F
A–Z	00 41–00 5A	41–5A
a–z	00 61–00 7A	61–7A

Table 4-20 lists the optional directly encoded characters of the UTF-7 encoding form. They are optional because in some contexts these characters may not be reliably transmitted. It is up to the client whether they should be directly encoded or transformed according to Base64.

Table 4-20. UTF-7 optional directly encoded characters

Character	UCS-2/UTF-16BE encoding form	UTF-7 encoding form
!	00 21	21
"	00 22	22
#	00 23	23
$	00 24	24
%	00 25	25
&	00 26	26
*	00 2A	2A
;	00 3B	3B
<	00 3C	3C
=	00 3D	3D
>	00 3E	3E
@	00 40	40
[	00 5B	5B
]	00 5D	5D
^	00 5E	5E
_	00 5F	5F
`	00 60	60
{	00 7B	7B
|	00 7C	7C
}	00 7D	7D

The "space" character, U+0020 (ASCII 0x20), may also be directly encoded. However, the four characters "+" (0x2B), "=" (0x3D), "\" (0x5C), and "~" (0x7E) are explicitly excluded from these sets of directly encoded characters, optional or otherwise.

Character strings that require Base64 transformation according to UTF-7 encoding begin with a "plus" character (+, 0x2B), which functions as an "escape" or "shift" to signify the start of a Base64 sequence. What follows are the Base64 characters themselves, and continue until a character not in the Base64 character set is encountered, including line-termination characters. A "hyphen" (-, 0x2D) can be used to explicitly terminate a Base64 sequence. Table 4-21 provides an example of UTF-7–encoded text, with UCS-2 encoding, in big-endian byte order. This is provided for reference and for demonstrating how the bit arrays are segmented into six-bit segments.

Table 4-21. UTF-7 encoding example

Encoding form	String
Characters	河豚
UCS-2	6C B3 8C 5A
UCS-2 bit arrays	01101100 10110011 10001100 01011010
Six-bit segments	011011 001011 001110 001100 010110 10
UTF-7	2B 62 4C 4F 4D 77 67 2D
UTF-7—visual	+bLOMwg-

Because UTF-7 is a seven-bit, byte-based encoding whose code units are seven-bit bytes, there is no need to explicitly indicate byte order. This means that the BOM can effectively be removed when converting to UTF-7 encoding. Of course, the BOM should be reintroduced when converting to the UCS-2, UCS-4, UTF-16, or UTF-32 encoding forms.

UTF-7 encoding, by definition, interoperates only with UCS-2 and UTF-16 encodings, and is applied in big-endian byte order. UTF-7 encoding can obviously be converted to UTF-8, UCS-4, or UTF-32 encoding, given that their encoding spaces are a superset of that of UTF-7 encoding. This means that if UTF-7–encoded data is encountered, it can be converted to a currently supported Unicode encoding form. The UTF-16 Surrogates are treated as though they were UCS-2–encoded, meaning no special treatment.

Also note how this example necessarily spans character boundaries. This is because Base64 transformation spans byte boundaries due to the splitting of multiple bytes into six-bit segments. UTF-7 clearly makes it challenging to perform many common text-processing tasks, such as character deletion or insertion. Converting to another Unicode encoding form prior to performing such text-processing tasks is obviously the prudent thing to do.

Given everything written in this section, it is easy to understand why UTF-7 encoding became obsolete and its use deprecated. The fact that it is limited to what is now referred to as the BMP contributed to the circumstances that helped to seal its fate, as did its somewhat cumbersome representation. For purposes that would suggest UTF-7 as the preferred encoding, the UTF-8 encoding form or the use of NCRs is probably more appropriate.

Comparing UTF Encoding Forms with Legacy Encodings

There is no clean code conversion algorithm between Unicode and legacy CJKV encodings, such as those used to encode GB 2312-80, GB 18030-2005, CNS 11643-2007, JIS X 0208:1997, JIS X 0213:2004, KS X 1001:2004, TCVN 6056:1995, and so on. The basic requirement for converting to and from Unicode is simply a set of mapping tables, and

luckily, there is a reliable source for some of these tables.* The Unicode Consortium has also developed, and continues to maintain, an incredibly useful tab-delimited database-like file called the *Unihan Database* that provides mappings from the tens of thousands of CJK Unified Ideographs and CJK Compatibility Ideographs to the major CJKV character set standards and to other references, including major dictionaries.† Still, it is best to consider using OS-level APIs or well-established libraries for performing code conversion between Unicode and legacy encodings. The important topic of code conversion is covered later in this chapter.

Because the UTF encoding forms are simply algorithmic transformations of one another, conversion from one UTF to another should be performed before conversion to legacy encodings is performed. Depending on the extent of non-BMP coverage, which is really a binary condition, BMP-only UTF-16 and the UTF-32 encoding form serve as the best encodings with which to interoperate with legacy encodings. This is simply common sense.

Unicode versus legacy encoding interoperability issues

When interoperating between Unicode and legacy encodings, which means that some form of code conversion is being performed (usually through the use of mapping tables), a small number of characters or character classes may require special attention. Or at least, you should look closely at them when checking or validating code conversion.

Using the Japanese standards as an example, we can explore this topic in some depth. The JIS X 0208:1997 standard includes single and double smart quotes at 01-38 through 01-41. These are represented in Unicode as U+2018, U+2019, U+201C, and U+201D, respectively. The issue with these characters is not the code conversion *per se*, but rather that font implementations that are not Unicode-based use full-width glyphs, and those that are Unicode-based use proportional glyphs.

JIS X 0208:1997 does include a character that can be problematic from the code conversion point of view. It is 01-29, and it represents a full-width dash. According to JIS standards, the Unicode mapping is U+2014 (EM DASH). Apple follows this, but Microsoft Windows maps this character to a different Unicode code point, specifically U+2015 (HORIZONTAL BAR).

JIS X 0213:2004 includes 25 characters that are not represented with a single Unicode code point, but instead map to Unicode code point sequences (base character plus combining mark). Code conversion software must be prepared to handle sequences, in addition to the usual one-to-one mappings.

* *http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/*

† *http://www.unicode.org/charts/unihan.html*

Legacy Encoding Methods

The following sections describe legacy encoding methods, which include locale-independent and locale-specific encoding methods. These encoding methods are important, and will continue to be important. Many documents are stored according to these encoding methods, and many systems that use these encoding methods still persist. Although their use is diminishing, it will take years or decades for them to go away completely. For this reason, environments that support Unicode must still interoperate with these legacy encoding methods.

Locale-Independent Legacy Encoding Methods

This is probably one of the most important sections of this book, and it lays the foundation for your complete understanding of the various CJKV encoding methods, beyond the encodings for Unicode, so be sure to take some extra time to read and study this material well. Some of the legacy encoding methods described here, specifically ISO-2022 and EUC, will serve as a basis for drawing comparisons between other encodings and for discussions that appear later in this book.

ASCII and CJKV-Roman encodings

ASCII and CJKV-Roman (GB-Roman, CNS-Roman, JIS-Roman, KS-Roman, and TCVN-Roman) are considered different character set standards, but they share the same encoding.* The definition of the encoding method for the ASCII character set is found in the document called ISO 646:1991. The encoding method for CJKV-Roman encoding is found in GB 1988-89 (GB-Roman), JIS X 0201-1997 (JIS-Roman), KS X 1003:1993 (KS-Roman), and TCVN 5712:1993 (TCVN-Roman). The ASCII/CJKV-Roman encoding method specifies that 7 bits be used, which in turn allows for 128 uniquely encoded characters. Of these 128 encoded characters, 94 comprise the ASCII/CJKV-Roman character set and are considered printable, meaning that they are displayed on the screen with something other than only whitespace. The remaining 34 characters are nonprinting, meaning that they are either control characters or whitespace. *Whitespace* refers to characters such as tabs and spaces, which take up space but contain no graphic elements. Table 4-22 lists the encoding ranges for the characters, both printable and nonprintable, in ASCII/CJKV-Roman encoding.

Table 4-22. ASCII and CJKV-Roman encoding specifications

Characters	Decimal	Hexadecimal
Control characters	0–31	00–1F
Space character	32	20

* With the possible exception of TCVN-Roman.

Table 4-22. ASCII and CJKV-Roman encoding specifications

Characters	Decimal	Hexadecimal
Graphic characters	33–126	21–7E
Delete character	127	7F

Note that these values can be represented with only seven bits; the importance of this is explained later in this chapter. For Japanese, this allows the mixture of the JIS-Roman and half-width katakana character sets when using eight-bit bytes. A graphic representation of the ASCII/CJKV-Roman encoding method is provided in Figure 4-1.

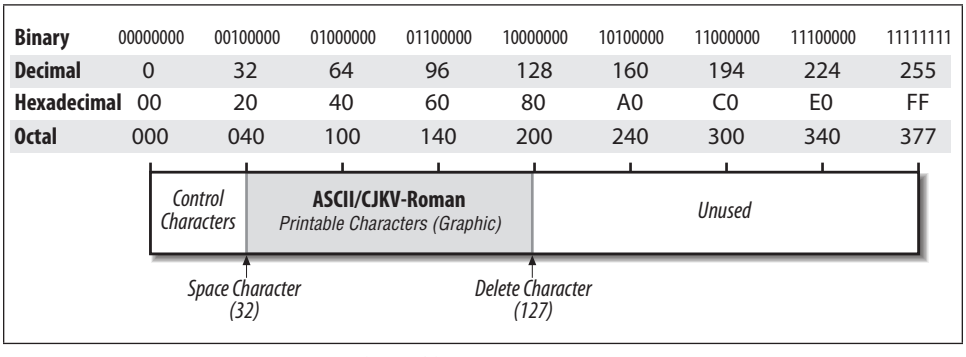


Figure 4-1. ASCII/CJKV-Roman encoding table

The extended ASCII character set encoding defined by ISO 8859 makes use of all eight bits, so more of the possible 256 eight-bit code points are available for encoding graphic characters. The first 128 code points are reserved for the ASCII character set and control characters, but the additional 128 code points made possible with the eighth bit can vary in terms of what characters are assigned to them. Precisely which characters are encoded in the extended portion of this encoding depends on the implementation. For ISO 8859, it depends on which of its 15 parts is being used. The extended ASCII characters specified in each of ISO 8859's 15 parts fall into the range `0xA1` through `0xFF`. However, not all of the code points in this extended range are used by every part of ISO 8859.

Figure 4-2 illustrates the encoding range for ASCII and extended ASCII as defined in the 15 parts of ISO 8859.

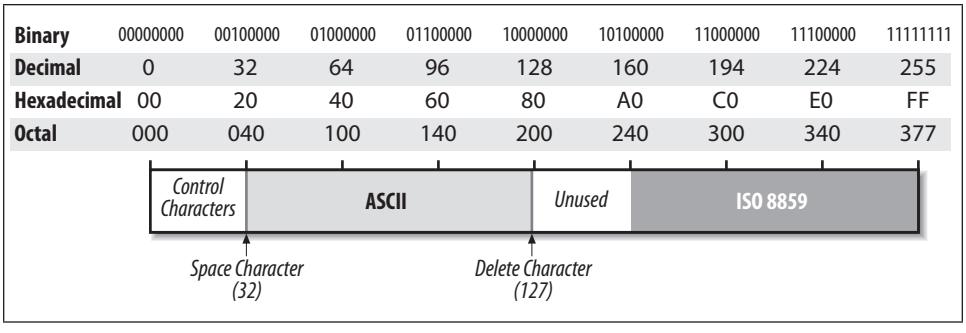


Figure 4-2. ISO 8859 encoding table

Most of the CJKV encoding methods, especially those used as internal codes for computer systems, make generous use of eight-bit characters. This makes it very difficult to mix characters from ISO 8859's 15 parts with CJKV text, because they all fall into the same encoded range. Some computer platforms deal with this much better than others. Apple's Mac OS (before Mac OS X) handled this problem simply by changing the font, which in turn changed the underlying script. Today, it is simply better practice to embrace Unicode, which doesn't exhibit this problem.

EBCDIC and EBCDIK encoding

IBM developed its own single-byte character set standard called EBCDIC (*Extended Binary-Coded-Decimal Interchange Code*). The number and type of printable characters are the same as ASCII, but the encoding for EBCDIC differs significantly. The main difference is that EBCDIC requires eight bits for full representation, whereas ASCII and CJKV-Roman require only seven bits. Table 4-23 lists the specifications for EBCDIC.

Table 4-23. EBCDIC encoding specifications

Characters	Decimal	Hexadecimal
Control characters	0–63	00–3F
Space character	64	40
Graphic characters	65–239	41–EF
Numerals	240–249	F0–F9
Undefined	250–254	FA–FE
Control character	255	FF

The EBCDIC encoding method is not used as often as the encoding for ASCII/CJKV-Roman, and appears to be slowly becoming obsolete. Unicode has effectively sealed its fate. Still, EBCDIC is a legacy encoding and cannot be completely forgotten. EBCDIC is included in these pages for the sake of completeness and for historical purposes, and because three of the encoding methods described elsewhere in this book include

EBCDIC as a subset (they are IBM's DBCS-Host, Fujitsu's JEF, and Hitachi's KEIS encoding methods—all of these are vendor-specific). Figure 4-3 illustrates the encoding space for EBCDIC encoding.

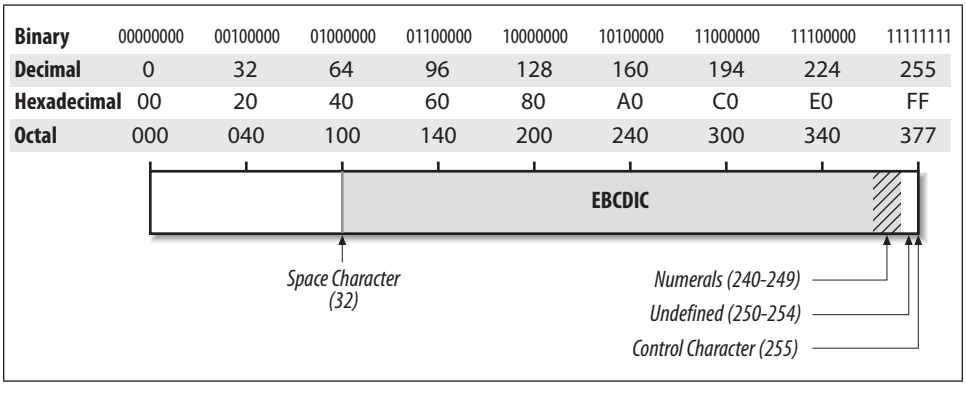


Figure 4-3. EBCDIC encoding table

There is also an encoding called EBCDIK, which stands for *Extended Binary-Coded-Decimal Interchange Kana*. It is an EBCDIC encoding that contains uppercase Latin characters, numerals, symbols, half-width katakana, and control characters (note that there are no lowercase Latin characters). Table 4-24 details the encoding ranges for the characters in EBCDIK encoding.

Table 4-24. EBCDIK encoding specifications

Characters	Decimal	Hexadecimal
Control characters	0–63	00–3F
Space character	64	40
Graphic characters (with katakana)	65–239	41–EF
Numerals	240–249	F0–F9
Undefined	250–254	FA–FE
Control character	255	FF

Note how the encoding space is identical to that of EBCDIC (see Figure 4-3). Characters appear to be randomly scattered throughout the encoding space, and not all code points have characters assigned to them. For a complete listing of EBCDIC and EBCDIK characters and their code points, please refer to Appendix M for the former, and Appendix J for the latter.

Half-width katakana encodings

Half-width katakana characters, specific to Japanese, have been encoded in a variety of ways. These characters were chosen to be the first Japanese characters encoded on

computers because they are used for Japanese telegrams. As single-byte characters, they are encoded using one of two methods. These two methods are described in the standard designated JIS X 0201-1997. Table 4-25 illustrates the one-byte encoding methods for this small collection of characters.

Table 4-25. Half-width katakana encoding specifications

Encoding type	Decimal	Hexadecimal
Seven-bit	33–95	21–5F
Eight-bit	161–223	A1–DF

ISO-2022-JP encoding makes use of both of these encoding methods. Shift-JIS encoding, specific to Japanese, makes use of only the eight-bit, half-width katakana encoding method. EUC-JP encoding takes a different approach and encodes these characters using two bytes. These encoding methods are discussed in detail later in this chapter. Table 4-26 provides the EUC-JP encoding specifications for half-width katakana.

Table 4-26. Half-width katakana encoding specifications—EUC-JP

Encoding type	Decimal	Hexadecimal
Packed format		
First byte	142	8E
Second byte range	161–223	A1–DF
Complete two-byte format		
First byte	0	00
Second byte range	161–223	A1–DF

The second byte range is identical to the eight-bit, half-width katakana encoding range in Table 4-25—EUC-JP encoding simply prefixes the byte with the value 0x8E, also known as EUC encoding's SS2 (*Single Shift 2*) character.

Figure 4-4 illustrates the encoding space for the half-width katakana character set in the various encodings. Now note how eight-bit, half-width katakana and seven-bit ASCII/JIS-Roman can coexist within the same eight-bit, one-byte encoding space. When these two character sets are mixed, the newly formed character set is called *Alphabet, Numerals, and Katakana* (ANK), as illustrated in Figure 4-5.

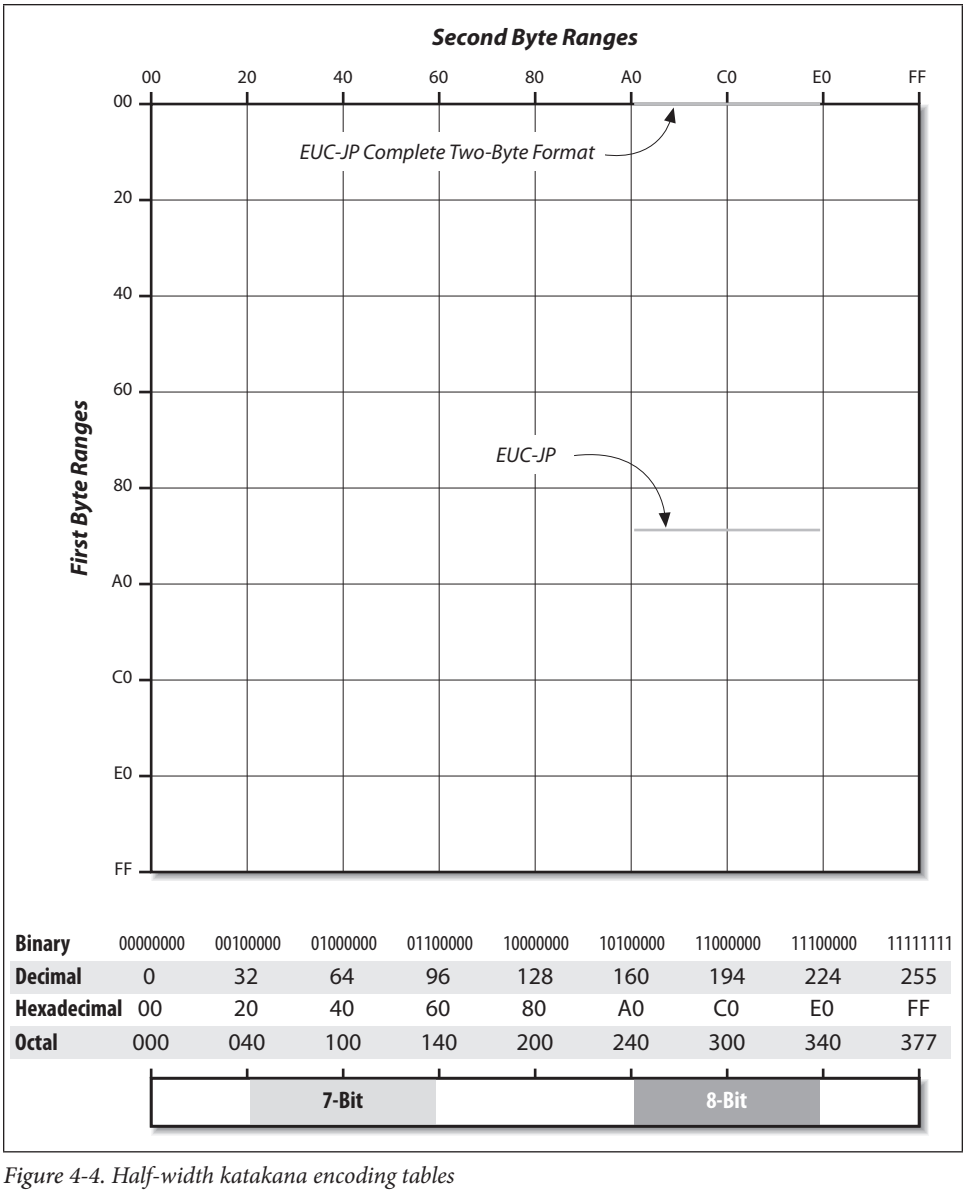


Figure 4-4. Half-width katakana encoding tables

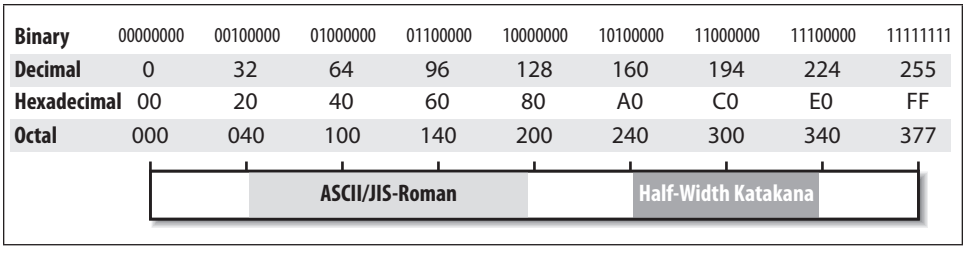


Figure 4-5. Half-width katakana plus ASCII/JIS-Roman encoding table—ANK

Row-Cell notation

Before we discuss ISO-2022 and other common multiple-byte encodings, I would like to draw your attention to Row-Cell notation. *Row-Cell* is simply a notational system for indexing characters. The Japanese version of TEX, a typesetting system developed by Donald Knuth (高德纳), is the only software I know of that can process Row-Cell codes internally or use them directly. The term *Row-Cell* itself refers to the rows and cells of a matrix. Character set standard documents commonly use Row-Cell notation to identify each character in its specification—as do the many appendixes in this book that contain character set tables. Row-Cell notation is used primarily as an encoding-independent method for representing characters within a specified matrix size, usually 94×94.

The encoding methods described thus far in this chapter more or less fall into the area of what we can safely call single-byte encodings (well, with the exception of EUC-JP–encoded half-width katakana—I guess we sort of got ahead of ourselves). Now we enter into the area of what are commonly referred to as multiple-byte encodings. Row-Cell notation is important because it provides a good introduction to the concept of encoding multiple-byte characters.

In the case of most CJKV character set standards, Row-Cell values range from 1 to 94 (using decimal notation). This constitutes a 94×94 matrix, with a total capacity of 8,836 cells. Table 4-27 provides a formal representation of the common Row-Cell ranges. Note how Row-Cell values can be expressed in notations other than decimal, although it is not very common to do so.

Table 4-27. Row-Cell notation specifications

Row and Cell	Decimal	Hexadecimal
Row range	1–94	01–5E
Cell range	1–94	01–5E

Figure 4-6 illustrates the 94×94 matrix as used by Row-Cell notation. You will quite often see this 94×94 matrix as the same dimensions of other encoding methods.

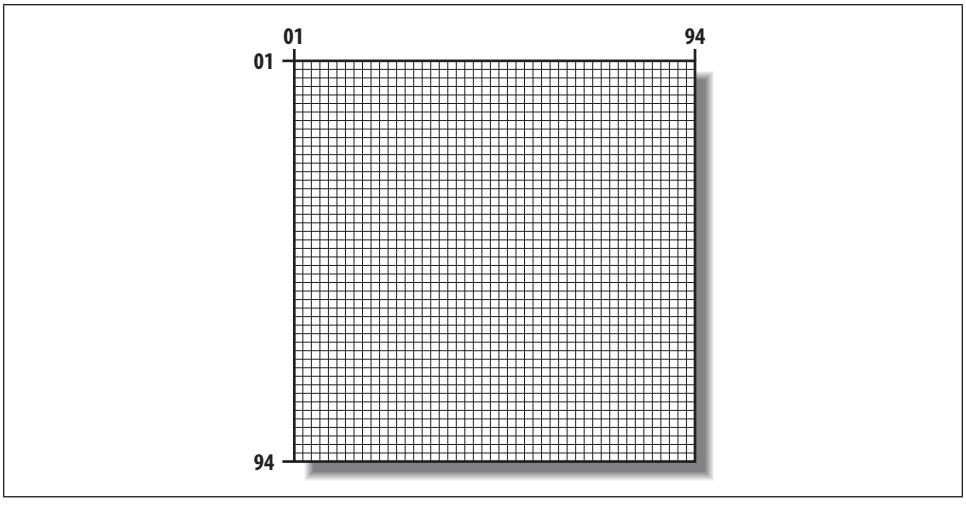


Figure 4-6. Row-Cell table

Interestingly, with some character sets expanding beyond a 94×94 matrix into multiple planes, each of which is a 94×94 matrix, the concept of Plane-Row-Cell seems appropriate. Table 4-28 expands Row-Cell notation into a more appropriate Plane-Row-Cell notation in order to handle character sets that are composed of two or more planes.

Table 4-28. Plane-Row-Cell notation specifications

Plane, row, and cell	Decimal	Hexadecimal
Plane range	1–94	01–5E
Row range	1–94	01–5E
Cell range	1–94	01–5E

CNS 11643-2007 and JIS X 0213:2004 are examples of character sets that contain two or more planes, each of which is a 94×94 matrix.

The section entitled "What Is Row-Cell and Plane-Row-Cell?" in Chapter 1 provides more details about Row-Cell and Plane-Row-Cell notation, including how they are referred to in Chinese, Japanese, and Korean.

ISO-2022 encoding

The ISO-2022 encoding method, documented in ISO 2022:1994, *Information technology—Character code structure and extension techniques*, is considered one of the most basic encoding methods used for CJKV text. It is a modal encoding; that is, escape sequences or other special characters are used to switch between different modes. Switching "modes" here can refer to either switching between one- and two-byte mode, or among character sets. Some character sets are one-byte, and some are two-byte. The character set to which you are switching determines whether you switch into one- or two-byte mode.

The use of "ISO-2022 encoding" in this book is as a generic reference to ISO-2022-CN, ISO-2022-CN-EXT, ISO-2022-JP, ISO-2022-KR, and similar encodings. The full ISO 2022:1994 standard defines a lot of machinery that is rarely used. The encodings described in this section, as a practical matter, implement only a subset of the ISO 2022:1994 standard. Some ISO-2022 encodings are, in fact, incompatible with ISO 2022:1994.

Actually, it is somewhat misleading and ambiguous to refer to the encodings in this section as "ISO-2022" because ISO 2022:1994 also sets the framework for EUC encoding, which is described later in this chapter.

ISO-2022 encoding is not very efficient for internal storage or processing on computer systems. It is used primarily as an information interchange encoding for moving text between computer systems, such as through email. This is often referred to as an external code. ISO-2022 encoding is also often referred to as a seven-bit encoding method because all of the bytes used to represent characters do not have their eighth-bit enabled.

There are some software environments that can process ISO-2022 encoding internally, the most notable of which is GNU Emacs version 20 and later. Many other programs, such as email clients, can create ISO-2022–encoded text, but do not necessarily process ISO-2022 encoding internally.

ISO-2022 encoding and Row-Cell notation are more closely related than you would think. ISO-2022 refers to encoded values, but Row-Cell refers to an encoding-independent notation for indexing characters.

There are locale-specific versions of the ISO 2022:1994 standard available, such as GB 2311-80 (China), CNS 7654-1989 (Taiwan), JIS X 0202:1998 (Japan), and KS X 1004:1995 (Korea),* but they are for the most part simply translations, and some are now outdated and obsolete.

There are several instances of ISO-2022 encodings for representing CJKV text. Table 4-29 lists these ISO-2022 encodings, along with what character sets they are known to support, plus a reference to the *Request For Comments* (RFC) in which they are officially defined and described. Of course, these RFCs are available online.† Those encodings that have been used extensively in the past, or continue to be used today for some purposes, have been highlighted.

Table 4-29. Character sets supported in ISO-2022 encodings

Encoding	Character sets	RFC
ISO-2022-CN	ASCII, GB 2312-80, CNS 11643-1992 Planes 1 and 2	1922
ISO-2022-CN-EXT	ISO-2022-CN plus CNS 11643-1992 Planes 3–7	1922
ISO-2022-JP[a]	ASCII, JIS-Roman, JIS C 6226-1978, JIS X 0208-1983	1468

* Previously designated KS C 5620-1995

† *http://www.ietf.org/rfc.html*

Table 4-29. Character sets supported in ISO-2022 encodings

Encoding	Character sets	RFC
ISO-2022-JP-1	ISO-2022-JP plus JIS X 0212-1990	2237
ISO-2022-JP-2[b]	ISO-2022-JP plus JIS X 0212-1990	1554
ISO-2022-KR	ASCII, KS X 1001:1992	1557

a. Also includes implied support for JIS X 0208-1990 and JIS X 0208:1997.

b. ISO-2022-JP-2 encoding, according to the RFC in which it is defined, also supports ISO 8859-1:1998, GB 2312-80, and KS X 1001:1992. But, from a practical point of view, ISO-2022-JP-2 encoding adds support for only JIS X 0212-1990, which makes it equivalent to ISO-2022-JP-1 encoding.

All of these encodings share one common attribute: the encoding ranges for one- and two-byte characters are identical. Table 4-30 provides these encoding ranges.

Table 4-30. One- and two-byte encoding ranges of ISO-2022 encoding

Encoding	Decimal	Hexadecimal
First byte range[a]	33–126	21–7E
Second byte range	33–126	21–7E

a. This also corresponds to the one-byte encoding range.

In other words, the values for all bytes used to encode characters are equivalent to the printable ASCII range. These characters range from the exclamation point (!) at `0x21` to the tilde (~) at `0x7E`. Figure 4-7 illustrates the ISO-2022 encoding space.

All instances of ISO-2022 encoding support one or more character sets that have been registered with ISO's International Registry for escape sequences. Table 4-31 lists CJKV character sets, along with their ISO International Registry numbers plus their final character (and its hexadecimal equivalent) for use in an escape or designator sequence. These are ordered by ascending ISO International Registry number, which more or less indicates the order in which these character sets were registered with ISO.

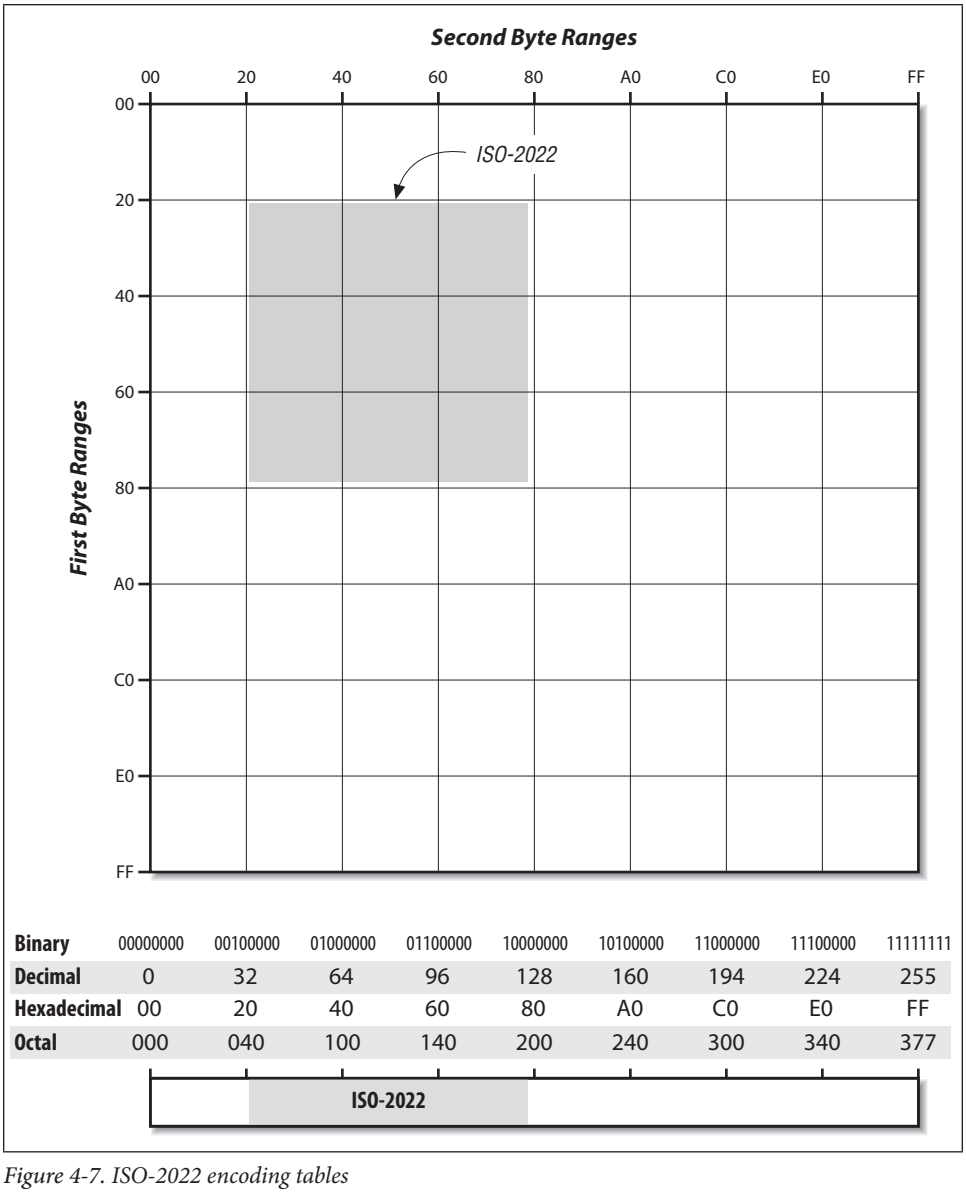


Figure 4-7. ISO-2022 encoding tables

Table 4-31. ISO International Registry numbers

Character set	ISO International Registry number	Final character
ANSI X3.4-1986 (ASCII)	6	0x42 B
JIS X 0201-1997 (JIS-Roman)	13	0x4A J
JIS X 0201-1997 (half-width katakana)	14	0x49 I
JIS C 6226-1978	42	0x40 @
GB 1988-89	57	0x54 T
GB 2312-80	58	0x41 A
JIS X 0208-1983	87	0x42 B
ISO 8859-1:1998	100	0x41 A
KS X 1001:1992	149	0x43 C
JIS X 0212-1990	159	0x44 D
ISO-IR-165:1992	165	0x45 E
JIS X 0208-1990	168	0x42 B
CNS 11643-1992 Plane 1	171	0x47 G
CNS 11643-1992 Plane 2	172	0x48 H
VSCII	180	0x5A Z
CNS 11643-1992 Plane 3	183	0x49 I
CNS 11643-1992 Plane 4	184	0x4A J
CNS 11643-1992 Plane 5	185	0x4B K
CNS 11643-1992 Plane 6	186	0x4C L
CNS 11643-1992 Plane 7	187	0x4D M
KPS 9566-97	202	0x4E N
JIS X 0213:2000 Plane 1	228	0x4F O
JIS X 0213:2000 Plane 2	229	0x50 P
JIS X 0213:2004 Plane 1	233	0x51 Q

More information about these ISO-registered character sets is available, including PDF versions of the actual ISO IR documents.*

Some references refer to GL and GR encodings, which are abbreviations for "Graphic Left" and "Graphic Right," respectively. What is the significance of left versus right? Figure 4-8 illustrates an eight-bit encoding table in which the GL and GR portions are shaded. Note how the GL region is on the left, and the GR region is on the right. This is where GL and GR come from. Apparently, the "everything is relative" expression also applies to encoding methods.

* *http://www.itscj.ipsj.or.jp/ISO-IR/*

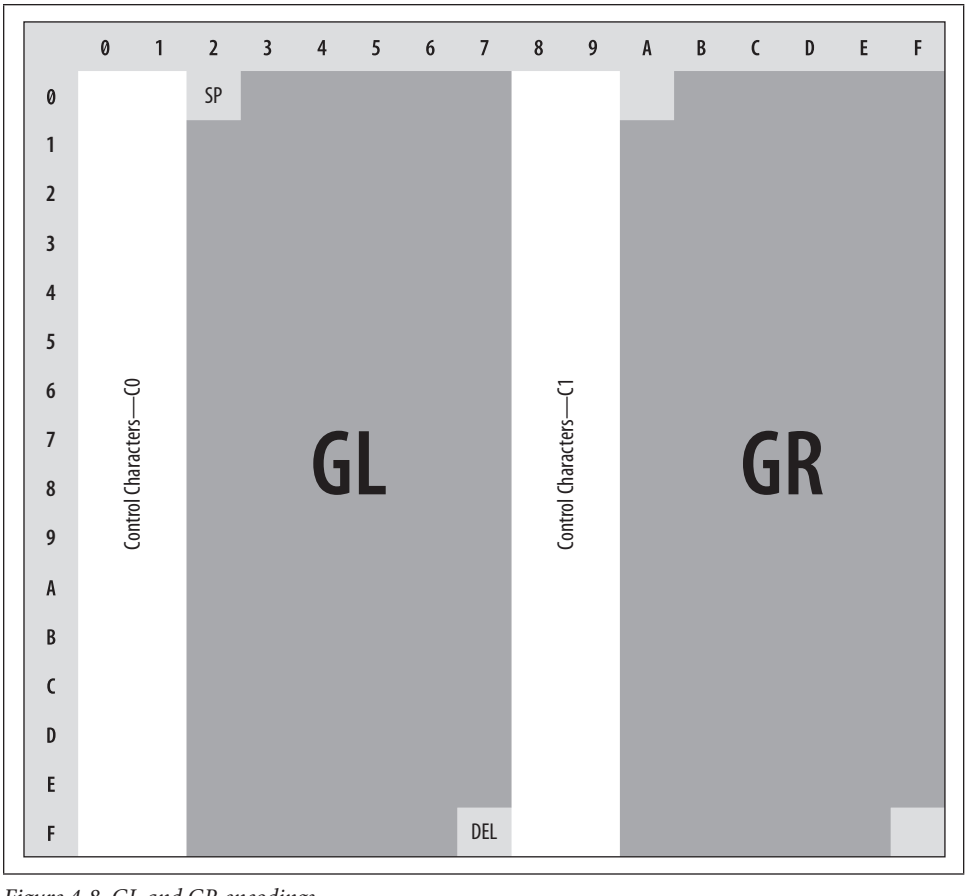


Figure 4-8. GL and GR encodings

The GL range is 0x21 through 0x7E, which is used by ISO-2022 encodings to encode all graphic characters, and by EUC encodings to encode the single-byte ASCII or CJKV-Roman character sets. The GR block is 0xA1 through 0xFE, which is used by EUC encodings to encode all two-byte graphic characters, as well as some single-byte graphic characters, such as half-width katakana.

My convention is to use "ISO-2022" or locale-specific instances thereof to indicate GL for one- and two-byte encodings, and "EUC" or locale-specific instances thereof to indicate GR for two- or more-byte portions of the encoding.

ISO-2022 encodings use special characters or sequences of characters called designator sequences, single shift sequences, shifting characters, and escape sequences. The following are descriptions of each:

Designator sequence
: Indicates what character set should be invoked when in two-byte mode. It does not invoke two-byte mode.

Single shift sequence
: Indicated by SS2 (Single Shift 2; <1B 4E>) or SS3 (Single Shift 3; <1B 4F>), invokes two-byte mode only for the following two bytes. Typically employed for rarely used character sets.

Shifting characters
: Indicated by SO (Shift-Out; 0x0E) or SI (Shift-In; 0x0F), switches between one- and two-byte modes. An SO invokes two-byte mode until an SI is encountered, which invokes one-byte mode.

Escape sequence
: Indicates what character set should be invoked, and invokes it as well.

Table 4-32 indicates what special characters or character sequences each locale-specific instance of ISO-2022 encoding uses. Note how there are two basic types of ISO-2022 encodings: those that use designator sequences, shifting characters, and perhaps single shift sequences; and those that use only escape sequences.

Table 4-32. The use of special characters or character sequences in ISO-2022 encodings

Type	Encodings
Designator sequences	ISO-2022-CN, ISO-2022-CN-EXT, ISO-2022-KR
Single shift sequences	ISO-2022-CN, ISO-2022-CN-EXT
Shifting characters	ISO-2022-CN, ISO-2022-CN-EXT, ISO-2022-KR
Escape sequences	ISO-2022-JP, ISO-2022-JP-1, ISO-2022-JP-2

The following sections describe each of these locale-specific ISO-2022 encoding instances in more detail, in the order of their importance, and provide illustrative examples of their use. You will soon discover that some instances are more complex than others.

ISO-2022-JP encodings—RFCs 1468, 1554, and 2237

ISO-2022-JP encoding, defined in RFC 1468, *Japanese Character Encoding for Internet Messages*, provides support for the ASCII, JIS-Roman, JIS C 6226-1978, and JIS X 0208-1983 character sets.* ISO-2022-JP-2 encoding, defined in RFC 1554, *ISO-2022-JP-2: Multilingual Extension of ISO-2022-JP*, is an extension to ISO-2022-JP that adds support for

* *http://www.ietf.org/rfc/rfc1468.txt*

the GB 2312-80, JIS X 0212-1990, and KS X 1001:1992 character sets in addition to two parts of ISO 8859 (Parts 1 and 7).* ISO-2022-JP-1, defined in RFC 2237, *Japanese Character Encoding for Internet Messages*, is a modest extension to ISO-2022-JP that simply adds support for JIS X 0212-1990.†

All three ISO-2022-JP encodings are incompatible with ISO 2022:1994 because they use escape sequences that do not follow the standard (for JIS C 6226-1978 and JIS X 0208-1983), and because the JIS X 0208-1983 escape sequence is used for introducing JIS X 0208-1990 (see Table 4-37).

From a practical point of view, ISO-2022-JP-2 adds support for only JIS X 0212-1990. There are other ISO-2022 encodings better suited to support the GB 2312-80 and KS X 1001:1992 character sets, specifically ISO-2022-CN and ISO-2022-KR, respectively.

Table 4-33 lists the escape sequences supported by ISO-2022-JP encoding.

Table 4-33. ISO-2022-JP encoding specifications

Character set	Decimal	Hexadecimal	Visual—ASCII
ASCII	27 40 66	1B 28 42	<ESC> (B
JIS-Roman	27 40 74	1B 28 4A	<ESC> (J
JIS C 6226-1978	27 36 64	1B 24 40	<ESC> $ @
JIS X 0208-1983	27 36 66	1B 24 42	<ESC> $ B

Escape sequences must be fully contained within a line; they should not span newlines or carriage returns. If the last character on a line is represented by two bytes, an ASCII or JIS-Roman character escape sequence should follow before the line terminates. If the first character on a line is represented by two bytes, a two-byte character escape sequence should precede it. Not all these procedures are necessary, but they are useful because they ensure that small communication errors do not render an entire Japanese document unreadable—each line becomes self-contained. These escape sequences are also known as kanji-in and kanji-out. Kanji-in corresponds to a two-byte character escape sequence, and kanji-out corresponds to a one-byte character escape sequence.

ISO-2022-JP-1 is identical to ISO-2022-JP encoding except that it adds support for the JIS X 0212-1990 character set. Table 4-34 provides the JIS X 0212-1990 escape sequence as described for ISO-2022-JP-1 encoding.

Table 4-34. ISO-2022-JP-1 encoding specifications

Character set	Decimal	Hexadecimal	Visual—ASCII
JIS X 0212-1990	27 36 40 68	1B 24 28 44	<ESC> $ (D

* *http://www.ietf.org/rfc/rfc1554.txt*

† *http://www.ietf.org/rfc/rfc2237.txt*

ISO-2022-JP-2, on the other hand, adds support for five additional character sets. These character sets, along with their escape sequences, are listed in Table 4-35.

Table 4-35. ISO-2022-JP-2 encoding specifications

Character set	Decimal	Hexadecimal	Visual—ASCII
GB 2312-80	27 36 65	1B 24 41	<ESC> $ A
JIS X 0212-1990	27 36 40 68	1B 24 28 44	<ESC> $ (D
KS X 1001:1992	27 36 40 67	1B 24 28 43	<ESC> $ (C
ISO 8859-1:1998	27 46 65	1B 2E 41	<ESC> . A
ISO 8859-7:1998	27 46 70	1B 2E 46	<ESC> . F

Let's take a look at some ISO-2022-JP–encoded material to see exactly how this encoding method works. Table 4-36 uses the example string is かな漢字, which means "kana [and] kanji," and the encoded values are expressed in hexadecimal notation.

Table 4-36. Japanese encoding example—ISO-2022-JP encoding

Encoding	String					
Characters		か	な	漢	字	
ISO-2022-JP	1B 24 42	24 2B	24 4A	34 41	3B 7A	1B 28 4A
Escape sequences	<ESC> $ B					<ESC> (J
ISO-2022-JP—visual		$ +	$ J	4 A	; z	

In this example, the first escape sequence signals a switch in mode to the JIS X 0208-1983 character set (two-byte mode); this is followed by the data for the four characters to be displayed. To terminate the string, a one-byte character escape sequence is used (in this case it is the one for switching to the JIS-Roman character set).

You have already learned that ISO-2022-JP encoding makes use of seven bits for representing two-byte characters. The actual encoded range corresponds to that used for representing the ASCII character set. Thus, the encoded values for the kanji in our example can be represented with ASCII characters, as shown in the last line of the example.

The predecessor of ISO-2022-JP encoding—JIS encoding

Before RFC 1468 was introduced in June of 1993, which effectively defined ISO-2022-JP encoding, ISO-2022–encoded Japanese text was commonly referred to as "JIS encoding" (and is often still referred to as such in many contexts). But, what most people do not realize is that what is now known as ISO-2022-JP encoding is actually a subset of what is still known as JIS encoding. Confused? You shouldn't be after reading this section.

JIS encoding can be thought of as a much richer variation of ISO-2022-JP encoding—richer in the sense that it supports more character sets. Table 4-37 provides the specifications for JIS encoding.

Table 4-37. JIS encoding specifications

Encoding	Decimal	Hexadecimal	Visual—ASCII
One-byte characters			
Byte range	33–126	21–7E	
Two-byte characters			
First byte range	33–126	21–7E	
Second byte range	33–126	21–7E	
Escape sequences			
JIS-Roman	27 40 74	1B 28 4A	<ESC> (J
JIS-Roman[a]	27 40 72	1B 28 48	<ESC> (H
ASCII	27 40 66	1B 28 42	<ESC> (B
Half-width katakana	27 40 73	1B 28 49	<ESC> (I
JIS C 6226-1978	27 36 64	1B 24 40	<ESC> $ @
JIS X 0208-1983	27 36 66	1B 24 42	<ESC> $ B
JIS X 0208-1990	27 38 64 27 36 66	1B 26 40 1B 24 42	<ESC> & @ <ESC> $ B
JIS X 0208:1997	27 38 64 27 36 66	1B 26 40 1B 24 42	<ESC> & @ <ESC> $ B
JIS X 0212-1990	27 36 40 68	1B 24 28 44	<ESC> $ (D
JIS7 half-width katakana			
Shift-out	14	0E	<SO>
Byte range	33–95	21–5F	
Shift-in	15	0F	<SI>
JIS8 half-width katakana			
Byte range	161–223	A1–DF	

a. This is improperly used on some implementations as the one-byte character escape sequence for JIS-Roman. According to the standard designated JIS X 0202:1998, it is actually the one-byte character escape sequence for the Swedish character set. It is a good idea for software to recognize, but not to generate, this one-byte character escape sequence. The correct sequence to use is <ESC> (J.

JIS encoding also supports half-width katakana, and has two different methods called JIS7 and JIS8. JIS7 encoding has all eighth bits cleared; JIS8 does not. JIS7 and JIS8 half-width katakana encodings are not widely used in products, so it is questionable whether all software should necessarily generate such codes. It is, however, important that software recognize and deal with them appropriately.

JIS7 encoding is identical to JIS encoding, but with the addition of another escape sequence for shifting into half-width katakana mode. This method is defined in the document JIS X 0202:1998. This means that a document containing two-byte Japanese,

one-byte ASCII, and half-width katakana characters may make use of at least three escape sequences, one for shifting into each of the three modes or character sets. An alternate method for encoding half-width katakana under JIS7 uses the ASCII Shift-Out (SO) and Shift-In (SI) characters instead of an escape sequence. Half-width katakana sequences begin with a Shift-Out character, and are terminated with a Shift-In character. This encoding method is described in the standard designated JIS X 0201-1997.

The encoding range for JIS8 includes eight-bit bytes (its range is `0xA1` through `0xDF`, and is identical to the half-width katakana range in Shift-JIS encoding). The text stream must be in one-byte mode. This encoding is also described in JIS X 0201-1997.

ISO-2022-KR encoding—RFC 1557

ISO-2022-KR encoding, defined in RFC 1557, *Korean Character Encoding for Internet Messages*, specifies how Korean text is to be encoded for email messages or other electronic transmission.* The ISO-2022-KR designator sequence must appear only once in a file, at the beginning of a line, before any KS X 1001:2004 characters. This usually means that it appears by itself on the first line of the file. ISO-2022-KR uses only two shifting sequences (for switching between one- and two-byte modes), and involves neither the escape character nor an escape sequence. Table 4-38 provides a listing of the ISO-2022-KR designator sequence and the two shifting sequences.

Table 4-38. ISO-2022-KR encoding specifications

Encoding	Decimal	Hexadecimal	Visual—ASCII
Designator sequence	27 36 41 67	1B 24 29 43	<ESC> $) C
One-byte shift	15	0F	<SI>
Two-byte shift	14	0E	<SO>

Table 4-39 illustrates how Korean text is encoded according to ISO-2022-KR encoding by using the Korean word 김치 (*gimchi*, meaning "kimchi") as the example.

Table 4-39. Korean encoding example—ISO-2022-KR encoding

Encoding	String				
Characters			김	치	
ISO-2022-KR	1B 24 29 43	0E	31 68	44 21	0F
Designator sequence	<ESC> $) C				
Shifts		<SO>			<SI>
ISO-2022-KR—visual			1 h	D !	

* *http://www.ietf.org/rfc/rfc1557.txt*

Note the similarities with the ISO-2022-CN and ISO-2022-CN-EXT encoding methods, specifically that a designator sequence and shifting characters are used for switching between one- and two-byte modes.

ISO-2022-CN and ISO-2022-CN-EXT encodings—RFC 1922

ISO-2022-CN and ISO-2022-CN-EXT encodings, defined in RFC 1922, *Chinese Character Encoding for Internet Messages*, are somewhat complex in that they involve designator sequences, single shift sequences, and shifting characters.* This is because these encodings support a rather large number of character sets, from both China and Taiwan.

ISO-2022-CN is distinguished from ISO-2022-CN-EXT in that it supports only ASCII, GB 2312-80, and CNS 11643-1992 Planes 1 and 2. ISO-2022-CN-EXT is equivalent to ISO-2022-CN, plus it supports many more character sets.

Table 4-40 provides the specifications for the designator sequences used in ISO-2022-CN and ISO-2022-CN-EXT encodings. Note how the designator sequences indicate a character set or subset thereof (such as each plane of CNS 11643-1992).

Table 4-40. ISO-2022-CN and ISO-2022-CN-EXT encoding specifications—Part 1

Character set	Decimal	Hexadecimal	Visual—ASCII
GB 2312-80	27 36 41 65	1B 24 29 41	<ESC> $) A
CNS 11643-1992 Plane 1	27 36 41 71	1B 24 29 47	<ESC> $) G
CNS 11643-1992 Plane 2	27 36 42 72	1B 24 2A 48	<ESC> $ * H
ISO-IR-165	27 36 41 69	1B 24 29 45	<ESC> $) E
CNS 11643-1992 Plane 3	27 36 43 73	1B 24 2B 49	<ESC> $ + I
CNS 11643-1992 Plane 4	27 36 43 74	1B 24 2B 4A	<ESC> $ + J
CNS 11643-1992 Plane 5	27 36 43 75	1B 24 2B 4B	<ESC> $ + K
CNS 11643-1992 Plane 6	27 36 43 76	1B 24 2B 4C	<ESC> $ + L
CNS 11643-1992 Plane 7	27 36 43 77	1B 24 2B 4D	<ESC> $ + M

ISO-2022-CN-EXT actually provides support for additional character sets, including GB/T 12345-90, GB 7589-87, GB 7590-87, and others, but these character sets are not listed in the table because they are not yet ISO-registered.

Table 4-41 provides the specifications for the single shift sequences and shifting characters used in ISO-2022-CN and ISO-2022-CN-EXT encodings.

* *http://www.ietf.org/rfc/rfc1922.txt*

Table 4-41. ISO-2022-CN and ISO-2022-CN-EXT encoding specifications—Part 2

Special characters	Decimal	Hexadecimal	Visual—ASCII
SS2	27 78	1B 4E	<ESC> N
SS3	27 79	1B 4F	<ESC> O
One-byte shift	15	0F	<SI>
Two-byte shift	14	0E	<SO>

Once you know the designator sequence, you then must find out what kind of shifting is required. Table 4-42 lists the three shifting types, SO, SS2, and SS3, along with what character sets are used for each method.

Table 4-42. ISO-2022-CN and ISO-2022-CN-EXT encoding specifications—Part 3

Shifting type	Character sets
SO	GB 2312-80, CNS 11643-1992 Plane 1, and ISO-IR-165:1992
SS2	CNS 11643-1992 Plane 2
SS3	CNS 11643-1992 Planes 3–7

The designator sequence must appear once on a line before any instance of the character set it designates. If two lines contain characters from the same character set, both lines must include the designator sequence (this is so that the text can be displayed correctly when scrolled back in a window). This is different behavior from ISO-2022-KR (described later in this chapter), where the designator sequence appears only once in the entire file (this is because ISO-2022-KR supports only one two-byte character set—shifting to two-byte mode is unambiguous).

The predecessor of ISO-2022-CN encoding—HZ encoding

HZ encoding was a simplistic yet powerful system for encoding GB 2312-80 text, which was developed by Fung Fung Lee (李枫峰 *lǐ fēng fēng*) for exchanging email and posting news in Chinese. HZ encoding was commonly used when exchanging email or posting messages to Usenet News (specifically, to *alt.chinese.text*).

The actual encoding ranges used for one- and two-byte characters are identical to ISO-2022-CN and ISO-2022-CN-EXT encodings, but, instead of using designator sequences and shift characters to shift between character sets, a simple string of two printable characters is used. Table 4-43 lists the two most important shift sequences used in HZ encoding.

Table 4-43. HZ encoding shift sequences

Character set	Shift sequence	Decimal	Hexadecimal
ASCII or GB-Roman	~}	126 125	7E 7D
GB 2312-80	~{	126 123	7E 7B

As you can see, the overline or tilde character (0x7E) is interpreted as though it were an escape character in HZ encoding, so it has special meaning. If an overline character is to appear in one-byte mode, it must be doubled (so "~~" would appear as "~"). There is also a fourth escape sequence, specifically "~" followed by a newline character. It is used for maintaining two-byte mode while breaking lines. This effectively means that there are four shift sequences used in HZ encoding, as shown in Table 4-44.

Table 4-44. Meanings of HZ escape sequences

Escape sequence	Meaning
~~	"~" in one-byte mode
~}	Shift into one-byte mode: ASCII or GB-Roman
~{	Shift into two-byte mode: GB 2312-80
~<NL>	Maintain two-byte across lines

HZ encoding typically works without problems because the shift sequences, while being valid two-byte code points, represent empty code points in the very last row of the GB 2312-80 character set's 94×94 matrix (actually, the second- and third-from-last code points). This effectively makes 93 of the 94 rows accessible.

The complete HZ specification is part of the HZ package,* described in Fung Fung Lee's RFC 1843, *HZ—A Data Format for Exchanging Files of Arbitrarily Mixed Chinese and ASCII characters*.†

In addition, RFC 1842, *ASCII Printable Characters-Based Chinese Character Encoding for Internet Messages*, establishes "HZ-GB-2312" as the charset designator for MIME-encoded email headers.‡ Its properties are identical to HZ encoding as described here and in RFC 1843.

Other possible ISO-2022 encodings

Although not even officially in existence, one could conceive of ISO-2022–style encodings for North Korea's KPS 9566-97 or Vietnam's TCVN 5773:1993 and TCVN 6056:1995 standards. Actually, the final escape sequence character for KPS 9566-97 has been

* *ftp://ftp.ifcss.org/pub/software/unix/convert/HZ-2.0.tar.gz*

† *http://www.ietf.org/rfc/rfc1843.txt*

‡ *http://www.ietf.org/rfc/rfc1842.txt*

registered with ISO, and is set to "N" (0x4E). So, the stage has been set for establishing ISO-2022-KP encoding. If an ISO-2022-KP encoding is to spring into existence, it is likely to be similar to that of ISO-2022-KR encoding, except that the designator sequence that is used for KPS 9566-97 would be the four-character string <ESC> $) N (<1B 24 29 4E>).

Likewise, a reasonable encoding name for TCVN 5773:1993 and TCVN 6056:1995 would be ISO-2022-VN, but before that can happen, ISO registration is necessary. However, the ever-increasing adoption of Unicode makes this very unlikely. Also, the use of ISO-2022 encodings has diminished, though it is still possible to encounter them in some contexts or for specific uses.

EUC encoding

Extended Unix Code (EUC) encoding was implemented as the internal code for most Unix software configured to support Japanese. EUC is also known in Japan as *Unixized JIS* (UJIS) and AT&T JIS. The definition of EUC, like that of ISO-2022, comes from the standard designated ISO 2022:1994, though its implementation dates back to the early 1980s.

EUC was developed as a method for handling multiple character sets, Japanese and otherwise, within a single text stream. The full definition of EUC encoding is quite rich and supports various multiple-byte encodings, but the specific implementations used for CJKV systems usually fall into two specific types: packed format and complete two-byte format. The packed format was far more common. The Japanese definition (or instance) of EUC, called EUC-JP, was standardized in 1991 by three organizations: *Open Software Foundation* (OSF), *Unix International* (UI), and *Unix System Laboratories Pacific* (USLP). This standardization has subsequently made it easier for other developers to implement Japanese systems, and at the same time reinforced the use of EUC-JP encoding. The definitions of EUC encoding for other CJKV locales have also been defined.

There was once a trend in software development to produce systems that processed EUC-JP—it is much more extensible than Shift-JIS. Most Unix operating systems process EUC-JP encoding internally. However, Unicode changed all that.

CJKV implementations of EUC encoding use one specific instance of multiple-length and one specific instance of fixed-length encoding.

The full definition of EUC encoding consists of four code sets. Code set 0 is always set to the ASCII character set or a country's own version thereof, such as KS-Roman for Korea. The remaining code sets are defined as a set of variants for which each country can specify what character sets they support. You will learn how each major CJKV locale has implemented EUC encoding in the sections that follow.

There are several reserved code points in EUC that cannot be used to encode printable characters. These code points and ranges consist of the "space" character, the "delete" character, and two disjointed ranges of control characters. Table 4-45 shows these code point ranges in more detail.

Table 4-45. EUC reserved code ranges and positions

Special characters	Decimal	Hexadecimal
Control set 0	0–31	00–1F
Space character	32	20
Delete character	127	7F
Control set 1	128–159	80–9F

This allocation of reserved code points permits two ranges for encoding graphic characters, specifically 0x21 through 0x7E (94 characters) and 0xA0 through 0xFF (96 characters). The second range, at least for CJKV implementations, is limited to the range 0xA1 through 0xFE as a way to remain compatible with encodings that support ranges of only 94 characters, such as ISO-2022. Table 4-46 lists these two encoding ranges in decimal and hexadecimal notations, and also indicates their GL and GR relationships.

Table 4-46. EUC graphic character encoding ranges

Encoding ranges	Decimal	Hexadecimal
First code range—GL	33–126	21–7E
Second code range—GR	160–255	A0–FF

There are also two special characters: SS2 and SS3. Like with ISO-2022 encodings, SS2 stands for *Single Shift 2*, and serves as a prefix for every character in code set 2. Likewise, SS3 stands for *Single Shift 3*, and serves as a prefix for every character in code set 3. Table 4-47 lists these characters in decimal and hexadecimal notations. Contrast these with the SS2 and SS3 characters used in ISO-2022-CN and ISO-2022-CN-EXT encodings in Table 4-42. These will become important later when we discuss code sets 2 and 3.

Table 4-47. EUC encoding's SS2 and SS3 characters

Encoding	Decimal	Hexadecimal
SS2	142	8E
SS3	143	8F

Table 4-48 illustrates the variable-length representation of the four EUC code sets, along with some of their possible permutations (hexadecimal notation is used here for the sake of space), as they may appear in locale-specific instances of EUC encoding. Note how code set 0 uses GL encoding ranges, and code sets 1 through 3 use GR encoding ranges.

Table 4-48. EUC variable-width representations

Code set	Variant 1	Variant 2	Variant 3
Code set 0	21–7E		
Code set 1	A0–FF	A0–FF + A0–FF	A0–FF + A0–FF + A0–FF
Code set 2	8E + A0–FF	8E + A0–FF + A0–FF	8E + A0–FF + A0–FF + A0–FF
Code set 3	8F + A0–FF	8F + A0–FF + A0–FF	8F + A0–FF + A0–FF + A0–FF

The representations shown in Table 4-48 are often referred to as *EUC packed format*, and represent the most commonly used instances of EUC encoding. Also, there can be as many variants of this representation as needed to represent a given locale's character set—or character sets if there is more than one that requires support.

The definition of EUC encoding is thus locale-specific—each locale implements its own instance of EUC encoding. This is also why specifying EUC encoding itself is somewhat ambiguous. The locale-specific instances of EUC encoding are known as EUC-CN (China), EUC-TW (Taiwan), EUC-JP (Japan), and EUC-KR (Korea). These are described in the following sections.

There are two fixed-length EUC representations: 16- and 32-bit.* The significance of these fixed-length representations is that all characters are represented by the same number of bits or bytes. While this may waste space, it does make internal processing more efficient for some text-processing tasks. Table 4-49 describes the 16-bit, fixed-length representations of EUC encoding. Be sure to note the mixed use of GL and GR encoding ranges for code sets 2 and 3.

Table 4-49. EUC 16-bit, fixed-length representations

Code set	Variant 1	Variant 2
Code set 0	00 + 21–7E	
Code set 1	80 + A0–FF	A0–FF + A0–FF
Code set 2	00 + A0–FF	21–7E + A0–FF
Code set 3	80 + 21–7E	A0–FF + 21–7E

This 16-bit, fixed-length representation is often referred to as *EUC complete two-byte format*, and is primarily used for internally processing (as opposed to external representation, such as for information interchange). Note that the SS2 and SS3 characters are not used in this representation—they are not necessary under a fixed-length encoding model. The 32-bit representation gets very long, and since it is not implemented for most locales, there is no need to illustrate its code ranges here.

* Of course, these can also be thought of as two- and four-byte representations.

The following sections describe the locale-specific instances of EUC encoding. Be sure to note their similarities and differences. You will find similarities in code sets 0 and 1, and the primary differences in code sets 2 and 3. Also, sometimes code sets 2 and 3 are unused.

EUC-CN encoding—China

The instance of EUC encoding used for the China locale is known as EUC-CN encoding. It is sometimes referred to as eight-bit GB or simply GB encoding outside the context of EUC encoding. Table 4-50 lists what character set is assigned to each EUC code set. The "Display width" column in this and similar tables in this chapter corresponds to the number of columns occupied by each character in a code set. A display width value of 1 corresponds to half-width or proportional, depending on the glyphs in the font, and a display width value of 2 corresponds to full-width.

Table 4-50. EUC-CN code set allocation

Code set	Character set	Display width	Number of bytes
Code set 0	ASCII or GB-Roman	1	1
Code set 1	GB 2312-80	2	2
Code set 2	unused		
Code set 3	unused		

Note that EUC code sets 2 and 3 are unused in EUC-CN encoding. Table 4-51 lists the EUC-CN encoding specifications.

Table 4-51. EUC-CN encoding specifications

Code set	Decimal	Hexadecimal
Code set 0		
Byte range	33–126	21–7E
Code set 1		
First byte range	161–254	A1–FE
Second byte range	161–254	A1–FE
Code set 2	unused	unused
Code set 3	unused	unused

EUC-CN encoding is virtually identical to EUC-KR encoding (covered in the next page or so), except for what particular character sets are allocated to each EUC code set, specifically code sets 0 and 1. Figure 4-9 illustrates the encoding regions for EUC-CN and EUC-KR encodings.

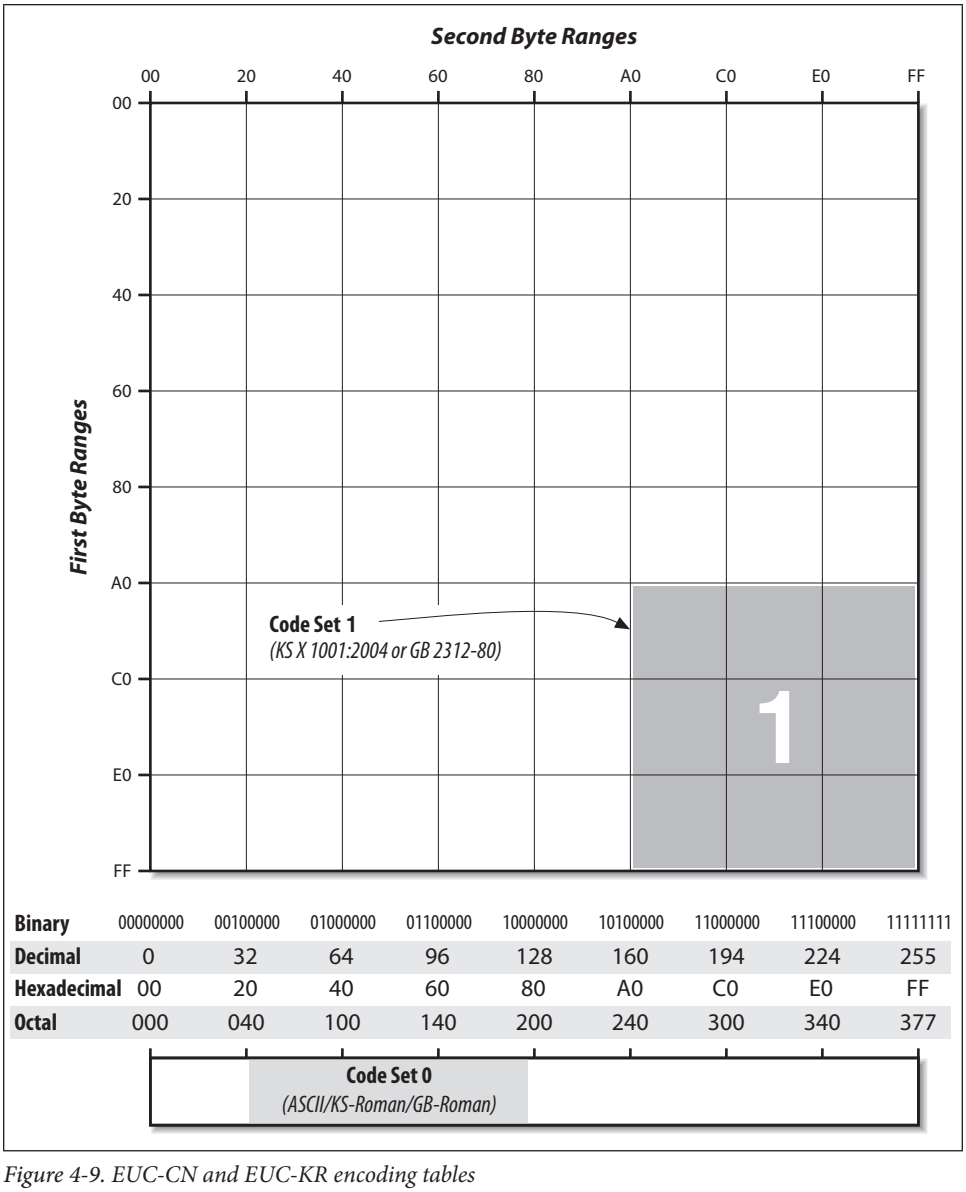


Figure 4-9. EUC-CN and EUC-KR encoding tables

EUC-TW encoding—Taiwan

The instance of EUC encoding used for the Taiwan locale is known as EUC-TW encoding. It is by far the most complex instance of EUC encoding in terms of how many characters it encodes, which is now approximately 75,000. Table 4-52 lists what character

set is assigned to what code set. In the case of CNS 11643-2007, its 80 defined planes are implemented across EUC code sets 1 and 2.

Table 4-52. EUC-TW code set allocation

Code set	Character set	Display width	Number of bytes
Code set 0	ASCII or CNS-Roman	1	1
Code set 1	CNS 11643-2007 Plane 1	2	2
Code set 2[a]	CNS 11643-2007 Planes 1–80	2	4
Code set 3	unused		

a. Note how CNS 11643-2007 Plane 1 is encoded in both EUC code set 1 and 2—this is by design.

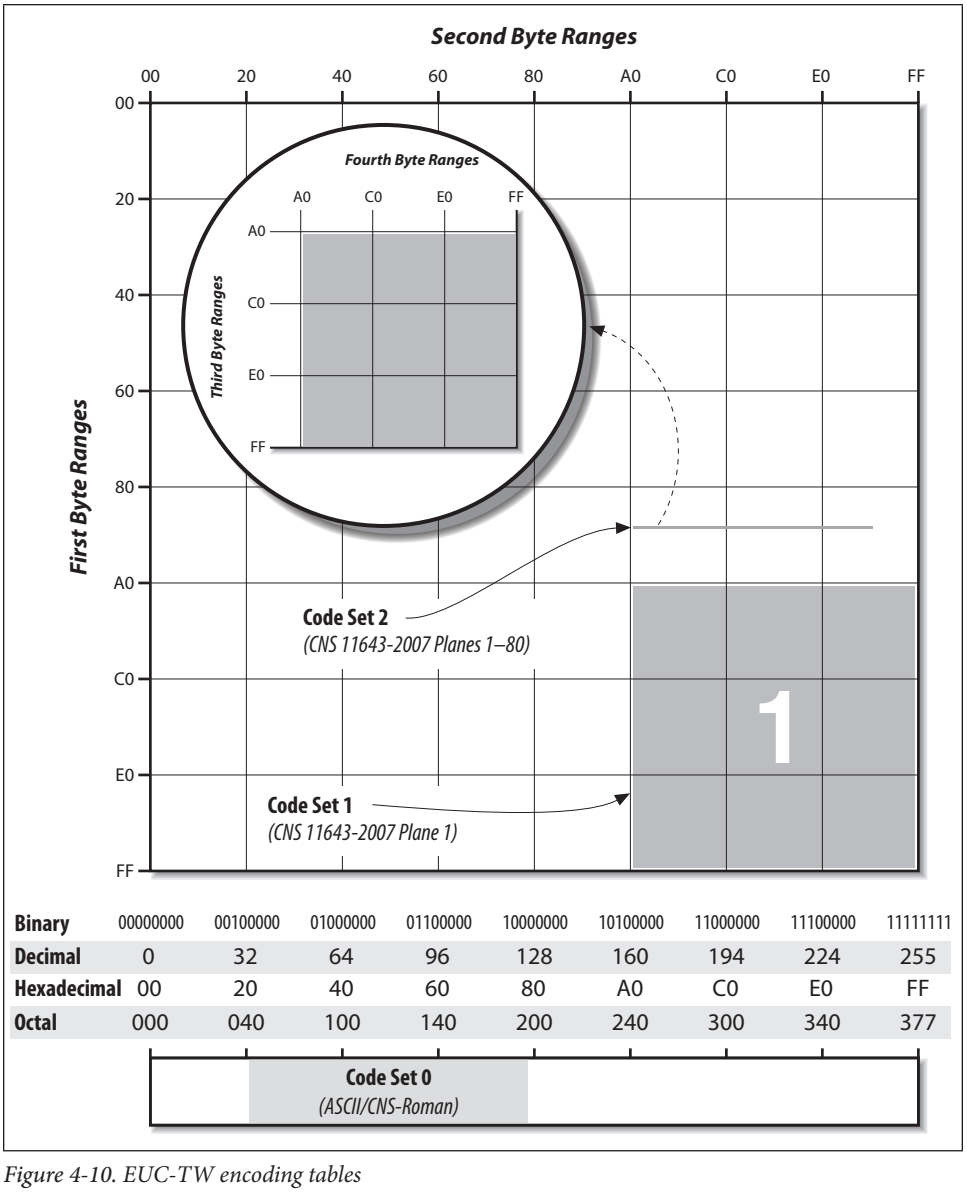
Note that EUC code set 2 is quite overloaded (over 700,000 code points are referenced using a 4-byte encoding), but EUC code set 3 is completely unused. Table 4-53 lists EUC-TW's encoding specifications.

Table 4-53. EUC-TW encoding specifications

Code set	Decimal	Hexadecimal
Code set 0		
Byte range	33–126	21–7E
Code set 1		
First byte range	161–254	A1–FE
Second byte range	161–254	A1–FE
Code set 2		
First byte (SS2)	142	8E
Second byte range[a]	161–240	A1–F0
Third byte range	161–254	A1–FE
Fourth byte range	161–254	A1–FE
Code set 3	unused	unused

a. This value indicates the plane number. Subtract decimal 160 (or 0xA0) from the value to calculate the plane number. For example, 161 (or 0xA1) means Plane 1, and 240 (or 0xF0) means Plane 80.

Note the use of four bytes to encode characters in EUC code set 2. CNS 11643-2007 Plane 7, for example, is encoded by using 167 (0xA7) as the second byte. Figure 4-10 illustrates the complete EUC-TW encoding regions.



Binary	00000000	00100000	01000000	01100000	10000000	10100000	11000000	11100000	11111111
Decimal	0	32	64	96	128	160	194	224	255
Hexadecimal	00	20	40	60	80	A0	C0	E0	FF
Octal	000	040	100	140	200	240	300	340	377

Figure 4-10. EUC-TW encoding tables

EUC-JP encoding—Japan

The official definition of EUC-JP encoding specifies the character sets that are allocated to each of the four EUC code sets. These allocations are illustrated in Table 4-54.

Table 4-54. EUC-JP code set allocation

Code set	Character set	Display width	Number of bytes
Code set 0	ASCII or JIS-Roman	1	1
Code set 1	JIS X 0208:1997	2	2
Code set 2	Half-width katakana	1	2
Code set 3	JIS X 0212-1990	2	3

Note that EUC-JP encoding encodes half-width katakana using two bytes rather than one, and that it includes a very large user-defined character space. Large enough, in fact, that EUC-JP encoding implements the JIS X 0212-1990 character set by placing it into this space.

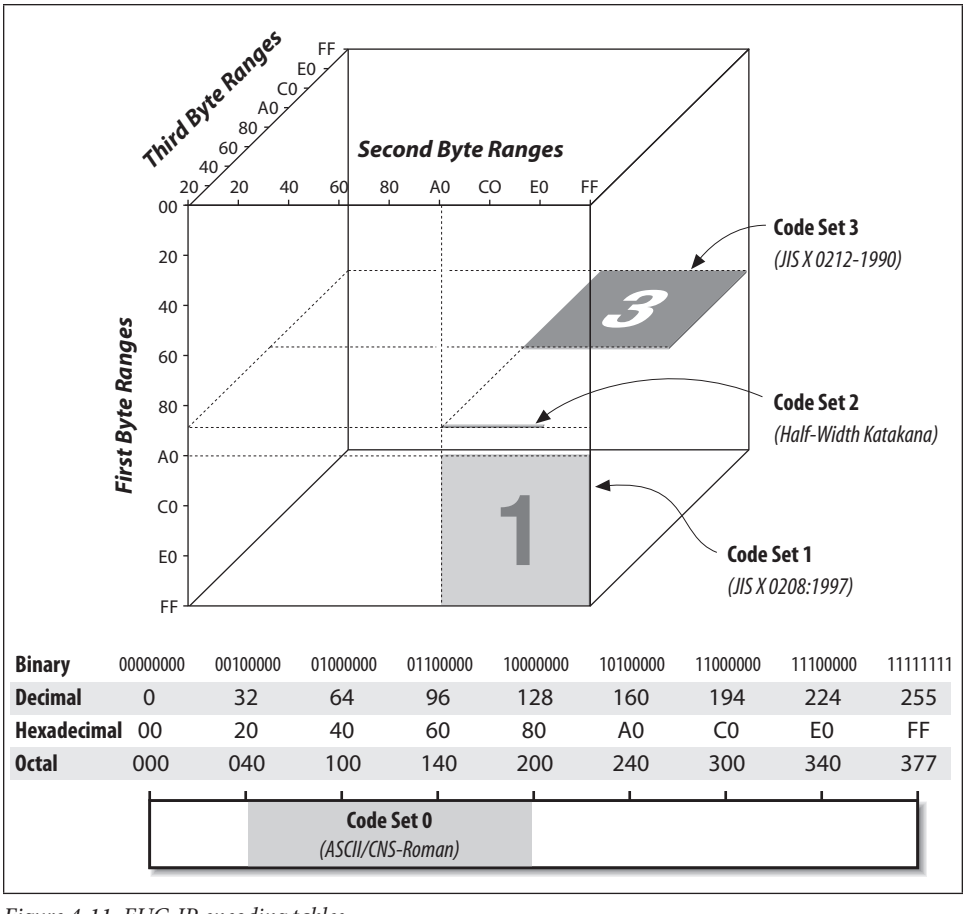
Unlike other CJKV instances of EUC encoding, EUC-JP encoding makes use of all four code sets.

As you already learned, EUC encoding consists of four code sets: the primary code set (code set 0) to which the ASCII/CJKV-Roman character set is assigned, and three supplemental code sets (code sets 1, 2, and 3) that can be specified by the locale and are usually used for non-Latin characters. Table 4-55 lists the code specifications for all the code sets of EUC-JP encoding.

Table 4-55. EUC-JP encoding specifications

Code set	Decimal	Hexadecimal
Code set 0		
Byte range	33–126	21–7E
Code set 1		
First byte range	161–254	A1–FE
Second byte range	161–254	A1–FE
Code set 2		
First byte (SS2)	142	8E
Second byte range	161–223	A1–DF
Code set 3		
First byte (SS3)	143	8F
Second byte range	161–254	A1–FE
Third byte range	161–254	A1–FE

See Figure 4-11 for an illustration of the EUC-JP encoding space. Note how it requires a three-dimensional space.



Binary	00000000	00100000	01000000	01100000	10000000	10100000	11000000	11100000	11111111
Decimal	0	32	64	96	128	160	194	224	255
Hexadecimal	00	20	40	60	80	A0	C0	E0	FF
Octal	000	040	100	140	200	240	300	340	377

Figure 4-11. EUC-JP encoding tables

You may have noticed that, for each byte of some of the code sets, EUC permits up to 96 characters (that is, the range 0xA0 through 0xFF). So why are the encoding ranges in the tables of these sections slightly smaller—specifically 0xA1 through 0xFE—or 94 characters instead of 96? As stated earlier, this is done for the sake of compatibility with character sets and encodings (most notably, ISO-2022-JP) that are based on a 94×94 matrix. This is not to say that code points 0xA0 or 0xFF are invalid, but that there are most likely no characters encoded at those rows.

Let's begin to expand our Japanese encoding example by including the EUC-JP–encoded equivalent of the example string かな漢字, in order to see how this encoding method works and how it compares to ISO-2022-JP encoding. Like before, the encoded values are in hexadecimal notation and correspond to characters in the JIS X 0208:1997 character set encoded in EUC-JP code set 1. Table 4-56 provides this encoding example.

Table 4-56. Japanese encoding example—ISO-2022-JP and EUC-JP encodings

Encoding	String					
Characters		か	な	漢	字	
ISO-2022-JP	1B 24 42	24 2B	24 4A	34 41	3B 7A	1B 28 4A
Escape sequences	<ESC> $ B					<ESC> (J
ISO-2022-JP—visual		$ +	$ J	4 A	; z	
EUC-JP		A4 AB	A4 CA	B4 C1	BB FA	

As shown in the table, there are absolutely no escape sequences or shifting sequences used in EUC-JP encoding as there are in ISO-2022-JP encoding.

Table 4-57 illustrates why ISO-2022 and EUC encodings are referred to as seven- and eight-bit encodings, respectively, by showing the corresponding bit arrays for the four characters of the example string かな漢字.

Table 4-57. Japanese encoding example—ISO-2022-JP and EUC-JP bit arrays

Encoding	String			
Characters	か	な	漢	字
ISO-2022-JP	24 2B	24 4A	34 41	3B 7A
ISO-2022-JP—binary	00100100 00101011	00100100 01001010	00110100 01000001	00111011 01111010
EUC-JP	A4 AB	A4 CA	B4 C1	BB FA
EUC-JP—binary	10100100 00101011	10100100 11001010	10110100 11000001	10111011 11111010

Although hexadecimal notation may not overtly or explicitly indicate the close relationship between ISO-2022 and EUC encodings (unless you are very good at working with hexadecimal digits), their bit arrays most certainly do, because the bit arrays differ only in that the most significant bits are set to "0" for ISO-2022 encodings, and are set to "1" for EUC encodings.

EUC-JP encoding for JIS X 0213:2004

The EUC-JP encoding shown in the JIS X 0213:2004 standard proper, for the characters that are outside of JIS X 0208:1997 and thus specific to JIS X 0213:2004, are *Informative*, not Normative, meaning that they are not meant to be implemented. In fact, I am not aware of any EUC-JP–based implementations of JIS X 0213:2004.

The preferred way in which JIS X 0213:2004 is to be supported is through Unicode. Unicode version 3.2 or greater fully supports JIS X 0213:2004, though 25 of its characters map to a sequence of two Unicode code points.

EUC-KR encoding

The instance of EUC encoding used for the Korean locale is known as EUC-KR encoding and is sometimes referred to as "KS_C_5601-1987" or "eight-bit KS" encoding, though these are incorrect and somewhat dangerous designations in my opinion. This encoding is defined in the standard designated KS X 2901:1992.* Table 4-58 lists what character sets are assigned to each corresponding EUC-KR code set.

Table 4-58. EUC-KR code set allocation

Code set	Character set	Display width	Number of bytes
Code set 0	ASCII or KS-Roman	1	1
Code set 1	KS X 1001:2004	2	2
Code set 2	unused		
Code set 3	unused		

EUC-KR encoding, like EUC-CN encoding covered earlier, does not make use of EUC encoding's code sets 2 and 3. This makes it virtually impossible to distinguish EUC-CN encoding from EUC-KR encoding without the use of any language or locale attribute. Table 4-59 details the specifications for EUC-KR encoding.

Table 4-59. EUC-KR encoding specifications

Code set	Decimal	Hexadecimal
Code set 0		
Byte range	33–126	21–7E
Code set 1		
First byte range	161–254	A1–FE
Second byte range	161–254	A1–FE
Code set 2	unused	unused
Code set 3	unused	unused

Note the similarities with EUC-CN encoding in Table 4-51—apart from character set allocation, the encoding ranges are identical. EUC-KR encoding is expressed visually in Figure 4-9.

Let's now take a peek at some EUC-KR–encoded material, using the same example string used for the ISO-2022-KR section from earlier in this chapter, specifically 김치. Table 4-60 provides the EUC-KR–encoded example string and contrasts it with its ISO-2022-KR–encoded representation. Like before, the encoded values are in hexadecimal notation.

* Previously designated KS C 5861-1992

Table 4-60. Korean encoding example—ISO-2022-KR and EUC-KR encodings

Encoding	String				
Characters			김	치	
ISO-2022-KR	1B 24 29 43	0E	31 68	44 21	0F
Designator sequence	<ESC> $) C				
Shifts		<SO>			<SI>
ISO-2022-KR—visual			1 h	D !	
EUC-KR			B1 E8	C4 A1	

This table shows that EUC-KR encoding does not use a designator sequence nor shifting characters, which are necessary elements for ISO-2022-KR encoding.

Other possible EUC encodings

Although not even officially in existence, one could conceive of EUC-style encodings for North Korea's KPS 9566-97 or Vietnam's TCVN 5773:1993 and TCVN 6056:1995 standards. If an EUC-KP encoding is to spring into existence, it is likely to be similar to EUC-KR. And, a likely encoding name for TCVN 5773:1993 and TCVN 6056:1995 is EUC-VN, but it is unclear how the entire TCVN-Roman character set, specifically the many characters adorned with diacritic marks, would be handled. I suspect that EUC code set 2 could be used, in a way similar to how half-width katakana are handled in EUC-JP encoding.

It needs to be explicitly pointed out that given the extent to which Unicode is used today in OSes and applications, the likelihood of these encodings becoming established is relatively low. There would be very little benefit in doing so, and would also represent a significant step backward, meaning a step in the wrong direction.

EUC versus ISO-2022 encodings

EUC encoding is closely related to ISO-2022 encoding. In fact, every character that can be encoded by an ISO-2022 encoding can be converted to an EUC-encoded equivalent. This relationship leads to better information interchange. Figure 4-12 draws a comparison between ISO-2022 and EUC encodings. It is critical to understand that the relationship between the encoding ranges is nothing more than that of seven- versus eight-bit.

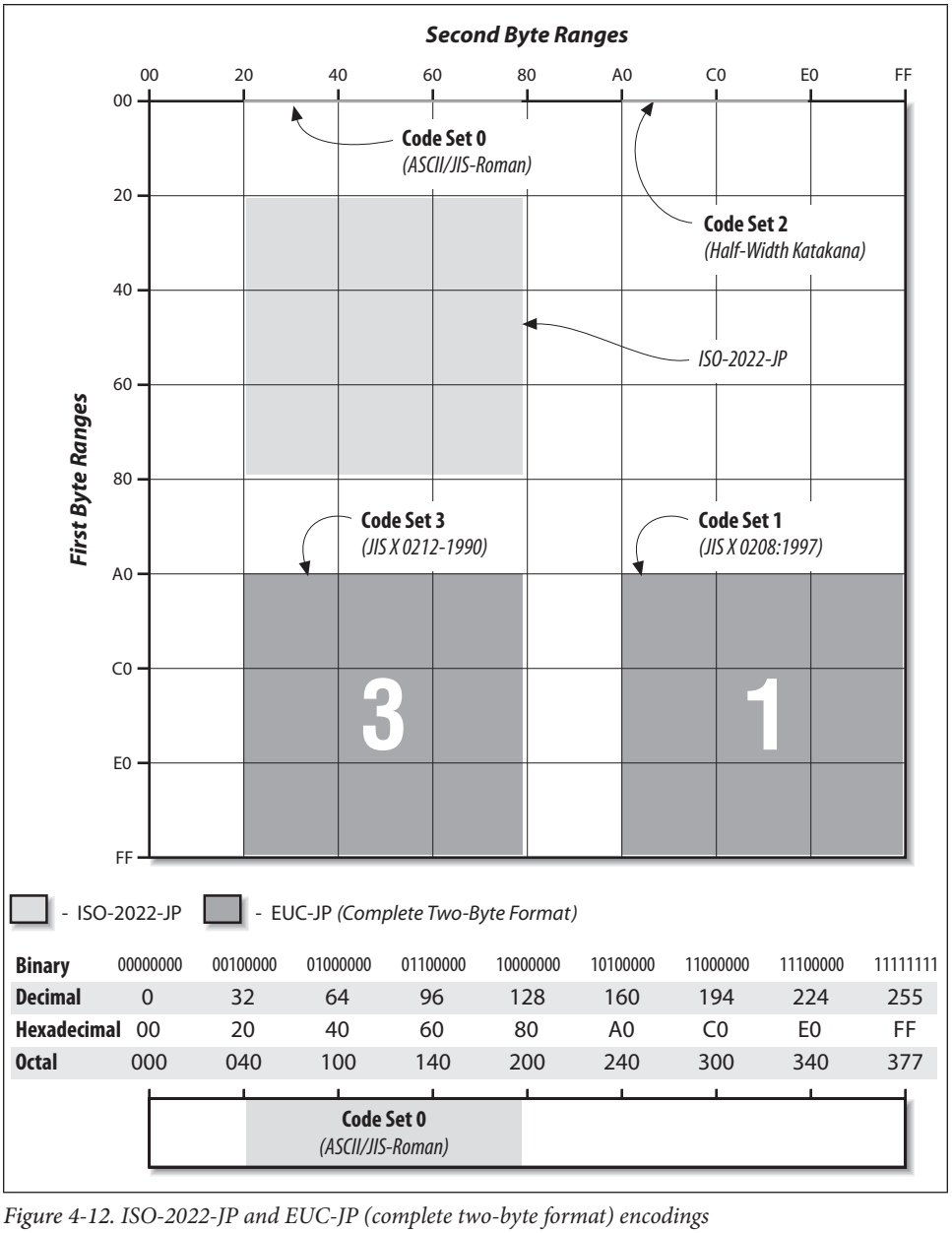


Figure 4-12. ISO-2022-JP and EUC-JP (complete two-byte format) encodings

In most cases, EUC encoding is simply ISO-2022 encoding with the high bits set and without escape or shift sequences of any kind.* Algorithms for code conversion are discussed in Chapter 9.

Locale-Specific Legacy Encoding Methods

Included here are descriptions of legacy encoding methods used for specific locales. I find it intriguing that all five of these CJKV locales each have at least one locale-specific encoding method. Table 4-61 lists some CJKV character set standards, along with their locale-specific encoding methods.

Table 4-61. Locale-specific legacy encoding methods

Character set	Encoding method	Locale
GBK	GBK	China
GB 18030-2005	GB 18030	China
Big Five[a]	Big Five	Taiwan
Big Five Plus	Big Five Plus	Taiwan
Hong Kong SCS-2008	Big Five	Hong Kong
JIS X 0208:1997[b]	Shift-JIS	Japan
KS X 1001:2004	Johab	Korea

a. Keep in mind that CNS 11643-2007 Planes 1 and 2 are equivalent to Big Five.

b. JIS X 0213:2004 is explicitly excluded from this table, because the Shift-JIS encoding for the characters specific to JIS X 0213:2004, meaning those characters beyond JIS X 0208:1997, is Informative, not Normative. JIS X 0213:2004 is expected to be implemented through the use of Unicode and its encodings.

The following sections cover each of these encoding methods in significant detail, and some also provide encoding examples similar to those found in earlier sections. Comparisons are also drawn between these encoding methods, specifically with each other, or with encoding methods that were already covered in this chapter.

GBK encoding—EUC-CN extension

GBK encoding was originally planned as a Normative annex of GB 13000.1-93 as a way to encode the Chinese subset of ISO 10646-1:1993. The "K" in "GBK" represents the letter for the first sound of the Chinese word that means "extension," specifically 扩展 (*kuòzhǎn*). GBK encoding has been implemented as the system Code Page for the Chinese (PRC) versions of Microsoft's Windows 95 and IBM's OS/2.

* Some may claim that EUC encoding's SS2 and SS3 characters are shift characters of sorts. It is "officially okay" to think so.

GBK is divided into five levels as indicated in Table 4-62, which also indicates their encoding ranges, the total number of code points, and the total number of characters that are encoded.

Table 4-62. GBK's five levels

GBK level	Encoding range	Total code points	Total characters
GBK/1[a]	A1 A1–A9 FE	846	717
GBK/2[b]	B0 A1–F7 FE	6,768	6,763
GBK/3	81 40–A0 FE	6,080	6,080
GBK/4	AA 40–FE A0	8,160	8,160
GBK/5	A8 40–A9 A0	192	166

a. Equivalent to the non-hanzi found in both GB 2312-80 and GB/T 12345-90, but without 10 vertical variants found in GB/T 12345-90's row 6. Also, lowercase Roman numerals 1 through 10 were added to the beginning of GB 2312-80's row 2.

b. Equivalent to GB2312-80's two levels of hanzi.

GBK also supports up to 1,894 user-defined code points, separated into three encoding ranges, as indicated in Table 4-63.

Table 4-63. GBK's user-defined regions

Encoding range	Total code points
AA A1–AF FE	564
F8 A1–FE FE	658
A1 40–A7 A0	672

GBK thus includes a total of 23,940 code points, 21,886 of which have characters assigned. Table 4-64 details the complete specifications for GBK encoding.

Table 4-64. GBK encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or GB-Roman		
Byte range	33–126	21–7E
GBK		
First byte range	129–254	81–FE
Second byte ranges	64–126, 128–254	40–7E, 80–FE

GBK encoding was rendered obsolete by GB 18030 encoding, which is described in the next section.

GB 18030 encoding—GBK extension

GB 18030 encoding is unique among the locale-specific encodings because it is the only one that is code-point–compatible with Unicode. This compatibility is by design and is a good thing. GB 18030 encoding is an extension of GBK encoding, and as you have learned, GBK is an extension of EUC-CN encoding. What this effectively means is that the way in which a character is encoded according to EUC-CN encoding is preserved in GBK encoding, and thus in GB 18030 encoding. Likewise, the way in which a character is encoded according to GBK encoding is preserved in GB 18030 encoding.

GB 18030 is a mixed one-, two-, and four-byte encoding method. Its one- and two-byte regions are identical to GBK encoding. Table 4-65 details the GB 18030 encoding specifications.

Table 4-65. GB 18030 encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or GB-Roman		
Byte range	33–126	21–7E
Two-byte encoding—identical to GBK encoding with 23,940 code points		
First byte range	129–254	81–FE
Second byte ranges	64–126, 128–254	40–7E, 80–FE
Four-byte encoding—1,587,600 code points		
First byte range	129–254	81–FE
Second byte range	48–57	30–39
Third byte range	129–254	81–FE
Fourth byte range	48–57	30–39

GB 18030 thus includes a total of 1,611,668 code points. Staggering? Yes, but certainly not overwhelming.

GB 18030 versus GBK versus EUC-CN encodings

Note that the EUC-CN code set 1 encoding range, <A1 A1> through <FE FE>, forms a subset of GBK encoding. This is by design and has the benefit of providing backward compatibility with EUC-CN encoding. Figure 4-13 illustrates the relationship between EUC-CN and GBK encodings.

Likewise, GBK's 23,940 code points form a subset of GB 18030 encoding. Again, this is by design and provides backward compatibility with GBK (and thus EUC-CN) encoding.

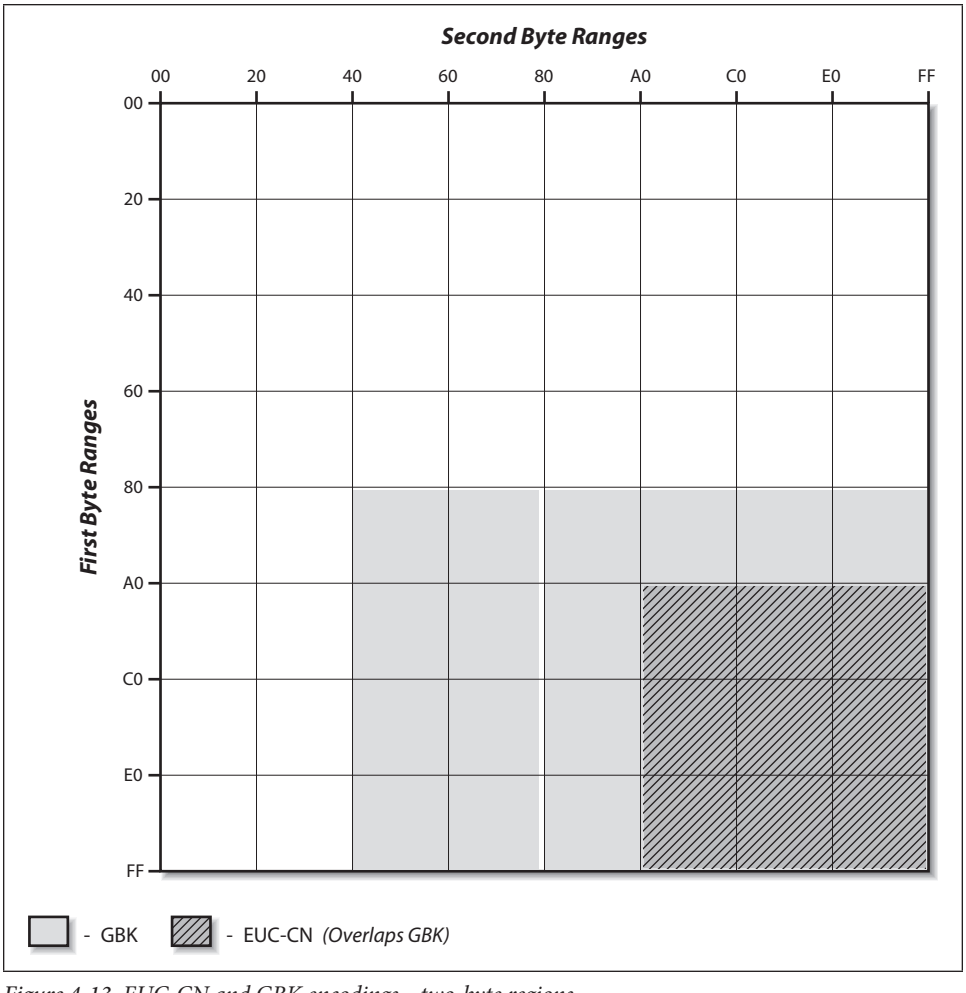


Figure 4-13. EUC-CN and GBK encodings—two-byte regions

GB 18030 versus Unicode encoding forms

There has been some speculation as to why GB 18030 encoding has a greater number of code points than Unicode. Consider the following facts:

- GB 18030 encoding includes a total of 1,611,668 code points, specifically 128 in its 1-byte region, 23,940 in its two-byte region, and 1,587,600 in its four-byte region.
- Unicode encoding forms support a total of 1,112,064 code points, specifically 63,488 in the BMP, and 1,048,576 in its 16 additional planes.

Conspiracy theories aside, GB 18030 encoding includes more code points than Unicode simply because its four-byte extension needed to be at least as large as Unicode in order

to be compatible with it. The GB 18030 code points that do not map to Unicode are considered unusable.

Table 4-66 demonstrates how GB 18030 encoding is both backward-compatible with EUC-CN and GBK encodings, and code-point–compatible with Unicode. The table also shows how the two- and four-byte regions of GB 18030 relate to BMP versus non-BMP coverage when the same characters are represented in Unicode.

Table 4-66. Chinese encoding example—EUC-CN, GBK, GB 18030, and Unicode encodings

Encoding	Examples			
Character	一	龥	北	乛
EUC-CN	B0 A1	n/a	n/a	n/a
GBK	B0 A1	FD 9B	n/a	n/a
GB 18030	B0 A1	FD 9B	81 39 EE 39	95 32 82 36
UTF-32BE	00 00 4E 00	00 00 9F A5	00 00 34 00	00 02 00 00
UTF-16BE	4E 00	9F A5	34 00	D8 40 DC 00
UTF-8	E4 B8 80	E9 BE A5	E3 90 80	F0 A0 80 80

This table suggests that ideographs that are in CJK Unified Ideographs Extension A map to the four-byte region of GB 18030 encoding. But, as I pointed out in Chapter 3, of the 6,582 ideographs in Extension A, 52 were in already included in GBK. This means that the vast majority of Extension A characters map to the four-byte region of GB 18030, although not all do. But, all of the ideographs in CJK Unified Ideographs URO map to the GBK-compatible two-byte region of GB 18030 encoding.

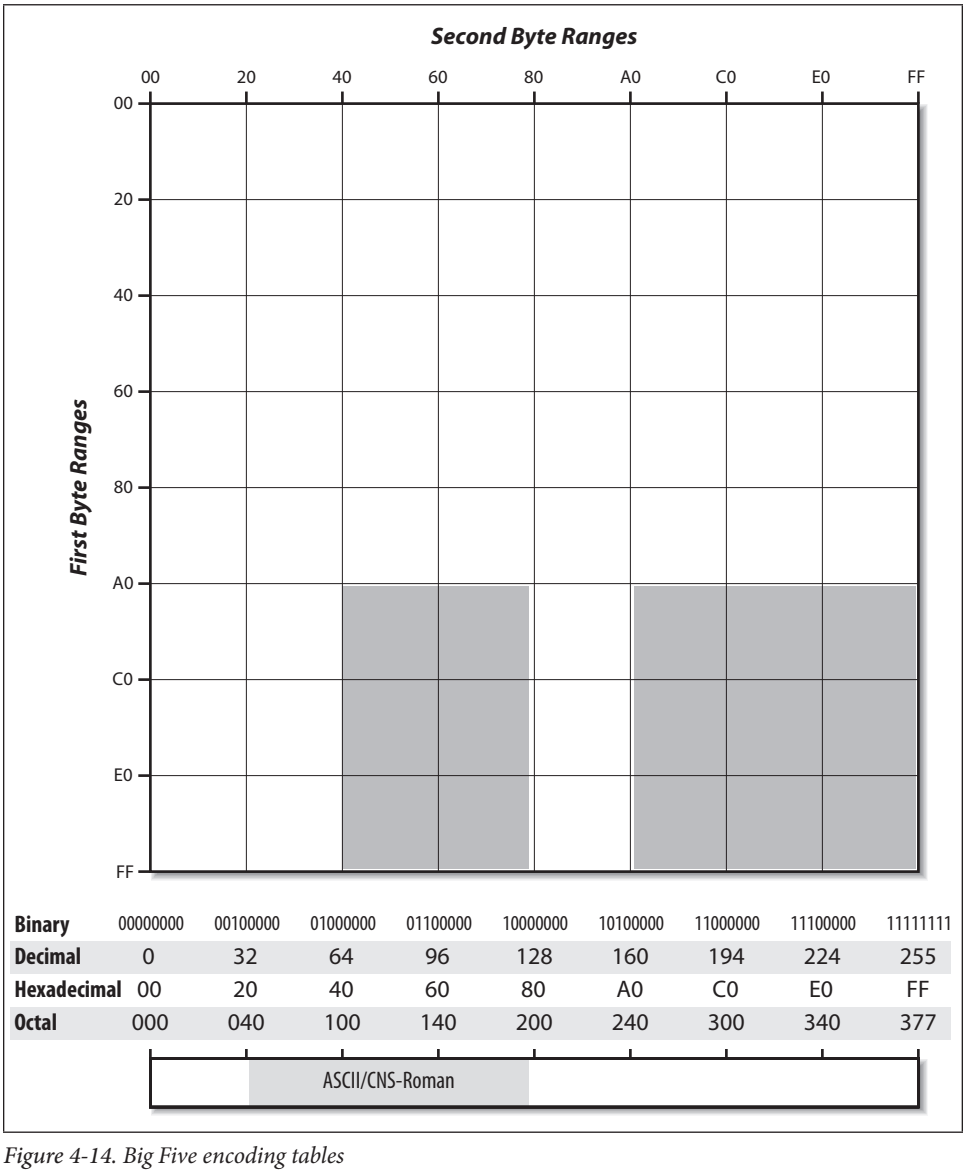
Big Five encoding

Big Five encoding has a lot in common with EUC-TW code sets 0 and 1 in terms of character repertoire. The main difference, encoding-wise, is that there is an additional encoding block used by Big Five, and that Big Five uses only one plane. This additional encoding block is required because the Big Five character set contains over 13,000 characters—EUC-TW code set 1 simply cannot encode that many characters. Table 4-67 illustrates its encoding specifications.

Table 4-67. Big Five encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or CNS-Roman		
Byte range	33–126	21–7E
Big Five		
First byte range	161–254	A1–FE
Second byte ranges	64–126, 161–254	40–7E, A1–FE

Figure 4-14 illustrates the Big Five encoding structure, which clearly shows its two separate encoding regions.



Binary	00000000	00100000	01000000	01100000	10000000	10100000	11000000	11100000	11111111
Decimal	0	32	64	96	128	160	194	224	255
Hexadecimal	00	20	40	60	80	A0	C0	E0	FF
Octal	000	040	100	140	200	240	300	340	377

Figure 4-14. Big Five encoding tables

Big Five versus EUC-TW encodings

It seems a bit silly to compare Big Five and EUC-TW encodings because they are so different from one another. Big Five encoding, on the one hand, is a mixed one- and two-byte

encoding whose second-byte values extend into the seven-bit region. EUC-TW, on the other hand, is a mixed one-, two-, and four-byte encoding that is fundamentally made up of planes. They are compatible only in that some parts are equivalent: Big Five Levels 1 and 2 are equivalent to CNS 11643-2007 Planes 1 and 2.

Big Five encoding—Hong Kong SCS-2008

Hong Kong SCS-2008, which is implemented as an extension to Big Five encoding, includes 5,009 characters, 4,568 of which are hanzi. The characters that are specific to Hong Kong SCS-2008 are encoded in two separate regions within Big Five encoding, one of which is considered a nonstandard or extended region. The portion of Hong Kong SCS-2008 that is encoded in the standard Big Five encoding region spans rows 0xFA through 0xFE. Because this Big Five encoding extension includes a nonstandard region, specifically the range that spans rows 0x87 through 0xA0, not all OSes should be expected to support Hong Kong SCS-2008, at least in the context of Big Five encoding.

Table 4-68 provides the specification for the Hong Kong SCS-2008 extension to Big Five encoding. This should be added to Table 4-67 to come up with a complete encoding definition.

Table 4-68. Hong Kong SCS-2008 Big Five encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or CNS-Roman		
Byte range	33–126	21–7E
Big Five—Hong Kong SCS-2008 extension		
First byte range	135–254	87–FE
Second byte ranges	64–126, 161–254	40–7E, A1–FE

Be aware that Hong Kong SCS-2008 extension to Big Five encoding conflicts with Big Five Plus encoding, which is described next. It also conflicts with some vendor-specific instances of Big Five encoding—in particular, that used by Apple's Mac OS-T.

Big Five Plus encoding—another Big Five extension

Due to influences from Unicode and CNS 11643-2007, the Big Five character set has recently been expanded to include additional characters, mostly hanzi. This necessitated an expansion of the encoding space. This expanded version of Big Five is known as Big Five Plus. Table 4-69 lists the complete encoding specification for Big Five Plus encoding.

Table 4-69. Big Five Plus encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or CNS-Roman		
Byte range	33–126	21–7E
Big Five Plus		
First byte range	129–254	81–FE
Second byte ranges	64–126, 128–254	40–7E, 80–FE

Look familiar? We'll draw a comparison between Big Five Plus and another encoding in the next section. The "standard" Big Five characters (including the ETen extensions as described in Appendix E) are encoded in the two-byte encoding range as provided in Table 4-67.

GBK versus Big Five versus Big Five Plus encodings

While I have normally preferred to compare encodings within a single locale in this chapter, this is a perfect time to draw comparisons between GBK and Big Five Plus encodings. They share many of the same attributes, but how they were designed pinpoints their differences. GBK began with EUC-CN code set 1 as its base, but Big Five Plus began with Big Five as its base.

Although GBK and Big Five Plus share the same overall encoding structure, their character allocation in the two-byte region is completely different. Figure 4-15 illustrates how Big Five encoding is a subset of Big Five Plus encoding. Compare this with Figure 4-13.

Shift-JIS encoding—JIS X 0208:1997

Shift-JIS encoding, originally codeveloped by ASCII Corporation* and Microsoft Corporation, was widely implemented as the internal code for a variety of platforms, including Japanese PCs and Mac OS-J (including the Japanese Language Kit). Shift-JIS is sometimes referred to as MS (an abbreviation for Microsoft) Kanji, MS Code, or SJIS (an abbreviated form of "Shift-JIS").† Historically, Shift-JIS was so named because of the way the code points for two-byte characters effectively "shifted" around the code points for half-width katakana. Japanese PC users were originally restricted to only half-width katakana, so Shift-JIS was developed in order to maintain backward compatibility with its code points. It was not until 1997 that the definition of Shift-JIS encoding became an official part of the JIS X 0208 standard. JIS X 0208:1997 contains its definition.

* ASCII here refers to the Japan-based company (*http://www.ascii.co.jp/*), not the character set.

† There are also less flattering permutations of this encoding's name, but they are not suitable for printing in this book.

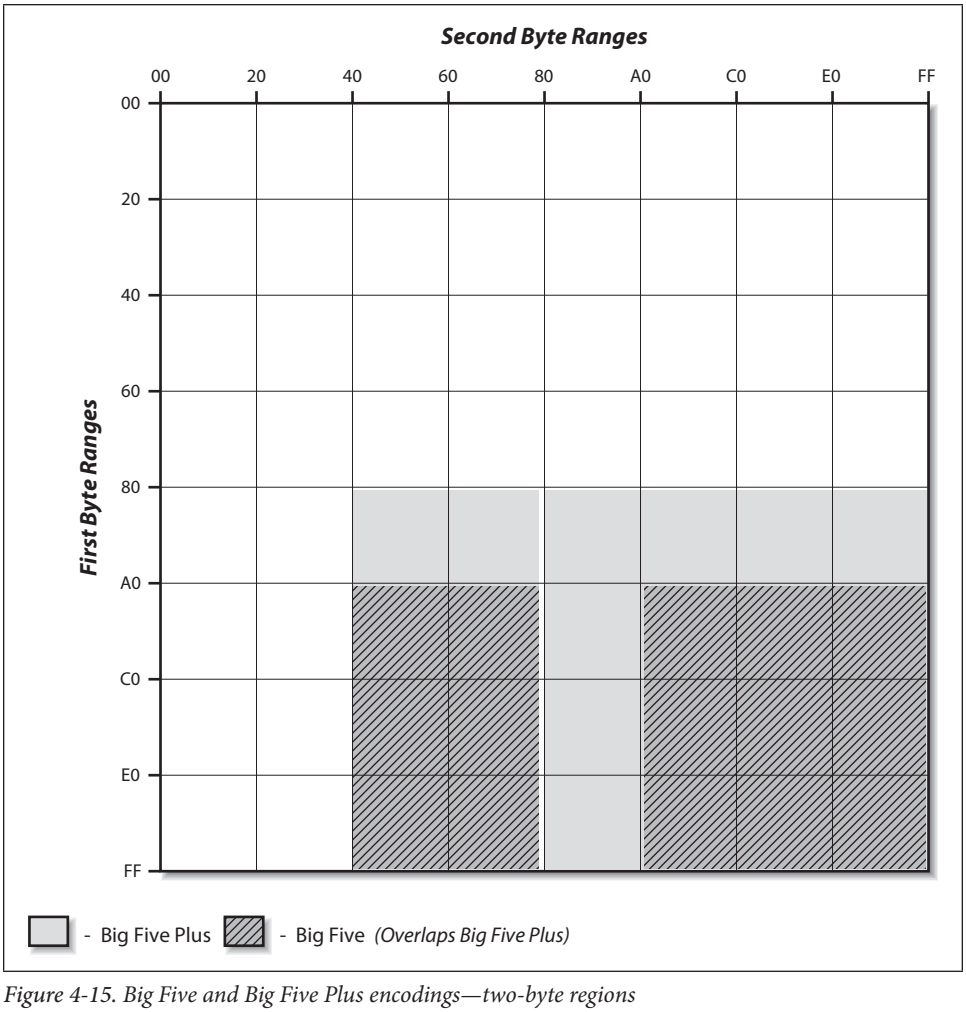


Figure 4-15. Big Five and Big Five Plus encodings—two-byte regions

The following list provides more examples of OSes and environments that can process Shift-JIS code internally:

- Virtually all Japanese PCs
- Some Unix implementations, such as HP-UX, AIX, and NEWS
- Japanese TEX (ASCII Corporation version)
- Mac OS-J and Mac OS with JLK*

* Japanese Language Kit

What follows is an explanation of how Shift-JIS encoding works. A two-byte character in Shift-JIS encoding is initiated with a byte having a decimal value of 129 through 159 or 224 through 239 (in hexadecimal, 0x81 through 0x9F or 0xE0 through 0xEF). This byte is subsequently treated as the first byte of an expected two-byte sequence. The following (second) byte must have a decimal value of 64 through 126 or 128 through 252 (in hexadecimal, 0x40 through 0x7E or 0x80 through 0xFC). Note that the first byte's range falls entirely in the extended ASCII range—in the eight-bit encoding range with the high-order bit turned on. Shift-JIS also encodes half-width katakana and ASCII/JIS-Roman. Table 4-70 provides the specifications for Shift-JIS encoding.

Table 4-70. Shift-JIS encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or JIS-Roman		
Byte range	33–126	21–7E
Half-width katakana		
Byte range	161–223	A1–DF
JIS X 0208:1997		
First byte ranges	129–159, 224–239	81–9F, E0–EF
Second byte ranges	64–126, 128–252	40–7E, 80–FC

Let's now complete our Japanese encoding example by including the Shift-JIS–encoded equivalent of the example string かな漢字, in order to better illustrate how this encoding method works, and perhaps more importantly, how it compares to ISO-2022-JP and EUC-JP encodings; see Table 4-71. Like before, the encoded values are in hexadecimal notation, and they correspond to characters in the JIS X 0208:1997 character set.

Table 4-71. Japanese encoding example—ISO-2022-JP, EUC-JP, and Shift-JIS encodings

Encoding	String					
Characters		か	な	漢	字	
ISO-2022-JP	1B 24 42	24 2B	24 4A	34 41	3B 7A	1B 28 4A
Escape sequences	<ESC> $ B					<ESC> (J
ISO-2022-JP—visual		$ +	$ J	4 A	; z	
EUC-JP		A4 AB	A4 CA	B4 C1	BB FA	
Shift-JIS		82 A9	82 C8	8A BF	8E 9A	

Note that no escape sequences are used for Shift-JIS encoding. This is typical of nonmodal encodings and produces a much tighter or smaller encoding, in terms of the total number of bytes required to represent the entire text string. There is, however, no ASCII representation for the two bytes that constitute these Shift-JIS code points.

Shift-JIS encoding does not support the characters defined in JIS X 0212-1990. There is simply not enough encoding space left in Shift-JIS to accommodate these characters, and there is currently no plan to extend Shift-JIS in a manner such that JIS X 0212-1990 can be accommodated. See Figure 4-16 for an illustration of the Shift-JIS encoding space.

Some definitions (in particular, corporate definitions) of Shift-JIS encoding also contain encoding blocks for user-defined characters, or even a code point for the half-width katakana "space" character. Such encoding blocks and code points are not useful if true information interchange is desired, because they are encoded in such a way that they do not convert to code points in other Japanese encoding methods, such as ISO-2022-JP and EUC-JP encodings. Table 4-72 lists these nonstandard Shift-JIS encoding blocks and code points.

Table 4-72. Shift-JIS user-defined character encoding specifications

Encoding	Decimal	Hexadecimal
Half-width katakana		
Half-width "space" character	160	A0
User-defined characters		
First byte range[a]	240–252	F0–FC
Second byte ranges	64–126, 128–252	40–7E, 80–FC

a. Some implementations of Shift-JIS encoding implement a smaller user-defined character range, such as rows 0xF0 through 0xF9 or 0xF0 through 0xFB.

Note how the second-byte range is unchanged from the standard definition of Shift-JIS—only the first-byte range differs for the user-defined range.

Figure 4-17 illustrates the standard Shift-JIS encoding space, along with the user-defined character region.

Shift-JIS encoding for JIS X 0213:2004

The Shift-JIS encoding shown in the JIS X 0213:2004 standard proper, for the characters that are outside of JIS X 0208:1997 and thus specific to JIS X 0213:2004, are Informative, not Normative, meaning that they are not meant to be implemented. In fact, I am not aware of any Shift-JIS–based implementations of JIS X 0213:2004.

The preferred way in which JIS X 0213:2004 is to be implemented is through Unicode. Unicode version 3.2 or greater fully supports JIS X 0213:2004, though 25 of its characters map to a sequence of two Unicode code points.

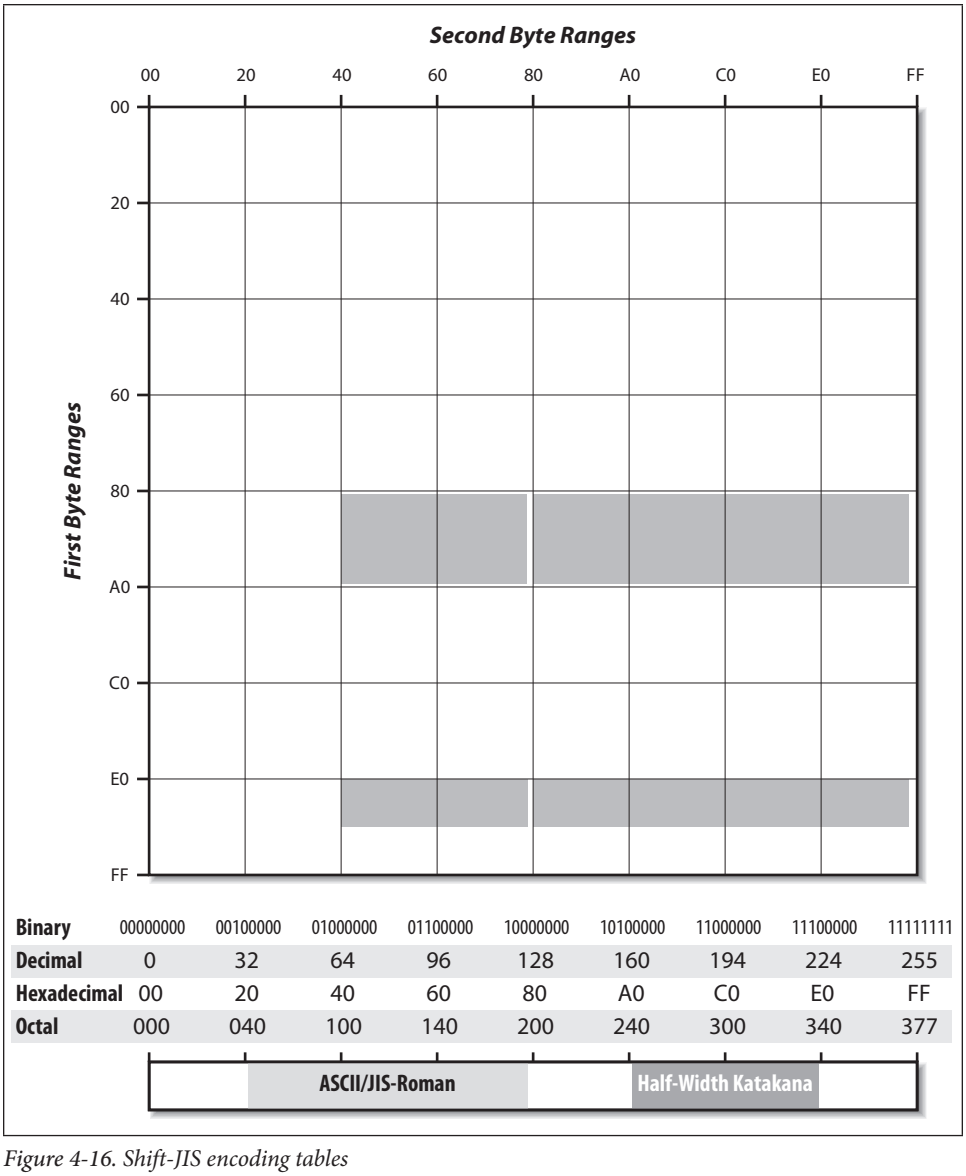


Figure 4-16. Shift-JIS encoding tables

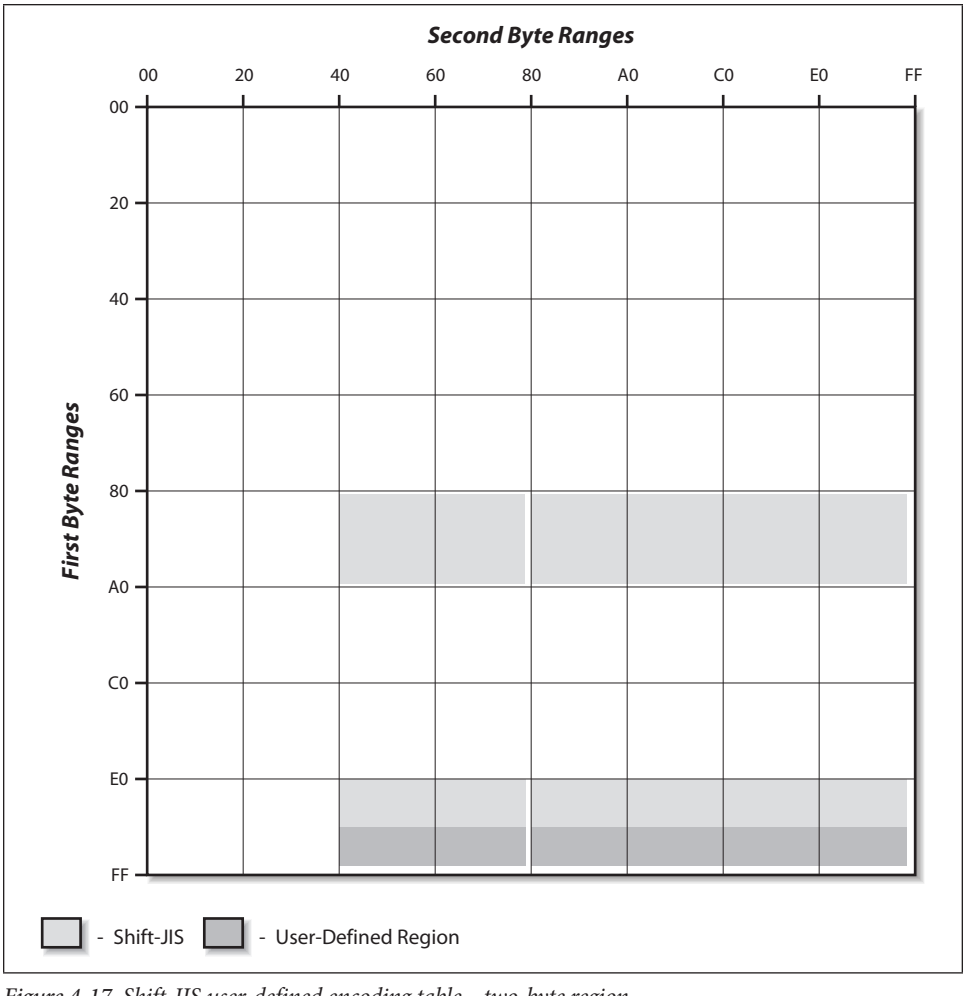


Figure 4-17. Shift-JIS user-defined encoding table—two-byte region

Shift-JIS versus ISO-2022-JP versus EUC-JP encodings

The relationship between Shift-JIS and EUC-JP encodings is not very apparent and requires the use of a somewhat complex code conversion algorithm, examined in detail in Chapter 9. Figure 4-18 illustrates the two-byte Shift-JIS encoding space and how it relates to EUC-JP and ISO-2022-JP encodings.

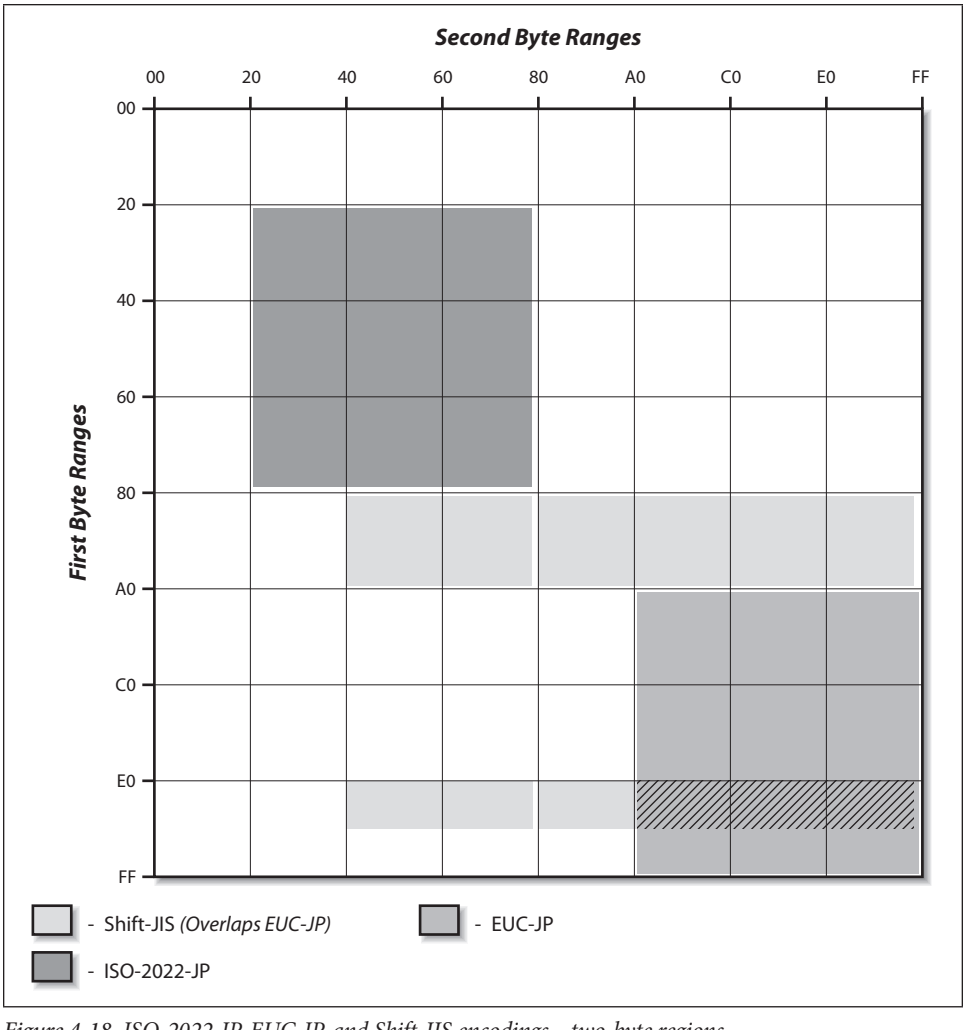


Figure 4-18. ISO-2022-JP, EUC-JP, and Shift-JIS encodings—two-byte regions

Johab encoding—KS X 1001:2004

The Korean character standard, KS X 1001:2004, is encoded very similarly to JIS X 0208:1997. It is supported by ISO-2022-KR and EUC-KR encodings, which are the Korean analogs to ISO-2022-JP and EUC-JP encodings. KS X 1001:2004 departs from JIS X 0208:1997 only in that we have not yet discussed an encoding method comparable to Shift-JIS encoding. There is a comparable encoding method for KS X 1001:2004, referred to as *Johab* (조합/組合 *johap*) encoding. Johab encoding is described in the KS X 1001:2004 standard as an alternate encoding that includes all possible modern hangul syllables, specifically 11,172, which is 8,822 more than can be encoded according to ISO-2022-KR or EUC-KR encodings. It's possible to encode these remaining 8,822 hangul

syllables using ISO-2022-KR or EUC-KR encodings through a KS X 1001:2004 provision that specifies them as jamo sequences, but this is not widely supported.

The *Unified Hangul Code* (UHC) character set and encoding—fully described in Appendixes E and F—is character-for-character identical to Johab encoding in terms of the character set that it supports, but its encoding is radically different. UHC encoding is designed to be backward compatible with EUC-KR encoding and forward compatible with Unicode—two good qualities. Johab is only forward compatible with Unicode. Table 4-73 provides the Johab encoding specifications as described in the KS X 1001:2004 standard.

Table 4-73. Johab encoding specifications

Encoding	Decimal	Hexadecimal
ASCII or KS-Roman		
Byte range	33–126	21–7E
Hangul syllables and 51 modern jamo[a]		
First byte range	132–211	84–D3
Second byte ranges	65–126, 129–254	41–7E, 81–FE
Symbols, hanja, and 42 ancient jamo[b]		
First byte ranges	216–222, 224–249	D8–DE, E0–F9
Second byte ranges	49–126, 145–254	31–7E, 91–FE

a. These 51 modern jamo are KS X 1001:2004 04-01 through 04-51.

b. These 42 ancient jamo are KS X 1001:2004 04-53 through 04-94.

Note how the hangul encoding range is quite different from that of the symbol and hanja encoding range. They both encode up to 188 characters per row, but the exact encoding range that defines these 188 code points is quite different. Figure 4-19 illustrates the two different two-byte encoding ranges of Johab encoding.

Also note how the hangul encoding range defines 15,040 code points (80 rows of 188 code points each), which means that the encoding of the 11,172 hangul is not contiguous.

The hangul portion of Johab encoding is fundamentally based upon three five-bit segments. Five bits are used to represent the three basic positions of the jamo that constitute a single hangul. Five bits, of course, can encode up to 32 unique entities. Knowing that there are 19 initial jamo (consonants), 21 medial jamo (vowels), and 27 final jamo (consonants; there are actually 28, to support the "fill" character for 2-jamo hangul), we can see that 5 bits can easily represent the number of unique jamo for each of the three positions. But three 5-bit units become 15 bits. The 16th bit, which is the most significant bit (meaning the first bit, not the 16th), is always set, meaning that its value is set to "1."

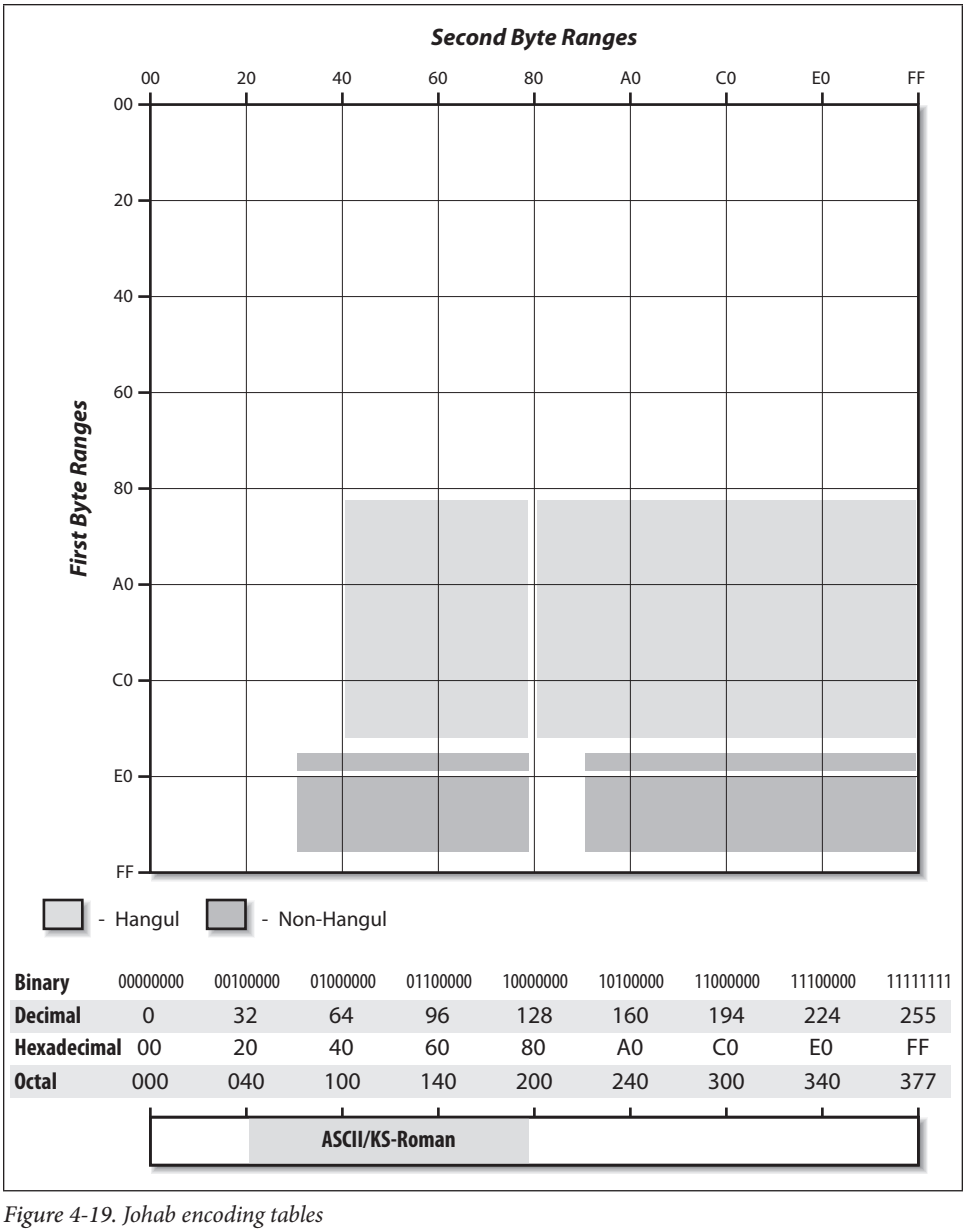


Figure 4-19. Johab encoding tables

Table 4-74 lists the 32 different binary patterns (bit arrays), along with the jamo they represent in the three positions for composing hangul.

Table 4-74. Johab encoding's 5-bit binary patterns

Binary pattern	Initial	Medial	Final
00000	unused	unused	unused
00001	"fill"	unused	"fill"
00010	ㄱ	"fill"	ㄱ
00011	ㄲ	ㅏ	ㄲ
00100	ㄴ	ㅐ	ㄳ
00101	ㄷ	ㅑ	ㄴ
00110	ㄸ	ㅒ	ㄵ
00111	ㄹ	ㅓ	ㄶ
01000	ㅁ	unused	ㄷ
01001	ㅂ	unused	ㄹ
01010	ㅃ	ㅔ	ㄺ
01011	ㅅ	ㅕ	ㄻ
01100	ㅆ	ㅖ	ㄼ
01101	ㅇ	ㅗ	ㄽ
01110	ㅈ	ㅘ	ㄾ
01111	ㅉ	ㅙ	ㄿ
10000	ㅊ	unused	ㅀ
10001	ㅋ	unused	ㅁ
10010	ㅌ	ㅚ	unused
10011	ㅍ	ㅛ	ㅂ
10100	ㅎ	ㅜ	ㅄ
10101	unused	ㅝ	ㅅ
10110	unused	ㅞ	ㅆ
10111	unused	ㅟ	ㅇ
11000	unused	unused	ㅈ
11001	unused	unused	ㅊ
11010	unused	ㅠ	ㅋ
11011	unused	ㅡ	ㅌ
11100	unused	ㅢ	ㅍ
11101	unused	ㅣ	ㅎ
11110	unused	unused	unused
11111	unused	unused	unused

Table 4-75 provides some examples of hangul that are encoded according to Johab encoding, and it demonstrates how the binary patterns of the jamo from which they are

composed are used to derive the complete or final encoded value. Note how individual modern jamo are encoded using this scheme through the use of two "fill" elements—hangul can use up to one "fill" element, for those that are composed of only two jamo.

Table 4-75. Composing hangul syllables from jamo according to johab encoding

Hangul	Bit 1	Bits 2–6	Bits 7–11	Bits 12–16	Johab encoding
가	1	00010	00011	00001	88 61—10001000 01100001
김	1	00010	11101	10001	8B B1—10001011 10110001
치	1	10000	11101	00001	C3 A1—11000011 10100001
ᄀ	1	00010	00010	00001	88 41—10001000 10100001
ᅡ	1	00001	00011	00001	84 61—10000100 01100001
ᄅᄀ	1	00001	00010	01010	84 4A—10000100 01001010

It is critical to understand that KS X 1001:2004 row 4 (the jamo) has special treatment according to Johab encoding. The first 51 characters, specifically 04-01 through 04-51, are encoded according to the standard Johab hangul encoding scheme, as illustrated in the last three examples of Table 4-75. The remaining 43 characters, specifically 04-52 through 04-94, are encoded according to the mapping as described in the KS X 1001:2004 manual.

Johab versus EUC-KR encodings

The relationship between Johab and EUC-KR encodings is an interesting one. First, their hangul portions are incompatible in that EUC-KR encoding can encode only 2,350. Johab encoding can encode all 11,172 hangul, which means an additional 8,822 hangul beyond the set of 2,350 that are supported by EUC-KR encoding. Their symbol and hanja portions, however, are compatible, and there is a well-understood and convenient code conversion algorithm for converting between them. Figure 4-20 illustrates how the encoding structure of Johab and EUC-KR encodings differ.

A complete machine-readable mapping table that provides correspondences between Johab encoding and Unicode is available.* Other mapping tables provide correspondences with other encodings for the hangul† and non-hangul‡ portions of the Korean character set.

* *http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/KSC/JOHAB.TXT*

† *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/map/hangul-codes.txt*

‡ *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/map/non-hangul-codes.txt*

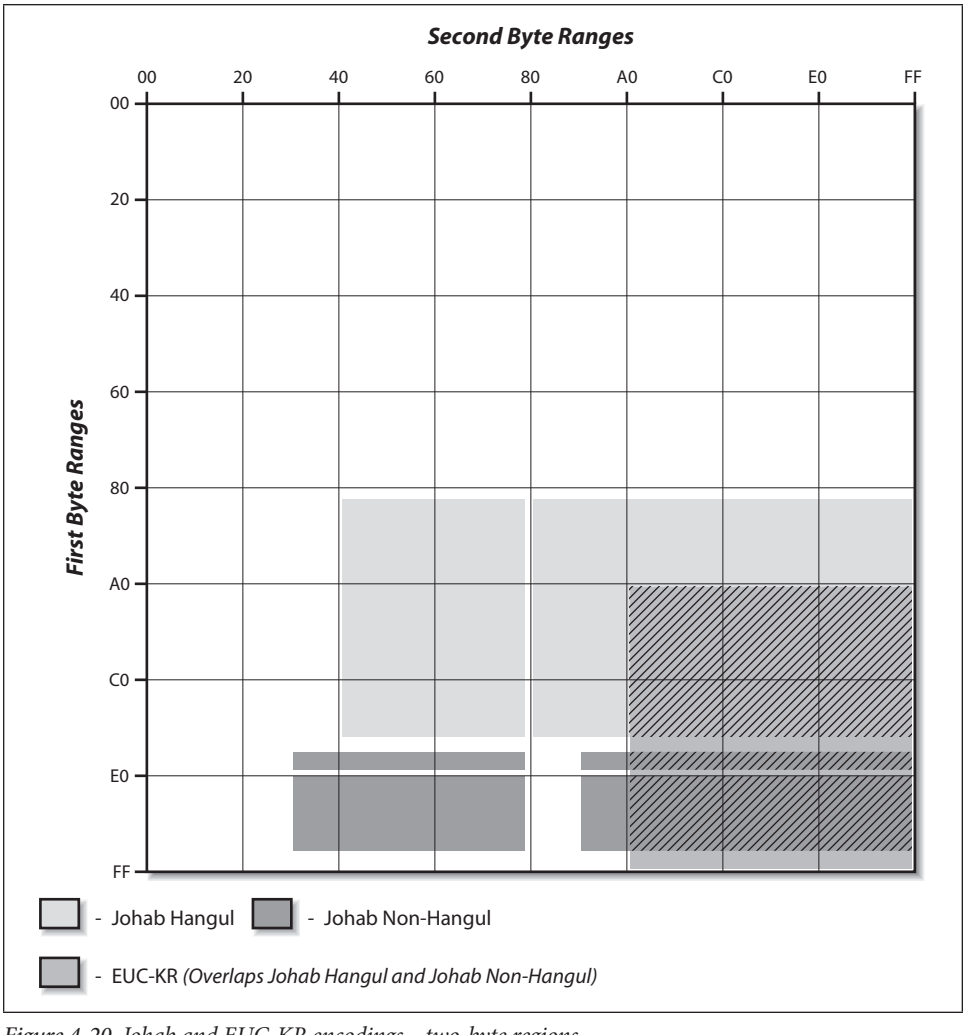


Figure 4-20. Johab and EUC-KR encodings—two-byte regions

Comparing CJKV Encoding Methods

The repertoire of characters, especially those of Chinese origin (meaning the ideographs), in the CJKV character set standards overlap considerably, although the arrangement and ordering of the characters is markedly different. This means that any attempt at conversion between them must be done through the use of mapping tables, and there may be cases—sometimes hundreds or thousands—when characters in one standard do not exist in another.

Table 4-76 lists the encoded values for the two ideographs 漢 (U+6F22 or U+FA47) and 字 (U+5B57). I am listing them under the most common encoding methods for each locale. Note how there is no correspondence among the encodings of these two characters across the different character sets.

Table 4-76. Ideographs encoded according to legacy CJKV encodings

Locale—character set	ISO-2022 encoding	Eight-bit encodings	
China—GB 2312-80, GBK, and GB 18030-2005		**EUC-CN**	
汉	3A 3A	BA BA	
字	57 56	D7 D6	
Taiwan—CNS 11643-2007 and Big Five		**EUC-TW**	**Big Five**
漢	69 47	E9 C7	BA 7E
字	47 73	C7 F3	A6 72
Japan—JIS X 0208:1997 and JIS X 0213:2004		**EUC-JP**	**Shift-JIS**
漢	34 41	B4 C1	8A BF
漢[a]	77 25	F7 A5	EC 44
字	3B 7A	BB FA	8E 9A
South Korea—KS X 1001:2004		**EUC-KR**	**Johab**
漢	79 53	F9 D3	F7 D3
字	6D 2E	ED AE	F1 AE
North Korea—KPS 9566-97			
漢	72 53	F2 D3	
字	66 2F	E6 AF	
Vietnam—TCVN 6056:1995			
漢	5D 3E	DD BE	
字	52 3E	D2 BE	

a. The EUC-JP and Shift-JIS encoding for this kanji, <F7 A5> and <EC 44>, respectively, which is specific to JIS X 0213:2004, is Informative, not Normative, which effectively means that they are not implemented. Unicode is the preferred way in which JIS X 0213:2004 is implemented.

Although these ideographs look nearly identical across locales (except for 汉, which is the simplified form of 漢), their character codes (in other words, their encodings) are quite different.

Now, let us consider Unicode. There are multiple forms of 漢 to consider, specifically its simplified form 汉 (U+6C49), and the CJK Compatibility Ideograph 漢 (U+FA47) that is specific to Japanese. Table 4-77 shows how these ideographs are encoded according to Unicode.

Table 4-77. Ideographs encoded according to Unicode

Ideograph	Unicode	UTF-8	UTF-16BE	UTF-32BE
漢	U+6F22	E6 BC A2	6F 22	00 00 6F 22
汉	U+6C49	E6 B1 89	6C 49	00 00 6C 49
漢	U+FA47	EF A9 87	FA 47	00 00 FA 47
字	U+5B57	E5 AD 97	5B 57	00 00 5B 57

Charset Designations

In this section you will learn about the difference between a character set and an encoding. You will also learn why this distinction is critically important in several important contexts, one of which is information interchange, whether in the form of email or other electronic media. You see, in order to explicitly indicate the content of a document, such as an email message or an HTML file, there is the notion of "charset" (character set), which is used as an identifier.

Character Sets Versus Encodings

The fundamental ways in which character sets are different from encodings—which are especially clear when in the context of CJKV—are as follows:

- Character sets, especially CJKV ones, can usually be encoded in more than one way. Consider ISO-2022 and EUC encodings, both of which are commonly used to encode most CJKV character sets.*
- Most CJKV encodings support more than one character set. Consider EUC-JP for Japan, which supports JIS X 0201-1997, JIS X 0208:1997, and JIS X 0212-1990 in a mixed one-, two-, and three-byte encoding.

Table 4-78 lists several CJKV encodings, along with the characters sets that they support. Table 4-2 provided similar information.

Table 4-78. CJKV encodings and the character sets they support

Encoding	Character sets
EUC-CN	GB 1988-89[a] and GB 2312-80
EUC-TW	CNS 5205-1989[a] and CNS 11643-2007
EUC-JP	JIS X 0201-1997,[b] JIS X 0208:1997, and JIS X 0212-1990
EUC-KR	KS X 1003:1993[a] and KS X 1001:2004

* Except for ISO-2022-JP and ISO-2022-KR encodings, the other ISO-2022 encodings described in this book were rarely used, if at all.

Table 4-78. CJKV encodings and the character sets they support

Encoding	Character sets
HZ	GB 1988-89[a] and GB 2312-80
GBK	ASCII and GBK
GB 18030	ASCII, GBK, and GB 18030-2005
Big Five	ASCII, Big Five, and Hong Kong SCS-2008
Big Five Plus	ASCII, Big Five, and Big Five Plus
Shift-JIS	JIS X 0201-1997[b] and JIS X 0208:1997
Johab	KS X 1003:1993,[a] KS X 1001:2004, and additional hangul syllables

a. Or ASCII.

b. Or ASCII instead of the JIS-Roman portion.

There are some character sets that are supported by a single encoding, such as GBK, GB 18030, Big Five, Big Five Plus, and Hong Kong SCS-2008. Their designations are unique in that they can refer to either their character set or their encoding.

I should also point out and make clear that although "charset" obviously is a contraction of "character set," its meaning is a combination of one or more character sets and an encoding method.

Charset Registries

Now that the distinction between a character set and an encoding has been made clear, the question becomes what designation is appropriate for the purpose of relaying content information for documents, such as in the context of email, HTML, XML, and so on. In my opinion, encoding names make the best charset designators because there is little or no ambiguity or confusion as to what character sets they can support.

There have been three primary registries for charset designators, all of which are listed here:

- The *European Computer Manufacturers Association* (Ecma) International Registry*
- The *Internet Assigned Numbers Authority* (IANA) Registry†
- The *Internet Corporation for Assigned Names and Numbers* (ICANN) Registry‡

IANA currently maintains the registry. Registering a new charset designation, or changing one, requires that the procedures set forth in Ned Freed and Jon Postel's RFC 2978

* *http://www.ecma-international.org/*

† *http://www.iana.org/*

‡ *http://www.icann.org/*

(obsoletes RFC 2278), *IANA Charset Registration Procedures*, be followed.* The latest charset registry is available online.†

Consider the case when a character set name is used as the charset designator, as is actually done in some environments. When a charset designator is set to a string such as "GB_2312-80" or "KS_C_5601-1987," it is ambiguous as to what encoding it specifies. This is because these character sets, GB 2312-80 and KS X 1001:2004, can be encoded in at least three ways, and many more if a Unicode encoding is considered, though one could argue that these charset designators are inappropriate for Unicode-encoded data.

Table 4-79 lists the official and preferred charset designators for the encodings covered in this book. Note that some encodings covered in this book still lack a registered charset designator.

Table 4-79. Charset designators

Encoding	Official charset designator	Preferred charset designator
ASCII	ANSI_X3.4-1968	US-ASCII
ISO-2022-CN	ISO-2022-CN	same
ISO-2022-CN-EXT	ISO-2022-CN-EXT	same
HZ	HZ-GB-2312	same
ISO-2022-JP	ISO-2022-JP	same
ISO-2022-JP-2	ISO-2022-JP-2	same
ISO-2022-KR	ISO-2022-KR	same
EUC-CN	GB2312	same
EUC-TW	none	n/a
EUC-JP	Extended_UNIX_Code_Packed_Format_for_Japanese	EUC-JP
EUC-KR	EUC-KR	same
GBK	GBK	same
GB 18030	GB18030	same
Big Five	Big5	same
Shift-JIS	Shift_JIS	same
Johab	none	n/a
UTF-7	UTF-7	same
UTF-8	UTF-8	same
UTF-16	UTF-16	same
UTF-16BE	UTF-16BE	same

* *http://www.ietf.org/rfc/rfc2978.txt*

† *http://www.iana.org/assignments/character-sets*

Table 4-79. Charset designators

Encoding	Official charset designator	Preferred charset designator
UTF-16LE	UTF-16LE	same
UTF-32	UTF-32	same
UTF-32BE	UTF-32BE	same
UTF-32LE	UTF-32LE	same
UCS-2	ISO-10646-UCS-2	same
UCS-4	ISO-10646-UCS-4	same

In order to be compatible with older or poorly implemented software, it is important to maintain an aliasing mechanism that effectively maps several known charset designations to the preferred one. The charset registry maintains known aliases for each charset designator.

Some charset designations can withstand the test of time, and others cannot. For example, consider Korean, for which there are two camps when it comes to charset designations. One camp prefers to use the designation "KS_C_5601-1987" for EUC-KR encoding. The other camp simply prefers to use "EUC-KR" for EUC-KR encoding. In case it is not obvious, I belong to the latter camp. In 1998, all KS character set standards changed designation. For example, KS C 5601-1992 became KS X 1001:1992 and is now KS X 1001:2004. As you can clearly see, the use of "KS_C_5601-1987" as a charset designator did not withstand the test of time. However, the use of "EUC-KR" is still completely valid and still preferred, and clearly has withstood the test of time.

Code Pages

In the context of IBM and Microsoft documentation, there is often mention of a Code Page. A Code Page is somewhat analogous to charset designations in that it indicates an encoding that supports one or more character sets.

Although some of the IBM and Microsoft Code Pages appear to be similar, they should be treated independently of one another because they have several important differences. Specifically, Microsoft's Code Pages define multiple-byte encodings. IBM's define fixed-length encodings that support a single character set, but can combine into multiple-byte entities. For example, IBM's *Coded Character Set Identifier* (CCSID) 00932 differs from Microsoft's Code Page 932 in that it does not include NEC Row 13 nor IBM Selected characters encoded in Rows 89 through 92 (that is, IBM Selected characters encoded according to NEC).

IBM Code Pages

The best way to describe IBM Code Page designations is by first listing the individual *Single-Byte Character Set* (SBCS), *Double-Byte Character Set* (DBCS), and *Triple-Byte*

Character Set (TBCS) Code Page designations (those designated by "Host" use EBCDIC-based encodings). The IBM terminology for "Code Page" is *Code Page Global Identifier* (CPGID), which refers to a number between 00001 and 65534 (decimal) that identifies a Code Page. Table 4-80 lists several IBM SBCS Code Pages, which are for the most part equivalent to CJKV-Roman as described in Chapter 3.

Table 4-80. IBM SBCS Code Pages

Code Page	Language	Encoding	Additional details
00367	English	SBCS-PC	ASCII
00836	Simplified Chinese	SBCS-Host	
00903	Simplified Chinese	SBCS-PC	
01115	Simplified Chinese	SBCS-PC	
00037	Traditional Chinese	SBCS-Host	
00904	Traditional Chinese	SBCS-PC	
01043	Traditional Chinese	SBCS-PC	
00037	Japanese	SBCS-Host	
00290	Japanese	SBCS-Host	EBCDIC
01027	Japanese	SBCS-Host	EBCDIK
00897	Japanese	SBCS-PC	
01041	Japanese	SBCS-PC	
00895	Japanese	SBCS-EUC	EUC-JP code set 0—JIS-Roman
00896	Japanese	SBCS-EUC	EUC-JP code set 2—half-width katakana
00833	Korean	SBCS-Host	
00891	Korean	SBCS-PC	
01088	Korean	SBCS-PC	
01129	Vietnamese	SBCS-PC	

Table 4-81 lists some IBM DBCS Code Pages, grouped by language. Some revealing information is provided in the "Additional details" column.

Table 4-81. IBM DBCS Code Pages

Code Page	Language	Encoding	Additional details
00837	Simplified Chinese	DBCS-Host	
00928	Simplified Chinese	DBCS-PC	
01380	Simplified Chinese	DBCS-PC	IBM GB
01385	Simplified Chinese	DBCS-PC	GBK
01382	Simplified Chinese	DBCS-EUC	EUC-CN

Table 4-81. IBM DBCS Code Pages

Code Page	Language	Encoding	Additional details
00835	Traditional Chinese	DBCS-Host	
00927	Traditional Chinese	DBCS-PC	
00947	Traditional Chinese	DBCS-PC	IBM BIG-5
00960	Traditional Chinese	DBCS-EUC	EUC-TW code set 1—CNS 11643-1992 Plane 1
00300	Japanese	DBCS-Host	
00301	Japanese	DBCS-PC	
00941	Japanese	DBCS-PC	Windows-J character set
00952	Japanese	DBCS-EUC	EUC-JP code set 1—JIS X 0208:1997
00953	Japanese	DBCS-EUC	EUC-JP code set 3—JIS X 0212-1990
00834	Korean	DBCS-Host	
00926	Korean	DBCS-PC	
00951	Korean	DBCS-PC	IBM KS Code
01362	Korean	DBCS-PC	UHC
00971	Korean	DBCS-EUC	EUC-KR

Table 4-82 lists the only IBM TBCS Code Page of which I am aware. One would expect EUC-TW code set 2 to be an IBM *Four-Byte Character Set* (FBCS) or *Quadruple-Byte Character Set* (QBCS) Code Page, but IBM does not count the SS2 as one of the bytes in terms of Code Page type designator. For the same reason, EUC-JP code set 3 is a DBCS, not TBCS, Code Page.

Table 4-82. IBM TBCS Code Pages

Code Page	Language	Encoding	Additional details
00961	Traditional Chinese	TBCS-EUC	EUC-TW code set 2—CNS 11643-1992 Plane 2

When we combine the SBCS, DBCS, and TBCS Code Pages into *Multiple-Byte Character Set* (MBCS) entities that are assigned unique CCSIDs, things may become a bit more revealing for those familiar with Microsoft Code Page designations, as shown in Table 4-83.

Table 4-83. IBM MBCS entities

CCSID	Language	Encoding	Code Page composition
05031	Simplified Chinese	MBCS-Host	00836 and 00837
00936	Simplified Chinese	MBCS-PC	00903 and 00928
00946	Simplified Chinese	MBCS-PC	01042 and 00928
01383	Simplified Chinese	MBCS-EUC	00367 and 01382

Table 4-83. IBM MBCS entities

CCSID	Language	Encoding	Code Page composition
05033	Traditional Chinese	MBCS-Host	00037 and 00835
00938	Traditional Chinese	MBCS-PC	00904 and 00927
00950	Traditional Chinese	MBCS-PC	01114 and 00947
25524	Traditional Chinese	MBCS-PC	01043 and 00927
00964	Traditional Chinese	MBCS-EUC	00367, 00960, and 00961
00930	Japanese	MBCS-Host	00290 and 00300
00939	Japanese	MBCS-Host	01027 and 00300
00932	Japanese	MBCS-PC	00897 and 00301
00943	Japanese	MBCS-PC	01041 and 00941
00954	Japanese	MBCS-EUC	00895, 00952, 00896, and 00953
00933	Korean	MBCS-Host	00833 and 00834
00934	Korean	MBCS-PC	00891 and 00926
00944	Korean	MBCS-PC	01040 and 00926
25525	Korean	MBCS-PC	01088 and 00951
00970	Korean	MBCS-EUC	00367 and 00971

You will see in the next section that many of Microsoft's Code Pages are similar, in both content and designation, to some of these CCSIDs specified by IBM.

More detailed information about IBM Code Pages can be found in IBM's *Character Data Representation Architecture* (CDRA), and the *CDRA Reference* is now available online.*

Microsoft Code Pages

Table 4-84 lists Microsoft's CJKV Code Pages, along with a brief description of their composition. Note how many of them have Code Page designations that are the same as IBM CCSIDs listed in Table 4-83 (when the leading zeros are ignored).†

Table 4-84. Microsoft Code Pages

Code Page	Characteristics
932	JIS X 0208:1997 character set, Shift-JIS encoding, Microsoft extensions (NEC Row 13 and IBM Selected Characters duplicately encoded in Rows 89 through 92 and Rows 115 through 119)
936	GBK character set, GBK encoding

* *http://www-306.ibm.com/software/globalization/cdra/index.jsp*

† Although a hunch, this may have resulted from the fact that Microsoft Code Pages began their life as IBM Code Pages, but then developed on their own.

Table 4-84. Microsoft Code Pages

Code Page	Characteristics
949	KS X 1001:2004 character set, Unified Hangul Code encoding, remaining 8,822 hangul as extension
950	Big Five character set, Big Five encoding, Microsoft extensions (actually, only the ETen extensions of Row 0xF9)
1258	TCVN-Roman character set
1361	Johab character set, Johab encoding

Although the Code Page designations of Microsoft Code Pages have remained the same over the years, their contents or definitions have been expanded to cover additional characters or new encodings that are true supersets of the previous version. Code Page 936, for example, was once the GB 2312-80 character set encoded according to EUC-CN encoding, but is now based on GBK. Also, Code Page 949 was once the KS X 1001:2004 character set encoded according to EUC-KR, but is now defined as Unified Hangul Code (UHC) encoding, which is detailed in Appendix F.

Code Conversion

Put simply, code conversion is interoperability between encodings. Given that encodings must interoperate, code conversion is a necessary—and a very fundamental and basic—task that even the simplest text-processing software must perform. And code conversion must be done correctly, or else any subsequent processing that is performed will propagate any errors that were introduced, such as incorrect or missing characters.

Conversion of CJKV text from one encoding to another requires that you alter the numeric values of the bytes or code units that are used to represent each character. There is a wide variety of CJKV-capable code conversion programs and libraries available today, and to some extent, some software have these routines built-in. Some of these code conversion programs are portable across platforms, but some are not. Software developers who intend to introduce products into these markets must make their software as flexible and robust as possible—meaning that software should be able to handle more than a single CJKV encoding method. This is not to say that such software should be able to process all possible encodings internally, but that it should at least have the ability to import and export as many encodings as possible, which will result in better information interchange among various platforms. This is interoperability. Adobe FrameMaker, for example, allowed users to import EUC-JP– and ISO-2022-JP–encoded Japanese text, although it now processes Unicode internally.*

Earlier in this chapter it was demonstrated how ISO-2022-JP encoding can support the designation of the 1978, 1983, 1990, and 1997 vintages of the JIS X 0208 character set

* I used Adobe FrameMaker to write and typeset the first edition of this book over 10 years ago. The second edition, which you are reading right now, was written and typeset using Adobe InDesign CS3.

through the use of different escape sequences (although the escape sequences for the 1990 and 1997 versions are identical). In Appendix J you will find material illustrating the differences among these versions of JIS X 0208. Should Japanese code conversion programs account for these differences? I don't recommend it, unless you specifically need to refer to a particular vintage of JIS X 0208. Keep in mind that Shift-JIS and EUC-JP encodings support the JIS X 0208 character set without any method for designating its vintage. Any conversion from ISO-2022-JP encoding to Shift-JIS or EUC-JP encoding effectively loses the information that indicates which vintage of JIS X 0208 was used to encode the original text. Likewise, converting Shift-JIS or EUC-JP encoding to ISO-2022-JP encoding requires that an arbitrary version of JIS X 0208 be selected, because a two-byte character escape sequence must be used for properly encoding ISO-2022-JP–encoded text.

I have written a tool called *JConv* that is designed to convert the Japanese encoding of text files.* JConv supports ISO-2022-JP, EUC-JP, and Shift-JIS encodings, and it is available in a variety of platform-specific executables:

- JConv (Mac OS with minimal interface)
- JCONV-DD (Mac OS)
- *jconv.exe* (MS-DOS and Windows)
- WinJConv (Windows)

I also distribute this tool as ANSI C source code so that other programmers may benefit from the algorithms used to convert between the Japanese encodings, and so that it can be compiled on a variety of platforms. This tool has many useful features: error checking, the ability to automatically detect an input file's encoding, the ability to manually specify the input file's encoding, selective conversion of half-width katakana to their full-width equivalents, a help page, automatic ISO-2022-JP encoding repair, and command-line argument support. A more complete description of this tool, including its help page, can be found at the end of Chapter 9.

Other CJKV code conversion tools are available. They all perform roughly the same tasks, but in different ways, and some are more portable than others. Available tools are listed in Table 4-85, though some have since become obsolete.

Table 4-85. CJKV code conversion tools

Tool name	Language	URL
NCF[a]	Chinese[b]	*http://www.edu.cn/c_resource/software.html* or *ftp://ftp.net.tsinghua.edu.cn/pub/Chinese/ncf/*
ConvChar	Japanese	*http://www.asahi-net.or.jp/~VX1H-KNOY/*
nkf[c]	Japanese	*http://sourceforge.jp/projects/nkf/*

* *http://lundestudio.com/j_tools.html*

Table 4-85. CJKV code conversion tools

Tool name	Language	URL
pkf[d]	Japanese	*ftp://ftp.iij.ad.jp/pub/IIJ/dist/utashiro/perl/*
hcode	Korean	*ftp://ftp.kaist.ac.kr/hangul/code/hcode/*

a. Network Chinese Filter.

b. Many additional Chinese code conversion tools are available at *http://www.mandarintools.com/*.

c. Network Kanji Filter.

d. A Perl version of nkf.

Contemporary Unix implementations often include a built-in code conversion utility called *iconv*[*] (it is based on a GNU library called *libiconv*[†] that can be used in other programs), which is also a standard Unix C library API. There is a web implementation of *iconv* available as well.[‡] Perl, Python, Java, and other programming languages now provide their own code conversion facilities. I must point out that *iconv* and related utilities and APIs are important because they provide the best overall support for Unicode.

The demand for standalone CJKV code conversion tools is decreasing due to the introduction of built-in code conversion facilities in most CJKV-capable text editors, word processors, and other text-intensive applications.

Chinese Code Conversion

The only frustrating problems that arise when dealing with Chinese code conversion are when it is between the GB 2312-80 and Big Five (or CNS 11643-2007) character sets. There are three basic reasons for this:

- GB 2312-80 contains only 6,763 hanzi, but Big Five contains 13,053—a two-fold difference in character complement.
- Approximately one-third of the 6,763 hanzi in GB 2312-80 are considered simplified forms, and these are simply not available in Big Five.
- Many of the simplified hanzi in GB 2312-80 correspond to two or more traditional forms. Word-level context tells you which form is appropriate.

The use of GB/T 12345-90 conversion tables can sometimes aid in this process, but it doesn't solve the simple fact that Big Five has approximately two times the number of hanzi than GB 2312-80. Even ignoring the simplified/traditional issue, there are characters in GB 2312-80 that simply do not exist in Big Five.

* *http://www.opengroup.org/onlinepubs/009695399/functions/iconv.html*

† *http://www.gnu.org/software/libiconv/*

‡ *http://www.iconv.com/iconv.htm*

Network Chinese Filter (NCF), a Chinese code conversion tool that is written in C and includes APIs, represented the core technology of several projects that took place in China (this was considered a "National Ninth Five-Year Plan Key Project"), such as:

WinNCF
: A Windows version of NCF.

NCFTTY
: A pseudo terminal emulator that uses NCF to handle different encodings.

NCF Proxy
: A proxy for use with web browsers for automatically handling Chinese code conversion.

NCFDT
: A tool for detecting Chinese encodings that is written in C and includes APIs.

CERNET's Network Compass (a Chinese search engine), developed by Tsinghua University, made use of NCF and NCFDT.* The URL for NCF and related technologies can be found in Table 4-85.

See Chapter 9, specifically the section entitled "Chinese-Chinese Conversion," for details about converting between Simplified and Traditional Chinese, which is not so simple.

Japanese Code Conversion

Japanese code conversion is not problematic if you are dealing with only the JIS X 0208:1997 character set. The JIS X 0212-1990 character set, for example, cannot be encoded using Shift-JIS. Interestingly, the newer JIS X 0213:2004 character set can be encoded using Shift-JIS, but it is not. In addition, there is no single method for encoding half-width katakana in an ISO-2022–based encoding, which causes some confusion (to the point, in my opinion, that half-width katakana were explicitly excluded from the ISO-2022-JP encodings as described earlier in this chapter). Shift-JIS encoding, as you've learned, was the most commonly used Japanese encoding, and it is still used today in many contexts and environments.

One issue that comes up again and again in the context of Japanese code conversion is how to deal with the Shift-JIS user-defined region, specifically the 1,880 code points in the range <F0 40> through <F9 FC>. ISO-2022-JP and EUC-JP encodings cannot handle these Shift-JIS code points well, but Unicode can. In fact, a mapping between this Shift-JIS range and Unicode's PUA has already been defined, along with a mapping for EUC-JP encoding. The corresponding Unicode range is U+E000 through U+E757. For EUC-JP

* *http://compass.net.edu.cn/* or *http://compass.net.edu.cn:8010/*

encoding, the last 10 rows of JIS X 0208:1997 and JIS X 0212-1990, meaning rows 85 through 94 of each character set standard, are used for this purpose.*

Table 4-86 provides this mapping on a per-row basis according to Shift-JIS encoding.

Table 4-86. Conversion of Shift-JIS user-defined region to Unicode an EUC-JP encodings

Shift-JIS	Unicode	EUC-JP
F0 40–F0 FC	U+E000–U+E0BB	F5 A1–F5 FE, F6 A1–F6 FE
F1 40–F1 FC	U+E0BC–U+E177	F7 A1–F7 FE, F8 A1–F8 FE
F2 40–F2 FC	U+E178–U+E233	F9 A1–F9 FE, FA A1–FA FE
F3 40–F3 FC	U+E234–U+E2EF	FB A1–FB FE, FC A1–FC FE
F4 40–F4 FC	U+E2F0–U+E3AB	FD A1–FD FE, FE A1–FE FE
F5 40–F5 FC	U+E3AC–U+E467	8F F5 A1–8F F5 FE, 8F F6 A1–8F F6 FE
F6 40–F6 FC	U+E468–U+E523	8F F7 A1–8F F7 FE, 8F F8 A1–8F F8 FE
F7 40–F7 FC	U+E524–U+E5DF	8F F9 A1–8F F9 FE, 8F FA A1–8F FA FE
F8 40–F8 FC	U+E5E0–U+E69B	8F FB A1–8F FB FE, 8F FC A1–8F FC FE
F9 40–F9 FC	U+E69C–U+E757	8F FD A1–8F FD FE, 8F FE A1–8F FE FE

Another issue that can affect software developers who deal with Japanese encodings is the Microsoft Windows Japanese character set in the context of Unicode. As described in Appendix E, this character set is an amalgamation of NEC and IBM character sets with a JIS X 0208:1997 base. This results in several duplicate characters at different code points. The code conversion problem can be separated into two categories:

- Code conversion of the 360 IBM Selected Kanji, because they are encoded at NEC and IBM code points
- Code conversion of the non-kanji, which affects JIS X 0208:1997 row 2, NEC Row 13, and IBM Selected Non-kanji

Code conversion of the 360 IBM Selected Kanji is easiest to describe. When converting from Unicode back into Shift-JIS encoding, the IBM code points are preferred, which adheres to the third rule provided in the following bullet list. Code conversion of the duplicately encoded non-kanji is not as trivial, and involves a bit of history. Keep in mind that the NEC Kanji character set was originally based on JIS C 6226-1978, which didn't include a lot of the characters that are currently in JIS X 0208:1997 row 2. In general, the following rules apply when deciding the preference during round-trip conversion:

- If the character is in both JIS X 0208-1983 (and later) row 2 and NEC Row 13, the JIS X 0208-1983 row 2 code point is preferred.

* It seems somewhat odd to me that 940 Shift-JIS user-defined code points, specifically <F0 40> through <F4 FC>, map back onto JIS X 0208:1997.

- If the character is in both NEC Row 13 and IBM Selected Non-kanji, the NEC Row 13 code point is preferred.
- If the character is IBM Selected at both NEC and IBM code points, the IBM code point is preferred.

Table 4-87 provides the non-kanji mappings for round-trip conversion of the Microsoft Windows-J character set.

Table 4-87. Round-trip Unicode mapping for Microsoft Windows-J character set

Character	Shift-JIS code points	To Unicode	Back to Shift-JIS
∪	81 BE, 87 9C	U+222A	81 BE
∩	81 BF, 87 9B	U+2229	81 BF
¬	81 CA, EE F9, FA 54	U+FFE2	81 CA
∠	81 DA, 87 97	U+2220	81 DA
⊥	81 DB, 87 96	U+22A5	81 DB
≡	81 DF, 87 91	U+2261	81 DF
≒	81 E0, 87 90	U+2252	81 E0
√	81 E3, 87 95	U+221A	81 E3
∵	81 E6, 87 9A, FA 5B	U+2235	81 E6
∫	81 E7, 87 92	U+222B	81 E7
Ⅰ	87 54, FA 4A	U+2160	87 54
Ⅱ	87 55, FA 4B	U+2161	87 55
Ⅲ	87 56, FA 4C	U+2162	87 56
Ⅳ	87 57, FA 4D	U+2163	87 57
Ⅴ	87 58, FA 4E	U+2164	87 58
Ⅵ	87 59, FA 4F	U+2165	87 59
Ⅶ	87 5A, FA 50	U+2166	87 5A
Ⅷ	87 5B, FA 51	U+2167	87 5B
Ⅸ	87 5C, FA 52	U+2168	87 5C
Ⅹ	87 5D, FA 53	U+2169	87 5D
№	87 82, FA 59	U+2116	87 82
℡	87 84, FA 5A	U+2121	87 84
㈱	87 8A, FA 58	U+3231	87 8A
ⅰ	EE EF, FA 40	U+2170	FA 40
ⅱ	EE F0, FA 41	U+2171	FA 41
ⅲ	EE F1, FA 42	U+2172	FA 42
ⅳ	EE F2, FA 43	U+2173	FA 43

Table 4-87. Round-trip Unicode mapping for Microsoft Windows-J character set

Character	Shift-JIS code points	To Unicode	Back to Shift-JIS
ⅴ	EE F3, FA 44	U+2174	FA 44
ⅵ	EE F4, FA 45	U+2175	FA 45
ⅶ	EE F5, FA 46	U+2176	FA 46
ⅷ	EE F6, FA 47	U+2177	FA 47
ⅸ	EE F7, FA 48	U+2178	FA 48
ⅹ	EE F8, FA 49	U+2179	FA 49
¦	EE FA, FA 55	U+FFE4	FA 55
'	EE FB, FA 56	U+FF07	FA 56
"	EE FC, FA 57	U+FF02	FA 57

Korean Code Conversion

Korean code conversion, when dealing with only EUC-KR and ISO-2022-KR encodings, is trivial. However, when Johab encoding is added to the mix, things become somewhat less trivial. Converting EUC-KR and ISO-2022-KR encodings to Johab encoding results in no loss of data, because all of their characters can be represented in Johab encoding. However, the 8,822 additional hangul made available in Johab encoding cannot convert to EUC-KR and ISO-2022-KR encodings, except as strings of individual jamo, all of which are encoded in EUC-KR and ISO-2022-KR encodings. *Hcode* is a very popular Korean code conversion program that supports a wide variety of Korean encodings. Its URL can be found in Table 4-85.

Perl subroutines that illustrate algorithmic conversion to and from Johab encoding—they apply to all KS X 1001:2004 characters, except to the 51 modern jamo and hangul syllables—are provided in Appendix C.

Code Conversion Across CJKV Locales

All of the difficult, yet very interesting, code conversion problems arise when attempting to convert text from one CJKV locale to another. For example, consider the times when someone sends to me some Chinese text, such as the title of a book or dictionary, that is encoded according to a Japanese encoding method, such as Shift-JIS encoding. If I wanted to render the title in its original Chinese form, I would be forced to deal with at least one complex issue, specifically that Japanese does not use the same set of simplified ideographs that are used in China for Chinese. Both languages use many of the same simplified ideographs, but there are plenty of exceptions. Some simplified ideographs are specific to Japanese, meaning that Chinese, as used in China, still uses what is considered the traditional form. Consider 黒 (U+9ED2) for Japanese, which contrasts with 黑 (U+9ED1) for Chinese. The former is a simplified ideograph that is specific to Japanese, and the latter

is a traditional ideograph that is used by Chinese, as used in China, as well as in Taiwan, Hong Kong, and Korea. There are many more examples of simplified ideographs that are specific to Chinese, as used in China.

This simplified ideograph issue can pose a problem if you are using an intermediate representation, such as Unicode. Using Unicode is a problem only if you are not aware of this issue. Unicode often encodes two versions of the same ideograph, specifically the simplified and traditional forms, as exemplified by U+9ED1 and U+9ED2. In the end, chances are that all the ideographs will convert to an appropriate form simply because the author of the original text was able to input them.

In order to effectively handle cases of characters that do not have a direct mapping to another character set according to Unicode, making use of correspondence tables, such as for simplified/traditional ideograph pairs and ideograph variants, can dramatically help to improve the accuracy of the conversion. But, there will always be cases of unmappable characters. It is unavoidable.

ICU (*International Components for Unicode*),* Basis Technology's RCLU (*Rosette Core Library for Unicode*),† available for Unix and Windows, *tcs*‡ (*Translate Character Sets*), also available for Unix and Windows, and my own home-grown *CJKVConv.pl*§ (written in Perl) are examples of tools or libraries that can perform code conversion across CJKV locales. ICU is an incredibly mature library for Unicode and globalization support. RCLU provides a plethora of options and features, including encoding autodetection. My own *CJKVConv.pl* uses multiple database-like ideographic variant tables that assist in effectively resolving unmappable characters.

Code Conversion Tips, Tricks, and Pitfalls

Understanding the relationships between the many CJKV encoding methods can work to your advantage when it comes time to perform code conversion of any kind. Luckily, though, a great many people, because of the environments that they use, are well insulated from having to know details about code conversion. And that is the way it should be. Software, not humans, should perform code conversion. This is typically the job of the OS and libraries, specifically through the APIs that they make available to applications. But someone must write these APIs. Given the critical nature of code conversion, meaning that any errors are a very bad thing, it is important that these code conversion APIs be written in such a way that they are robust, up-to-date, and correct.

The first determination that one must make with regard to code conversion is whether the data that you are attempting to display or manipulate actually requires code conversion, or is simply damaged and in need of some type of repair. If you are using a

* *http://www.icu-project.org/*

† *http://www.basistech.com/unicode/*

‡ *http://swik.net/tcs/*

§ *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/perl/cjkvconv.pl*

Korean-enabled OS and receive an ISO-2022-KR–encoded file, it may be difficult to determine whether code conversion is necessary. Even well-formed ISO-2022-KR–encoded files may not display on a Korean-enabled OS, because it may instead expect data to be encoded according to EUC-KR or Johab encodings.

Consider the following past scenario. You were using a Japanese-capable operating system, such as Mac OS-J. You received two files. One file was EUC-JP–encoded, and the other was ISO-2022-JP–encoded, but it was damaged because its escape characters had been stripped away. Both files displayed equally *un*-well when opened in a typical text editor or other text-processing application. Until the EUC-JP–encoded file was converted to Shift-JIS encoding, it could not be used on Mac OS-J. Likewise, until the ISO-2022-JP–encoded file was repaired *and* converted to Shift-JIS encoding, it also could not be used on Mac OS-J.

The subject of handling damaged or otherwise unreadable CJKV text, which is closely related to code conversion, is covered in the next section.

Repairing Damaged or Unreadable CJKV Text

CJKV text files can be damaged in different ways depending on which encoding was used in the file. ISO-2022–encoded text is usually damaged by unfriendly email clients (and news readers) that remove control characters, including the all-important escape and shift characters. You will learn that this type of damage is relatively easy to repair for some ISO-2022 encodings because the remaining characters that constitute a valid escape or designator sequence can serve as the context for properly restoring the characters that were removed. However, EUC, GBK, GB 18030, Big Five, Shift-JIS, and Johab encodings make generous use of eight-bit bytes, and many email clients (and news readers) used in the past were not what many would call eight-bit clean, meaning that they effectively turned off the eighth bit of every byte. This had the nasty and unpleasant effect of scrambling the encoding, which rendered the text unreadable.

Some data are not damaged *per se*, but rather transformed or reencoded in a way that is intended to preserve the original encoding. This reencoding refers to Quoted-Printable and Base64 transformations. Files that contain a small amount of binary data or have few bytes with the MSB set are usually converted to Quoted-Printable to retain some level of human-readability, and those that contain mostly binary data or are filled with bytes that have the MSB set are converted to Base64.

Quoted-Printable Transformation

Quoted-Printable transformation simply converts nonalphanumeric characters into a three-character form that is composed of an "equals" symbol (`=`, `0x3D`) followed by the two hexadecimal digits that represent the original character's encoded value. Instances of a genuine "equals" symbol are converted to the three-character string `=3D` according to Quoted-Printable transformation rules.

Quoted-Printable transformation is defined in Ned Freed and Nathaniel Borenstein's RFC 2045, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*.* RFCs 2047, *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*, and 2231, *MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations*, should also be consulted.† While most Quoted-Printable–transformed data is found in email message bodies, it can sometimes be found in email message headers, such as in the following one that I received (the emboldened portions represent the actual data; the rest is part of the transformation process that is explained in Table 4-90):

```
Subject: =?Big5?Q?=A6=5E=C2=DO_=3A_Reply_=3A_Tze=2Dloi_Input_Method?=
```

The following short Perl program converts Quoted-Printable data back into its original form:

```
while (defined($line = <STDIN>)) {
  $line =~ s/=([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/ge;
  $line =~ s/=[\n\r]+$//;
  print STDOUT $line;
}
```

This and many more Perl programs are provided in Appendix C to serve as examples from which you can write your own routines, whether in Perl or your programming language of choice.

Base64 Transformation

Base64 transformation is more complex than Quoted-Printable in that it involves manipulation of data at the bit level and is applied to an entire file, not only those bytes that may represent binary data. Base64 transformation, put simply, is a method for transforming arbitrary sequences into the safest 64-character ASCII subset, and like Quoted-Printable, is defined in RFCs 2045 and 2047.

Base64 transformation is easy to describe. Every three bytes are transformed into a four-byte sequence. That is, the 24 bits that constitute three bytes are split into four 6-bit segments. Six bits can encode up to 64 unique characters. Each 6-bit segment is then converted into a character in the Base64 character set. A 65th character, "=" (0x3D), functions as a "pad" character if a full three-byte sequence is not achieved. Zero bits added to the right are used to pad instances of incomplete six-bit segments. The Base64 character set and mappings can be found in Table 4-18.

My name, written in Japanese and UTF-8–encoded, can serve as an example to illustrate the Base64 transformation. The three characters are 小林剣 (U+5C0F, U+6797, and U+5263), and their UTF-8 values are <E5 B0 8F>, <E6 9E 97>, and <E5 89 A3>. When the

* *http://www.ietf.org/rfc/rfc2045.txt*

† *http://www.ietf.org/rfc/rfc2047.txt* and *http://www.ietf.org/rfc/rfc2231.txt*

Base64 transformation is applied, the result becomes the following Base64-encoded string consisting of 12 characters:

```
5bCP5p6X5Ymj
```

When the three characters are represented as binary strings, or bit arrays, the result is as shown in Table 4-88.

Table 4-88. UTF-8 encoding bit array examples

Character	UTF-8 encoding	Bit arrays
小	E5 B0 8F	11100101 10110000 10001111
林	E6 9E 97	11100110 10011110 10010111
剣	E5 89 A3	11100101 10001001 10100011

Splitting these bit arrays into six-bit segments results in the bit arrays and the corresponding Base64 characters illustrated in Table 4-89.

Table 4-89. Base64 transformation example

Binary string	Decimal and hexadecimal equivalent	Base64 character
111001	57—0x39	5
011011	27—0x1B	b
000010	02—0x02	C
001111	15—0x0F	P
111001	57—0x39	5
101001	41—0x29	p
111010	58—0x3A	6
010111	23—0x17	X
111001	57—0x39	5
011000	24—0x18	Y
100110	38—0x26	m
100011	35—0x23	j

Compare this with the contents of Table 4-18. The following brief Perl program decodes Base64 data (it requires the MIME module):

```
use MIME::Decoder;
my $enc = 'base64';
my $decoder = new MIME::Decoder $enc or die "$enc unsupported";
$decoder->decode(\*STDIN, \*STDOUT);
```

Base64 data is typically delivered in two forms: either embedded within an SMTP header field or else as an attachment whereby the entire attachment has been Base64-transformed. The following is a Base64-encoded SMTP header field:

```
From: lunde@adobe.com (=?UTF-8?B?5bCP5p6X5Ymj?=)
```

The emboldened portion represents the actual Base64-encoded segment. Table 4-90 lists each component of the data within the parentheses (the parentheses themselves are genuine and are not considered part of the Base64-encoded string).

Table 4-90. Base64 in SMTP header fields

Component	Explanation
=?	Signals the start of the Base64-encoded string
UTF-8	Charset designation
?	Delimiter
B	Transformation type—"B" for Base64; "Q" for Quoted-Printable
?	Delimiter
5bCP5p6X5Ymj	Base64-encoded data
?=	Signals the end of the Base64-encoded string

The following is an example of a text stream that had Base64 transformation applied:

```
SSBhbSB3b25kZXJpbmcgaG93IG1hbnkgcGVvcGxlIHdpbGwgYWNOdWFsbHkgdHlwZSBOaGlzIG
luIHRvIGZpbmQgb3VOCndoYXQgaXQgc2F5cy4gV2VsbCwgaXQgZG9lc24ndCBzYXkgbXVjaC4K
```

I have never encountered a need to Base64-encode data, because *Mail User Agents* (MUAs) and *Mail Transport Agents* (MTAs) automatically and routinely apply Base64 transformation when required.* If you really have a strong desire to Base64-encode some data, you can use the following Perl program (which, again, requires the MIME module):

```
use MIME::Base64;
undef $/;
my ($data, $encoded);
$data = <STDIN>;
$encoded = encode_base64($data);
print STDOUT $encoded;
```

Sendmail is an excellent example of an MTA that implements RFC 1652† faithfully in that it handles Base64 and Quoted-Printable conversion as necessary.‡

* But frankly, I did need to apply Base64 transformation in order to create the examples for this section.

† *http://www.ietf.org/rfc/rfc1652.txt*

‡ *http://www.sendmail.org/*

Other Types of Encoding Repair

As opposed to being reencoded by a well-understood and proven transformation, some data can become truly damaged in the sense that information is removed, either in terms of entire bytes, or the values of specific bits. Such data appears as garbage, meaning it is unreadable. This is referred to as *mojibake* (文字化け *mojibake*) in Japanese.

We first discuss the repair procedure for ISO-2022-JP–encoded files as an example of how encodings may become damaged by having key characters removed, and then repaired. In the past, one might have received Japanese email messages or attempted to display Japanese articles from Usenet News, which had their "escape" characters stripped out by unfriendly email or news reading software. Sometimes the escape characters are simply mangled—converted into a single "space" character (0x20) or into Quoted-Printable (discussed previously). This was a very annoying problem because one usually threw out such email messages or articles, rather than suffering through the manual and the somewhat grueling task of manually restoring the escape characters. For example, look at the ISO-2022-JP–encoded string in Table 4-91. It is first shown as it should appear when displayed properly, and then shown as it could appear if it were damaged, specifically with its escape characters missing. There are certainly lots of dollar signs, but these ones clearly won't make you rich, and may have the opposite effect, especially if your software is expected to correctly handle such situations but doesn't.*

Table 4-91. Damaged encoding example—ISO-2022-JP encoding

Encoding	String
Original text	これは和文の文章の例で、それは English の文章の例です。
ISO-2022-JP—damaged	B3IO0BJ8$NJ8>O$NNc$G! $=$I$O(J English BNJ8>ONNcG$9!#(J

Many years ago, I wrote a tool that was designed to repair damaged ISO-2022-JP–encoded files by simply scanning for the printable-character portions of the escape sequences that were left intact, keeping track of the state of the data stream (that is, whether it is in one- or two-byte mode), and then using this as the context for restoring the escape characters that were removed. This tool is called *JConv*, which was briefly described in the previous section—its full description is at the end of Chapter 9.

Because ISO-2022-CN, ISO-2022-CN-EXT, and ISO-2022-KR encodings use single control characters—the ASCII Shift-In and Shift-Out control characters—for switching between one- and two-byte modes, there is not sufficient context for restoring them if they are inadvertently or mistakenly removed. So, unfortunately, there is no elegant way of repairing these encodings.

* These dollar signs, which correspond to hexadecimal 0x24, represent the first byte of the hiragana characters in row 2 of JIS X 0208:1997. Given that 70% of Japanese text is made up of hiragana, the number of dollar signs in damaged ISO-2022-JP–encoded text will be relatively high.

Some encodings are damaged at the bit level, meaning that specific bits change value. Bits exist in a binary condition and are either on or off. The bits that are damaged are effectively switched from on to off. Eight-bit encodings that have had their eighth bits stripped are such cases. Manual repair of EUC encoding is not terribly painful: you simply turn on or enable the eighth bit of byte sequences that appear as garbage. The only problem is detecting which bytes are used to compose two-byte characters—this is when human intervention and interaction is required. Table 4-92 uses the same Japanese string as used in the ISO-2022-JP–encoded example, but demonstrates how EUC-JP and Shift-JIS encodings can become damaged.

Table 4-92. Damaged encoding example—ISO-2022-JP, EUC-JP, and Shift-JIS encodings

Encoding	String
Original text	これは和文の文章の例で、それは English の文章の例です。
ISO-2022-JP—damaged	B3IOOBJ8$NJ8>O$NNc$G! $=$I$O(J English BNJ8>ONNcG$9!#(J
EUC-JP—damaged	3I$OOBJ8$NJ8>ONNcG! $=$I$O English $NJ8>O$NNcG9!#
Shift-JIS—damaged	1 j M a 6 L 6 M L a E A ; j M English L 6 M L a E 7 B
Shift-JIS—damaged	1jMa6L6MLaEA;jM English L6MLaE7B

For EUC-JP encoding, the crucial context-forming byte sequences, specifically "$B" and "(J" from the ISO-2022-JP encoding escape sequences, are missing. For Shift-JIS encoding, the results are much different, in terms of what appears and in that it bears no resemblance to the damaged ISO-2022-JP– and EUC-JP–encoded string. The two examples of damaged Shift-JIS–encoded data differ in that the second example does not contain spaces as place holders for the bytes that had their eighth bits disabled.

The bottom line is that any encoding repair solution that deals with eight-bit encodings with mixed single- and multiple-byte representations requires interaction with a user. There is simply too much judgement involved, and a literate (meaning someone who has more than simply a pulse) human is needed to guide the repair process. It is clearly better to make a concerted effort to avoid the damage in the first place.

Advice to Developers

The information presented in this chapter may have given you the impression or feeling that CJKV encoding is a real mess. Well, to be quite frank, it can be if you don't have a clue what you're doing. That's the whole point of this book, and specifically this chapter. Encodings are systematic, which means that they are predictable regardless of the number of code points that they can address, and regardless of how many characters have been assigned to its code points. Systematic simply means that it can be programmed, categorized, or easily referenced.

There are three basic pieces of advice that I strongly feel should be conveyed to software developers, as follows:

- Embrace Unicode and support its encoding forms, meaning UTF-8, UTF-16, and UTF-32. Although only one of these encoding forms typically needs to be used for internal processing, all three must be supported in terms of interoperability.
- Do not forget to support the legacy encodings—the more the better—in terms of being able to interoperate with them.
- When dealing with Unicode and legacy encodings, interoperability must be properly tested. Any implementation must be properly and exhaustively tested prior to its release to its users, whether they are other developers or end users.

The following sections detail these three areas of advice and provide examples of why they are applicable to software development.

Embrace Unicode

Embracing Unicode has its rewards, and they are great, but rewards are usually the result of dealing with challenges. Moving from a system that uses legacy encodings to one that uses Unicode is certainly a challenge, but the rewards are significant. For example, GB 18030 certification, for software developers who wish to market their products in China, represents a significant hurdle, but it has been demonstrated time and time again that proper Unicode support significantly eases the path to GB 18030 compliance.

When supporting Unicode, although it is clear that only one of its encodings needs to be supported in terms of the processing that is done within your software, it is prudent to support all of the Unicode encodings, specifically UTF-8, UTF-16, and UTF-32, in terms of interoperability. OpenType fonts that include glyphs that are mapped from non-BMP code points include 'cmap' tables that use UTF-32 encoding. UTF-8 and UTF-16 encodings are more common for application use, meaning that at some level, the UTF-8 or UTF-16 encoding that is used in an application must interoperate with the UTF-32 encoding in OpenType fonts.

Speaking of non-BMP code points, supporting them in your application is critical in this day and age, when roughly half of the characters in Unicode are outside the BMP, and not all are considered rarely used. *Ideographic Variation Sequence* (IVS) support is another example of why non-BMP capability is critical, because the 240 *Variation Selectors* (VSes) that serve as the second component of an IVS are encoded in Plane 14. Properly supporting JIS X 0213:2004, Hong Kong SCS-2008, and GB 18030-2005 also requires non-BMP support.

It is also important to reiterate the importance of byte order when using the Unicode encodings. Proper use and interpretation of the *Byte Order Mark* (BOM) is critical, especially when interoperating with other encodings. The BOM is also useful, but not necessary, for UTF-8 encoding in disambiguating it from legacy encodings.

Legacy Encodings Cannot Be Forgotten

If you embrace Unicode as you should, even very tightly, legacy encodings must still be supported at some level, in terms of interoperability. GB 18030 certification is an area that demonstrates the need to interoperate between Unicode and legacy encodings. While embracing Unicode certainly simplifies the support of GB 18030-2005, one of the GB 18030 compliance requirements is to interoperate with GB 18030 encoding. There is no escaping this. To not support GB 18030 encoding in the context of interoperability means that your software cannot achieve GB 18030 certification.

In terms of legacy encodings, you should have learned that there are two basic CJKV encoding methods, specifically ISO-2022 and EUC, and a small number of locale-specific ones. These legacy encoding methods, at least within a single locale, are designed to interoperate with one another through the use of published and well-proven algorithms. Legacy encodings also interoperate with Unicode, some better than others, through the use of mapping tables. For the most part, Unicode is considered a superset of the legacy encodings. This means that data can be converted from a legacy encoding to Unicode, but mapping from Unicode back to a legacy encoding may not be possible for some characters, depending on the target legacy encoding.

How many legacy encodings should be supported? As many as possible, considering that it is being done for the purpose of interoperability. To some extent, the OS-level APIs can be used to support legacy encodings, in terms of interoperability with Unicode. There are also libraries that perform the same functions. The advantage of using an OS-level API or a library is that the work has been done and can generally be considered proven.

Because code conversion plays an absolutely critical role in encoding interoperability, ensuring that the algorithms and mapping tables are current and correct should be a primary concern, and plenty of testing is necessary to guarantee this. This is true whether you build the code conversion routines yourself or use OS-level APIs or libraries. The Unicode Consortium provides a small number of useful and necessary mapping tables that can be used for this purpose.*

Code conversion is comparable to the foundation of a building. If the foundation is weak, the building that is built on top of it will not last long, and the bigger the building, the sooner it will fail. As a software developer, a strong foundation should always be the goal. In fact, it is your responsibility to your company, and more importantly, to its customers.

There are, of course, locale considerations that come into play. If the software is designed to be used in Japan, and for handling only Japanese text, then just the Japanese-specific legacy encodings need to be considered, in addition to embracing Unicode.

* *http://www.unicode.org/Public/MAPPINGS/OBSOLETE/EASTASIA/*

Testing

When dealing with encodings, which almost always involves transcoding between Unicode and legacy encodings, I cannot emphasize enough the importance of testing. Even if the code conversion is between the various Unicode encodings, there are pitfalls, along with special ranges of code points that require specific treatment, such as the Surrogates Area.

The bottom line is that if there is a flaw in the conversion table or algorithm, the resulting data loss can be disastrous for your customers. Developing a rigorous testing harness, along with a clear method for verifying the results, is prudent. In fact, doing so is your responsibility.

CHAPTER 5
Input Methods

In earlier chapters you were introduced to the complexities of CJKV writing systems, their character set standards, and their encoding methods. Now it is time for you to learn something about how a user is able to input the tens of thousand of characters that are made available in these character set standards, and which are encoded for today's OSes.

Mainly due to the vast number of characters set forth in CJKV character set standards, there is no simple solution for user input, such as direct keyboard input, as you would find in the West. Instead, you will find that CJKV input methods fall into the following two general categories:

Direct methods
: Employ a unique value for the target character, usually one of its encoded values.

Indirect methods
: Usually more convenient or intuitive, these obtain the encoded value of the target character or characters, usually by typing out the reading or shape on a standard or specialized keyboard.

Examples provided in this chapter will demonstrate that the indirect input methods are the most commonly used, and appropriately so, because they usually involve the reading or structure of a character, both of which are more intuitive than any other type of input. After all, native speakers of CJKV languages learn ideographs by their readings. There are ideographs, however, that are not commonly known, and these are typically input by means other than their reading. Of course, direct and indirect input methods are covered in this chapter.

To facilitate the explanations to come, at least two examples will be provided per input method. In most cases, the two examples will be the two ideographs 漢 and 字, which represent the two ideographs for expressing "ideograph" in most CJKV locales, specifically *hanzi* in Chinese, *kanji* in Japanese, and *hanja* in Korean. Other characters shall be substituted or included as appropriate.

There is an easy explanation for why so many different input methods have been developed for entering CJKV text. First, it is obviously impossible to fit thousands of characters

on a keyboard array. Mind you, such keyboards do exist, but they are not designed for the mortal operator, and also do not provide coverage for all the character set standards discussed in Chapter 3. An example of this type of keyboard is provided later in this chapter. This limitation effectively forced the CJKV locales to develop more efficient means of input. Second, CJKV text, as you learned in Chapter 4, can be encoded through the use of a variety of encoding methods, though Unicode has become the most broadly supported encoding among them. This makes using the encoded value of a character in each encoding method a viable method for input, though it is far from being intuitive. Input through the use of an encoded value is what I consider to be a last-resort method. There are circumstances when this input method is useful.

Other references that either describe various input methods or list input codes for ideographs include the following books and dictionaries:

- A Chinese book entitled 常用汉字输入法操作速成 (*chángyòng hànzì shūrùfǎ cāozuò sùchéng*, meaning "Quick Input Method Operations for Frequently Used Hanzi") describes many contemporary Chinese input methods.
- A Chinese book entitled 计算机汉字输入与编辑实用手册 (*jìsuànjī hànzì shūrù yù biānjí shíyòng shǒucè*) also describes contemporary Chinese input methods, but includes a large appendix that has readings and input codes for all GB 2312-80 hanzi.
- A Chinese dictionary entitled 常用汉字编码字典 (*chángyòng hànzì biānmǎ zìdiǎn*, meaning "Dictionary of Codes for Frequently Used Hanzi") lists nearly two dozen (some obscure) input codes for all 6,763 hanzi in GB 2312-80.
- The manuals for a Chinese word processor called NJStar (南极星/南極星 *nánjíxīng*) provide useful descriptions, including examples, for nearly two dozen different input methods.
- A Japanese dictionary entitled 増補改訂 JIS 漢字字典 (*zōho kaitei JIS kanji jiten*, meaning "Expanded and Revised JIS Kanji Dictionary") includes all of the kanji found in the JIS X 0208:1997 and JIS X 0213:2004 standards, and provides their readings and character codes.
- A Japanese book entitled 中国入力方法の話 (*chūgoku nyūryoku hōhō-no hanashi*, meaning "Discussions of Chinese Input Methods") describes the principles behind various Chinese input methods.

If you need more information about input methods than what is provided in this chapter, I encourage you to explore these references and seek out additional ones. I have also found Wikipedia to be a good source for input method information, including details about keyboard arrays.*

* *http://www.wikipedia.org/*

Transliteration Techniques

Before we can begin to discuss the basics of CJKV input, to include input methods and keyboard arrays, it is necessary to first cover the issue of transliteration. This is because the most broadly used input methods provide a facility to use the universally accepted QWERTY keyboard array. In other words, all CJKV scripts can be expressed using Latin characters or with a set of native alphabetic or native elements, through the use of transliteration techniques and principles.

Zhuyin Versus Pinyin Input

Zhuyin (also known as *bopomofo*) and Pinyin are related in that there is always a way to represent the same set of sounds using either writing system. Both systems are phonetic, but Pinyin is based on Latin characters.

There are three types of Pinyin input, as follows:

- Half Pinyin (简拼/簡拼 *jiǎnpīn*)
- Full Pinyin (全拼 *quánpīn*)
- Double Pinyin (双拼 *shuāngpīn*)

Fortunately, these three methods are easily accommodated using the standard QWERTY keyboard array.

Full Pinyin functions by simply typing the Pinyin equivalent of hanzi. Note that any Pinyin-based input methods result in a (sometimes long) list of candidate hanzi from which the user must choose. This is the nature of most indirect input methods.

Double Pinyin functions by first dividing the Pinyin reading into two parts. Certain letter combinations are replaced with single characters according to a set of standardized rules. The resulting characters are used for input. The primary advantage of Double Pinyin is that hanzi can be input using only one or two keystrokes.

Table 5-1 illustrates zhuyin and Pinyin input using both Full Pinyin and Double Pinyin methods of transliteration. Those transliterations that are different from Full Pinyin have been shaded.

Table 5-1. Keystrokes for zhuyin and Pinyin characters

Zhuyin	Full Pinyin	Half Pinyin	Double Pinyin
ㄅ	B	B	B
ㄆ	P	P	P
ㄇ	M	M	M
ㄈ	F	F	F
ㄉ	D	D	D

Table 5-1. Keystrokes for zhuyin and Pinyin characters

Zhuyin	Full Pinyin	Half Pinyin	Double Pinyin
ㄊ	T	T	T
ㄋ	N	N	N
ㄌ	L	L	L
ㄍ	G	G	G
ㄎ	K	K	K
ㄏ	H	H	H
ㄐ	J	J	J
ㄑ	Q	Q	Q
ㄒ	X	X	X
ㄓ	ZH	A	A
ㄔ	CH	I	U
ㄕ	SH	U	I
ㄖ	R	R	R
ㄗ	Z	Z	Z
ㄘ	C	C	C
ㄙ	S	S	S
ㄚ	A	A	A
ㄛ	O	O	O
ㄜ	E	E	E
ㄝ	EI	EI	W
ㄞ	AI	L	S
ㄟ	EI	EI	W
ㄠ	AO	K	D
ㄡ	OU	OU	P
ㄢ	AN	J	F
ㄣ	EN	F	R
ㄤ	ANG	H	G
ㄥ	ENG	G	T
ㄦ	ER	ER	Q
ㄧ	I	I	I
ㄧㄚ	IA	IA	B

Table 5-1. Keystrokes for zhuyin and Pinyin characters

Zhuyin	Full Pinyin	Half Pinyin	Double Pinyin
ㄧㄝ	IE	IE	M
ㄧㄠ	IAO	IK	K
ㄧㄡ	IU	IU	N
ㄧㄢ	IAN	IJ	J
ㄧㄣ	IN	IN	L
ㄧㄤ	IANG	IH	H
ㄧㄥ	ING	Y	;
ㄨ	U	U	U
ㄨㄚ	UA	UA	B
ㄨㄛ	UO	UO	O
ㄨㄞ	UAI	UL	X
ㄨㄟ	UI	UI	V
ㄨㄢ	UAN	UJ	C
ㄨㄣ	UN	UN	Z
ㄨㄤ	UANG	UH	H
ㄨㄥ	ONG	S	Y
ㄩ	V	V	U
ㄩㄝ	UE	UE	V
ㄩㄥ	IONG	IS	Y

This means that Full Pinyin requires anywhere from one to six keystrokes per hanzi, Half Pinyin requires one to three keystrokes per hanzi, and Double Pinyin requires only one or two keystrokes per hanzi. Table 5-2 provides some examples of Pinyin input to better illustrate how these three types of Pinyin work.

Table 5-2. Pinyin input examples

Hanzi	Zhuyin	Full Pinyin	Half Pinyin	Double Pinyin
啊	ㄚ	A	A	A
酷	ㄎㄨ	KU	KU	KU
处	ㄔㄨ	CHU	IU	UU
汆	ㄘㄨㄢ	CUAN	CUJ	CC

Table 5-2. Pinyin input examples

Hanzi	Zhuyin	Full Pinyin	Half Pinyin	Double Pinyin
张	ㄓㄤ	ZHANG	AH	AG
双	ㄕㄨㄤ	SHUANG	UUH	IH

Needless to say, Pinyin input is very important for the input of ideographs in the context of Chinese.

Kana Versus Transliterated Input

There are two ways to provide reading-based input to Japanese input methods through the keyboard array:

- Transliterated using Latin characters
- Kana

Ultimately, most Japanese input methods require kana input, which means that there must be a mechanism in place for converting transliterated Japanese strings into kana on the fly. Almost all—if not all—such software support such a mechanism, which permits Western keyboard arrays, such as the QWERTY keyboard array, to be used to input kana and thus Japanese text.

Table 5-3 lists the basic set of kana characters (hiragana in this case, but these are equally applicable to katakana because they represent the same sounds), along with the most common keystroke or keystrokes that are necessary to produce them. Some Japanese input methods support more than one way of entering some kana, through the use of alternate keystrokes. These multiple methods for entering kana are provided in Table 5-3 and are separated by slashes.

Table 5-3. Keystrokes to produce kana characters

"A" row		"I" row		"U" row		"E" row		"O" row	
あ	A	い	I	う	U	え	E	お	O
か	KA	き	KI	く	KU	け	KE	こ	KO
が	GA	ぎ	GI	ぐ	GU	げ	GE	ご	GO
さ	SA	し	SI/SHI	す	SU	せ	SE	そ	SO
ざ	ZA	じ	ZI/JI	ず	ZU	ぜ	ZE	ぞ	ZO
た	TA	ち	TI/CHI	つ	TU/TSU	て	TE	と	TO
だ	DA	ぢ	DI	づ	DU/DZU	で	DE	ど	DO
な	NA	に	NI	ぬ	NU	ね	NE	の	NO
は	HA	ひ	HI	ふ	HU/FU	へ	HE	ほ	HO

Table 5-3. Keystrokes to produce kana characters

"A" row		"I" row		"U" row		"E" row		"O" row	
ば	BA	び	BI	ぶ	BU	べ	BE	ぼ	BO
ぱ	PA	ぴ	PI	ぷ	PU	ぺ	PE	ぽ	PO
ま	MA	み	MI	む	MU	め	ME	も	MO
や	YA			ゆ	YU			よ	YO
ら	RA/LA	り	RI/LI	る	RU/LU	れ	RE/LE	ろ	RO/LO
わ	WA	ゐ	WI			ゑ	WE	を	WO
ん	N/NN/N'								

There are also combinations of two kana characters that require special transliteration techniques. These consist of one of the kana in Table 5-3 plus the small versions of や (*ya*), ゆ (*yu*), and よ (*yo*). These are listed in Table 5-4. Like Table 5-3, optional keystrokes are separated by a slash.

Table 5-4. Keystrokes to produce palatalized kana characters

Small "Ya" row		Small "Yu" row		Small "Yo" row	
きゃ	KYA	きゅ	KYU	きょ	KYO
ぎゃ	GYA	ぎゅ	GYU	ぎょ	GYO
しゃ	SYA/SHA	しゅ	SYU/SHU	しょ	SYO/SHO
じゃ	ZYA/JA	じゅ	ZYU/JU	じょ	ZYO/JO
ちゃ	TYA/CHA	ちゅ	TYU/CHU	ちょ	TYO/CHO
ぢゃ	DYA	ぢゅ	DYU	ぢょ	DYO
にゃ	NYA	にゅ	NYU	にょ	NYO
ひゃ	HYA	ひゅ	HYU	ひょ	HYO
びゃ	BYA	びゅ	BYU	びょ	BYO
ぴゃ	PYA	ぴゅ	PYU	ぴょ	PYO
みゃ	MYA	みゅ	MYU	みょ	MYO
りゃ	RYA	りゅ	RYU	りょ	RYO

These three small kana characters—や, ゆ, and よ—can usually be generated by either typing an "x" before their transliterated forms (for example, や can be input with the three-character string "xya") or by pressing the Shift key while typing their transliterated forms. Actually, all small kana characters, such as あ, い, う, え, お, か, け, つ, や, ゆ, よ, and わ, can be handled in this way. Check the documentation of your Japanese input method to find out which of these methods is supported. Chance favors that both of them are. Also check

the documentation for transliteration tables, similar to those just shown, that specify other special key combinations that can be used.

There are additional kana characters that require special transliteration techniques for input, which are are illustrated in Table 5-5. They are expressed in katakana because they are typically used for transliterating loan words, such as foreign places, names, and words.

Table 5-5. Keystrokes to produce special katakana sequences

Katakana	Latin keystrokes
ファ	FA
フィ	FI
フェ	FE
フォ	FO
ヴァ	VA
ヴィ	VI
ヴ	VU
ヴェ	VE
ヴォ	VO

Japanese long vowels, while transliterated using macroned vowels, are input according to the reading of the kana used to express them. For example, おう (*ō*) is entered using the two keystrokes "ou," not "oo."

Writing a Latin-to-kana conversion routine is not terribly difficult. Sometimes the authors of Japanese text-processing programs encourage others to use their routines, and most Japanese input methods that are freely available come with source code that includes conversion tables. As an example, a Perl version of a Latin-to-kana conversion library called *romkan.pl* is available. It requires the Perl library file called *jcode.pl* to function.*

Hangul Versus Transliterated Input

Like bopomofo and kana, each hangul—whether it is composed of multiple or a single jamo—can be represented by equivalent Latin characters. There appears to be several ways to transliterate hangul characters when performing keyboard input. A variety of ways to transliterate jamo are shown in the "Keyboard input" column of Table 5-6, along with the Ministry of Education, Korean Language Society, and both ISO/TR 11941:1996 systems for comparison (only the syllable-initial transliterations are provided for these systems). Note that case is significant in some instances, in particular an alternate keystroke for the double consonants that involves the use of uppercase.

* *http://srekcah.org/~utashiro/perl/scripts/?lang=en*

Table 5-6. Keystrokes to produce jamo—consonants

Jamo	Keyboard input	MOE	KLS	ISO—ROK	ISO—DPRK
ㄱ	g	K/G	G	G	K
ㄴ	n	N	N	N	N
ㄷ	d	T/D	D	D	T
ㄹ	l/r	R/L	R	R	R
ㅁ	m	M	M	M	M
ㅂ	b	P/B	B	B	P
ㅅ	s	S/SH	S	S	S
ㅇ	ng/x	none/NG	none/NG	none/NG	none/NG
ㅈ	j	CH/J	J	J	C
ㅊ	c	CH′	CH	C	CH
ㅋ	k	K′	K	K	KH
ㅌ	t	T′	T	T	TH
ㅍ	p/f	P′	P	P	PH
ㅎ	h	H	H	H	H
ㄲ	gg/G	KK	GG	GG	KK
ㄸ	dd/D	TT	DD	DD	TT
ㅃ	bb/Bh	PP	BB	BB	PP
ㅆ	ss/S	SS	SS	SS	SS
ㅉ	jj/J	TCH	JJ	JJ	CC

Table 5-7 illustrates the keystrokes for inputting the jamo that represent vowels. Alternate keystrokes are separated by a slash.

Table 5-7. Keystrokes to produce jamo—vowels

Jamo	Keyboard input	MOE	KLS	ISO—ROK and DPRK
ㅏ	a	A	A	A
ㅑ	ya/ia	YA	YA	YA
ㅓ	eo	Ŏ	EO	EO
ㅕ	yeo/ieo/ie	YŎ	YEO	YEO
ㅗ	o	O	O	O
ㅛ	yo/io	YO	YO	YO
ㅜ	u/oo	U	U	U

Table 5-7. Keystrokes to produce jamo—vowels

Jamo	Keyboard input	MOE	KLS	ISO—ROK and DPRK
ㅠ	yu/yw/iu	YU	YU	YU
ㅡ	eu/ew	Ŭ	EU	EU
ㅣ	i/wi	I	I	I
ㅐ	ae/ai	AE	AE	AE
ㅒ	yae/iai	YAE	YAE	YAE
ㅔ	e/ei	E	E	E
ㅖ	ye/ie/iei	YE	YE	YE
ㅘ	wa/ua/oa	WA	WA	WA
ㅙ	wae/uae/oai	WAE	WAE	WAE
ㅚ	oe/oi	OE	OE	OE
ㅝ	weo/ueo/ue	WŎ	WEO	WEO
ㅞ	we/uei	WE	WE	WE
ㅟ	wi/ui	WI	WI	WI
ㅢ	eui/yi/w	ŬI	EUI	YI

Of course, you should always check an input method's documentation to determine exactly what keystrokes are necessary to input each jamo.

Input Techniques

This chapter is intended to describe CJKV input in a platform- and software-independent way. What you learn here can then be applied to a wider variety of CJKV input programs, possibly even ones that have yet to be developed.

Unlike English and other Latin-based languages, to include Vietnamese, along with typical Korean text, which permits direct keyboard entry for the majority of its characters, there are two ways to input CJKV characters, as follows:

- Direct
- Indirect

Input by encoded value is a direct means of input, and unambiguously allows you to access CJKV characters.* However, this is not very intuitive. One can certainly memorize

* This is not always true. For example, when an input method can accept ISO-2022 and Row-Cell codes, there are many ambiguous cases, such as the four-digit code 3021. In GB 2312-80, this code can result in either 啊 (hexadecimal ISO-2022-CN encoding) or 镜 (decimal Row-Cell notation). However, if one works with a single encoding, such as Unicode, and uses Unicode scalar values, the original statement is true.

the hexadecimal ISO-2022-JP value 0x5178 or the Unicode scalar value U+528D for the Japanese kanji 劍 (*ken* or *tsurugi*), but imagine doing this for thousands or tens of thousands of ideographs. While input by reading may yield more than one candidate character from which to choose, it is—in a seeming paradox—the most productive and most widely used method yet invented. Figure 5-1 illustrates the four possible stages of input: it shows how the flow of input information travels and how different input methods and keyboards interface at each stage.

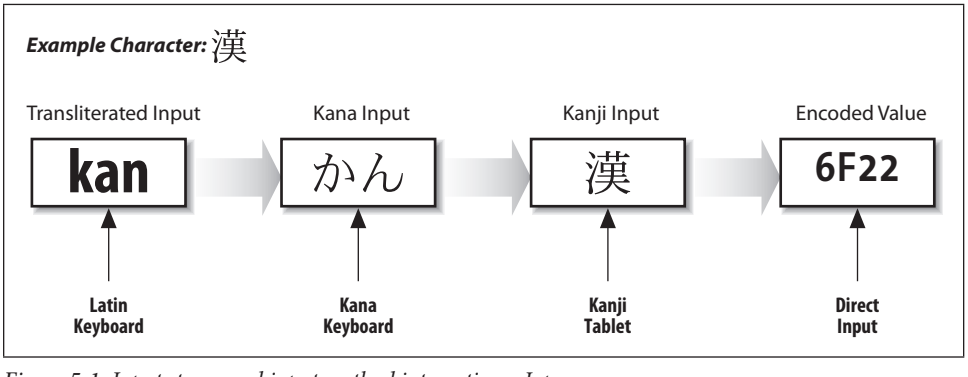


Figure 5-1. Input stages and input method interaction—Japanese

The Input Method

CJKV input software is usually referred to as an *input method* (or *FEP*, which is an abbreviation for *front-end processor*). The input method is aptly named because it captures the keyboard input, processes it to become the intended text, and then sends it to the application or other software. Typically, the input method runs as a separate process in its own input window, largely independent of the underlying application. Now, thanks to OS-level APIs, the use of input methods is much more seamless than in the past.

The input methods that were available at the time when the first edition of this book was published was already quite rich. Microsoft Windows OSes, at least the Japanese-enabled versions thereof, had several input methods from which to choose. These included ATOK, egbridge, Katana (刀 *katana*), VJE, and Wnn. Some industrious people had even adapted Unix-based Japanese input methods so that they could run under non-Japanese MS-DOS systems. One such example was the adaptation of an input method called SKK for use with MOKE, a Japanese text editor that ran on MS-DOS. Mac OS-based systems, specifically those that ran Mac OS-J or Mac OS with the Japanese Language Kit, offered several Japanese input methods, such as ATOK, egbridge, Katana, MacVJE, Wnn, and ことえり (*kotoeri*). Kotoeri was bundled with the OS. Unix offered input methods such as SKK, Wnn, Canna, and kinput2—these were available from a variety of online sources or bundled with other programs. Some of these Japanese input methods are described in more detail at the end of this chapter.

Now the landscape is different. The OS-bundled input methods have matured and offer rich functionality. Mac OS X still provides Kotoeri as its Japanese input method, but it is very different from earlier iterations. Windows provides Global IME. It seems that JustSystems' ATOK remains as the most popular third-party Japanese input method.

Most input methods run as separate processes from the programs that ultimately use the input, such as text editors and word processors. This allows you to use a single input method with many programs. It can also allow the use of more than one input method (not simultaneously, though, but by switching between them with a special keystroke, menu item, or control panel).

Many of these Japanese input methods allow you to emulate others, at least they did so in the past when many were available, and each enjoyed a loyal following. For example, MacVJE for Mac OS allowed users to emulate egbridge, ATOK, Wnn, and TurboJIP keyboard commands. Some others offer the same flexibility. This seemed to be useful, especially when using multiple OSes where the Japanese input method you prefer to use is not available on the current OS. However, the emulation was not always complete. For example, egbridge for Mac OS could emulate MacVJE keyboard commands, but its emulation did not have the same set of functions. One expected to find slight differences—after all, egbridge was merely *emulating* the other input method.

All CJKV input methods share many similar features and functionalities. Although there are other tasks that they must perform, sometimes specific to an input method, at a minimum an input method must provide a way to perform the following basic operations:

- Switch between CJKV and Latin writing modes
- Convert the input string into one or more ideographs
- Select from a list of candidate ideographs
- Accept the selected or converted string, which is subsequently sent to the application in which the string is to be used

The ability to switch between writing modes is necessary because CJKV text can be composed of both CJKV and Latin text. Most input methods do not input ideographs directly, so a keystroke is used to tell the input software to convert the input text and, because many ideographs and ideograph compounds share the same reading, it is necessary to list all the ideographs or ideograph compounds with the same reading. The user then selects the ideograph or ideograph compound that they intended to input. Finally, the converted input string is provided to the application, which is referred to as accepting the input string. Table 5-8 lists several popular Japanese input methods and the keystrokes used to perform the input tasks just described. The Xfer and Nfer keystrokes in Table 5-8 refer to special keys sometimes found on keyboard arrays.

Table 5-8. Keystrokes for common input tasks—Japanese

Operation	Kotoeri	Wnn	Canna	SKK
Japanese ⇒ English	Cmd-space	F3	Xfer or C-o	l
English ⇒ Japanese	Cmd-space	F3	Xfer or C-o	C-j
Conversion	space or C-c	C-w or C-z	Xfer or space	uppercase
Candidate selection	arrows	up/down arrows	arrows	space
Accepting	return or enter	C-l	Nfer or return	C-j

As you can clearly see from Table 5-8, the basic input tasks are common among these various Japanese input methods, but the keystrokes used to invoke them are necessarily different. For Korean input, the keystroke "Shift-space" is usually used to toggle between English and Korean modes, though Windows uses a dedicated toggle key, and Mac OS X users Command-space. These and other input methods are described at the end of this chapter.

Our discussion will proceed from the most widely used input method to the least widely used. Since most symbols and kana can be input directly, these discussions focus primarily on the input of ideographs and the problems inherent in that process. Ideographs are problematic because they number in the thousands or tens of thousands, and thus require an indirect means of input.

The Conversion Dictionary

While the input method provides the mechanical and algorithmic power behind the ability to enter CJKV text, it is the conversion dictionary that allows input strings to be converted into ideographs, by matching user input with one or more entries in a dictionary that contains words and other language elements.

Conversion dictionaries come in a variety of formats, and define more than simply readings for ideographs. It is also possible to specify additional information to help the input method decide how to use the entry, such as grammatical information (part of speech, type of verb or adjective, and so on). Contemporary input methods include hundreds of thousands of entries, and provide users with very rich a complete input experiences.

JSA, in cooperation with several Japanese input method developers, has published the standard designated JIS X 4062:1998, *Format for Information Interchange for Dictionaries of Japanese Input Method* (仮名漢字変換辞書交換形式 *kana kanji henkan jisho kōkan keishiki*). This standard sets a precedent for establishing a common interchange format for representing the contents of input methods' conversion dictionaries.

Input by Reading

The most frequently used CJKV input method is by reading, which is also referred to as *pronunciation*. Input by reading is by far the most intuitive way to input CJKV text. There are three basic units by which input readings can be converted into ideographs:

- Single ideograph
- Ideograph compound, meaning a string of two or more ideographs
- Ideograph phrase, meaning a string of one or more ideographs, along with additional nonideograph characters (this is not applicable to all CJKV locales)

You may have heard of input methods that claim to be able to convert whole sentences at once. In fact, this is actually describing the ability to input whole sentences, which are parsed into smaller units, usually ideograph phrases, and then converted. This often introduces parsing errors, and it is up to the user to adjust each phrase.

For example, if you want to input the Japanese phrase 漢字は, pronounced *kanji-wa*, and meaning "the kanji," you have three choices: you can input each character, you can input a phrase as a compound, or you can input a phrase as a string of characters. Table 5-9 shows how you can input each character one at a time, and how this results in candidates from which you may choose for each character.

Table 5-9. Input by reading—single ideograph

Target	Latin input	Kana input	Candidates
漢	KAN(N)[a]	かん	乾 侃 冠 寒 刊 勘 勧 巻 喚 堪 姦 完 官 寛 干 幹 患 感 慣 憾 換 敢 柑 桓 棺 款 歓 汗 漢 澗 潅 環 甘 監 看 竿 管 簡 緩 缶 翰 肝 艦 莞 観 諌 貫 還 鑑 間 閑 関 陥 韓 館 舘
字	JI	じ゛	事 似 侍 児 字 寺 慈 持 時 次 滋 治 爾 璽 痔 磁 示 而 耳 自 蒔 辞
は	HA	は	n/a[b]

a. Whether you need to type one or two Ns depends on the input method.

b. The character は resolves as itself—no conversion is necessary.

Table 5-10 illustrates how you can input this phrase as an ideograph compound plus the following hiragana, and how a shorter list of candidates results.

Table 5-10. Input by reading—ideograph compound

Target	Latin input	Kana input	Candidates
漢字	KANJI	かんじ゛	漢字 感じ 幹事 監事 完司
は	HA	は	n/a[a]

a. As in Table 5-9, the character は resolves as itself—no conversion is necessary.

Note how the candidate list became shorter, making selection among them much easier. Table 5-11 shows the effect of inputting the entire phrase as a single string of characters. The ability to perform this type of conversion depends on the quality of your input method.

Table 5-11. Input by reading—phrase

Target	Latin input	Kana input	Candidates
漢字は	KANJIHA	かんしゞゝは	漢字は 感じは 幹事は 監事は 完司は

For Japanese, the kana-to-kanji conversion dictionary (仮名漢字変換辞書 *kana kanji henkan jisho*) makes input by reading possible. Most Japanese input systems are based on a conversion dictionary that takes kana strings as input, and then converts them into strings containing a mixture of kanji and kana. The conversion dictionary uses a key (the reading) to look up possible replacement strings (candidates). Quite often a single search key has multiple replacement strings assigned to it. Conversion dictionaries typically contain tens of thousands of entries, and some now contain over 100,000.

Most kanji even have multiple readings assigned to them. How many really depends on the conversion dictionary used by a Japanese input method. For example, the kanji 日 can have up to nine unique readings, depending on the context (and the conversion dictionary!). They are び (*bi*), ひ (*hi*), に (*ni*), ぴ (*pi*), か (*ka*), じつ (*jitsu*), にち (*nichi*), につ (*nitsu*), and たち (*tachi*).* While this is an extreme example, this phenomenon is not unique.

As you just saw, the most widely used and most efficient method for inputting kanji by reading is to handle them as kanji compounds or kanji phrases, not as single characters. If you desire a single kanji, it is often more efficient to input it as a compound or phrase, and then delete the unwanted character or characters.

Up to this point we have dealt primarily with ideographs. What about other characters, such as symbols? Typical CJKV character set standards contain several hundred symbols. As you have seen, Japanese kana and Korean hangul can be input more or less directly. Symbols and other miscellaneous characters may also be input using the symbol's name, reading, or close relative on the keyboard. Table 5-12 provides examples of symbols, along with their candidates.

* There are even more readings for this kanji if you consider readings for Japanese names.

Table 5-12. Input by symbol name for nonideographs

Target	Latin input	Kana input	Candidates
〒	YUUBIN(N)	ゆうびん	郵便 〒[a]
】	KAKKO	かっこ	括弧 （〔「『【）〕」』】

a. The character 〒 is not an ideograph, but rather the Japanese postal code symbol.

Note how ideograph compounds are also listed as candidates. There are usually perfectly good ideograph equivalents for these symbol names.

Chinese is unique in that its reading-based input methods provide one additional way to reduce the amount of candidate hanzi: indication of tones. Table 5-13 lists all the hanzi in GB 2312-80 with the reading *han*, followed by how they break down into candidates for each of the four tones. Note how some hanzi have more than one reading—well, the same reading, but a different tone. One such example has both instances underlined in Table 5-13.

Table 5-13. Reducing the number of hanzi candidates through tones

Reading	Number of hanzi	GB 2312-80 hanzi
han—*no tone*	31	顸酣蚶鼾憨邗邯韩晗含焓涵寒函喊罕阚翰菡撼捍撖旱颔憾悍焊汗瀚汉
hān	5	酣憨顸蚶鼾
hán	10	邯韩含涵寒函汗邗晗焓
hǎn	3	喊罕阚
hàn	13	翰撼捍旱憾悍焊汗汉菡撖瀚颔

Nearly all input software uses its own proprietary conversion dictionary or dictionaries. This means that no two input programs behave the same. Luckily, such software allows the user to create a user dictionary in which new search keys can be created and candidate replacement strings assigned. Whether or not there is a logical ordering in a conversion dictionary depends on the dictionary itself. Entries are typically ordered by reading (remember that the key for lookup is usually kana, which implies ordering by reading). The values associated with the keys are usually ordered by their relative frequency of use—more obscure ideographs or ideograph compounds appear further down the list of candidates. It is quite common to find CJKV input method programs that include a learning function (学習機能 *gakushū kinō*). This learning function allows the subtle rearrangement of values associated with keys. The default value for a particular key can change depending on how often the user selects a given candidate. For example, if the default value for the input string けん (*ken*) is 犬 (meaning "dog"), and if I constantly select the candidate 剣 (the ideograph I use to write my given name), 剣 eventually becomes the default value associated with the key けん. Exactly when the default changes depends on the software—sometimes only one occurrence is sufficient.

More advanced input methods implement parsers that use grammatical information for making decisions. This allows users to input whole sentences by reading, and then lets the software slice the input into units that are manageable for conversion. Like I mentioned earlier, errors in parsing are quite common.

Input by Structure

Input by reading, whether transliterated using Latin characters or not, is by far the most common method for inputting CJKV text, and it is also the easiest to learn. But there are times when input by reading fails to locate the desired ideographs, or when it is simply not fast enough.

Ideographs, as described in Chapter 2, are composed of radicals or radical-like elements, which in turn are composed of strokes. Ideographs with related shapes—which sometimes means that they have somewhat related meanings—can thus be grouped together. All of the following input techniques involve the structure of the ideograph:

- By indexing radical
- By number of strokes—total or residual
- By stroke shapes
- By corner

In case it is not yet obvious, there exist input techniques that use both structure and reading. Some examples are covered later in this chapter in the section entitled "Input by Multiple Criteria."

Input by indexing radical

The most common way to organize and arrange ideographs, other than by reading, is by indexing radical. In the past, some input methods allowed only certain ideographs to be input by indexing radical. This was usually because that is how the ideographs were arranged in a character set, and it is quite trivial to slice up this collection of characters into sets indexed by the same radical. For example, the hanzi in GB 2312-80 Level 2 and the kanji in JIS X 0208:1997 Level 2 are ordered by radical followed by the number of residual strokes. The current trend, fortunately, is for input methods to allow users to input all ideographs by radical, even those that are ordered differently, such as by reading (GB 2312-80 Level 1, JIS X 0208:1997 Level 1, and KS X 1001:2004) or total number of strokes (Big Five and CNS 11643-2007). The CJK Unified Ideographs in Unicode are ordered by indexing radical, but are arranged in different blocks, such as the URO and Extensions A through C. The URO even includes a small number of CJK Unified Ideographs that are not part of the URO proper, such as those after U+9FA5. There are now CJK Unified Ideographs from U+9FA6 through U+9FC3, at least through Unicode version 5.1

Table 5-14 illustrates examples of input by indexing radical, using two kanji from JIS X 0208:1997 Level 2. Obviously, if the additional kanji in JIS X 0213:2004 were to be

considered, the lists of candidate kanji would be longer and would include the traditional form of the kanji 漢, specifically 漢, as one of the candidates.

Table 5-14. Input by indexing radical

Target	Indexing radical	Candidates
漢	氵	水 氷 永 氾 汀 汁 求 汎 汐 汕 汗 汚 汝 汞 江 池 汢 汨 汪 汰 汲 汳 決 汽 汾 沁 沂 沃 沈 沌 沐 漢 滝 滞 滬 滲 滷 滸
字	子	子 孔 孕 字 存 孚 孛 孜 孝 孟 季 孳 学 孩 孤 孫 孰 孱 孳 孵 孺

More sophisticated input methods display a graphic palette of the 214 radicals, or a reduced set of 186 radicals if you are using an input method based on GB 2312-80. These 214 radicals originated in a Chinese dictionary entitled 康熙字典 (*kāngxī zìdiǎn*), which was published in 1716. Each of these 214 radicals has a name or number assigned to it. Some of these radicals even have variants, most of which have unique names. The number represents the relative order within the set of radicals. For example, the last radical is assigned the number 214. You may encounter input methods that require that radicals be input through their names, numbers, or number of strokes. Table 5-15 provides some examples of radicals, along with their names (in Japanese), meanings, numbers, and number of strokes.

Table 5-15. Radical names and numbers

Radical	Name	Meaning	Number	Stroke count
一	一 (ichi)	one	1	1
丨	棒 (bō)	bar	2	1
女	女 (onna)	woman	38	3
子	子 (ko)	child	39	3
疒	病垂 (yamaidare)	illness enclosure	104	5
石	石 (ishi)	stone	112	5
貝	貝 (kai)	shell	154	7
鳥	鳥 (tori)	bird	196	11
鼻	鼻 (hana)	nose	209	14

Appendix J provides a complete listing of the classical 214 radicals, along with their names in Japanese. China's reduced set of 186 radicals, as used in the GB 2312-80 standard, is provided in Appendix G. Other Chinese dictionaries use a different set of radicals, such as 现代汉语词典 (*xiàndài hànyǔ cídiǎn*), which uses a reduced set of 188 radicals.

There are some ideographs for which the indexing radical is not obvious. But it is almost always possible to count the number of strokes. This brings us to the next input technique that involves the number of strokes of an ideograph.

Input by number of strokes

Almost all ideographs have a unique number of strokes, meaning that the number of strokes for an ideograph is unambiguous. Clearly, for a given number of strokes, there may be a large number of corresponding ideographs. The ability to count strokes paves the way for an input technique whereby the number of strokes is entered by the user and all the candidates are displayed for selection.

Interestingly, there are a small number of ideographs whose actual number of strokes can be debated, sometimes due to locale differences. This difference is typically manifested as a single stroke.* Quality input methods account for these differences in number of strokes, and they allow the user to input characters using multiple stroke counts. Also note that the number of strokes occasionally depends on the glyph. For example, an input method based on the JIS C 6226-1978 character set may behave strangely if the JIS X 0208:1997 character set were to be swapped. Consider Row-Cell 16-02, whose JIS C 6226-1978 glyph is 唖 (11 strokes), and whose JIS X 0208:1997 glyph is 唖 (10 strokes).†

Table 5-16 provides a candidate list for two different numbers of strokes, based on the two ideographs that have been serving as our targets.

Table 5-16. Input by stroke count

Target	Stroke count	Candidates
漢	13	愛 葦 飴 暗 意 違 溢 碓 園 煙 猿 遠 鉛 塩 嫁 暇 禍 嘩 蛾 雅 解 塊 慨 碍 蓋 該 較 隔 楽 滑 褐 蒲 勧 寛 幹 感 漢 頑
字	6	旭 扱 安 伊 夷 衣 亥 芋 印 因 吋 宇 羽 迂 臼 曳 汚 仮 会 回 灰 各 汗 缶 企 伎 危 机 気 吉 吃 休 吸 朽 汲 兇 共 匡 叫 仰 曲 刑 圭 血 件 交 光 向 后 好 江 考 行 合 此 艮 再 在 旨 死 糸 至 字 寺 次 而 耳 自

Interestingly, because the traditional form of kanji 漢, specifically 漢, has 14 strokes, it is thus not listed as one of the candidates.

Input by stroke shapes

Most users believe that input by indexing radical or by number of strokes is more than sufficient when input by reading fails to produce the desired ideograph. One of the input methods that allows users to enter ideographs by other shape-based criteria is known as the *Wubi Method* (五笔输入法 *wǔbǐ shūrùfǎ*), developed by Yongmin Wang (王永民 *wáng yǒngmín*). The title literally means "Five-stroke Input Method." A typical Wubi code consists of the ideograph's overall shape plus its first, second, and final stroke.

* Some examples of radical or radical-like elements that have an ambiguous number of strokes are provided in Table 11-4.

† When we consider Unicode and JIS X 0213:2004, these two characters have separate and distinct code points. The JIS C 6226-1978 glyph is in JIS X 0213:2004 at Plane-Row-Cell 1-15-08, which corresponds to U+555E in Unicode. The JIS X 0208:1997 glyph corresponds to U+5516 in Unicode.

The *Shouwei Method* (首尾输入法 *shǒuwěi shūrùfǎ*) is an input method that specifies the first and final strokes or shapes of an ideograph.

Input by corner

If we ignore the indexing radical and number of strokes of an ideograph, it is still possible to input them by categorizing the shapes that are at each corner. The *Four Corner Code*, which is described in Chapter 11, is typically used for locating characters in ideograph dictionaries.

However, the *Three Corner Code*, invented by Jack Huang (黃克東 *huáng kèdōng*) and others, is designed specifically for character input.

Input by other structures

There are ways to describe the structure of ideographs that go beyond indexing radicals, strokes, and corners. Many Chinese input methods take advantage of the fact that ideographs can be categorized by shapes.

An input method known as the *Cangjie Method* (倉頡輸入法 *cāngjié shūrùfǎ*), developed by Bangfu Zhu (朱邦復 *zhū bāngfù*), allows the user to input ideographs by radical-like elements. More details about the Cangjie Method can be found later in this chapter when its keyboard array is discussed. This input method is commonly used in Taiwan and Hong Kong. In fact, many dictionaries provide Cangjie codes for hanzi.

Also, the *Zheng Code Method* (郑码输入法 *zhèngmǎ shūrùfǎ*), developed by Yili Zheng (郑易里 *zhèng yìlǐ*) and Long Zheng (郑珑 *zhèng lóng*), also uses radical-like elements for inputting ideographs. It is used in all Chinese locales by virtue of being included with Microsoft Windows.

Input by Multiple Criteria

Some tools were specifically designed to input ideographs or other characters with more than one of the input methods just described. This is very useful because a user can significantly narrow a search by giving more than one input criterion.

Like a typical bibliographic search, the more search parameters you provide to the software, the shorter the candidate list becomes. However, don't expect to find very many programs, other than dedicated CJKV character dictionary software, that accept multiple search criteria.

As you will learn in the next section, inputting CJKV characters by their encoded value or dictionary number is a direct method, and it always results in a single character match.

Input by structure and reading

The ultimate goal of any indirect input technique should be to reduce the number of candidates from which the user must choose. This is known as reducing the number of collisions, that is, reducing the number of characters that share the same attributes according

to an input technique. It is possible to combine reading and structure attributes to form a new input technique that can effectively reduce the number of collisions.

Examples of input methods that combine reading and structure information include the *Tze-loi Method* (子來輸入法 *zǐlái shūrùfǎ*), developed by Tze-loi Yeung (楊子來 *yáng zǐlái*), and the *Renzhi Code Method* (认知码输入法 *rènzhī mǎ shūrùfǎ*). A Tze-loi code consists of three keystrokes. The first two are based on the character's structure (the upper-left and lower-right corner), and the third represents the first sound in the character's reading. A Renzhi code, like a Tze-loi code, consists of three keystrokes. The first keystroke is the first character of its Pinyin reading, and the last two keystrokes are its first and last strokes. Renzhi codes, however, can also consist of other types of elements, and can consist of as little as two keystrokes or as many as four. Table 5-17 lists a handful of hanzi, along with their Tze-loi input codes.

Table 5-17. Examples of Structure Plus Reading input methods—Tze-loi Method

Hanzi	Tze-loi code	Tze-loi—QWERTY
晶	日＋日＋J	JJJ
品	口＋口＋B	HHB
法	丶＋厶＋F	6ZF

Input by Encoding

This CJKV input method is based on fixed values for each character, specifically their encoded values. This is a direct way to input and is typically used as a last resort for inputting characters. This means that one can unambiguously (that is, without candidates) input a single character. Note that it is most common to use hexadecimal values when inputting characters, with the exception of Row-Cell, which usually requires four decimal digits.

Input by encoding makes use of the encoded values of the target characters. As most systems process only a single code internally, yet accept different encoded values, some sort of conversion between codes is still being performed. For example, many Japanese systems can accept both hexadecimal ISO-2022-JP and Shift-JIS codes since they can be easily distinguished from one another—they occupy separate encoding regions.

Most input software includes character tables indexed by one or more of these encoding methods. Some programs even have these tables built in as on-screen palettes so that you need not ever consult the printed documentation.

Many input methods allow the user to select which encoding to use when inputting by code, and those which are really good automatically detect which code the user has selected. This is simple for ISO-2022-JP and Shift-JIS encodings because they occupy different encoding regions, but Row-Cell notation and EUC encoding pose special problems. Some implementations require that a period or other delimiter separate the Row from the

Cell of a Row-Cell code. Input by EUC code is quite rare for Japanese, but quite common for Chinese (GB 2312-80) and Korean (KS X 1001:2004).

Table 5-18 lists two hanja, along with their encoded values according to a variety of Korean encodings, including Unicode.

Table 5-18. Input by encoded value—Korean

Hanja	Row-Cell	EUC-KR	Unicode	ISO-2022-KR	Johab
漢	89-51	F9D3	6F22	7953	F7D3
字	77-14	EDAE	5B57	6D2E	F1AE

Most input systems provide at least two of these input methods, usually Row-Cell and hexadecimal ISO-2022. Sometimes the encoding method supported by the software you are running is another option. For example, Japanese OSes that processed Shift-JIS internally, such as Mac OS-J, Windows 3.1J, and Windows 95J, provided input methods that gave the user the ability to perform code input by Row-Cell, hexadecimal ISO-2022-JP, and hexadecimal Shift-JIS.

If you need to convert lots of these "codes" into actual characters, you are better off writing a quick tool for this purpose (perhaps using Perl), or else finding software that provides this functionality. JCode, described in Chapter 9, provides this level of functionality, at least for Japanese.

Input by Other Codes

China's *Telex Code* (电报码/電報碼 *diànbáomǎ*),* developed in 1911 for the purpose of hanzi interchange, is yet another code that can be used to unambiguously input ideographs—sort of. A Telex Code is composed of four decimal digits, and it ranges from 0001 to 9999. The ordering of ideographs in Telex Code is by radical, and then number of strokes.

It is important that you know that a Telex Code does not distinguish between simplified and traditional ideographs; they have been effectively merged into a single Telex Code. So, for example, the two related hanzi, 剑 and 劍, share the same Telex Code, specifically 0494.

Other possible codes may include numbers used to index ideographs in specific dictionaries. The Four Corner Code is also used, but is usually restricted to indexes in ideograph dictionaries. See Chapter 11's section entitled "Four Corner Code" for a brief description.

* Sometimes referred to as 电报明码/電報明碼 (*diànbáomíngmǎ*).

Input by Postal Code

Japanese postal codes (郵便番号 *yūbin bangō*) consist of three or seven digits.* Some conversion dictionaries include entries that correspond to these postal codes, and the strings associated with those entries represent the place or places that correspond to the codes. Table 5-19 includes some examples of three-digit postal code input.

Table 5-19. Input by postal code—three-digit

Postal code	Candidate locations
001	北海道札幌市北区
500	岐阜県岐阜市
999	山形県酒田市, 山形県最上郡, 山形県上山市, 山形県飽海郡, 山形県北村山郡, 山形県尾花沢市, 山形県長井市, 山形県西置賜郡, and so on

Japanese place names, especially for nonnative speakers of Japanese, can be difficult to learn and pronounce. Japanese addresses usually contain postal codes, and for those who use Japanese input methods that support this type of input, much digging in dictionaries to find out how to pronounce, and thus easily enter, each kanji can be avoided.

Input by Association

Input by association (連想入力 *rensō nyūryoku*) is an older Japanese input method, and it is often referred to as the *two-stroke input method* (二ストローク入力方式 *ni sutorōku nyūryoku hōshiki*). It is unlike input by reading in that there is only one kanji associated with each pair of keystrokes—no candidate selection is required.

Input by association works by associating two characters, usually kana, to a single kanji. These two kana are usually associated with the kanji by reading or meaning. For example, the two katakana ハハ (pronounced *haha*) are associated with the kanji 母, whose reading happens to be *haha*.

Needless to say, this input method has a long learning curve, but skilled text entry operators can use it quite effectively.

There are many in Japan who feel that *input by unassociation* (無連想入力 *murensō nyūryoku*) is better. This means that the relationship between a kanji and its two keystrokes is arbitrary. In fact, such an input method has been developed and has quite a following. It is called *T-Code*, and it is available for a variety of OSes, such as Mac OS, MS-DOS, Unix, and Windows. More information about T-Code itself is provided later in this chapter.

* Seven-digit Japanese postal codes were introduced in early 1998.

User Interface Concerns

The ability to simply input CJKV text using a variety of input methods and techniques is not enough to satisfy a large number of users. This is the start of ergonomic considerations. For example, having the input and selection of characters take place at the cursor position is highly desired. This is referred to as *inline conversion*.

Inline Conversion

CJKV input methods typically provide their own input window, because they run as a separate process from the application in which the text will be inserted. CJKV input takes place in the dedicated window, is then sent to the current application, and finally is set into the current cursor position. As you can expect, this is far from ideal, because the user must keep his eyes trained on the current cursor position and the input method's own UI. The solution to this UI problem is inline conversion (インライン変換 *inrain henkan*). In the past, there were standard protocols developed by input method developers that could be used in applications such that inline conversion could take place. Now, the facilities for inline conversion are provided by the OS and are used by input methods and applications.

Many CJKV-capable word processors come bundled with their own input method, which usually means that there is support for inline conversion, at least for the bundled input method. Be sure to read the application manual to learn whether there is inline conversion support for a particular input method. Inline conversion support is, fortunately, very common these days, thanks to the facilities that are provided by the OS, such as Mac OS X and Windows.

Keyboard Arrays

Our discussion continues with a description of a number of keyboard arrays in use in CJKV locales, along with accompanying figures and tables so that comparisons can be more easily drawn between them. The keyboard arrays that are covered in this chapter have been divided into the following eight categories:

- Two Western keyboard arrays—*QWERTY* and *Dvorak*
- One ideograph keyboard array—*kanji tablet*
- Two Chinese input method keyboard arrays—*Wubi* and *Cangjie*
- Three zhuyin keyboard arrays
- Eight kana keyboard arrays—*JIS*, *New-JIS*, *Thumb-shift*, two variations of *50 Sounds*, and two variations of *TRON*
- Two hangul keyboard arrays—*KS* and *Kong*

- Two Latin keyboard arrays—*M-style* and *High-speed Roman*
- Three mobile keyboard arrays—one for Japanese, and two for Korean

The market for dedicated Japanese word processors experienced enormous flux in keyboard designs and usage. It seemed that for every new computer model that a hardware manufacturer introduced to the market, the same manufacturer introduced two or three dedicated Japanese word processor models. These word processors are much like computers, but the basic software for word processing is usually fixed and thus not upgradable.

Genuine computer OSes are designed for more general usage, so there is far less variety in keyboard arrays. In fact, some dedicated Japanese word processor keyboards may have more than one keyboard array imprinted on their keys. This is done by imprinting more than one character on each key. I once owned and used two dedicated Japanese word processors, specifically NEC's 文豪ミニ 5G and 7H.* On the tops of their keys were imprints for the QWERTY and JIS keyboard arrays, and on the sides of the keys were imprints for the 50 Sounds keyboard array.

The intent of this chapter is not to teach you how to use these keyboard arrays effectively, but rather to tell you a little bit about them and their characteristics. I should also point out that this book does not definitively cover all keyboard arrays. It cannot. Doing so is well beyond the scope of this book. However, when it comes to practical usage and market dominance, the QWERTY keyboard is still the most popular.

Western Keyboard Arrays

Western keyboard arrays are used quite commonly for CJKV input because typical CJKV input methods allow the user to input text phonemically through the use of Latin characters. This is referred to as *transliterated input*. The input method subsequently converts this transliterated input into CJKV characters. In fact, one study claimed that over 70% of Japanese computer users input Japanese through the use of Latin characters. This is not to say that keyboard arrays designed specifically for Japanese do not exist, but as you will see in the following sections, there are many from which to choose.

QWERTY array

The most widely used keyboard on the planet is known as the QWERTY keyboard array, so named because its first six alphabetic keys are for the characters *q*, *w*, *e*, *r*, *t*, and *y*. It was originally developed so that frequently used keys were spaced far from each other. In the days of mechanical typewriters, having such keys in proximity would often lead to a mechanical jam. However, most keyboards today are not mechanical, but electrical, so the original need for spacing out frequently used keys is no longer valid. However, the QWERTY keyboard array is so well entrenched that it is doubtful that it will ever be

* The Japanese word 文豪 is read *bungō* and means "literary master."

replaced. There have been many attempts at doing so, the most famous of which is the Dvorak keyboard array, covered in the next section.

Although written in Japanese, Koichi and Motoko Yasuoka (安岡孝一 *yasuoka kōichi* and 安岡素子 *yasuoka motoko*) have published a comprehensive book entitled キーボード配列 QWERTY の謎 (*kībōdo hairetsu QWERTY-no nazo*; NTT 出版, 2008) that details the history of the QWERTY keyboard array.* It is the most exhaustive book on the subject that I have ever seen. Figure 5-2 illustrates the basic QWERTY keyboard array.

Figure 5-2. The QWERTY keyboard array

To further demonstrate the true *de facto* nature of the QWERTY keyboard array, consider mobile devices, such as mobile phones. The mobile keyboard arrays that are used for such devices are based on a numeric keyboard, and the ability to enter nonnumeric data, meaning text, is accomplished through the use of specialized assignments of characters to the numeric keys. Some mobile devices do provide full keyboard arrays, sometimes in addition to the numeric keyboard. Interestingly, when full keyboard arrays are provided, it is based on the QWERTY keyboard array.

Dvorak array

There have been attempts at replacing the QWERTY keyboard array by providing an improved layout of keys, but none of them have succeeded to date. One such attempt was called the Dvorak keyboard array, developed in the 1930s by August Dvorak and William Dealey. Keys on the Dvorak keyboard array are positioned such that approximately 70% of English words can be typed with the fingers in the home position. Compare this with

* *http://www.nttpub.co.jp/vbook/list/detail/4176.html*

only 32% in the case of the QWERTY keyboard array. See Figure 5-3 for an illustration of the Dvorak keyboard array.

Figure 5-3. The Dvorak keyboard array

To date, the Dvorak keyboard array has not succeeded in replacing the QWERTY array. This only goes to show that efficiency does not always make an item more appealing.

Ideograph Keyboard Arrays

The first Japanese keyboards that were able to accommodate the Japanese writing system were called *kanji tablets*. These were huge keyboards that contained thousands of individual keys.

The standard designated JIS X 6003-1989, *Keyboard Layout for Japanese Text Processing* (日本語文書処理用文字盤配列 *nihongo bunsho shori-yō mojiban hairetsu*), defines a keyboard array that contains a total of 2,160 individual keys.* The kanji tablet shown in Figure 5-4 is 60 keys wide by 36 keys deep (it is also available in another orientation with fewer keys). The 780 most frequently used kanji are in Level 1, 1,080 additional kanji are in Level 2, and 300 non-kanji are in the remaining keys.

Some Japanese corporations have even defined their own kanji tablet layouts, but this type of Japanese input device is quickly becoming obsolete. Japanese input methods have developed to the point where much smaller keyboard arrays, such as those already discussed, are more efficient and easier to learn.

* Previously designated JIS C 6235-1984

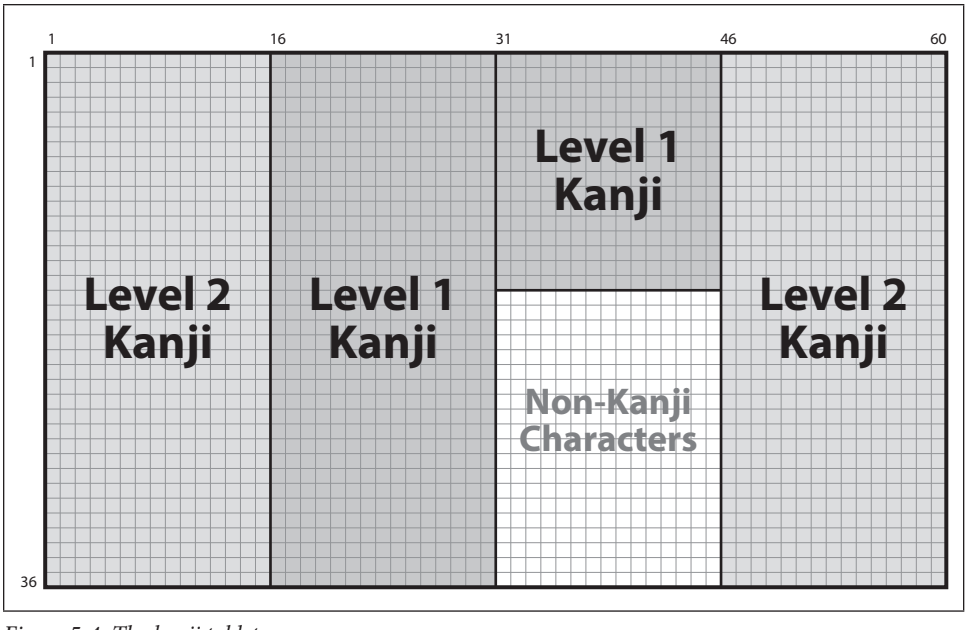


Figure 5-4. The kanji tablet array

Chinese Input Method Keyboard Arrays

There have been dozens of keyboard arrays designed for use with specific Chinese input methods. The keys of such keyboard arrays have imprinted on them ideographs or ideograph elements (such as radicals). The two keyboard arrays that are discussed in this section are the Wubi and Cangjie arrays, which appear to be the most popular.

Wubi array

The Wubi array is designed to be used with the Wubi Method, introduced earlier in this chapter in the section entitled "Input by stroke shapes." A Wubi code consists of three or four elements, but four elements is the most common. The Wubi Method specifies five basic categories of character roots, according to the first stroke's shape, as shown in Table 5-20. Also provided are the key cap hanzi and their location on the QWERTY keyboard array.

Table 5-20. Wubi Method character roots

Row		Cell	Code	Key cap	QWERTY	Example character roots
Horizontal	1	1	11	王	G	王主一五戋
		2	12	土	F	土士二十干寸雨
		3	13	大	D	大犬三古石厂
		4	14	木	S	木丁西
		5	15	工	A	工匚七弋戈廾廿艹
Vertical	2	1	21	目	H	目丨卜上止
		2	22	日	J	日曰刂早虫
		3	23	口	K	口川
		4	24	田	L	田甲口四皿车力
		5	25	山	M	山由冂贝几
Curve[a]	3	1	31	禾	T	禾丿竹彳夂攵
		2	32	白	R	白手扌斤
		3	33	月	E	月舟彡衣乃用豕
		4	34	人	W	人亻八
		5	35	金	Q	金钅勹儿夕
Curve[b]	4	1	41	言	Y	言讠亠丶广文方圭
		2	42	立	U	立冫丬六辛疒门
		3	43	水	I	水氵小
		4	44	火	O	火灬米
		5	45	之	P	之辶廴一宀
Corner	5	1	51	已	N	已己巳乙尸心忄羽
		2	52	子	B	子孑也凵了阝耳卩
		3	53	女	V	巛女刀九彐臼
		4	54	又	C	又厶巴马
		5	55	纟	X	纟幺弓匕

a. Curves that extend from upper-right to lower-left.

b. Curves that extend from upper-left to lower-right.

Figure 5-5 illustrates the Wubi array, which demonstrates how the 25 hanzi used as key caps are assigned to the keys of the QWERTY array—all Latin character keys except for "Z."

Figure 5-5. The Wubi keyboard array

Three of the principles of the Wubi Method can be summarized, in simple terms, as follows:

- If the hanzi is one that is used as a key cap (a limited set of 25 hanzi), then its Wubi code is four instances of that keystroke—for example, the Wubi code for the hanzi 言 is "YYYY."
- If the hanzi is one of the character roots, as shown in the "Example character roots" column of Table 5-20, its Wubi code consists of that keystroke plus the first, second, and final strokes of the character—for example, the Wubi code for the hanzi 西 is SGHG ("S" for the 木 key cap, "G" for the first stroke 一, "H" for the second stroke 丨, and "G" for the final stroke 一). If the number of strokes in the character root is three, its Wubi code consists of that keystroke plus the first and final strokes of the character—for example, the Wubi code for the hanzi 丁 is SGH ("S" for the 木 key cap, "G" for the first stroke 一, and "H" for the final stroke 丨).
- If the hanzi is not a character root, but contains more than three character roots, its Wubi code is the key caps for the first, second, third, and final character roots—for example, the Wubi code for the hanzi 照 is JVKO ("J" for the first character root 日, "V" for the second character root 刀, "K" for the third character root 口, and "O" for the final character root 灬). If the character is not a character root, but contains less than four character roots, its Wubi code is the key caps for all the roots plus an extra identification code—for example, the Wubi code for the hanzi 苗 is ALF ("A" for the

first character root 廾, "L" for the second character root 田, and "F" for the extra identification code).*

The Row and Cell values that make up the Code column of Table 5-20 are used for other Wubi principles, and describing them is beyond this book's scope.

The dictionary entitled 標準中文輸入碼大字典 (*biāozhǔn zhōngwén shūrùmǎ dà zìdiǎn*) provides Wubi codes for most GB 2312-80 hanzi. Mac OS X provides two types of Wubi input methods for Simplified Chinese.

Cangjie array

There is a special keyboard array designed for the Cangjie input method, which is one of the most popular input methods for entering ideographs through the structure of hanzi.† A Cangjie code can consist of as few as one keystroke or as many as five, depending on the complexity of the hanzi. In many cases, the Cangjie code perfectly reflects the complete structure of hanzi, as illustrated in Table 5-21.

Table 5-21. Intuitive Cangjie codes

Hanzi	Cangjie code—QWERTY	Cangjie code—Graphic
一	M	一
二	MM	一＋一
三	MMM	一＋一＋一
日	A	日
昌	AA	日＋日
晶	AAA	日＋日＋日
曇	AAAA	日＋日＋日＋日
曇	AAAA	日＋日＋日＋日
品	RRR	口＋口＋口
塭	GMRW	土＋一＋口＋田
矗	MWMWM	一＋田＋一＋田＋一

* Not to be confused with *ALF*. See *http://en.wikipedia.org/wiki/ALF_(TV_series)*

† Yes indeed, this is the very same keyboard array that effectively stumped British Secret Agent James Bond (007) who always portrays himself as a know-it-all or all-around expert. Chinese Secret Agent Colonel Wai Lin, in the 1997 film *Tomorrow Never Dies*, had a hideout somewhere in Vietnam where 007 was confronted with the possibility of using this keyboard array. He threw up his hands and gave up. Too bad this book wasn't available in 1997—imagine 007 whipping out this blowfish-clad tome, which he then uses to learn the ins and outs of this keyboard array within moments, just in time to save all of mankind. Curiously, this keyboard array is not used in China, but rather Taiwan. So what was a Chinese Secret Agent doing with a keyboard array developed in Taiwan?

Table 5-22 illustrates less-intuitive Cangjie codes, which result from the fact that some graphic Cangjie codes can also represent simple strokes.

Table 5-22. Less-intuitive Cangjie codes

Hanzi	Cangjie code—QWERTY	Cangjie code—Graphic
酷	MWHGR	一＋田＋竹＋土＋口
劍	OOLN	人＋人＋中＋弓

Once you understand that 竹 can represent a downward curved stroke, 中 can represent a vertical stroke, and 弓 can represent a stroke with an angle at the end, these examples become more intuitive.

Figure 5-6 illustrates the Cangjie keyboard array, which illustrates the correspondence between QWERTY Cangjie codes and their graphic counterparts.

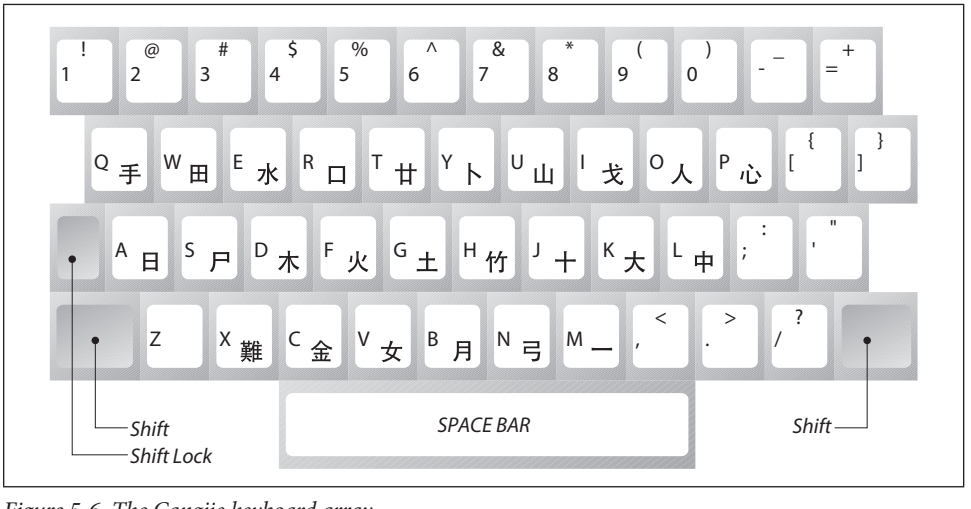


Figure 5-6. The Cangjie keyboard array

The dictionary entitled 標準中文輸入碼大字典 (*biāozhǔn zhōngwén shūrùmǎ dà zìdiǎn*) provides Cangjie codes for all Big Five hanzi plus the 3,049 hanzi in Hong Kong GCCS. Mac OS X provides Cangjie as one of its many input methods for Traditional Chinese.

Zhuyin Keyboard Arrays

In order to ease the input of Chinese by reading, several keyboard arrays that include zhuyin characters have been developed over the years. Luckily, only one of these keyboard arrays seems to have taken on the status of being the *de facto* standard.

Figure 5-7 illustrates the most popular instance of a zhuyin keyboard array, which is considered the *de facto* standard, and used on Mac OS X and Windows.

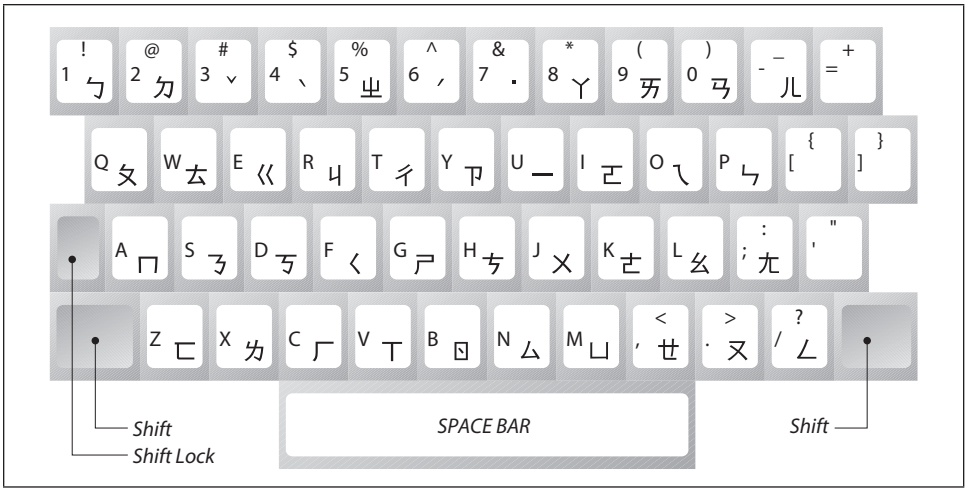


Figure 5-7. The most popular zhuyin keyboard array

Figure 5-8 illustrates yet another zhuyin keyboard array, specifically one that expresses the keyboard mappings used by TwinBridge.

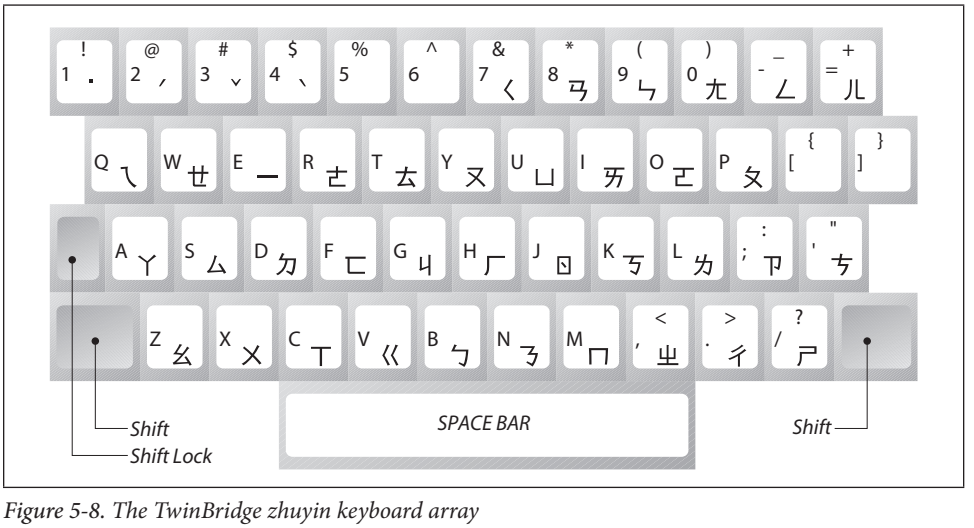


Figure 5-8. The TwinBridge zhuyin keyboard array

Figure 5-9 illustrates a zhuyin keyboard array that was designed for use with the Dai-E input method (大一輸入法 *dàyī shūrùfǎ*), developed by Timothy Huang (黃大一 *huáng dàyī*).

Figure 5-9. The Dai-E zhuyin keyboard array

More detailed information about Dai-E can be found in a book coauthored by Timothy Huang, entitled *An Introduction to Chinese, Japanese and Korean Computing* (World Scientific Publishing, 1989).

Kana Keyboard Arrays

The keyboard arrays discussed in this section have kana imprinted on their keys. One word of caution, though: simply because such keyboard arrays are considered standard doesn't mean that they have been widely accepted in the Japanese marketplace. Like the QWERTY array in the West, the Japanese have a similar keyboard array called the JIS array—one that is not very efficient, yet is the most commonly used and learned.

JIS array

The standard designated JIS X 6002-1985, *Keyboard Layout for Information Processing Using the JIS 7 Bit Coded Character Set* (情報処理系けん盤配列 *jōhō shori kei kenban hairetsu*), specifies what is known as the JIS keyboard array (JIS配列 *JIS hairetsu*).* This keyboard array is the most widely used in Japan (after the QWERTY array, that is) and can be found with almost every computer system sold there. This standard also defines that the QWERTY array be superimposed on the keys of the keyboard. Incidentally, this is how one accesses numerals.

The JIS array is not terribly efficient for Japanese input. Keys are arranged such that all four banks are required for it. This means that users must move their fingers a lot during typing, and must shift mode in order to access numerals, which are imprinted on the

* Previously designated JIS C 6233-1980

fourth bank of keys, along with kana. In addition, the keys are not logically arranged, so it is difficult to memorize the positions. Figure 5-10 provides an illustration of the JIS array.

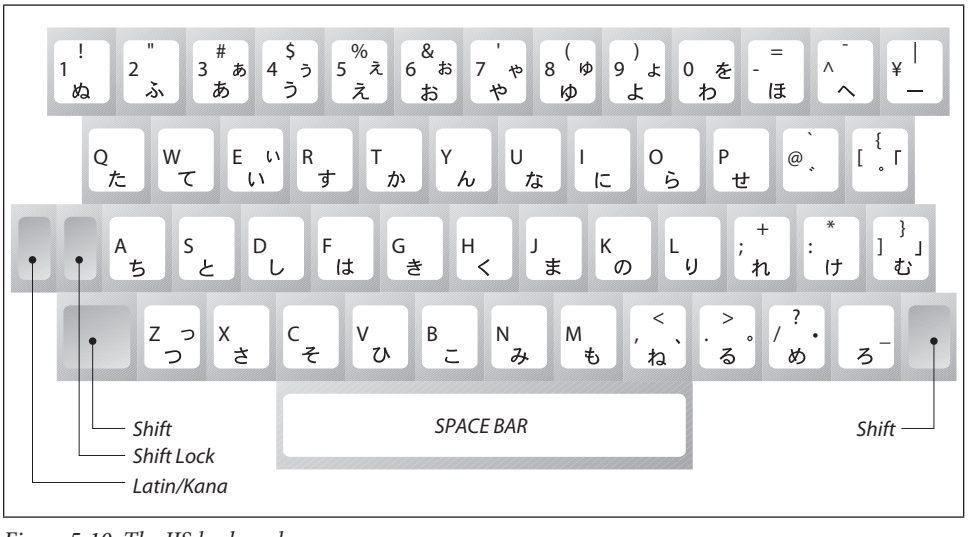


Figure 5-10. The JIS keyboard array

Note that the *dakuten* (゛) and *handakuten* (゜) have their own keys. This means that characters such as が (hiragana *ga*) must be input as the two keystrokes か (hiragana *ka*) and ゛ (*dakuten*). The same character が can be input as the two keystrokes *g* and *a* in the case of the QWERTY and other Latin keyboard arrays.

New-JIS array

The standard designated JIS X 6004-1986, *Basic Keyboard Layout for Japanese Text Processing Using Kana-Kanji Translation Method* (仮名漢字変換形日本文入力装置用けん盤配列 *kana kanji henkan kei nihonbun nyūryoku sōchi-yō kenban hairetsu*), specifies what is known as the New-JIS keyboard array (新 JIS 配列 *shin JIS hairetsu*).* This keyboard array, too, specifies that the QWERTY array be superimposed on the keyboard keys.

The kana on the keyboard are arranged on the first three banks of keys, and each key holds up to two kana (a Shift key is required to access all the kana). This allows the input of numerals without the use of a mode change. Figure 5-11 illustrates the New-JIS keyboard array.

* Previously designated JIS C 6236-1986

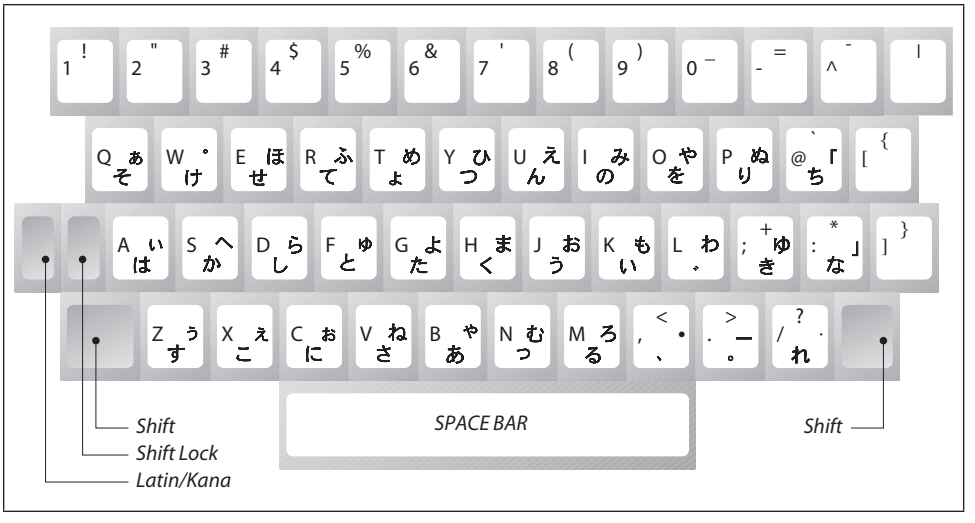


Figure 5-11. The New-JIS keyboard array

Although this keyboard array seems to be an improvement over the JIS array, it has not been widely accepted in industry. To put it mildly, it failed to replace the standard JIS array (which I just covered). You will see that its design is similar in some ways to Fujitsu's Thumb-shift array, which is described in the next section.

Thumb-shift array

In an attempt to improve the input of Japanese text on computers, Fujitsu developed a keyboard known as the Thumb-shift array (親指シフト配列 *oyayubi shifuto hairetsu*). It is very similar in design and concept to the New-JIS array, but it has a slightly different keyboard arrangement and places two special modifier keys in the vicinity of the user's thumbs (these act to shift the keyboard to access more characters).

Like the New-JIS array, the Thumb-shift array assigns two kana characters per key for the first three banks of keys (the fourth bank of keys is reserved for numerals and symbols), but diverges in how the *dakuten* (゛) and *handakuten* (゜) are applied to kana characters. This is where the thumb-shift keys play a vital role.

The two thumb-shift keys each serve different functions. The left thumb-shift key converts the default character into the version that includes the *dakuten*. The right thumb-shift key simply shifts the keyboard so that the second character on the key is input. Table 5-23 illustrates some keys and shows how to derive all possible characters from them (secondary characters for each are in parentheses).

Table 5-23. The effect of the thumb-shift keys

Key	No thumb-shift	Left thumb-shift	Right thumb-shift
は(み)	は	ば	み
と(お)	と	ど	お
せ(も)	せ	ぜ	も
け(ゅ)	け	げ	ゅ

The trickery used by this keyboard array is that all the characters that can be modified by the *dakuten* are placed in the no thumb-shift location of each key (that is, the default character). There is a special key used for modifying a character with a *handakuten*. Figure 5-12 illustrates the entire Thumb-shift keyboard array.

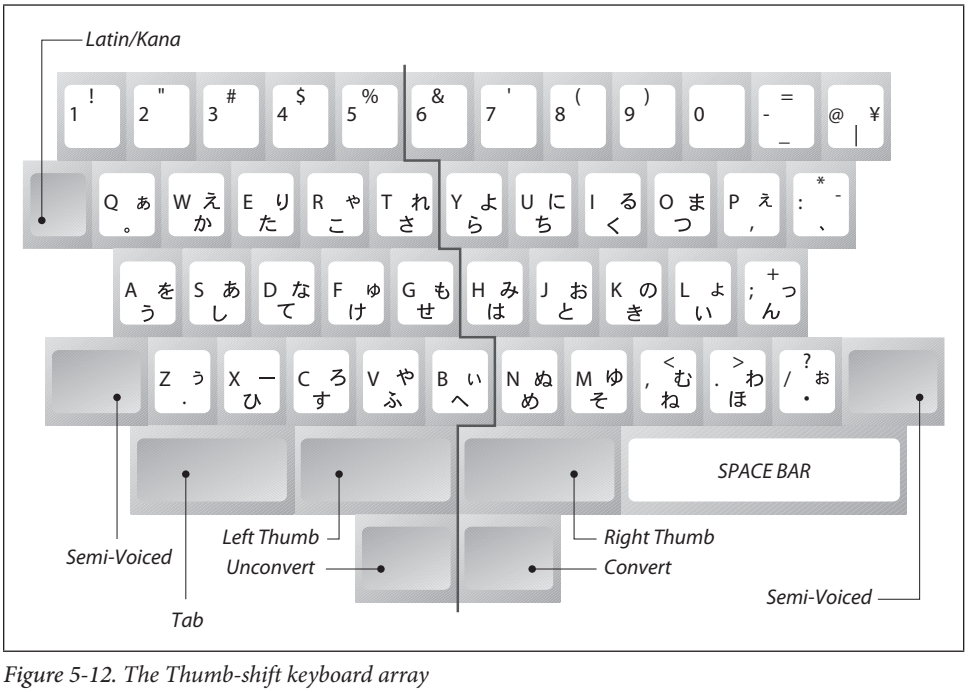


Figure 5-12. The Thumb-shift keyboard array

The Thumb-shift keyboard array is probably one of the most widely used in Japan (behind the QWERTY and JIS arrays, that is). In fact, other manufacturers have licensed it for use with their own computer systems.

50 Sounds array

As you may recall from discussions in Chapter 2, the term *50 Sounds* (50音 *gojūon*) refers to the 5×10 matrix that holds the basic kana character set. The 50 Sounds array (50音配

五十音順列 *gojūon hairetsu*) is based on this same matrix. On one side of the matrix are five vowels: *a*, *i*, *u*, *e*, and *o*. On the other side are nine consonants: *k*, *s*, *t*, *n*, *h*, *m*, *y*, *r*, and *w*. On the same axis as the consonants is also a place that represents no consonant, where the vowels can stand alone. This arrangement of kana characters was used as the basis for a Japanese keyboard array. This array is illustrated in Figure 5-13.

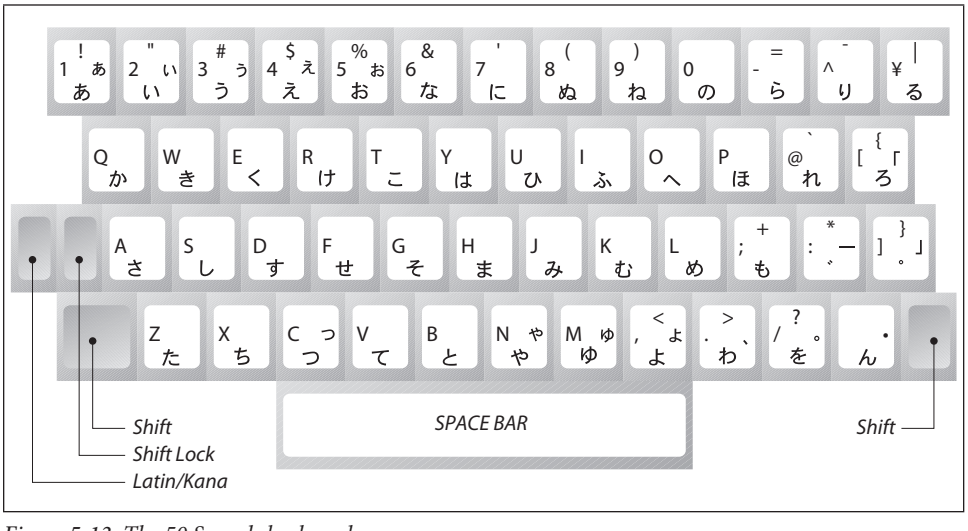


Figure 5-13. The 50 Sounds keyboard array

Figure 5-14 illustrates a different arrangement of the 50 Sounds keyboard array. Instead of being arranged horizontally in what is effectively a two-column layout, this alternate arrangement of the 50 Sounds keyboard array is arranged vertically. This keyboard array, however, requires a fifth row of keys in order to accommodate this vertical arrangement, because most of the logical kana rows require five keys.

This arrangement of keys is not very efficient: it is almost like having a keyboard array in which the 26 letters of the alphabet are arranged in order—most of the time is spent searching for keys. In fact, there are Western keyboard arrays on which the keys are arranged in alphabetical order! This problem is multiplied for Japanese, which requires nearly 50 separate keys! This keyboard array also suffers from the same problems of the JIS array, specifically that all four banks of keys are required for kana, and that the *dakuten* (゛) and *handakuten* (゜) require separate keys. This keyboard array never gained widespread acceptance, and is thus not very much used in Japan.

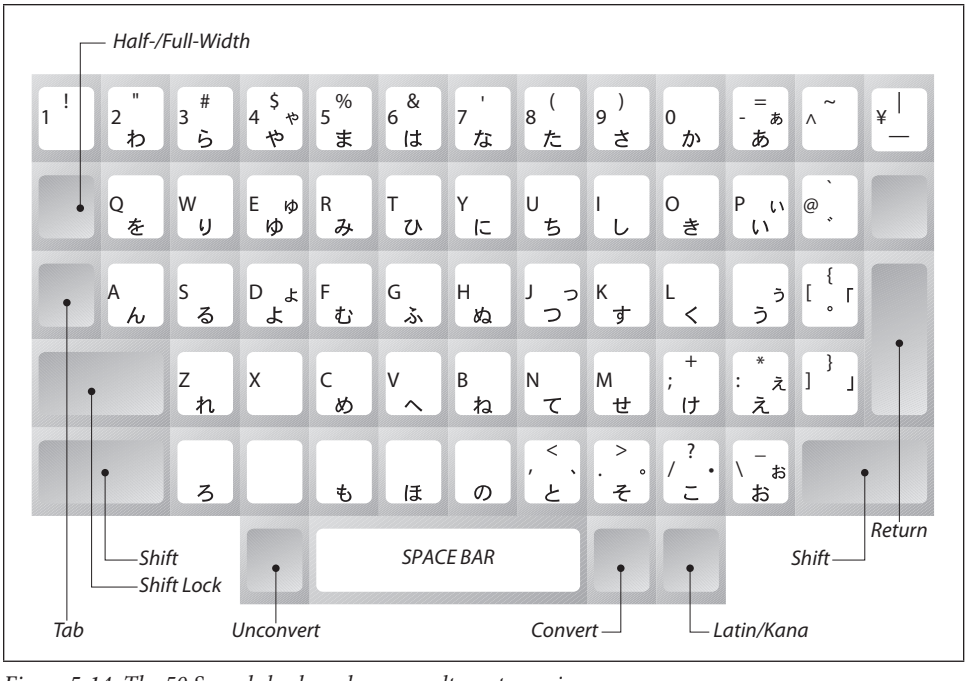


Figure 5-14. The 50 Sounds keyboard array—alternate version

TRON arrays

Developed by Ken Sakamura (坂村健 *sakamura ken*), the TRON keyboard array is similar in concept to other Japanese keyboard arrays in that several ergonomic and other optimizations were made in its design. TRON stands for *The Real-time Operating system Nucleus*. More information about the TRON Project is available in Appendix E.

There are two common instances of the TRON keyboard array: TK1 and μTRON (Micro TRON). Both instances of the TRON keyboard include the Dvorak array for accessing Latin characters (although the figures in this chapter do not show them). The TK1 design is laid out in an ergonomic fashion, as illustrated in Figure 5-15.

The μTRON design is more conventional in arrangement so that it can be used for notebook computers. It is illustrated in Figure 5-16.

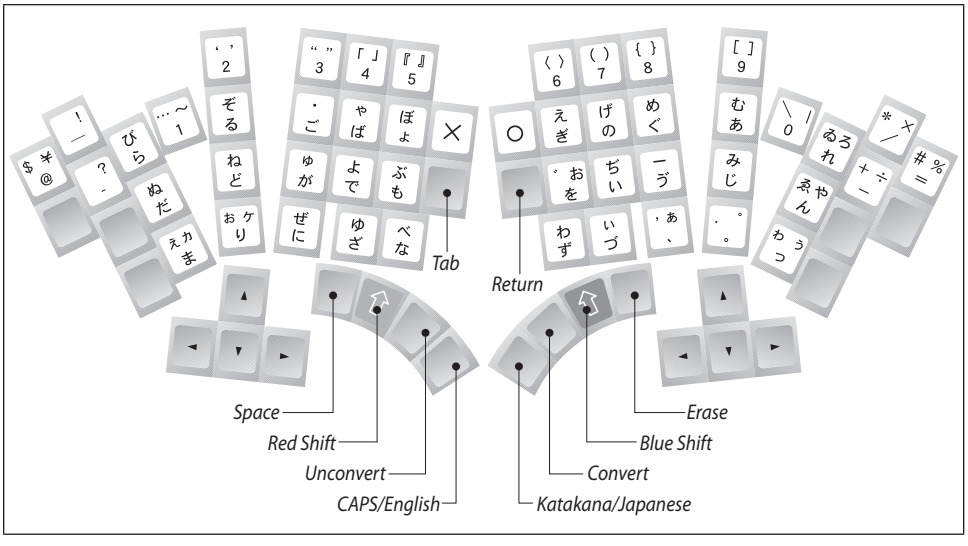


Figure 5-15. The TRON TK1 keyboard array

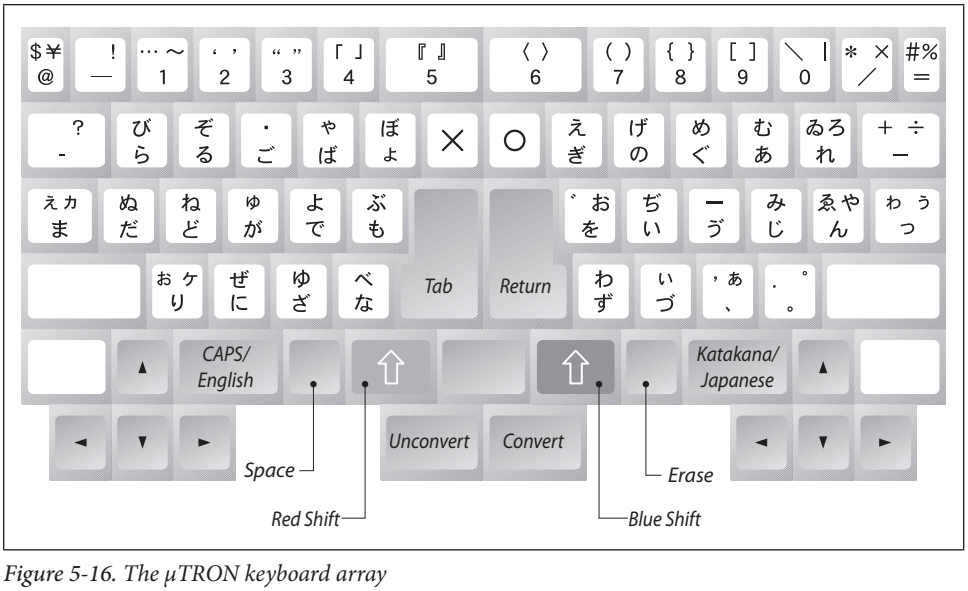


Figure 5-16. The μTRON keyboard array

Like the thumb-shift keys of Fujitsu's Thumb-shift array, the TRON keyboard arrays include two Shift keys that allow the typist to gain access to additional characters. The left Shift key is colored red, and the right Shift key is colored blue. Table 5-24 lists the standard characters of the TRON keyboard arrays, along with the characters that are made accessible by pressing each of the two Shift keys. Keys whose imprinted characters could be

confused due to their positioning, such as some punctuation and small kana, are enclosed in boxes with corner reference marks to ease identification.

Table 5-24. Characters made accessible using red and blue Shift keys

Unshifted key	Red Shift key	Blue Shift key
@	$	¥
゛		!
゜	…	〜
2	‘	’
3	“	”
4	「	」
5	『	』
6	〈	〉
7	(	)
8	{	}
9	[	]
0	＼	|
／	*	×
=	#	%
^		?
ら	ひ	び
る	そ	ぞ
こ	・	ご
は	ゃ	ば
ょ	ほ	ぼ
き	ぎ	え
の	げ	け
く	ぐ	め
あ		む
れ	ゐ	ろ
ー	＋	÷
ま	ぇ	カ
た	ぬ	だ
と	ね	ど

Table 5-24. Characters made accessible using red and blue Shift keys

Unshifted key	Red Shift key	Blue Shift key
か	ゆ	が
て	よ	で
も	ふ	ぶ
を	ゃ	お
い	ぢ	ち
う	づ	ー
し	じ	み
ん	ゑ	や
っ	わ	ゅ
り	ぉ	ケ
に	せ	ぜ
さ	ゆ	ざ
な	へ	べ
す	ず	わ
つ	づ	ぃ
ゃ	、	ぁ
。	・	。

Hangul Keyboard Arrays

Hangul syllables (and hanja) are input through the basic hangul elements, jamo. Up to four jamo comprise a complete *hangul syllable* (also referred to as *precombined* or *precomposed hangul*). A complete hangul syllable can also correspond to a single hanja, in which case a hangul-to-hanja conversion dictionary is necessary. Of course, better hangul-to-hanja conversion dictionaries also provide word-based conversion, which is much more efficient than character-based. There are several hangul keyboard arrays available, most of which are described and illustrated in the next sections. The most broadly used Korean keyboard is the KS array.

KS array

The most common hangul keyboard array, known as the KS array, is documented in the standard designated KS X 5002:1992, *Keyboard Layout for Information Processing* (정보처리용 건반 배열 *jeongbo cheoriyong geonban baeyeol*), which was originally established

in the 1980s.* A total of 27 hangul elements are accessed using the three lowest banks of keys (leaving the fourth bank open for numeral access). An additional seven hangul elements are accessed by shifting the third bank of keys.

Figure 5-17 illustrates the KS keyboard array. Note the additional key, beside the right-side Shift key, for switching between Korean and Latin mode (labeled 한/영, transliterated *han/yeong*, and meaning "Korean/English").

Figure 5-17. The KS keyboard array

Although this is the most commonly used hangul keyboard array, its biggest drawback is that there is no method for distinguishing initial and final consonants. This is also why the KS array is known as the *Two-Set* keyboard array, which is expressed as 두벌식 (*dubeolsik*) in Korean. The two sets are initial and final consonants and medial vowels. The Kong array, described next, remedies this situation, but at the expense of additional complexity.

Kong array

The *Kong* array, originally developed in the 1940s for use by Korean mechanical typewriters, is sometimes referred to as the Dvorak of hangul keyboard arrays and is considered a three-set keyboard array. This means that all three hangul element positions—initial, medial, and final—are easily distinguished by the keyboard itself. The KS array required distinguishing to be done by software at the OS level.

The unshifted state provides access to 39 hangul elements using all four banks of keys, and the shifted state provides access to 19 additional hangul elements, again using all four banks of keys.

* Previously designated KS C 5715-1992

Figure 5-18 illustrates the Kong keyboard array. Note the position of the numerals, specifically in the shift state of the second and third banks. Also note that some hangul elements are repeated. This is because they are to distinguish between initial and final instances.

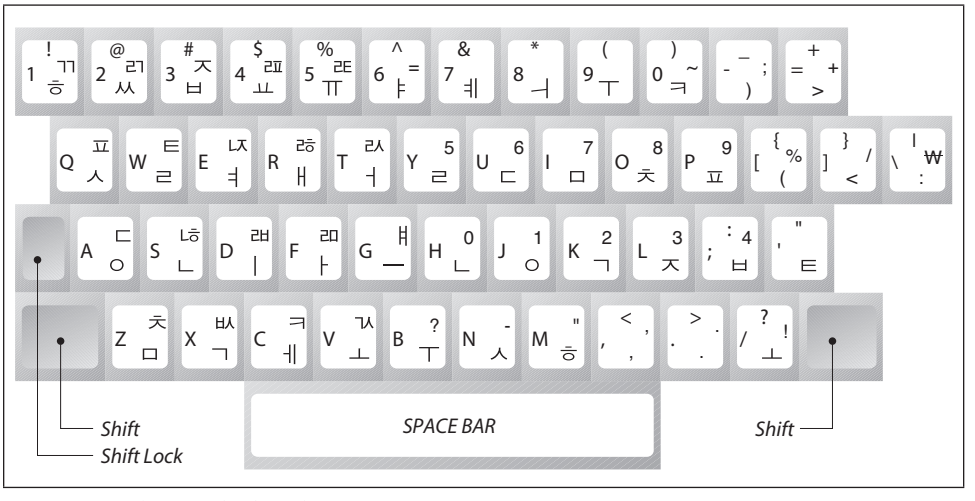


Figure 5-18. The Kong keyboard array

The Kong array is named after its inventor and is sometimes called the Three-Set Final keyboard array, which is expressed as 세벌식 최종 (*sebeolsik choejong*) in Korean.

Latin Keyboard Arrays for CJKV Input

Keyboard arrays appearing in this section make use of Latin characters rather than kana—there are a smaller number of Latin characters than kana, and these keyboard designs take advantage of that fact. These are unlike the QWERTY and Dvorak keyboard arrays in that they are optimized for Japanese input.

M-style array

Developed by NEC in the early 1980s, the M-style array (M式配列 *emu shiki hairetsu*) defines not only a keyboard, but also a new Japanese input method. The "M" in the name of this keyboard array comes from the last name of its designer, Masasuke Morita (森田正典 *morita masasuke*), a senior engineer at NEC. He has even written two books about this keyboard array and input method. I had a chance to try out the M-style keyboard array connected to two different machines while briefly visiting Japan in 1988. I was impressed with the feel of the keyboard and the efficiency of input.

This keyboard array makes use of only 19 keys for inputting Japanese text. Five are vowels, specifically *a*, *i*, *u*, *e*, and *o*. The remaining 14 are consonants, specifically *k*, *s*, *t*, *n*, *h*, *g*, *z*, *d*, *b*, *m*, *y*, *r*, *w*, and *p*. There are, of course, additional keys for the remaining seven characters necessary to input English text, specifically *q*, *l*, *j*, *f*, *c*, *x*, and *v*. Memorizing the locations

for 19 keys is easier than trying to for 26 for English, and it is considerably easier than the nearly 50 required for kana keyboard arrays. See Figure 5-19 for an illustration of the M-style keyboard array.

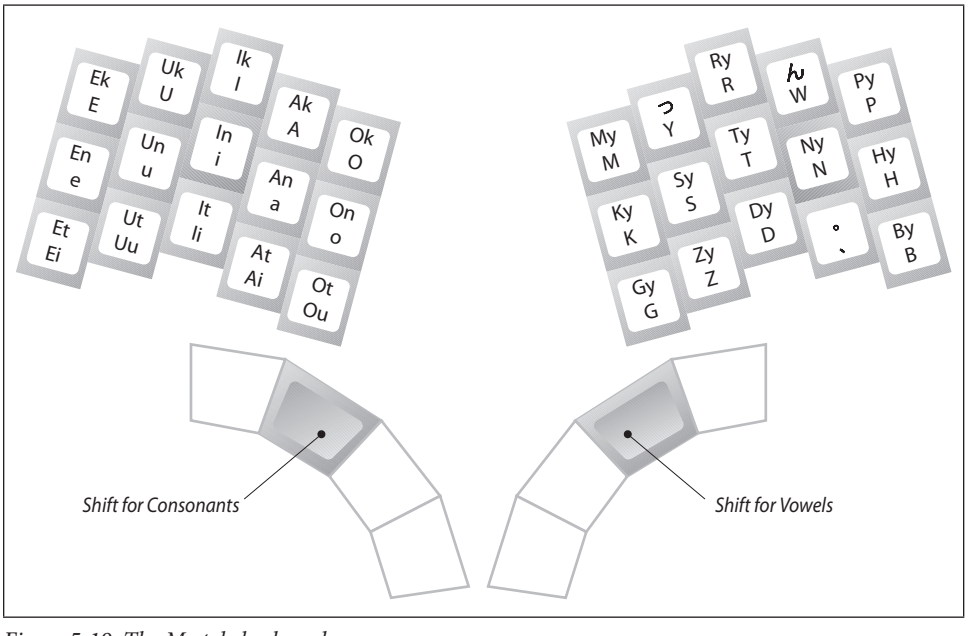


Figure 5-19. The M-style keyboard array

At first glance you should notice that the keyboard has a unique design; this is to make it more comfortable to use (less strain on the hands). Next, you should have noticed that the vowels are on the left set of keys, and the consonants are on the right set of keys. Japanese is a syllable-based language, so this vowel and consonant key arrangement provides a right-to-left typing rhythm.

The most important feature of this keyboard array and input system is that the user decides which parts of the input string are to be converted to kanji, which to hiragana, and which to katakana. There are three banks of vowel keys, and each bank can also shift to provide even more variations. These different banks specify the target character type for the input segments. With conventional input method software, the user simply inputs a string of characters then lets the software decide which parts convert to kanji and which parts remain as hiragana—this can lead to misconversions.

There are three banks of vowel keys. Each bank contains five keys, one for each of the five Japanese vowels. In addition, a vowel-shift key is located near the user's right thumb. This, in effect, gives you six banks of vowel keys with each bank having a unique purpose. The consonant keys permit only two states, shifted by depressing the consonant-shift key located near the user's left thumb.

The shifted state for the consonant keys, in general, adds a "y" or palatalized element. This is used to make combinations such as ぎょ (*gyo*) or しゅ (*shu*). Table 5-25 illustrates how the consonant keys shift, and their purpose. Note that some of the shifted states are not used for kanji input and are instead used for hiragana, such as っ and ん.

Table 5-25. The M-style keyboard array's consonant keys

Unshifted	Use	Shifted	Use
G	kanji or hiragana	Gy	kanji or hiragana
Z	kanji or hiragana	Zy	kanji or hiragana
D	kanji or hiragana	Dy	kanji or hiragana
、	comma (punctuation)	。	period (punctuation)
B	kanji or hiragana	By	kanji or hiragana
K	kanji or hiragana	Ky	kanji or hiragana
S	kanji or hiragana	Sy	kanji or hiragana
T	kanji or hiragana	Ty	kanji or hiragana
N	kanji or hiragana	Ny	kanji or hiragana
H	kanji or hiragana	Hy	kanji or hiragana
M	kanji or hiragana	My	kanji or hiragana
Y	kanji or hiragana	っ	hiragana
R	kanji or hiragana	Ry	kanji or hiragana
W	kanji or hiragana	ん	hiragana
P	kanji or hiragana	Py	kanji or hiragana

The shifted state for the vowel keys is a bit more complex and requires more detailed tables. Some banks of vowel keys are specifically designed to generate hiragana, not kanji. See Tables 5-26 and 5-27 to find out how the vowel keys work, in both unshifted and shifted states.

Table 5-26. The M-style keyboard array's vowel keys—unshifted

First bank	Use	Second bank	Use	Third bank	Use
Ei	kanji (long vowel)	e	hiragana	E	kanji (single vowel)
Uu	kanji (long vowel)	u	hiragana	U	kanji (single vowel)
Ii	kanji (long vowel)	i	hiragana	I	kanji (single vowel)
Ai	kanji (long vowel)	a	hiragana	A	kanji (single vowel)
Ou	kanji (long vowel)	o	hiragana	O	kanji (single vowel)

Table 5-27. The M-style keyboard array's vowel keys—shifted

First bank	Use	Second bank	Use	Third bank	Use
Et	kanji (+ち/つ/っ)	En	kanji (+ん)	Ek	kanji (+き/く/っ)
Ut	kanji (+ち/つ/っ)	Un	kanji (+ん)	Uk	kanji (+き/く/っ)
It	kanji (+ち/つ/っ)	In	kanji (+ん)	Ik	kanji (+き/く/っ)
At	kanji (+ち/つ/っ)	An	kanji (+ん)	Ak	kanji (+き/く/っ)
Ot	kanji (+ち/つ/っ)	On	kanji (+ん)	Ok	kanji (+き/く/っ)

Okay, but what happened to katakana? The M-style keyboard must be shifted into a special state for katakana input to be in effect. Katakana input is quite similar to kanji input in that you can use different vowel keys as shortcuts. Some of the vowel keys change in ways that speed katakana input.

Table 5-28 shows how the M-style keyboard array can be used to specify which characters convert to kanji, and how many kanji to select during conversion. The examples all share the same reading: *daku*.

Table 5-28. Comparisons among M-style, Latin, and kana input

Word	M-style input	Conventional Latin input[a]	Conventional kana input
駄句	DAKU	DAKU	だ゛く
諾	DAk	DAKU	だ゛く
抱く	DAKu	DAKU	だ゛く
だく	DaKu	DAKU	だ゛く

a. Case makes no difference.

A lot—but not all—of the ambiguity that you are likely to encounter when using conventional Japanese keyboard arrays and their associated input methods is remedied by the M-style keyboard array.

The M-style keyboard is standard equipment on many of NEC's dedicated Japanese word-processing systems.

High-speed Roman array

NEC also developed a keyboard array, partly based on the M-style array, called the High-speed Roman array (快速ローマ字配列 *kaisoku rōmaji hairetsu*). It is set into a conventional keyboard shape, though. The keys are basically set into the same arrangement as with the M-style array, but without the extra vowel keys and the special vowel and consonant modifier keys for distinguishing kanji from kana during input. Figure 5-20 provides an illustration of the High-speed Roman array.

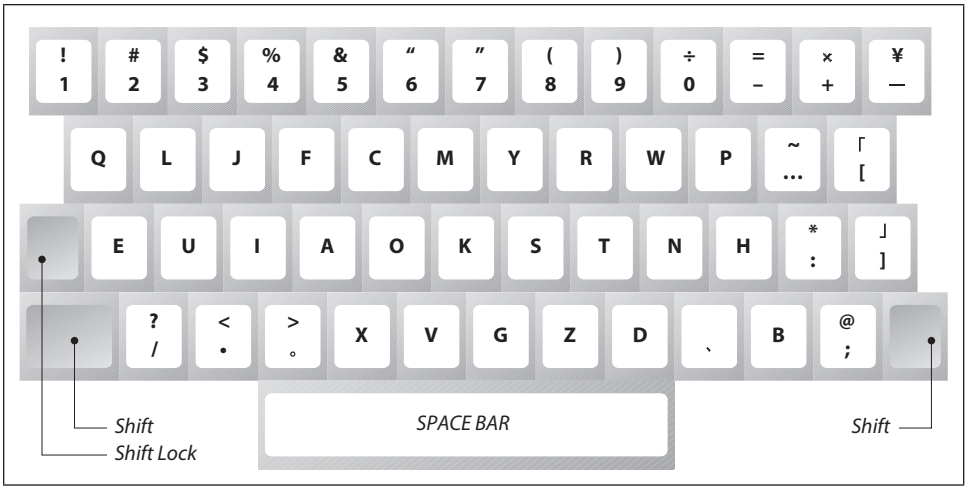


Figure 5-20. The High-speed Roman keyboard array

In the past, I have experimented with rearranging the keyboard resources within the Mac OS "System" file by using ResEdit, the standard resource editor for Mac OS, such that this keyboard array can be used for Japanese input. This effectively gave me an opportunity to experiment with this keyboard array without purchasing special hardware. Modifying keyboard arrays in software is something that almost every platform allows, and this keyboard array can be easily implemented through such means.

For those who use Mac OS X, I suggest using SIL International's Ukelele, which is a Unicode-based keyboard layout editor for Mac OS X.* For those who use Windows OS, Microsoft provides the Microsoft Keyboard Layout Creator (MSKLC), which allows users to define their own keyboard layouts.†

Mobile Keyboard Arrays

In the 10 years or so since the first edition of this book was published, mobile device use has increased exponentially, especially in China, Japan, Korea, and Taiwan. The number of users can be measured in the millions. Likewise, the number of text messages that are sent increases on a yearly basis.

As common sense now dictates, mobile devices are not merely used as telephones, but they also serve as mini-computers, allowing users to send and receive text messages and to run some limited applications. In Japan, mobile devices are even used to access blogs and to pay for commodities and services through the use of built-in 3D barcode scanners.

* *http://scripts.sil.org/ukelele*

† *http://www.microsoft.com/globaldev/tools/msklc.mspx*

Clearly, for some users, a mobile device provides sufficient features and services such that owning a standalone computer, even a laptop, is not necessary.

As mobile devices become more sophisticated, it is obvious that the applications that they run will become less and less limited. A good look into the future is manifested in Apple's iPhone, which has effectively taken mobile devices to an all new level, and I suspect that Apple is not yet done making strides and innovations in this area.*

Because of the importance of mobile devices in today's society, and because of the significantly reduced size of such devices, the keyboard arrays are necessarily small. Given the average size of the human hand, and the digits that are connected to them that ultimately manipulate keyboard keys, the keys can become only so small. There is also a minimal spacing threshold to consider. These are issues best dealt with by ergonomics experts.

Short of being a self-proclaimed ergonomics expert, it is sufficient to state that there are effectively four ways provide a full keyboard array for mobile devices, described as follows:

- One entire side of the mobile device is a touch panel LCD screen that can be used to emulate a full keyboard array.
- The mobile device opens to reveal a full keyboard array—referred to as *clamshell* designs because they open like a clamshell.
- The mobile device is sufficiently large to barely accommodate a full keyboard array.
- The mobile device uses a laser to project a full keyboard array onto suitable surfaces.

In the future, perhaps we can look forward to using holographic keyboard arrays that project the image of the keys into the air in front of us.

Interestingly, when full keyboard arrays *are* implemented for mobile devices, the most broadly supported keyboard array is QWERTY. This demonstrates once again that QWERTY has become, or has remained, for better or for worse, the universal or *de facto* keyboard array.

In terms of genuine mobile keyboard arrays, which are best described as a full keyboard array crammed into the confines of what is effectively a numeric keypad, there are *de facto* standards. In Japan, one such mobile keyboard array is dominant, and in Korean two are dominant.

Because multiple characters are necessarily mapped to a single numeric key on mobile keyboard arrays, a method known as *multi-tap* is used to access the desired character. A single numeric key press results in the first character that is assigned to it, and a second numeric key press results in the second character that is assigned to it, and so on. One characteristic of the multi-tap method is that if the subsequent character is assigned to the same numeric key, one must wait until the cursor advances before entering the next

* *http://www.apple.com/iphone/*

character. Consider entering the three-character string "abc" using a typical mobile phone. These three Latin characters are mapped to the numeric key labeled "2." The numeric key sequence is "2" for "a," "2 2" for "b," and "2 2 2" for "c." But, if you repeatedly press the "2" numeric key six times, the result is a single character, not the three-character string "abc." There is a one-second or so lag until the cursor advances, thus disabling multi-tap for the current character.

Bear in mind that the multi-tap issue affects any sequence in which two or more desired characters are assigned to the same numeric key. There is a way to work around the multi-tap issue, which is to use a separate cursor key to manually or explicitly advance to the next character, thus completing the entry of the current character and resetting multi-tap mode for the next character.

Some claim that mobile keyboard arrays hamper one's ability to enter text. From a sheer "number of keystrokes per character" point of view, along with the multi-tap issue, that may be the case, and for most people it is far easier—and significantly faster—to use a conventional keyboard array. However, an increasingly large number of books in Japan, including novels and short stories, have been written by entering their text into a mobile phone.* These books obviously weren't typeset on a mobile phone, but the text entry, which represents the author's main effort in the authorship of a book, was possible at almost any location. After all, mobile phone are *mobile*, and were specifically designed to be brought and used nearly anywhere. This demonstrates that the more one practices something, the better the skills become.

Also of interest is that I am not aware of any national or international standard for mobile keyboard arrays. This area is still somewhat volatile, and establishing such standards may be premature, at least for some markets, such as Korea. Clearly, innovation is still taking place. Once a *de facto* standard emerges, such as what has already happened in the U.S. and Japan, establishing a national standard makes sense and is the prudent thing to do in order to establish consistency across the industry.

Japanese mobile keyboard arrays

In Japan, the dominant mobile keyboard array closely adheres to the 50 Sounds ordering. For example, each logical "row" of kana characters is assigned to a single numeric key, so the "A" row (あ行 *a gyō*), consisting of あ, い, う, え, and お, is assigned to the "1" numeric key. If a row of kana characters includes any small versions, such as ぁ, ぃ, ぅ, ぇ, and ぉ for the "A" row, they are also assigned to the same numeric key, and immediately follow all of the standard instances. Thus, a single "1" keystroke results in あ, two "1" keystrokes result in ぃ, six keystrokes result in ぁ, and so on.

Figure 5-21 provides an example of a Japanese keyboard array for a mobile device. In addition to showing how hiragana characters are assigned to the numeric keys, it also shows how Latin characters are assigned to the same keys.

* For the record, this book was not written by entering its text into a mobile phone.

Figure 5-21. Mobile device keyboard array—Japanese

Table 5-29 lists the 12 common mobile keyboard array keys, the same ones as illustrated in Figure 5-21, along with how they map to the 26 characters of the Latin alphabet, and how they map to hiragana for Japanese mobile devices.

Table 5-29. Mobile keyboard array key assignments—Japanese

Numeric key	Latin	Japanese key cap	Characters
1	.@	あ	あいうえおぁぃぅぇぉ
2	ABC	か	かきくけこ
3	DEF	さ	さしすせそ
4	GHI	た	たちつてとっ
5	JKL	な	なにぬねの
6	MNO	は	はひふへほ
7	PQRS	ま	まみむめも
8	TUV	や	やゆよゃゅょ
9	WXYZ	ら	らりるれろ
*		゛゜	゛゜
0		わ、。	わをんゎ。、ー？！／¥＆＊＃
#	space		

Dakuten- and handakuten-annotated kana characters are entered by first entering their unannotated forms, followed by one or two presses of the "*" key. For those kana that can be annotated with both, specifically the characters from the "H" row, such as は (*ha*), one press of the "*" key results in the dakuten-annotated form, ば (*ba*), and two presses

result in the handakuten-annotated form, ぱ (*pa*). A third press of the "*" key causes the character to cycle back to its unannotated form.

The assignments of the kana to the numeric keys is fairly standard, as is basic punctuation. There are separate keys for entering katakana by switching modes or to convert hiragana sequences into kanji.

Korean mobile keyboard arrays

Although there are well over 20 mobile keyboard arrays in use in Korea, at least as of this writing, those developed by Samsung and LG (the leading mobile device manufacturers in Korea) are the most popular, and are the ones that are covered in this section. Of these, the one developed by Samsung is the most popular, and thus exhibits the broadest use in Korea.

Samsung's Korean mobile keyboard array is called *Chonjiin Hangul*[*] (천지인 한글 *cheonjiin hangeul*), and LG's is called *EZ Hangul* (EZ 한글 *EZ hangeul*). Figure 5-22 illustrates both of these mobile keyboard arrays.

Figure 5-22. Mobile device keyboard arrays—Korean

Because hangul is alphabetic, not syllabic like Japanese kana, efficient text entry is performed by entering individual jamo, or in some cases, more primitive elements. Table 5-30 again lists the 12 common mobile keyboard array keys, along with how they map to the Latin alphabet. More importantly, it lists how they map to the jamo for Korean mobile devices, based on the mobile keyboard arrays developed by Samsung and LG, specifically Chonjiin Hangul and EZ Hangul, respectively. Note that the jamo and other elements that

* *Chonjiin Hangul* can also be written using hanja, specifically 天地人 한글 (*cheonjiin hangeul*). The three hanja mean *Heaven*, *Earth*, and *Man*, respectively. This name comes from the design principles for vowels as described in 훈민정음 (訓民正音). According to this book, the vowels of hangul syllables are composed of the following three primitive elements: a dot or 천/天 (*cheon*), meaning "Heaven"; a horizontal bar or 지/地 (*ji*), meaning "Earth"; and a vertical bar or 인/人 (*in*), meaning "man."

are provided in Table 5-30 represent what is shown or printed on the keys themselves. See Tables 5-31 and 5-32 to find out how multiple key presses, of the same or different numeric keys, result in the desired jamo.

Table 5-30. Mobile keyboard array key assignments—Korean

Numeric key	Latin[a]	Chonjiin Hangul—Samsung	EZ Hangul—LG
1	.@	ㅣ	ㄱ
2	ABC	ㆍ	ㄴ
3	DEF	ㅡ	ㅏ ㅓ
4	GHI	ㄱ ㅋ	ㄹ
5	JKL	ㄴ ㄹ	ㅁ
6	MNO	ㄷ ㅌ	ㅗ ㅜ
7	PQRS	ㅂ ㅍ	ㅅ
8	TUV	ㅅ ㅎ	ㅇ
9	WXYZ	ㅈ ㅊ	ㅣ
*			Add a stroke
0		ㅇ ㅁ	ㅡ
#	space		Double the jamo

a. Breaking from the norm, Samsung mobile devices assign up to three Latin characters per numeric key, which affects the "7" and "9" numeric keys. The Latin characters "Q" and "Z" are instead assigned to the "1" numeric key.

Table 5-31 lists complete jamo consonants, along with the Chonjiin Hangul and EZ Hangul key sequences that produce them. The former uses only numeric keys to enter all consonants, but the latter requires the "*" and "#" keys to enter all consonants.

Table 5-31. Keystrokes to produce jamo—consonants

Jamo consonant	Chonjiin Hangul key sequence	EZ Hangul key sequence
ㄱ	4	1
ㄴ	5	2
ㄷ	6	2*
ㄹ	55	4
ㅁ	00	5
ㅂ	7	5*
ㅅ	8	7
ㅇ	0	8
ㅈ	9	7*

Table 5-31. Keystrokes to produce jamo—consonants

Jamo consonant	Chonjiin Hangul key sequence	EZ Hangul key sequence
ㅊ	99	7**
ㅋ	44	1*
ㅌ	66	2**
ㅍ	77	5**
ㅎ	88	8*
ㄲ	444	1#
ㄸ	666	2*#
ㅃ	777	5*#
ㅆ	888	7#
ㅉ	999	7*#

The Chonjiin Hangul mobile keyboard array distinguishes itself by how vowels are entered through the use of only three keys. The first three keys, specifically 1, 2, and 3, are assigned to primitive elements that are used to compose complete vowels, and are dedicated for that purpose. The key sequence corresponds to the left-to-right and top-to-bottom arrangement of the elements, which is also the stroke order when writing them by hand. The EZ Hangul mobile keyboard array dedicates four keys for vowels, specifically 3, 6, 9, and 0, but also requires the "*" key for some combinations. Table 5-32 lists complete jamo vowels, along with the Chonjiin Hangul and EZ Hangul key sequences that produce them.

Table 5-32. Keystrokes to produce jamo—vowels

Jamo vowel	Chonjiin Hangul key sequence	EZ Hangul key sequence
ㅏ	12	3
ㅑ	122	3*
ㅓ	21	33
ㅕ	221	33*
ㅗ	23	6
ㅛ	223	6*
ㅜ	32	66
ㅠ	322	66*
ㅡ	3	0
ㅣ	1	9
ㅐ	121	39

Table 5-32. Keystrokes to produce jamo—vowels

Jamo vowel	Chonjiin Hangul key sequence	EZ Hangul key sequence
ㅒ	1221	3*9
ㅖ	211	339
ㅖ	2211	33*9
ㅘ	2312	63
ㅙ	23121	639
ㅚ	231	69
ㅝ	3221	6633
ㅞ	32211	66339
ㅟ	321	669
ㅢ	31	09

At some point, I hope that a single *de facto* Korean mobile keyboard array emerges, such as what has happened in Japan. Given the somewhat volatile nature of the mobile device market, along with the complexity of hangul, it may be years until a *de facto* Korean mobile keyboard array emerges. However, if I were to guess based solely on market share, ease and intuitiveness of input, and other factors, Samsung's Chonjiin Hangul seems like the best candidate for the *de facto* standard.

Other Input Hardware

What has been described so far falls into the category of keyboard arrays—that is, keys that are used to input kana, hangul, Latin, or other characters. Other more recent hardware methods, such as pen input, do not require use of conventional keys. Optical Character Recognition (OCR) is another input system that deals with the problem of transcribing already printed information. Finally, there are voice input systems.

This area is rapidly changing. Since the first edition of this book was published, OCR and voice recognition systems have advanced significantly. Interestingly, pen input hasn't quite taken off, and at this point, it may never do so.

Pen Input

GO Corporation, which has since gone out of business after being acquired by AT&T, developed a pen-based OS and programming environment called PenPoint.* PenPoint did not require the use of conventional keys, but instead used a tablet on which the user

* For an interesting account of GO Corporation's rise and fall, I suggest Jerry Kaplan's *Startup: A Silicon Valley Adventure* (Penguin Books, 1994).

physically wrote what was intended. Pen input depends on another technology, specifically OCR, to be covered in the next section. Perhaps of historical interest, GO Corporation had enhanced their pen-based OS to handle Japanese through the use of Unicode.

Microsoft's MS-IME, which was the standard input method provided with various versions of their Windows OS, provided the user with the ability to input characters by writing them on a special onscreen tablet.

Optical Character Recognition

Several OCR systems currently accept CJKV character input, although there are, of course, limitations. The clearer and larger the typefaces, the more reliable such a system is. Some systems do not recognize all the characters in a character set (in the case of GB 2312-80, for example, some recognize only Level 1 hanzi), and some are restricted to certain typeface styles.

You encounter OCR systems more frequently in the West where recognition of a much smaller collection of characters is done. The recognition of thousands of individual characters becomes much more difficult, particularly when each one is fairly complex in structure.

NeocorTech's KanjiScan OCR, available only for Windows, can convert printed material that contains Japanese and English text into a form that can be manipulated as normal text.* For those characters that it cannot recognize, there is a Kanji Search System for looking up unrecognized kanji, which automatically suggests alternatives from which to choose.

WinReader PRO and e.Typist, both developed by Media Drive Corporation, are examples of OCR software that handles Japanese and English text. They are available only for Windows.†

Voice Input

Years ago, voice input systems required users to register their voice patterns so that the voice recognition software could more predictably match the user's voice input with correct character strings. Voice recognition systems are more sophisticated now, and many of them include automatic-learning modes. Voice recognition is widely used on mobile devices. Examples of voice input systems include IBM's Embedded ViaVoice (once called VoiceType),‡ with broad multilingual support, and Apple's Chinese Dictation Kit.§

Voice-driven software is now becoming more widespread, so expect more sophisticated systems to enter the market. If you think your office environment is distracting now, you

* *http://www.coyer.com/neocor/kanjiscan/*

† *http://mediadrive.jp/*

‡ *http://www-306.ibm.com/software/pervasive/embedded_viavoice/*

§ *http://speech.apple.com/speech/cdk/cdk.html*

can look forward to the joys of entire buildings full of people in cubicles yelling at their computers! Aren't sound-proof walls and doors nice?

Input Method Software

Input methods are the software programs that perform the actual CJKV input. A less common name for these is FEP, which stands for *front-end processor*. They make use of one or more conversion dictionaries, and often use special rules to more effectively convert input strings into ideographs, or into a mixture of kana and kanji (Japanese) or hangul and hanja (Korean). The actual mechanics of CJKV input were described in detail earlier in this chapter.

Here you will learn a little bit about a select few input methods. Under most environments, these programs are separate modules from the text-editing or word-processing applications with which they are used. This allows them to be used with a variety of applications. Some of these programs are dedicated for use within a specific application. For example, Quail is the dedicated input method for Japanese-capable versions of Emacs, such as GNU Emacs version 20 and greater.

Virtually all input methods have a facility that allows the user to add entries to the conversion dictionary (either to the main conversion dictionary or to a separate user conversion dictionary). This allows users to create entries for difficult-to-enter ideographs or ideograph compounds. This also allows adventurous users to create specialized or field-specific conversion dictionaries. Adding entries to a conversion dictionary is sometimes a simple task of providing a key in the form of a reading string (using Latin characters, kana, or hangul) followed by one or more names that will act as candidates when the key is encountered during the conversion process. More complex input methods require additional grammatical information, such as part of speech (noun, verb, and so on) or other information to assist in the conversion process.

CJKV Input Method Software

Microsoft has been offering their CJKV-capable *Global IME* input method since the Windows 95, 98, and NT time frame, and it is even more function under the Windows XP and Vista OSes.* Likewise, Apple provides a wide range of input methods for Mac OS X. Linux distributions provide the richest collection of CJKV-capable input methods. *Smart Common Input Method* (SCIM) is the input method framework that is widely used by Linux and Unix OSes.†

This is a significant development, because it means that it is now easier than ever before to send CJKV email, browse CJKV web pages, or fill out web-based CJKV forms. Thanks to

* *http://www.microsoft.com/windows/ie/ie6/downloads/recommended/ime/default.mspx*

† *http://www.scim-im.org/*

the efforts of Microsoft, Apple, and the open source community, much of what was once considered to be nontrivial is now trivialized.

Chinese Input Method Software

The number of input methods available for Chinese far outnumber those available for Japanese and Korean—combined! One source claims that well over 500 different Chinese input methods have been developed over the years.* For this reason, it is well beyond the scope of this book to even begin to cover them. Instead, I urge you to explore the Chinese input methods that are included with the current versions of Windows or Mac OS X, both of which have matured in terms of their multilingual support. Modern Linux distributions typically provide a larger number of Chinese input methods, such as those based on SCIM. The default installations of modern OSes are fully CJKV-capable, and thus include fully functional Chinese input methods.

NJStar

NJStar? You must be wondering why a Chinese word processor is listed as a Chinese input method. This is because it provides the user with nearly 20 Chinese input methods, each of which can be easily invoked through its UI.†

More information about NJStar's use as a word processor can be found in Chapter 10.

Japanese Input Method Software

Some popular Japanese input methods, such as ATOK, VJE, and Wnn, have been ported to multiple platforms, which often means you can exchange their conversion dictionaries among platforms.

The following are some of the commercial Japanese input methods that have been available over the years—some are bundled with OSes, some are added on as third-party software, and some are no longer developed or offered for sale:

- ATOK (JustSystems)
- egbridge (ERGOSOFT)
- Global IME (Microsoft)
- Katana (SomethingGood)
- Kotoeri (Apple)
- OAK (Fujitsu)
- SKK

* Yucheng Liu's MS thesis entitled *Chinese Information Processing* (University of Nevada, Las Vegas, 1995).

† *http://www.njstar.com/*

- VJE (VACS)
- Wnn (Wnn Consortium)

Some of the previously mentioned input methods, such as Kotoeri, Global IME, and Wnn, are bundled as part of OSes. After all, a Japanese-capable OS is meaningless without an input method. The other input methods were developed because some users found the input methods bundled with their OS inadequate. Finally, some of these input methods have been ported to multiple platforms, such as ATOK, VJE, and Wnn.

Noncommercial input methods include Canna, Quail, SKK, T-Code, Wnn (earlier than version 6), and so on. Quail and SKK are more or less closely related to GNU Emacs. Quail is actually part of GNU Emacs version 20 and greater, and SKK is an add-on input method for GNU Emacs. The latest version of Quail uses SKK's conversion dictionary.

ATOK

ATOK (エイトック *ētokku*), developed by JustSystems (the developer of the popular Ichitaro Japanese word-processing application), is one of the most powerful and popular Japanese input methods available, and for very good reason.* ATOK stands for *Advanced Technology of Kana-Kanji* transfer. It is available for both Mac OS X and Windows, as well as for Linux, and has been adapted for mobile phone use. There are also specialized versions of ATOK tailored toward specific markets, such as for newspaper use and for supporting the U-PRESS character set.

It is safe to state that ATOK is the most popular third-party input method in Japan, and its market share rivals the OS-bundled input methods. This is no small feat and is a testament to the features and capabilities of ATOK as an input method.

Canna

Canna (かんな *kanna*)† is the name of a Japanese input method originally developed by Akira Kon (今昭 *kon akira*) and several others at NEC, which offers features and a set of conversion dictionaries similar to Wnn, described later in this section. It is easily customized by the user, and comes with additional utilities for performing tasks such as conversion dictionary maintenance. Much of the customizing is done with LISP-like commands. Canna was one of the first freely available Japanese input methods for Unix that used automatic conversion and provided a unified UI. Canna had become available for Windows 95J and Java. Although it hasn't been updated for years, it is still available.

* *http://www.atok.com/*

† *http://www.nec.co.jp/canna/*

egbridge

ERGOSOFT is the developer of a popular Japanese input method called egbridge.* This input method had been available for many years and was a close competitor to ATOK, which is developed by JustSystems. It runs on Mac OS X, but is no longer developed or sold.

The main conversion dictionary used by egbridge boasted over 280,000 entries. Its personal and place name conversion dictionary had nearly 300,000 entries.

Global IME

The input method that was bundled with older versions of Windows, specifically 95J, 98J, or NT-J, was called *MS-IME*, which was an abbreviation for Microsoft Input Method Editor. MS-IME provided the user with a full range of input facilities, along with the ability to input characters by writing them on a special onscreen tablet. It was also possible to search for difficult-to-input kanji through the use of built-in radical and stroke count indexes.

Now, contemporary versions of Windows OS, such as XP and Vista, are instead bundled with Global IME. This is part of Microsoft's globalization strategy, which aims to include multilingual facilities in the standard or default installation of the OS.† This is a good thing for users. It eliminates the need to download Language Packs. It solves the font and input method problem.

Because of its status as the bundled input method for Windows, don't expect to see a Mac OS X version of Global IME anytime soon! Given the capabilities of the input method that is bundled with Mac OS X, specifically Kotoeri, there is no need to develop a Mac OS X version of Global IME.

Kotoeri

Kotoeri (ことえり *kotoeri*) is the name of the Japanese input method bundled with Mac OS-J version 7.1 and later, which included Mac OS with the Japanese Language Kit installed. Kotoeri has obviously evolved and is still part of Mac OS X, though its functionality has been merged into Apple's strategy of bundling as many input methods as possible in the standard or default installation of the OS.

Some might find it interesting that Kotoeri's name during its initial development was *Akiko*, which was an acronym for "Apple's Kana In Kanji Out." *Kotoeri* literally means "word selector." こと (*koto*) is an abbreviated form of the Japanese word that means "word," and えり (*eri*) means "select." It is much improved over the input method bundled with earlier versions of Mac OS-J, specifically 2.1 変換. The latest version of Kotoeri adds improvements and functionality that rival third-party input methods, such as the ability

* *http://www.ergo.co.jp/*

† *http://www.microsoft.com/windows/ie/ie6/downloads/recommended/ime/default.mspx*

to input kanji by using a special Unicode palette. It does, however, remove the ability to input characters by ISO-2022-JP and Shift-JIS codes, but ISO-2022-JP– and Shift-JIS–based palettes are still included. Input by code is now effectively restricted to four-digit Row-Cell codes.

User-defined conversion dictionary entries are entered into a special user conversion dictionary. The main conversion dictionary is considered fixed and thus read-only. This is considered to be fairly standard behavior among input methods.

SKK

SKK, which is an abbreviation for *Simple Kana to Kanji conversion program*, is the name of a freely available Japanese input method intended for use with Japanese-capable versions of text editors based upon GNU Emacs, such as Demacs, GNU Emacs itself, Mule, and NEmacs (most of these are described in Chapter 10).* This means that Japanese input under SKK is restricted to using Japanese Emacs. Many people use Emacs as their working environment. Various tasks—sending and receiving email, reading Usenet News, writing and compiling programs, and so on—can be done from within the context of Emacs. Yes, it is more than merely a text editor. It is really an environment, and to some extent, it can be considered a platform.

There are four main conversion dictionaries available for SKK: S, M, ML, and L. Obviously, S, M, and L refer to Small, Medium, and Large. ML refers to a size between Medium and Large. There is no need to install all three of these conversion dictionaries, because they are inclusive of each other. In other words, the L conversion dictionary contains all the entries of the M one, and so on. There are also nearly 20 specialized conversion dictionaries available for SKK.

In the past, SKK and its associated files have been adapted for use on non-Unix systems. For example, MOKE, a commercial Japanese text editor for MS-DOS systems, made use of the SKK conversion dictionary format for its own Japanese text entry system.

The development of SKK is still being managed by Masahiko Sato† (佐藤雅彦 *satō masahiko*), who was also responsible for first developing it in 1987. Users can register new words, which become part of the SKK conversion dictionaries, through a web form.‡ There is also an interactive tutorial, invoked from within GNU Emacs, that is useful for learning the SKK Japanese input method.

T-Code

T-Code is a freely available Japanese input method developed at Yamada Laboratory at Tokyo University.§ T-Code has been adapted to run under a variety of platforms, such

* *http://openlab.jp/skk/*

† *http://www.sato.kuis.kyoto-u.ac.jp/~masahiko/masahiko-e.html*

‡ *http://openlab.jp/skk/registdic.cgi*

§ *http://openlab.ring.gr.jp/tcode/*

as Mac OS, Unix, and Windows, and uses a two-stroke input method. Each kanji is thus input through the use of two arbitrary keystrokes. This is effectively the opposite of input by association, which was an input technique described earlier in this chapter.

VJE

VJE, which is an abbreviation for *VACS Japanese Entry system*, is a commercial Japanese input method that had been adapted for a variety of OSes. It is no longer developed or sold. VJE has been available for Japanese PCs, and MacVJE was available for Mac OS. Mac OS X required a newer version called MacVJE-Delta. VJE itself was developed by VACS,* and MacVJE was developed by Dynaware.†

All versions of VJE came with utilities for exchanging conversion dictionaries among operating systems. This was useful if you have more than one version of VJE, or want to send someone your conversion dictionary.

MacVJE came with two conversion dictionaries. The main conversion dictionary contained well over 100,000 entries. The other dictionary contained Japanese postal codes, along with candidate place names associated with those postal codes. Only one dictionary could be used at a time, but utilities were included for merging dictionaries. The main conversion dictionary could also be decompiled (that is, converted into a large text file for viewing individual entries) with the same utility—this is a feature that many other Japanese input programs do not have. User-specific entries were added directly to the main conversion dictionary.

Wnn

Wnn, which is an abbreviation of the transliterated form of the Japanese sentence 私の名前は中野です (*watashi-no namae-wa nakano desu*, which means "my name is Nakano"), is a freely available Japanese input method for Unix systems developed by the now-dissolved Wnn Consortium. One of the early goals of the Wnn project was to properly parse the Japanese sentence that Wnn represents. Wnn supports a multilingual environment—not only for Japanese, but also for Chinese, Korean, and many European scripts.

Wnn was actually the name of the conversion program that provided a consistent interface between *jserver* (a Japanese multiclient server) and actual Japanese input methods. Wnn also provided a set of conversion dictionaries. The Japanese input method that is provided with Wnn is called *uum*—this represents the word "wnn" rotated 180˚ so that it is upside down. Uum is the client program that is invoked by the user for Japanese input, and it defines the keystroke combinations necessary for Japanese input. Wnn, on the other hand, refers to the entire collection of software included in the distribution.

* *http://www.vacs.co.jp/*

† *http://www.dynaware.co.jp/*

The conversion dictionaries included with Wnn consist of a main dictionary (about 55,000 entries), a single kanji dictionary, a personal name dictionary, a place name dictionary, grammatical dictionaries, and several field-specific conversion dictionaries (for computer science and biological terms, for example). All these dictionaries are used in the conversion process.

The Ministry of Software—a company, not a government body in Japan—was the first company to adapt Wnn for use with Mac OS. Yinu System has also adapted Wnn for use with Mac OS. The best way to explore Wnn is through the FreeWnn Project.*

Korean Input Method Software

Because of the hangul-only nature of contemporary Korean text, the fundamental requirements for Korean input methods are simpler compared to those for Chinese and Japanese. The input methods bundled with Korean-capable OSes, such as Mac OS X and Windows, are popular with users. The Korean input methods that are bundled with these OSes support the input of hanja through the process of hangul-hanja conversion. In particular, Saenaru† (새나루 *saenaru*), developed by Wonkyu Park (박원규 *pak wonkyu*) and Hye-shik "Perky" Chang (장혜식 *jang hyesik*), and Nalgaeset‡ (날개셋 *nalgaeset*), developed by Yongmook Kim (김용묵 *gim yongmuk*), have become popular third-party Korean input methods for Windows. Hanulim (하늘입 *haneulim*) should be pointed out as a popular third-party Korean input method for Mac OS X.§

In addition to SCIM, Korean input methods for Linux or Unix, to include the X Window System, include imhangul¶ for GTK+ 2.0, qimhangul** for Qt, and nabi†† (나비 *nabi*). The primary developer for these Korean input methods is Hwanjin Choe (최환진 *choe hwanjin*), who also developed libhangul, which is an open source, cross-platform Korean input method library.‡‡ It needs to be pointed out that many of the Korean input methods described in this section make use of libhangul.

Although not specifically part of the input method, Korean input does have one unique feature not shared by Chinese and Japanese. This is the ability or option to delete hangul as entire characters (hangul) or by element (jamo) while still in "input mode." As soon as the character string is inserted or set into the application, the character must be deleted on a per-character, not per-jamo, basis. In any case, good input methods are the ones that make these "delete" options available to the user.

* *http://freewnn.sourceforge.jp/*
† *http://kldp.net/projects/saenaru/*
‡ *http://moogi.new21.org/prg4.html*
§ *http://code.google.com/p/hanulim/*
¶ *http://kldp.net/projects/imhangul/*
** *http://kldp.net/projects/qimhangul/*
†† *http://kldp.net/projects/nabi/*
‡‡ *http://kldp.net/projects/hangul/*

CHAPTER 6

Font Formats, Glyph Sets, and Font Tools

One of the most important considerations in displaying or printing CJKV text is the availability of appropriate fonts. Fonts form the most basic foundation of document writing and text input—no matter which writing system is involved—and are available in a wide variety of formats, though OpenType has clearly emerged as the preferred format. One could argue that a fully functional CJKV-capable application is completely worthless and meaningless without adequate font support.* Although the internal representation of the glyphs in these formats, which can range from bitmapped patterns to outline descriptions, may differ considerably, the final result, whether printed, displayed, or otherwise outputted, is simply an organized collection of bits or pixels. This is a very important point, so let me reiterate: *regardless of the format a font happens to be, the final result consists of nothing but BDPs.*†

Typical CJKV fonts include glyphs for the characters that correspond to the most common character set standards for a given locale. Nearly 20 years ago, for example, Japanese fonts that included glyphs for JIS X 0208:1997 Level 2 kanji were relatively uncommon, but now you can start expecting to find font products that include glyphs for the characters in CJK Unified Ideographs Extension B. However, the 64K glyph barrier—an important issue to be addressed in this chapter—prevents one from creating a single font instance that includes glyphs for all ideographs that are in Unicode.

First and foremost, selecting the point size of a particular font is probably one of the most common tasks one performs within an application.‡ The size of a font is usually described in units called *points*. The point is a term used in typography that represents a measurement that is approximately 1/72 of an inch, or 1/72.27 of an inch in more precise measurements, as standardized in the late 19th century. This means that a 72-point ideograph occupies a space that is roughly one inch wide and one inch tall. Ten- and 12-point

* Now you know why I like my job at Adobe Systems so much!

† Bits, dots, or pixels

‡ One could argue that typing in the characters themselves, which are rendered using glyphs in a font, is a much more common task.

fonts are the most common sizes used for text. The text of this book, for example, is set in 10.2-point Minion Pro. The subject of typographic units of measurement, including other versions or interpretations of the point, is a topic covered in Chapter 7.

This chapter covers what I consider to be the most popular font formats, either because they are widely used or have easily obtainable specifications for developers, or both. There are a myriad of font formats out there—bitmapped, vector, and outline—and it would be impossible (and impractical) to describe them all in this book. One font format, specifically OpenType, stands out from the others because it has become the most widely used and accepted. Your attention should thus be focused on the details of OpenType fonts.

Only the major CJKV type foundries have the appropriate level of design expertise and human resources necessary to create CJKV typefaces that contain thousands or tens of thousands of high-quality glyphs. Some of these CJKV type foundries include Adobe Systems (U.S. and Japan), Arphic Technology (Taiwan), Biblos Font (Japan), Changzhou SinoType Technology (China), Dainippon Screen (Japan), DynaComware (Taiwan and Japan), Enfour Media Laboratory (Japan), Fontworks (Japan), Founder Group (China), Hanyang Information & Communications (Korea), Hanyi Keyin (China), Jiyu-Kobo (Japan), Monotype Imaging (U.S., UK, and Hong Kong), Morisawa & Company (Japan), Nippon Information Science (Japan), Ryobi Imagix (Japan), Sandoll (Korea), Shaken (Japan), Type Project (Japan), TypeBank (Japan), TypeSolution (Korea), URW++ (Germany), Yoon Design Institute (Korea), and many others. CJKV type was originally cast in metal or wood, or handwritten, but of course the current trend is toward digital type that uses mathematically defined outlines to describe the glyph contours.

I should point out that some sections of this chapter intentionally contain sections that describe somewhat out-of-date or legacy font formats. This purpose is to encapsulate part of history, but the more practical purpose is for comparison to contemporary font formats, such as OpenType, and to demonstrate the clear progress that has been made in this area of software development. In the past, support for CJKV fonts was restricted to specific or specialized font formats that were supported only by specific environments or in limited versions of OSes. OpenType has changed this for the better. Support for CJKV fonts is now more uniform and widespread than ever before.

An excellent place to find or initiate font-related discussions is the Typophile forum.* Also consider Thomas Phinney's Typblography blog,† and the CCJK Type blog.‡

Typeface Design

A font's life begins as a typeface design. This may begin on paper. Younger typeface designers take advantage of computers to initiate typeface designs through the digital

* *http://typophile.com/forum/*

† *http://blogs.adobe.com/typblography/*

‡ *http://blogs.adobe.com/CCJKType/*

medium. A font simply cannot exist unless a typeface is designed, which is a process that involves a lot of effort on the part of the designer or design team.

When dealing with multiple typeface designs and different relative weights, there are many differences that may become clear. Whether or not these differences are readily apparent depends on whether you use multiple designs or weights in a single document and the size at which the glyphs are rendered. Of course, glyph differences become significantly more evident as the size increases.

Figure 6-1 illustrates differences in typeface design that may be observed when transitioning from lighter to heavier weights, using Adobe Systems' 小塚明朝 Pr6N EL and 小塚明朝 Pr6N H for the example.* You should take note of two major types of differences, both of which may appear to be somewhat subtle until pointed out:

- As the relative weight increases, some vertical strokes become thinner as they pass through and are enclosed by box-like elements. These instances of this characteristic are circle-shaded in Figure 6-1.
- For serif designs, the horizontal strokes often decrease in weight (thickness) as the overall weight of the typeface increases.

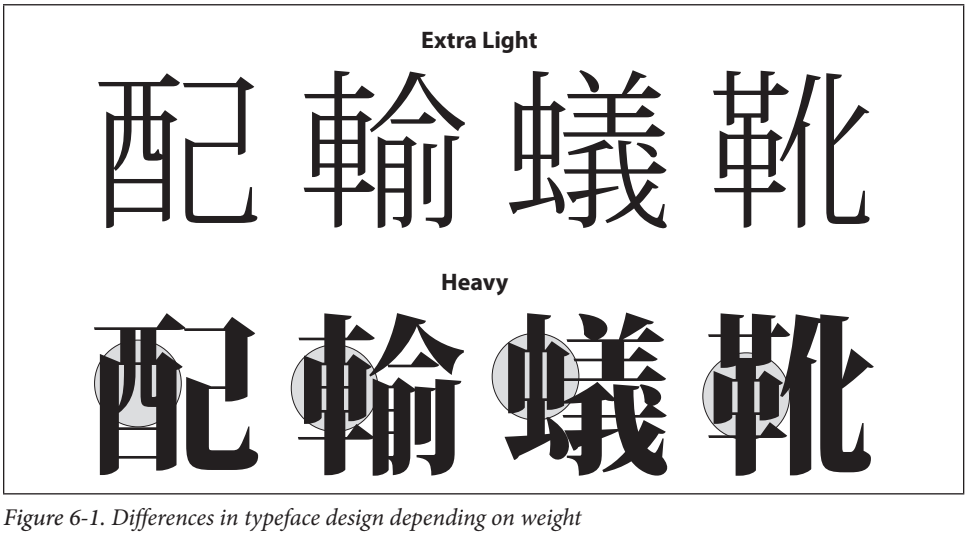


Figure 6-1. Differences in typeface design depending on weight

At first glance, or to the untrained eye, some typeface designs might look the same, or at least very similar. But when the same typeface designs are set at a much larger point size, designs that looked the same or similar when set at text sizes suddenly appear to be quite

* The corresponding PostScript font names are KozMinPr6N-ExtraLight and KozMinPr6N-Heavy, respectively. KozMin is short for Kozuka Mincho. It is named after its designer, Masahiko Kozuka (小塚昌彦 *kozuka masahiko*).

different. Some of the difference are in the shapes of the serifs or terminals of the strokes, in the angle of diagonal strokes, or in the relative thickness of the horizontal and vertical strokes. Let's take a closer look at the kanji 語 set at 150-point using two typefaces, KozMinPr6N-Heavy and HeiseiMinStd-W9, in Table 6-1.

Table 6-1. KozMinPr6N-Heavy and HeiseiMinStd-W9 at 150-point

KozMinPr6N-Heavy	HeiseiMinStd-W9
	

Do they look identical? Of course not. At least, I hope not. Are they similar? Sure. They represent the same character and are the same relative weight. If you look closely at the glyphs, you will observe that KozMinPr6N-Heavy has slightly curved terminal strokes, whereas HeiseiMinStd-W9 has squared ones. In addition, the horizontal strokes of the KozMinPr6N-Heavy glyph are noticeably thinner than those of HeiseiMinStd-W9.

Now, carefully observe the glyphs in Table 6-2, and compare them to those in Table 6-1.

Table 6-2. KozMinPr6N-Regular and HeiseiMinStd-W3 at 150-point

KozMinPr6N-Regular	HeiseiMinStd-W3
	

Carefully note how the horizontal strokes of the KozMinPr6N-Heavy glyph are significantly thinner than those of the KozMinPr6N-Regular glyph, but that the horizontal strokes of HeiseiMinStd-W9 and HeiseiMinStd-W3 are nearly identical in relative thickness. This is merely one of many ways in which typeface designs are distinguished from one another.

How Many Glyphs Can a Font Include?

When building fonts that include glyphs for CJKV characters, one obvious concern is file size. Another is how many glyphs can be included, especially because Unicode version 5.1 finally breaks the 100,000-character barrier. Are there limitations on file size or the number of glyphs? Any limitations could be exposed by using the tools that build fonts, such as AsiaFont Studio, or by using the fonts on the clients that make use them, meaning OSes, applications, and printers.

Being a professional CJKV font developer, the largest CIDFont resource that I have ever built contains 55,880 CIDs (Character ID, meaning glyphs) and is approximately 40 MB in size. It contains glyphs for all the characters in CNS 11643-1992 and CNS 11643-1986 Plane 15 and was developed specifically for printing Appendix G of the first edition of this book. It continues to be used for this edition in the form of an OpenType font.

There was once a project in Japan whose goal was to build a font that contains approximately 64,000 kanji.* The font was to be called GT 明朝 (*GT minchō*) or 東大明朝 (*tōdai minchō*) font. The "G" stands for 学術振興会 (*gakujutsu shinkōkai*), which means "Society for the Promotion of Science." The "T" stands for 東大 (*tōdai*), short for 東京大学 (*tōkyō daigaku*), which means "Tokyo University." It was likely that the first operating system to act as a client for this font is TRON, probably the BTRON instantiation.† Additional details about this project and its progress can be found at its website, and at LineLabo's website.‡

Today's font formats, such as those based on PostScript and TrueType outlines, are limited to 64K glyphs. For name-keyed PostScript fonts, the limit is 64,000 glyphs. For CID-keyed or GID-based fonts, the limit is slightly higher, a true 64K, specifically 65,535 glyphs. Given that the first CID or GID (Glyph ID, also meaning glyphs) is 0, this means that the highest possible value is at index 65534. For OpenType fonts, to include TrueType, the maximum GID is controlled by the *maxp.numGlyphs* value. (In other words, the *numGlyphs* value in the 'maxp' table.) 0xFFFF (65,535) is the largest possible value for this 'maxp' table field; because this value represents the *number* of glyphs, which includes GID at index 0, it means that the valid GID range is thus 0 through 65534 (65,535 GIDs).

* *http://www.um.u-tokyo.ac.jp/DM_CD/DM_TECH/KAN_PRJ/HOME.HTM*

† *http://www.l.u-tokyo.ac.jp/KanjiWEB/00_cover.html*

‡ *http://www.linelabo.com/*

The following section dives into the issue of the 64K glyph barrier in more detail. The difference between CID and GID is discussed later in this chapter, so if you are incredibly curious right now, please skip ahead to the section entitled "CID Versus GID."

Composite Fonts Versus Fallback Fonts

It is important to distinguish Composite Fonts, which are carefully crafted font recipes that are specifically designed to display glyphs from different component fonts in a harmonious way, and Fallback Fonts, which are designed to ensure that something useful, meaningful, or reasonable is always shown for any given character.

Composite Fonts allow developers and users to create a *virtual* font, which appears in application font menus as a new and unique font instance, takes specific ranges of glyphs from one or more component fonts, and combines those ranges to represent the glyph repertoire of the virtual font. In order for the glyphs of the component fonts to work harmoniously, it is sometimes necessary to adjust their relative sizes, escapements, or baselines. The end result of the Composite Font should be typographically pleasing, and suitable for publishing purposes.

Fallback Fonts, on the other hand, are not expected to appear in application font menus, and are thus virtual fonts in a genuine sense. Fallback Fonts are simply intended to display as many glyphs as possible, using fonts that are available to the OS or application, and are typically defined by OS or application developers. Fallback Fonts are defined by specifying a chain of fonts, listed in order of preference, and perhaps categorized by glyph coverage or Unicode encoding ranges. As the OS or application renders text, the first font in the Fallback Font list, appropriate for the glyph class or Unicode range, is used. If that font does not include a glyph for a specific code point, the subsequent fonts in the Fallback Font list are tried. Some OSes and applications make use of a last-resort font that includes glyphs that represent character classes or display the Unicode code point, and serve to indicate that no available font inludes an appropriate glyph. Apple has developed such a font,* which is made available through The Unicode Consortium.†

Of course, it is possible for Composite Fonts to participate in Fallback Font recipes. Both types of virtual fonts serve different purposes. To some extent, Fallback Fonts are like a double-edged sword. On the one hand, they become a user convenience, especially for purposes such as displaying or printing arbitrary web pages. As Unicode has grown, including more characters than can be included as corresponding glyphs in a single font resource, one could claim that Fallback Fonts have become a necessity. And, even if it were possible to include glyphs for all of Unicode's characters in a single font, doing so may not seem appropriate. Building fonts based on character class is much more manageable. On the other hand, when testing a font or application, Fallback Fonts can make it difficult to

* *http://developer.apple.com/textfonts/LastResortFont/*

† *http://www.unicode.org/policies/lastresortfont_eula.html*

know precisely what font is being used to render specific characters. Is it the font being tested? If the font being tested is broken, is a Fallback Font being used instead?

In any case, Composite Fonts and Fallback Fonts are similar in that they can allow the developer and user to address more code points than in standard font instances that have a fairly hard 64K glyph limit. Composite Fonts are further explored, in terms of specific formats, at the end of this chapter.

Breaking the 64K Glyph Barrier

Regardless of the font format, there is a very hard and fixed limit in terms of the number of glyphs that can be addressed or included in a single font instance. OpenType and TrueType fonts, along with the CIDFont resource, can include a maximum of 64K glyphs. For OpenType fonts with name-keyed 'CFF' tables, the limit is actually 64,000 glyphs. For TrueType fonts, OpenType fonts with 'glyf' tables, OpenType fonts with CID-keyed 'CFF' tables, and CIDFont resources, the maximum number of glyphs is genuinely 64K, meaning 65,535 glyphs.

Although this is often a point of confusion, I should point out that the 64K glyph barrier is completely unrelated to the total number of code points that can be addressed in the individual 'cmap' subtables of OpenType and TrueType fonts, and the ability to address Unicode code points that are beyond the BMP. CMap resources, such as 'cmap' tables, can address more than 64K code points, and can also address code points beyond the BMP as long as its encoding is specified accordingly.

CFF FontSets seem to provide a mechanism for easily breaking the 64K glyph barrier, at least in the context of OpenType fonts that include a 'CFF' table. Each individual FontSet is subject to the 64K glyph barrier, but a single 'CFF' table can contain up to 65,535 FontSets. At first glance, this seems to provide a trivial way for OpenType fonts with 'CFF' tables to effectively support more than 64K glyphs. Unfortunately, the world is not so simple. The other tables in an OpenType font that reference the GIDs specified in the 'CFF' table, such as the 'cmap', 'GPOS', and 'GSUB' tables, are blissfully unaware of CFF FontSets, and would need to be extensively revised to enable indexing into multiple CFF FontSets. While such an effort would certainly be noble, it is an incredible amount of work, and affects not only the fonts themselves, but clients that use the fonts. Furthermore, it would benefit only OpenType fonts with 'CFF' tables. Those with 'glyf' tables would still be bound to the 64K glyph barrier. For this reason, CFF FontsSets do not represent a viable solution to breaking the 64K glyph barrier in a cross-platform way, and perhaps more importantly, in a way that supports both PostScript and TrueType outlines.

Given the complex and difficult nature of forcibly breaking the 64K glyph barrier by revising the specifications of the affected tables, which involves extensive research and study, a far easier approach would be to define a genuinely cross-platform Composite Font format. Such a Composite Font would constitute a recipe whereby a single virtual font is established, and its description includes and depends upon one or more component fonts, the combination of which includes more than 64K glyphs. One important consideration

is that glyphs are not isolated objects, but rather interact with one another. For example, glyph substitution is now common, and kerning defines a context that affects more than one glyph. In terms of suitable language for establishing such a cross-platform composite font mechanism, XML seems like a clear and somewhat logical choice. XML's human-readable property is an obvious benefit.

Should the Composite Font be instantiated as a separate file, perhaps as XML, or encapsulated in a new or existing OpenType table? As an XML file, its contents become more visible to users. As an OpenType table, the Composite Font specification can become part of the base, primary, or parent font. Also, should the component fonts be allowed to function as standalone fonts? These are all very important questions and considerations. What must be made clear is that a major motivation for such a Composite Font format is the ability to address more than 64K glyphs through a single selectable font instance, and any solution needs to bear this in mind.

Bitmapped Font Formats

Let us take a step back to a time when outline fonts were not common. The first CJKV fonts for use on computer systems were bitmapped. This meant that each glyph was constructed from a matrix of dots or pixels, each of which could be turned on or off—this is referred to as a *dot-matrix*. The limitation of such font formats is that the resulting bitmapped patterns are restricted to a single point (or pixel) size. Any scaling applied to a bitmapped font almost always produces irregular-looking results, often referred to as the "jaggies" or the "Lego* effect." Skip ahead to Figure 8-1 in Chapter 8, which provides an example of a scaled 12-pixel bitmapped glyph.

Obviously, the larger the dot-matrix pattern, the more memory such a bitmapped font requires, especially when designing a bitmapped CJKV font that contains several thousand glyphs, or tens of thousands of glyphs, depending on which character set or character sets are supported. There is also the issue of needing to design a complete set of glyphs for every pixel size that is required.

It must be stated, however, that bitmapped fonts do have their merits. More advanced font technologies, which you will learn about later in this chapter, may produce poor-quality results at small point sizes and on low-resolution output devices, such as computer displays or mobile devices. With bitmapped fonts, the user can take full advantage of hand-tuned glyphs for commonly used point sizes. And, when few bitmap sizes are necessary, bitmapped fonts can be quite compact compared to outline fonts.

There are a myriad of bitmapped font formats available, but there are two very common formats that can be used for representing CJKV glyphs: BDF and HBF. The following sections briefly describe each of these.

* *http://www.lego.com/*

As advanced as OpenType fonts are, they can also include bitmapped font data, in the form of the 'EBDT', 'EBLC', and 'EBSC' tables whose heritage lies in TrueType fonts. AAT fonts, which are a flavor of TrueType fonts, along with sfnt-wrapped CIDFonts, can include the 'bdat' and 'bloc' tables for specifying bitmapped font data. Interestingly, the OpenType fonts produced by Adobe Systems do not include any bitmapped font data whatsoever. OpenType fonts do not require any bitmapped font data.

BDF Font Format

One of the most commonly used bitmapped font formats is called BDF, short for *Bitmap Distribution Format*. This was developed by Adobe Systems, and was subsequently adopted by the X Consortium for use in the X Window System. (Although the latest version of the BDF specification is version 2.2, the X Consortium has adopted version 2.1.)

A BDF file is composed of two main sections: a BDF header in which font-level attributes are specified; and the individual BDF records, one for each glyph in the BDF file. An example BDF header, which can be quite long, is illustrated later in this chapter, in the section entitled "PostScript extensions for X Window System fonts."

It is relatively easy to use and manipulate BDF fonts. In fact, many people have used freely available CJKV BDF fonts for developing their own software, such as for the UI (*user interface*) font. Figure 6-2 represents a complete BDF description for the ideograph 剣. This BDF record corresponds to the 24×24 dot-matrix pattern that is illustrated later in Figure 6-3.

```
STARTCHAR 3775
ENCODING 14197
SWIDTH 1000 0
DWIDTH 24 0
BBX 24 24 0 -2
BITMAP
018007
018006
0360E6
0318C6
060CC6
0C06C6
1866C6
37F0C6
C180C6
218CC6
3FFEC6
318CC6
318CC6
318CC6
318CC6
3FFCC6
318CC6
0340C6
033006
061806
0C1C06
180C06
300C3E
C0000C
ENDCHAR
```

Figure 6-2. A BDF record example

You must be wondering how to interpret this eye-catching BDF bitmap record data. Table 6-3 lists the BDF format keywords, along with a brief description of each.

Table 6-3. BDF bitmap record keyword descriptions

Keyword	Example	Description
STARTCHAR name	STARTCHAR 3775	Glyph name—can be anything; 3775 is hexadecimal ISO-2022-JP
ENCODING n	ENCODING 14197	Decimal encoding of glyph; 14197 is decimal ISO-2022-JP
SWIDTH x y	SWIDTH 1000 0	Scalable width expressed as a vector
DWIDTH x y	DWIDTH 24 0	Device width expressed as a vector
BBX w h x y	BBX 24 24 0 -2	The bounding box for the character in device units
BITMAP	BITMAP	Beginning of bitmapped pattern
ENDCHAR	ENDCHAR	End of glyph description

The bitmapped data in each bitmap record of a BDF file that describes 24×24 dot-matrix patterns consists of 24 lines of 6 characters each. Each line represents 24 pixels, thus each character represents 4 pixels. Each of these six characters can have a value in the range 0x00 through 0x0F (that is, 0x00 through 0x09 or 0x0A through 0x0F). This allows for up to 16 unique values, which is the total number of unique patterns that can be generated using 4 pixels. Table 6-4 illustrates these 16 values as the value's corresponding binary patterns.

Table 6-4. The 16 unique binary patterns in BDF data

Value	Binary pattern
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Let's take a closer look at the beginning of the BDF bitmap record as shown in Figure 6-2, specifically the following six hexadecimal digits:

018007

The binary pattern that is represented by these data corresponds to a pixel pattern: zeros correspond to white pixels, and ones correspond to black pixels. Table 6-5 illustrates how these binary patterns correspond to their equivalent pixel patterns.

Table 6-5. Binary and pixel patterns in BDF data

BDF data	0	1	8	0	0	7
Binary pattern	0000	0001	1000	0000	0000	0111
Pixel pattern	□□□□	□□□■	■□□□	□□□□	□□□□	□■■■

Note how there are 24 pixels across, and 24 such lines. Compare this pixel pattern with what you see in the first row of pixels in Figure 6-3. In fact, it is more efficient to read these binary patterns two characters at a time, into a single byte. A byte can store a binary pattern eight digits long. For example, the first two characters, 01, are stored into a single byte, and its binary pattern is 00000001 (or, more graphically, □□□□□□□■).

BDF files can be reduced in size by converting them into binary representations, such as *Server Natural Format* (SNF) or *Portable Compiled Format* (PCF). The standard X Window System utilities, called *bdftosnf* and *bdftopcf*, respectively, are used for this purpose. The X Window System (X11R5 and later) prefers PCF. SNF is older, preferred by X11R4 and earlier, and skips or ignores unencoded BDF records.* PCF is obviously the newer format, and it is considered to be superior to SNF in many respects because it uses a more efficient representation. For a complete and authoritative description of BDF, see the document entitled *Glyph Bitmap Distribution Format (BDF) Specification* (Adobe Systems Technical Note #5005).†

Figure 6-3 illustrates example 24×24 and 16×16 dot-matrix patterns for the single ideograph 剣. The 24×24 and 16×16 dot-matrix patterns illustrated in this figure were taken directly from the JIS X 9052-1983 and JIS X 9051-1984 standards, respectively.‡ These manuals were once useful if one's OS or environment did not support all the characters in JIS X 0208. This case is now extremely rare, and would arise only if one is using an ancient OS or a closed environment with little or no Japanese support. I once put these two manuals to practical use when I was working with EGWord version 2.2 by ERGOSOFT on Mac OS back in the late 1980s; that word-processing application supported only JIS

* An unencoded BDF record is one whose ENCODING field has its value set to −1.

† All of Adobe Systems' Technical Notes that are referenced in this chapter are available as PDF (Adobe Acrobat) files at *http://www.adobe.com/devnet/font/*.

‡ Note that these manuals are based on JIS X 0208-1983—they were never updated by JSA to conform to JIS X 0208-1990 or JIS X 0208:1997.

Level 1 kanji. Interestingly, versions of EGWord beyond version 2.2 included support for JIS Level 2 kanji by virtue of running under Mac OS-J.

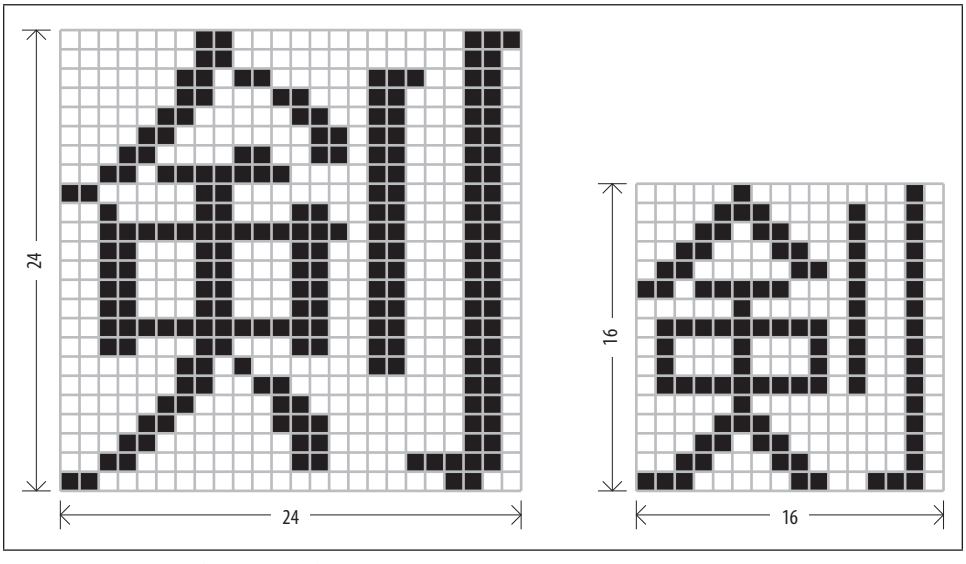


Figure 6-3. Japanese bitmapped characters

You can find bitmapped CJKV fonts at FTP sites, usually in the form of BDF files.* There are easily obtainable tools that allow BDF files to be converted into other formats. Some of these tools, such as *bdftosnf* or *bdftopcf*, may be already installed if you are running the X Window System. You can also use Mark Leisher's *XmBDFEditor*, or his newer *gbdfed*, to create and edit BDF files; this X Window System tool is described later in the chapter. The FreeType Project has developed a TrueType-to-BDF converter called *ttf2bdf*, which is useful for creating CJKV BDF fonts when a suitable BDF font cannot be found, and is based on FreeType version 1.† This tool has been surpassed by Mark's *otf2bdf* tool, which is based on FreeType version 2.‡

HBF Font Format

The *Hanzi Bitmap Font* (HBF) format is similar to BDF in how it describes the actual character bitmap patterns, but it also creates a space-saving advantage by assuming that all the characters have certain attributes in common, such as a common display and pixel width.

* I suggest that you try *ftp://etlport.etl.go.jp/pub/mule/fonts/*, *ftp://ftp.ifcss.org/pub/software/fonts/*, or even *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/fonts/*.

† *http://www.freetype.org/*

‡ *http://www.math.nmsu.edu/~mleisher/Software/otf2bdf/*

The HBF format stores the font attributes and common per-bitmap attributes in a human-readable text file, which is based on BDF. This is called the "HBF specification file." This HBF-specific file introduces several new keywords not present in BDF fonts. Table 6-6 lists these new keywords, along with an example of their use.

Table 6-6. HBF-specific keywords

Keyword	Example
HBF_BYTE_2_RANGE n-n	HBF_BYTE_2_RANGE 0xA1-0xFE
HBF_CODE_RANGE n-n file offset	HBF_CODE_RANGE 0xA1A1-0xA8FE jis97.24 0

The `HBF_BYTE_2_RANGE` keyword defines the valid byte-value range for the second byte of the encoding. The `HBF_CODE_RANGE` keyword defines an encoding block, followed by the font filename, then an offset value, expressed as the number of bytes, into the file. The font filename is specified because it is possible to refer to more than one font file according to the HBF format.

The actual bitmapped patterns are stored as a binary representation in one or more font files. The HBF format is suitable if all the glyphs share common attributes, such as being the same width—for example, half- or full-width. If variable-width characters are necessary, HBF is not suitable for your needs.

The HBF format is further described in the article entitled "The HBF Font Format: Optimizing Fixed-pitch Font Support" (pages 113–123 of *The X Resource*, Issue 10) and in online resources.*

Outline Font Formats

During the early 1980s there was a revolution in the desktop publishing industry made possible by the introduction of PostScript, the page-description language developed by Adobe Systems. PostScript provides a seamless mixture of text and graphics, and, with the development of laser printers, brought high-quality publishing capability to many more people than ever before. Now, for very little money, anyone can purchase the necessary hardware and software to produce high-quality CJKV text by themselves. As far as CJKV fonts are concerned, the PostScript language supports a number of font formats that represent glyphs as scalable outlines.

In the late 1980s, Microsoft and Apple collaborated to develop a scalable font format known as TrueType. The two companies went their separate ways (mainly due to the fact that they also develop competing OSes), and independently enhanced the TrueType format, but not to the point that they became completely incompatible with one another. TrueType, of course, is fully capable of representing CJKV fonts.

* *http://www.ibiblio.org/pub/packages/ccic/software/info/HBF-1.1/*

We will discuss both PostScript and TrueType font formats in greater detail later in this chapter. However, the principles and concepts behind each of these formats have much in common. In both cases, glyphs are constructed from outlines. This effectively means that each glyph is described mathematically as a sequence of line segments, arcs, and curves. This outline is scaled to the selected point size, filled, and then rendered as BDPs to the output device.* A glyph from an outline font can be used at any conceivable point size and resolution.† The design process is also simplified in that the designer need not worry about the point size of the font, and thus does not need to design more than a single point size.‡

Figure 6-4 represents an example glyph from an outline font. The outline is constructed from line segments and curves. The anchor points that describe the outline are marked with small black squares along the actual outline of the glyph. The offline control points that define the Bézier curves (explained briefly in the next paragraph) are represented by small black circles with lines drawn to their respective anchor points. In this example, the glyph for the ideograph 剣 from FDPC's 平成角ゴシック Std W5 (HeiseiKakuGoStd-W5) typeface is used.

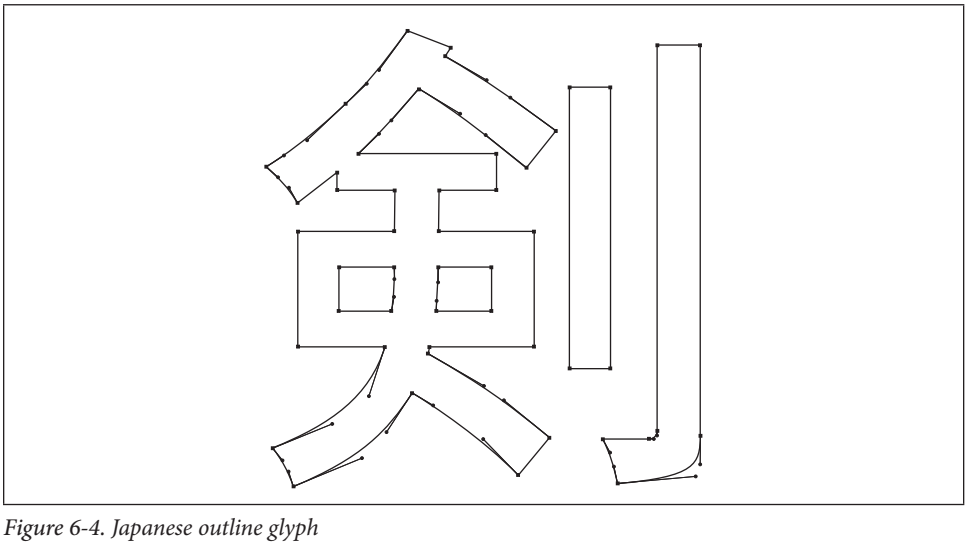


Figure 6-4. Japanese outline glyph

* In case you're wondering, rendering (or scan converting or rasterizing) is the process of converting a scaled outline into a bitmapped image. The outline format is only for more efficient internal representation and storage.

† Unless, of course, the font is *resolution-restricted*. This means that whether you have a low- or high-resolution device will decide what version of a font you must purchase. Needless to say, the high-resolution version costs more. This is, of course, an artificial restriction.

‡ That is, unless the designer intends to support optical size, in which case Adobe Systems' multiple master font technology, described later in this chapter, comes to the rescue.

Currently, the two most commonly used and encountered outline font technologies are PostScript and TrueType. These have been effectively merged into a newer font format know as OpenType. In terms of PostScript and TrueType outlines, specifically the way in which the glyphs are represented, their primary difference lies in that PostScript uses Bézier curves, which are also known as cubic splines, and TrueType uses quadratic splines. Exactly what this means, and how they are different, is well beyond the scope of this book, but it is useful to note that Bézier curves require fewer control points to represent the same contours. This generally leads to fonts that are smaller, and easier to transfer, and that require less disk storage space. Most users honestly do not care on what underlying font technology their fonts are based, so long as they achieve the desired results. In other words, users are happy as long as their fonts work as expected. The vast majority of users are—and honestly should be—insulated from knowing the details of the underlying font format.

PostScript Font Formats

PostScript is a powerful page-description language backed by a complete programming language. As a programming language, PostScript is syntactically similar to Forth—both are stack-based languages. As a page-description language, PostScript supports both graphics and text, and to render text effectively, provides built-in support for a wide variety of font formats.*

The most widely used format supported by PostScript is called Type 1. Table 6-7 summarizes these PostScript font formats with a brief description of their purpose.

Table 6-7. PostScript font formats

Font Type	Description
0	Composite font format
1	The basic and most widely used font format
2	Charstring format primarily for use with CFF—a lossless compaction of Type 1
3	User-defined font format
4	Disk-based font format—a Type 1 font stored in a way that saves RAM[a]
5	ROM-based font format—actually a Type 1 font stored in a special compressed format for ROM storage
9[b]	CIDFont with Type 1 glyph procedures—equivalent to CIDFontType 0
10[b]	CIDFont with PostScript BuildGlyph procedures, which is similar to Type 3 fonts—equivalent to CIDFontType 1
11[b]	CIDFont with TrueType glyph procedures—equivalent to CIDFontType 2

* But I don't recommend that you go out and use PostScript for your general programming needs. For one thing, most PostScript interpreters are available only in PostScript printers, although Ghostscript and DPS are implemented elsewhere.

Table 6-7. PostScript font formats

Font Type	Description
14	Chameleon ROM font format
42	A TrueType font inside a PostScript wrapper—requires that a TrueType rasterizer be present
CFF	Compact representation for fonts that use Type 1 charstrings—these charstrings can be Type 1 or 2

a. PostScript aficionados usually refer to RAM as VM (*virtual memory*).

b. Font Types 9 through 11 represent native-mode implementations of a CIDFont usable as a font resource instance, and only the "glyphshow" operator can be used as their "show" operator.

The following provides a more detailed description and synopsis of each of these PostScript font formats:

- Type 0 fonts are composite fonts, which are made up of two or more descendant fonts. A composite font constitutes a hierarchical structure in which large character sets can be supported. A descendant font can be a Type 1, Type 3, or even Type 0 font.* PostScript CJKV fonts are Type 0 fonts. CID-keyed fonts are also Type 0 fonts (with an FMapType value of 9—FMapType is described later in this chapter).
- Type 1 fonts are the most commonly used PostScript fonts. They use a special, restricted subset of PostScript (along with additional operators), which allows for a more compact (and faster-rendering) font. The Type 1 font format is typically used for all Latin (that is, non-CJKV) fonts issued by Adobe Systems. It is also the format specified by the international standard designated ISO 9541:1991 (*Information Technology—Font Information Interchange*), Parts 1, 2, and 3. The Japanese equivalent of this standard has been published as a series of four manuals. Detailed information on the Type 1 font format can be found in Adobe Systems' *Adobe Type 1 Font Format, Version 1.1* (Addison-Wesley, 1990). Extensions to the Type 1 format are described in *The Type 1 Font Format Supplement* (Adobe Systems Technical Note #5015).
- Type 2 is not a font format, but rather a charstring format used within a CFF (*Compact Font Format*) structure. The word *charstring* (an abbreviated form or contraction of "character string") here means the code that forms the description of the outline—think of it as a mini program that describes how to draw a glyph—as opposed to the entire wrapping mechanism used to contain all the charstrings for a single font. Type 2 operators form a superset of the Type 1 operators, which results in a smaller, more compact and more efficient representation, and offers the opportunity for better rendering quality than Type 1. This charstring format can also be considered as a lossless compaction of Type 1. More information on Type 2 can be found in *The Type 2 Charstring Format* (Adobe Systems Technical Note #5177).

* A Type 0 font that is composed of Type 0 descendent fonts is referred to as a *nested composite font*.

- Type 3 is sometimes called a user-defined font format. Type 3 fonts are much like Type 1 fonts, but allow for more complex outlines, such as logos and designs, and permit the use of the full range of PostScript operators. Type 3 fonts are not that common, do not work with Adobe Type Manager (ATM) software (as discussed in Chapter 8) and are not supported natively by Mac OS X, Windows, or other OSes. They also cannot be hinted (this is true for all practical purposes, but it is possible to hint Type 3 charstrings if, and only if, you write your own hint procedures—but this still means they will not work with ATM).
- Type 4 fonts are Type 1 fonts, but are disk-based rather than printer-resident. Type 1 fonts now have all the benefits of Type 4, such as being able to be read from disk on an as-needed basis.
- Type 5 fonts are Type 1 fonts packaged in a special way for storage within printer ROM.
- Type 9 fonts are CIDFonts that use Type 1 charstrings. All CIDFonts built by Adobe Systems are Type 9. Type 9 is also referred to as CIDFontType 0.
- Type 10 fonts are CIDFonts that use Type 3 charstrings. That is, they implement the PostScript BuildGlyph procedure. Type 10 is also referred to as CIDFontType 1.
- Type 11 fonts are CIDFonts that use TrueType charstrings. A TrueType rasterizer must be present on the PostScript device in order to use Type 11 fonts. Type 11 is also referred to as CIDFontType 2. More information on this TrueType CIDFont format is available in *PostScript Language Extensions for CID-Keyed Fonts* (Adobe Systems Technical Note #5213).
- Type 14 fonts are so-called Chameleon fonts that were designed to represent a large number of fonts in a compact fashion, in order to consume only a small amount of storage space, such as for printer ROM. Adobe Systems never documented this font format.
- Type 42 fonts are actually TrueType fonts with a PostScript wrapper so that they can reside within PostScript printers, and act much like PostScript fonts. A TrueType rasterizer must be present on the PostScript device in order to use Type 42 fonts.
- CFF (*Compact Font Format*) is a method that represents Type 1 and CIDFonts much more compactly than ever before. It is a font wrapper or container designed to be used primarily with Type 1 or Type 2 charstrings, though Type 2 charstrings are the default and also more size-efficient. Proper tools can convert Type 1 and CIDFonts to CFF and back again, with no loss of data, so CFF can be thought of as a way to transform existing font formats into a more compact representation. CFF is natively supported in PostScript 3 and higher. Information about CFF is found in *The Compact Font Format Specification* (Adobe Systems Technical Note #5176).

But, why the number 42? What is the significance of selecting such a seemingly high number?*

Henry McGilton and Mary Campione's book entitled *PostScript by Example* (Addison-Wesley, 1992) is an excellent tutorial, and has a chapter on fonts, including a superb tutorial on composite fonts. Adobe Systems' *PostScript Language Reference Manual*, Third Edition (Addison-Wesley, 1999), provides a complete description of the PostScript language, including information on Type 0 and Type 3 fonts. The many Technical Notes provided by Adobe Systems, several of which I have authored, provide additional information about PostScript and related font technologies. These Technical Notes are also freely available.†

FMapType—for Type 0 fonts

All Type 0 (composite) fonts must specify an FMapType. An FMapType is a PostScript language key, represented by an integer, that indicates which mapping algorithm to use when interpreting the sequence of bytes in a string. There are currently eight FMapTypes that have been defined, all of which are described in the *PostScript Language Reference Manual*. Table 6-8 lists the FMapTypes, along with explanations as provided in the *PostScript Language Reference Manual*.

* It has been rumored that the number 42 was chosen by an unidentified (Apple) employee who was being humorous. In *The Hitchhiker's Guide to the Galaxy* and its sequels (written by Douglas Adams and published by Pocket Books), a god-like computer named Deep Thought is asked to calculate the answer to the *Ultimate Question of Life, the Universe, and Everything*. After computing for seven and a half million years, Deep Thought returns the value 42.

Reality sets in. In a September 27, 1995 post to *comp.fonts* that I happened to spot, Kevin Andresen revealed to the world that he was the person who chose the number 42. I quote:

> *I named it, and for you conspiracy theorists out there, it did not mean TrueType was the answer to Type 1! Maybe the real story later.…*

In a private email communication, Kevin conveyed the details to me:

> *My group back at Xerox had changed their workstation/server/printer naming theme from Mad Max to Hitch Hiker's Guide just before I left—our main development printer was named "Forty-Two." I picked 42 because I knew that Adobe couldn't accidentally go there next, and as a wink and a nudge to my friends back in Rochester. It was only after Adobe and Apple resumed their business relationship that I heard the "conspiracy theory" about the answer to Type 1. By then, we had already released the spec to TrueType developers, so the name stuck.*

Kevin also told me that he wrote an unimplemented specification for the disk-resident form of Type 42—similar to Type 4 fonts—called Type 44. He chose 44 because he had a cold at the time, and was guzzling Vicks Formula 44D. Oh, and Kevin no longer works for Apple.

As a footnote to this footnote, Douglas Adams had a few words to say about the significance of 42, from an email originally sent to Kevin:

> *Everybody tries to find significances for 42. In fact, it's the other way around—many more significances have been created than previously existed. (Of course, the number that previously existed was zero—it was just a joke.)*

Sadly, Douglas Adams passed away on May 11, 2001 at the age of 49.

Perhaps on a somewhat brighter note, almost all of the second edition of this book was written while I was 42 years of age.

† *http://www.adobe.com/devnet/font/*

Table 6-8. FMapTypes for Type 0 fonts

FMapType	Algorithm	Explanation
2	8/8 mapping	Two bytes are extracted from the "show" string. The first byte is the font number, and the second byte is the character code.
3	Escape mapping	One byte is extracted from the "show" string. If it is equal to the value of the EscChar entry, the next byte is the font number, and subsequent bytes (until the next escape code) are character codes for that font. At the beginning of a "show" string, font 0 is selected.
4	1/7 mapping	One byte is extracted from the "show" string. The most significant bit is the font number, and the remaining seven bits are the character code.
5	9/7 mapping	Two bytes are extracted from the "show" string and combined to form a 16-bit number, high-order byte first. The most significant nine bits are the font number, and the remaining seven bits are the character code.
6	SubsVector mapping	One or more bytes are extracted from the "show" string and decoded according to information in the SubsVector entry of the font.
7	Double escape mapping	Similar to FMapType 3. When an escape code is immediately followed by a second escape code, a third byte is extracted from the "show" string. The font number is the value of this byte plus 256.
8	Shift mapping	This mapping provides exactly two descendant fonts. A byte is extracted from the "show" string. If it is the ShiftIn code, subsequent bytes are character codes for font 0. If it is the ShiftOut code, subsequent bytes are character codes for font 1. At the beginning of the "show" string, font 0 is selected.
9	CMap mapping	One or more bytes are extracted from the "show" string and decoded according to information in the font's CMap entry.

FMapTypes 7 and 8 are available only in PostScript Level 2 implementations. FMapType 9 is available only on CID-capable PostScript devices, including PostScript 3.

FMapTypes 3, 7, and 8 support modal encodings (that is, some bytes, such as escape or shifting characters, are used to indicate information other than characters themselves). FMapTypes 2, 4, 5, 6, and 9 support nonmodal encodings—all the bytes are used to encode characters themselves.

The composite fonts that are supported in Adobe Systems' Japanese OCF fonts use either FMapType 2 (for strictly two-byte encodings, such as ISO-2022-JP and EUC-JP)* or FMapType 6 (for mixed one- and two-byte encodings, such a Shift-JIS). CID-keyed fonts, of course, use FMapType 9.

* As you'll soon discover, Adobe Systems' Japanese OCF fonts support only EUC code set 1 (JIS X 0208:1997). CID-keyed fonts add support for code sets 0 and 2 (ASCII/JIS-Roman and half-width katakana, respectively).

Composite fonts

The very first composite fonts offered by Adobe Systems are now referred to as *Original Composite Format* (OCF) fonts.* Adobe Systems produced OCF fonts only for Japanese—Chinese and Korean fonts offered by Adobe Systems were originally available only in the newer CID-keyed font file specification.† Interestingly, the actual charstrings (that is, outlines) of OCF fonts and CID-keyed fonts are identical, including their hinting. The CID-keyed font file specification uses a much more compact and efficient packaging mechanism and file structure and is highly extensible when it comes to supporting additional or complex encodings.

PostScript CJKV fonts are implemented as collections of composite fonts that support various character sets and encoding methods. They also have special naming conventions not found in other fonts offered by Adobe Systems. A fully qualified PostScript CJKV font name consists of several parts: family name, face name (usually indicating the weight or other characteristic that distinguishes it from other members of the same typeface family), character set, encoding, and writing direction. The combination of family name and face name represents the base font name. Table 6-9 shows possible values for each of these parts (using examples from now-obsolete OCF fonts).

Table 6-9. PostScript CJKV font naming conventions

Base font name	Character set	Encoding	Writing direction
Ryumin-Light	83pv	RKSJ	H (horizontal)
GothicBBB-Medium	Add	EUC	V (vertical)
FutoMinA101-Bold	Ext	and so on…	
MidashiGo-MB31	NWP		
HeiseiMin-W3	and so on…		
and so on…			

The following are notes that accompany Table 6-9:

- When the "Character Set" is not specified, the default is the standard JIS character set (meaning JIS X 0208-1983 or JIS X 0208:1997, depending on the vintage of the font).
- When the Encoding is not specified, the default is ISO-2022-JP encoding.
- 83pv stands for "JIS X 0208-1983 plus verticals" (or perhaps "JIS X 0208-1983 plus proportional and verticals"—no one at Adobe Systems really knows for sure), and it

* Contrary to popular belief or rumor, OCF does not, I repeat, does not stand for *Old Crappy Font*. When they were first conceived, OCF fonts were quite revolutionary.

† All of Adobe Systems' Japanese OCF fonts have been upgraded to become CID-keyed fonts, most of which conform to the Adobe-Japan1-2 character collection, described shortly.

represents the KanjiTalk version 6 character set, which also contains the vertically set characters. The writing direction is always specified as H because most of the characters are horizontal.

- RKSJ stands for Roman, (half-width) Kana, and Shift-JIS.
- The Add character set represents a version of Fujitsu's FMR Japanese character set.
- EUC, as you might expect, stands for Extended Unix Code. The only recognized EUC-JP code set for OCF fonts is code set 1, for JIS X 0208:1997. CID-keyed fonts, described shortly, support EUC-JP code sets 0, 1, and 2.
- The Ext character set represents the NEC Japanese character set.
- The NWP character set, short for *NEC Word Processor*, represents the NEC Japanese character set as implemented on NEC dedicated Japanese word processors.

Note that not all combinations of Character Set and Encoding specified in Table 6-9 are supported. The following are the fully qualified PostScript font names for the typeface called HeiseiMin-W3, using the OCF version for the example:

HeiseiMin-W3-H
HeiseiMin-W3-V
HeiseiMin-W3-Add-H
HeiseiMin-W3-Add-V
HeiseiMin-W3-Ext-H
HeiseiMin-W3-Ext-V
HeiseiMin-W3-NWP-H
HeiseiMin-W3-NWP-V
HeiseiMin-W3-EUC-H
HeiseiMin-W3-EUC-V
HeiseiMin-W3-RKSJ-H
HeiseiMin-W3-RKSJ-V
HeiseiMin-W3-Add-RKSJ-H
HeiseiMin-W3-Add-RKSJ-V
HeiseiMin-W3-Ext-RKSJ-H
HeiseiMin-W3-Ext-RKSJ-V
HeiseiMin-W3-83pv-RKSJ-H

Compare this with Table 6-9 to find out what character sets and encoding methods each font instance supports. There are also some special one-byte–encoded fonts that make up

Adobe Systems' PostScript Japanese fonts, indicated as follows (again, using the OCF version of HeiseiMin-W3 for the example):

HeiseiMin-W3.Hankaku
HeiseiMin-W3.Hiragana
HeiseiMin-W3.Katakana
HeiseiMin-W3.Roman
HeiseiMin-W3.WP-Symbol

You may notice that the character set, encoding, and writing direction naming conventions that were used for OCF fonts have carried forward to the naming conventions for CMap resources (covered later in this chapter). In particular, the Adobe-Japan1-6 CMap resource names that are provided in Table 6-34 reflect this heritage.

Over the years, the Adobe Type Library has included a large number of Japanese typefaces (designed by Adobe Systems as Adobe Originals, and licensed from FDPC,* Kamono Design Laboratory, Morisawa, and TypeBank), a small number of Simplified Chinese typefaces (purchased from Changzhou SinoType Technology), a small number of Traditional Chinese typefaces (licensed from Monotype Imaging and purchased from Arphic Technology), and several Korean typefaces (originally licensed from SoftMagic, and licensed or purchased from Hanyang Information & Communications). Table 6-10 lists the OpenType CJKV fonts that are designed or owned by Adobe Systems. Of course, Adobe Systems licenses and offers additional OpenType CJKV fonts not shown in this table.† The typefaces that Adobe Systems offers are licensed or purchased from a type foundry, then produced by Adobe Systems or their font tools licensees to strictly conform to PostScript font specifications. The majority of the Japanese typefaces offered by Adobe Systems are designed by its Tokyo-based Japanese type design team.

Table 6-10. Adobe Systems' OpenType CJKV typefaces

Typeface name[a]	PostScript name	ROS	Sample text
かづらき Std L	KazurakiStd-Light	Adobe-Identity-0	わえ文じもじモジ
小塚ゴシック Pr6N EL	KozGoPr6N-ExtraLight	Adobe-Japan1-6	和文文字もじモジ
小塚ゴシック Pr6N L	KozGoPr6N-Light	Adobe-Japan1-6	和文文字もじモジ
小塚ゴシック Pr6N R	KozGoPr6N-Regular	Adobe-Japan1-6	和文文字もじモジ
小塚ゴシック Pr6N M	KozGoPr6N-Medium	Adobe-Japan1-6	**和文文字もじモジ**
小塚ゴシック Pr6N B	KozGoPr6N-Bold	Adobe-Japan1-6	**和文文字もじモジ**
小塚ゴシック Pr6N H	KozGoPr6N-Heavy	Adobe-Japan1-6	**和文文字もじモジ**

* FDPC, if you recall, stands for *Font Development and Promotion Center* (文字フォント開発・普及センター *moji fonto kaihatsu fukyū sentā*). FDPC is now defunct, and JSA (*Japanese Standards Association*) has since taken over the licensing and development of the Heisei fonts.

† *http://www.adobe.com/type/*

Table 6-10. Adobe Systems' OpenType CJKV typefaces

Typeface name[a]	PostScript name	ROS	Sample text
小塚明朝 Pr6N EL	KozMinPr6N-ExtraLight	Adobe-Japan1-6	和文文字もじモジ
小塚明朝 Pr6N L	KozMinPr6N-Light	Adobe-Japan1-6	和文文字もじモジ
小塚明朝 Pr6N R	KozMinPr6N-Regular	Adobe-Japan1-6	和文文字もじモジ
小塚明朝 Pr6N M	KozMinPr6N-Medium	Adobe-Japan1-6	和文文字もじモジ
小塚明朝 Pr6N B	KozMinPr6N-Bold	Adobe-Japan1-6	**和文文字もじモジ**
小塚明朝 Pr6N H	KozMinPr6N-Heavy	Adobe-Japan1-6	**和文文字もじモジ**
りょうゴシック PlusN EL	RyoGothicPlusN-ExtraLight	Adobe-Japan1-3 + 144	和文文字もじモジ
りょうゴシック PlusN L	RyoGothicPlusN-Light	Adobe-Japan1-3 + 144	和文文字もじモジ
りょうゴシック PlusN R	RyoGothicPlusN-Regular	Adobe-Japan1-3 + 144	和文文字もじモジ
りょうゴシック PlusN M	RyoGothicPlusN-Medium	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょうゴシック PlusN B	RyoGothicPlusN-Bold	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょうゴシック PlusN H	RyoGothicPlusN-Heavy	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょうゴシック PlusN UH	RyoGothicPlusN-UltraHeavy	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょう Text PlusN EL	RyoTextPlusN-ExtraLight	Adobe-Japan1-3 + 144	和文文字もじモジ
りょう Text PlusN L	RyoTextPlusN-Light	Adobe-Japan1-3 + 144	和文文字もじモジ
りょう Text PlusN R	RyoTextPlusN-Regular	Adobe-Japan1-3 + 144	和文文字もじモジ
りょう Text PlusN M	RyoTextPlusN-Medium	Adobe-Japan1-3 + 144	和文文字もじモジ
りょう Disp PlusN M	RyoDispPlusN-Medium	Adobe-Japan1-3 + 144	和文文字もじモジ
りょう Disp PlusN SB	RyoDispPlusN-SemiBold	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょう Disp PlusN B	RyoDispPlusN-Bold	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょう Disp PlusN EB	RyoDispPlusN-ExtraBold	Adobe-Japan1-3 + 144	**和文文字もじモジ**
りょう Disp PlusN H	RyoDispPlusN-Heavy	Adobe-Japan1-3 + 144	**和文文字もじモジ**
Adobe 黑体 Std R	AdobeHeitiStd-Regular	Adobe-GB1-5	**中文简体字**
Adobe 宋体 Std L	AdobeSongStd-Lght	Adobe-GB1-5	中文简体字
Adobe 仿宋 Std R	AdobeFangsongStd-Regular	Adobe-GB1-5	中文简体字
Adobe 楷体 Std R	AdobeKaitiStd-Regular	Adobe-GB1-5	中文简体字
Adobe 明體 Std L	AdobeMingStd-Light	Adobe-CNS1-5	中文繁體字
Adobe 명조 Std M	AdobeMyungjoStd-Medium	Adobe-Korea1-2	한글과 漢字 샘플

a. These are the actual font names that appear in application font menus. All of Adobe Systems' OpenType CJKV fonts share the same menu name across operating systems, though in the past there were a handful that did not when implemented using a legacy font format. These are listed in Table 6-54.

Companies that develop PostScript CJKV fonts, either by licensing font data or designing font data themselves, are far too numerous to list here, though a significant sampling was provided very early in this chapter.

There have been projects that resulted in freely available PostScript CJKV fonts. One of these projects, at WadaLab of Tokyo University (東京大学 *tōkyō daigaku*), resulted in several JIS X 0208:1997 and JIS X 0212-1990 fonts.* Another such project, sponsored by Korea's Ministry of Culture and Tourism (문화 관광 부/文化觀光部 *munhwa gwang-wang bu*), resulted in a series of Korean fonts named "Munhwa" (문화/文化 *munhwa*, meaning "culture") that support a hangul-only subset of KS X 1001:2004, specifically 2,350 glyphs. I have since added proportional- and half-width Latin characters to these fonts, which makes them slightly more usable. I have also heard of a similar set of free outline fonts from Taiwan. I have built and made available CIDFonts—and sfnt-wrapped CIDFont and OpenType versions for some of them—from the freely available PostScript font data as time and resources have permitted.†

In any case, there has been much talk thus far about two common PostScript font formats suitable for building CJKV fonts: OCF and CID-keyed fonts. The sections that immediately follow will set the record straight about their origins and differences. Still later in this chapter, you will learn that OpenType trumps them both, but that CID-keyed fonts represent a path toward OpenType.

OCF fonts—a legacy font format

As mentioned earlier, OCF fonts represent the original composite fonts offered by Adobe Systems. The OCF specification was never published or otherwise disclosed, so nearly every major CJKV type foundry took it upon themselves to reverse engineer (er, uh, I meant creatively re-engineer) its format.

A typical Japanese OCF font offered by Adobe Systems is made up of nearly 100 separate files spread throughout four directories on a PostScript filesystem. There are three reasons for the large number of files:

- Adobe Systems' OCF fonts support a large number of character sets and encodings
- The character set and encoding information, although common across all OCF fonts offered by Adobe Systems, is duplicated for every font
- The file format itself is complex

The more OCF fonts you install onto a PostScript filesystem, the more of a headache file management can become. There is nothing inherently wrong or bad about OCF fonts, but their format did not prove itself to be suitably extensible. Work then began on a simpler file format specification: CID-keyed fonts.

* *http://gps.tanaka.ecc.u-tokyo.ac.jp/wadalabfont/*

† *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/samples/*

CID-keyed fonts

CID-keyed fonts represent the basic underlying technology used for building today's (and tomorrow's) PostScript CJKV fonts. CID stands for *Character ID*, and it is a character and glyph access type for PostScript. Technically speaking, a CID-keyed font is a Type 0 (composite) font with FMapType 9. A CID-keyed font can also be CIDFontType 0, 1, or 2, depending on the type of character descriptions that are included. CIDFontType 0 is by far the most common CID-keyed font.

There are two components that constitute a valid a CID-keyed font: a CIDFont resource that contains the glyph descriptions (outlines), along with other data necessary to properly render them, such as hinting information; and one or more *Character Map* (CMap) resources that are used to establish character-code to CID mappings. The CIDs are ultimately used to index into the CIDFont resource for retrieving glyph descriptions.

A valid CID-keyed font instance consists of a CIDFont plus CMap concatenated using one or two hyphens.* Table 6-11 lists some CIDFonts and CMaps, along with the corresponding valid CID-keyed fonts.

Table 6-11. CIDFont and CMap resources versus CID-keyed fonts

CIDFont resource	CMap resource	CID-keyed font
Munhwa-Regular	KSC-H	Munhwa-Regular--KSC-H
HeiseiMin-W3	H	HeiseiMin-W3-H
MSung-Light	CNS-EUC-V	MSung-Light--CNS-EUC-V

When is it appropriate to use one versus two hyphens as glue? For CID-keyed fonts that were once available as OCF fonts, such as HeiseiMin-W3-H (that is, the "HeiseiMin-W3" CIDFont resource name plus "-" plus the "H" CMap resource name), a single hyphen is recommended for compatibility, because the OCF font machinery does not understand two hyphens, and will fail to execute, or "find," the font. Otherwise, two consecutive hyphens are recommended.

I intentionally did not label CID-keyed fonts as a legacy font format. This is because the CIDFont and CMap resources for CID-keyed fonts still serve as fundamental source material for building OpenType CJKV fonts, which is a topic that is fully covered later in this chapter. Although end-user use of CID-keyed fonts is effectively deprecated and no longer encouraged, font developers are still given the option to develop CIDFont and CMap resources in order to more easily build OpenType fonts.

The best way to learn how to use CID-keyed fonts in a conforming manner is to read *CID Font Tutorial* (Adobe Systems Technical Note #5643).

* Two hyphens are recommended because it explicitly tells PostScript and other clients what portions represent the CIDFont and CMap resources. For some CID-capable PostScript clone implementations, two hyphens are required.

The CIDFont resource

The CIDFont resource, which is a container for the character descriptions, assigns a unique character ID (CID) for every glyph. This CID is independent of any normal encoding and is simply an enumeration beginning at zero. Table 6-12 provides an example of a few CIDs, along with a graphic representation of the characters that are associated with them (examples taken from the Adobe-Japan1-6 character collection).

Table 6-12. Adobe-Japan1-6 CIDs and their graphic representations—samples

CID	Glyph
0	*.notdef*
1	(proportional-width space)
…	(thousands of CIDs omitted)
7474	堯
7475	槇
7476	遙
7477	瑤
…	(hundreds of CIDs omitted)
8284	凜
8285	熙
…	(thousands of CIDs omitted)
23057	鑾

The CIDFont resource is constructed from two basic parts, each of which is outlined below, along with a brief description:

A header
: Provides global font attributes, such as the font name, number of CIDs, private dictionaries, and so on.

A binary portion
: Contains offsets to subroutines, the subroutines themselves, offsets to charstrings, and the charstrings themselves.

While the header of a CIDFont resource is simply PostScript-like ASCII text, the remainder is binary data.

The CMap resource

The information that associates encoded values with CIDs is defined in the CMap resources. In most cases, or in a perfect world, an encoding range is associated with a CID

range. The following are example lines taken from a standard Adobe-Japan1-6 CMap resource, named simply "H," which specifies ISO-2022-JP encoding:

```
18 begincidrange
… (16 CID ranges omitted)
<7421> <7424> 7474
<7425> <7426> 8284
endcidrange
```

The convention is to use hexadecimal values enclosed in arrow brackets for character codes, and to use decimal values, not enclosed, for CIDs. Carefully note how the encoding ranges are associated with a range of CIDs. The following describes how it works: the encoding range is specified by two encoded values, one for each end of the encoding range—sometimes, noncontiguous encoding or CID ranges make one-to-one mappings necessary. In the preceding case, the two ranges are <74 21> through <74 24> and <74 25> through <74 26>. The CID associated with each range indicates the starting point from which encoded positions are associated with CIDs. For example, the two ranges just listed result in the associations between encoded values and CIDs, as described in Table 6-13.

Table 6-13. Encoded values versus CIDs

Encoded value	CID
<7421>	7474
<7422>	7475
<7423>	7476
<7424>	7477
<7425>	8284
<7426>	8285

Note how only two lines of a CMap resource can be used to associate many code points with their corresponding glyphs, expressed as CIDs. In the case of a complete row of characters, such as the range <30 21> through <30 7E> (94 code points), the following single line can be used:

```
<3021> <307e> 1125
```

For this example, the encoding range <30 21> through <30 7E> is mapped to CIDs 1125 through 1218.

The ordering of the glyphs in character collections generally favors a specific character set, and thus adheres to its ordering. The glyphs of the Adobe-Japan1-6 character collection that correspond to the JIS X 0208:1997 character set are ordered accordingly. This is why it is possible to map large contiguous encoding ranges to correspondingly large contiguous ranges of CIDs. When these same six CIDs are mapped from their corresponding

Unicode code points, the following lines are the result, using the UniJIS2004-UTF32-H CMap resource as the example:

```
6 begincidchar
<000051DC> 8284
<0000582F> 7474
<000069C7> 7475
<00007199> 8285
<00007464> 7477
<00009059> 7476
endcidchar
```

There are two very important things to note in the preceding example CMap lines:

- The character codes are necessarily in ascending order, but because the Unicode order is significantly different than that set forth in the JIS X 0208:1997 character set, the CIDs to which they map are no longer in ascending order. There is absolutely nothing wrong with this.
- The character codes and CIDs are no longer in contiguous ranges, which necessitates the use of the `begincidchar` and `endcidchar` operators that are used to specify single mappings. The `begincidrange` and `endcidrange` operators are appropriate only when character codes and their corresponding CIDs are in contiguous ranges.

The following is an example CMap resource, named UniCNS-UTF32-H, that maps a single Unicode code point, specifically U+4E00 expressed in UTF-32BE encoding, to its corresponding glyph in the Adobe-CNS1-5 character collection, specifically CID+595:

```
%!PS-Adobe-3.0 Resource-CMap
%%DocumentNeededResources: ProcSet (CIDInit)
%%IncludeResource: ProcSet (CIDInit)
%%BeginResource: CMap (UniCNS-UTF32-H)
%%Title: (UniCNS-UTF32-H Adobe CNS1 5)
%%Version: 1.006
%%EndComments
/CIDInit /ProcSet findresource begin
12 dict begin

begincmap

/CIDSystemInfo 3 dict dup begin
  /Registry (Adobe) def
  /Ordering (CNS1) def
  /Supplement 5 def
end def

/CMapName /UniCNS-UTF32-H def
/CMapVersion 1.006 def
/CMapType 1 def

/XUID [1 10 25593] def

/WMode 0 def
```

```
1 begincodespacerange
<00000000> <0010FFFF>
endcodespacerange

1 beginnotdefrange
<00000000> <0000001f> 1
endnotdefrange

1 begincidchar
<00004e00> 595
endcidchar
endcmap
CMapName currentdict /CMap defineresource pop
end
end

%%EndResource
%%EOF
```

I have emboldened the sections that define the encoding space and the character code to CID mappings themselves. Of course, the actual UniCNS-UTF32-H CMap resource includes thousands of mappings. More detailed information about how to build your own CMap resources is available in *Building CMap Files for CID-Keyed Fonts* (Adobe Systems Technical Note #5099).

These sections provided merely a taste of what the CID-keyed fonts offer. This technology was specifically designed so that font developers could make CJKV fonts much smaller, and more easily and efficiently than ever before. CID-keyed fonts are also portable across platforms, at least in the past. For Mac OS, ATM version 3.5J or later supported CID-keyed fonts as did ATM version 3.8 (non-Japanese) or later. For Windows, but not Windows NT or later, ATM version 3.2J (but not version 4.0!) supported CID-keyed fonts. Contemporary versions of these OSes, meaning Mac OS X and Windows (2000, XP, and Vista), support OpenType fonts, which can be built from CID-keyed font sources. Perhaps more importantly, CIDFont and CMap resources represent critical raw materials for building OpenType fonts. Any effort in building CIDFont and CMap resources is not wasted. OpenType, covered later in this chapter, represents a truly cross-platform font format.

The document entitled *Adobe CMap and CIDFont Files Specification* (Adobe Systems Technical Note #5014) describes CID-keyed fonts in more detail and is considered to be the engineering specification needed by font developers. A more gentle introduction is available in the document entitle *CID-Keyed Font Technology Overview* (Adobe Systems Technical Note #5092). Adobe Systems Technical Note #5213 (a PostScript version 2016 supplement) should also be of interest to developers because it more fully describes the various CIDFontTypes. If you are a font developer, I strongly encourage you to request a copy of the CID SDK (*CID-keyed Font Technology Software Development Kit*) from the Adobe Developers Association, which is delivered on a single CD-ROM. The CID SDK includes all the documentation and software necessary to implement CID-keyed font technology, including sample CIDFonts.

sfnt-wrapped CIDFonts—a legacy font format

A past twist to CID-keyed font technology was referred to as sfnt-wrapped CIDFonts. Font geeks and nerds alike are fully aware that TrueType fonts, at least those for Mac OS, resided within what is known as an 'sfnt' (*scalable font*) resource. But, weren't CID-keyed fonts the best thing since sliced bread? Well, sure they were, but….

CID-keyed font technology provides an extremely flexible mechanism for supporting large character sets and multiple encodings, but lacks host-based support such as user-selected and context-sensitive glyph substitution, alternate metrics (such as half-width symbols and punctuation, proportional kana, and even proportional ideographs), and easy vertical substitution. These types of advanced typographic features will be covered in greater detail when we get to Chapter 7.

An sfnt-wrapped CIDFont looks, smells, and behaves as though it were a TrueType—actually, AAT/QuickDraw GX—font, but instead of using TrueType outlines for its glyphs in the 'glyf' table, there is a CIDFont resource there instead in the 'CID' table that was established by Adobe Systems. Basically, the CIDFont file itself becomes one of the many tables within the 'sfnt' resource. Table 6-14 lists the additional 'sfnt' tables that are present in an sfnt-wrapped CIDFont.

Table 6-14. Additional 'sfnt' tables in sfnt-wrapped CIDFonts

Table Tag[a]	Description
ALMX	Alternate metrics
BBOX	Font bounding box
CID[b]	CIDFont resource
COFN	AAT font name for each component in a rearranged font
COSP	Code space for a rearranged font
FNAM	For AAT compatibility
HFMX	Horizontal font metrics
PNAM	Fully qualified PostScript font name
ROTA	Character rotation
SUBS	Subset definition
VFMX	Vertical font metrics
WDTH	Set widths

a. Note that all of these table tags are uppercase. This was intentional because only Apple is allowed to use all-lowercase table tags.

b. Note that this tag, like the others, consists of four characters: the three letters "CID" followed by a single space.

Note that the use of all-lowercase table tags is reserved by Apple, which explains why those table tags listed in Table 6-14 are entirely uppercase. The only exception to this

policy is the 'gasp' table that was defined by Microsoft, which should have included at least one uppercase letter. Gasp!

In addition, sfnt-wrapped CIDFonts typically include the 'bdat', 'bloc', 'cmap', 'feat', 'mort', 'name', 'post', and 'subs' tables, all of which are listed or described later in this chapter in the section entitled "AAT—formerly QuickDraw GX." The TrueType 'cmap' table in sfnt-wrapped CIDFonts functions in a way comparable to the CMap resource for CID-keyed fonts.

The first applications to recognize the advanced typographic features provided in sfnt-wrapped CIDFonts were Adobe Illustrator version 5.5J and Adobe PageMaker version 6.0J—both for Mac OS. Macromedia FreeHand 8.0J and higher also recognized these fonts' advanced typographic features. AAT applications could also recognize and use these fonts, as long as ATM version 3.9 or later was installed and enabled. More information on sfnt-wrapped CIDFonts can be found in *CID-Keyed sfnt Font File Format for the Macintosh* (Adobe Systems Technical Note #5180).

Now, it is clearly a better practice to build OpenType fonts. The good news is that the same CIDFont resource that is used to build an sfnt-wrapped CIDFont can be used to build an equivalent OpenType font. In fact, there are much better tools available for building OpenType fonts than for building sfnt-wrapped CIDFonts. This makes perfect sense considering the obsolete and deprecated nature of sfnt-wrapped CIDFonts.

Multiple master—a pseudo-legacy font format

Multiple master technology is an extension to the Type 1 Font Format described earlier in this chapter. This technology allows for the dynamic interpolation of a typeface's attributes, such as width, weight, optical size, and style. Multiple master fonts are a big design effort and require that many master outlines be made for each character. Exactly how many master outlines there are generally depends on the number of design axes. Table 6-15 illustrates the usual or typical relationship between design axes and master outlines. Although it may appear that the number of design axes determines the number of master outlines, that is not the case. The number of master outlines must be greater than or equal to the number of axes plus one. For example, Adobe Jenson and Kepler are multiple master fonts that do not follow the "power of two" model.

Table 6-15. Multiple master design axes and number of corresponding master outlines

Number of design axes	Number of master outlines
1	2
2	4
3	8
4	16

These master outlines are interpolated to produce a particular font instance. Think of a single axis where one end contains a light version of a character, and the other end contains a bold version of the same character. Now traverse along the axis. The closer you get to the end containing the bold version, the bolder the character becomes. Now imagine doing this with four axes!

Let's take a look at a two-axis multiple master character. Table 6-16 illustrates the four master designs of the letter "A." The two axes are for weight and width.

Table 6-16. Sample character for a two-axis multiple master font—Myriad Pro

Weight	Condensed	SemiExtended
Light	A	A
Black	**A**	**A**

The number of axes relates to the number of dimensions. For example, a typeface with a single design axis is represented by a straight line, and a line needs two points to be defined. Extend this all the way up to four design axes, at which time you have defined a hypercube (four dimensions). Needless to say, designing a multiple master font is a time-consuming task, but clearly possible. For those of you who are so inclined, the latest versions of Fontographer and FontLab Studio allow designers to create multiple master fonts. Fontographer is currently limited to two-axis designs, whereas FontLab supports up to four-axis designs.

These techniques can also be applied to a CJKV font. Table 6-17 illustrates several intermediate instances of a kanji from a real-world, single-axis, multiple master font—the axis is for weight. The two master designs for ExtraLight and Heavy are located at each extreme.

Table 6-17. Sample kanji for a one-axis multiple master font—Kozuka Mincho Pr6N

ExtraLight	Light	Regular	Medium	Bold	Heavy
蛇	蛇	蛇	蛇	蛇	蛇

Unfortunately, there are no genuine multiple master CJKV fonts available as of this writing. The design used for Table 6-17 originated as multiple master, but is implemented as individual fonts for each of its six weights. The design of a CJKV font is itself a large project—designing a multiple master CJKV font is much more work!

In terms of fonts that are used by OSes and applications and are sold, multiple master is effectively dead, primarily because the OpenType specification was never enhanced to support such fonts. Adobe Systems ceased selling and supporting multiple master fonts. Although CFF, described earlier in this chapter, allows up to 16 master outlines and 15 design axes, multiple master is simply not supported in the context of OpenType fonts.

Still, multiple master continues to be an extremely effective technology for assisting the typeface design process. Multiple weights are easily generated through interpolation and snapshotting techniques. In fact, the vast majority of Adobe Originals are designed through the use of multiple master, though the font instances that are released—in multiple weights, and in some cases, multiple widths—no longer carry the multiple master baggage. This is why the title of this section states that multiple master is a pseudo-legacy font format. It is legacy format in terms of delivering multiple master fonts to users, but it is current font format in terms of its advantages during the typeface design process.

How can I accelerate PostScript fonts?

The publishing workflow has changed from one that made extensive use of or required printer-resident fonts to one that uses PDF to encapsulate all of the information to display and print documents, from simple to incredible complex, including the embedding of the glyphs. Naturally, the way in which the performance of fonts can increase has necessarily changed.

In the past, in the days of printer-resident fonts, there were two primary ways to accelerate the speed at which Type 1 fonts—or font formats that use Type 1 charstrings, such as Adobe Systems' CID-keyed fonts—print, both of which are hardware-dependent:*

- Purchase a PostScript printer whose fonts are stored in ROM (as opposed to being resident on a hard disk, internal or external).
- Purchase a PostScript printer that is equipped with the Adobe Type 1 Coprocessor (such printers are typically bundled with fonts in ROM).

Fonts that are stored in printer ROM are considerably faster than those resident on a hard disk—reading from silicon chips is inherently faster than reading from a disk drive.

The Adobe Type 1 Coprocessor (T1C), also known as Typhoon, was an ASIC (*Application Specific Integrated Circuit*) developed by Adobe Systems for use under license by their PostScript OEMs (*Original Equipment Manufacturers*). This chip was essentially a hardware version of Adobe Systems' ATM renderer, which significantly accelerated the rasterization of Type 1 fonts. It was developed primarily to improve the performance of PostScript output devices that support CJKV fonts by reducing the time it takes to process their complex character descriptions.

* As far as host-based (that is, using Adobe Type Manager software) rasterization was concerned, simply using a faster computer usually did the trick.

The first commercial products to support the Adobe Type 1 Coprocessor were Oki Electric's series of PostScript Level 2 Japanese printers: the ML800PSII LT, ML801PSII, and ML801PSII+F. They were introduced to the market in January 1993. Other printer manufacturers have also shipped products that include the Adobe Type 1 Coprocessor. It is important to note that T1C is no longer being developed, and as a result does not support Type 2 charstrings as used by CFF.*

Now, given the current PDF-based document workflow that eliminates the need for printer-resident fonts, accelerating the performance of PostScript fonts or of fonts in general can be accomplished through the use of a faster computer or a faster network connection.

TrueType Font Formats

TrueType fonts, like PostScript fonts, are described mathematically as outlines, and are therefore fully scalable. TrueType curves are represented as quadratic splines. TrueType fonts are able to reside on PostScript printer hard disks, or in printer RAM or ROM, because the Type 42 font format defines a PostScript wrapper for TrueType fonts.

Many TrueType fonts are also available in Type 1 format—many type vendors market their fonts in both formats to appeal to those dedicated to a particular format.

TrueType CJKV fonts are a more recent addition to TrueType font technology, and are currently usable on Mac OS and Windows. In fact, Mac OS-J was bundled with seven TrueType Japanese fonts. Apple's Language Kits (CLK, JLK, and KLK) also came bundled with TrueType CJKV fonts—but not nearly as many as were bundled with the fully localized versions of the OS. These fonts required approximately 20% to 40% more disk space than the equivalent PostScript CJKV fonts.

A TrueType font is contained within a file resource known as 'sfnt' (an abbreviated form of Scalable Font) and consists of many tables. The required and optional tables are listed and described in Table 6-18.

Table 6-18. Standard TrueType tables

Table tag	Description	Required table?
cmap[a]	Character to glyph mapping	Yes
glyf	Glyph data	Yes
head	Font header	Yes
hhea	Horizontal header	Yes
hmtx	Horizontal metrics	Yes
loca	Index to location	Yes

* This is a polite way of stating that T1C is more or less dead.

Table 6-18. Standard TrueType tables

Table tag	Description	Required table?
maxp	Maximum profile	Yes
name	Naming table	Yes
post	PostScript information	Yes
OS/2	OS/2- and Windows-specific metrics	Yes
cvt	Control Value Table	No
EBDT	Embedded bitmap data	No
EBLC	Embedded bitmap location data	No
EBSC	Embedded bitmap scaling data	No
fpgm	Font program	No
gasp	Grid-fitting and scan conversion procedure—grayscale	No
hdmx	Horizontal device metrics	No
kern	Kerning data	No
LTSH	Linear threshold table	No
prep	CVT program	No
PCLT	PCL5	No
VDMX	Vertical device metrics table	No
vhea	Vertical metrics header	No
vmtx	Vertical metrics	No

a. This is different from Adobe Systems' CMap, which is a resource used for CID-keyed fonts and is instantiated as a file. This is a case when "case" can make a big difference.

It is important to realize that many TrueType developments took place at Apple and Microsoft independently, leading to differences between the two formats.

More detailed information about the TrueType font format can be found in Microsoft's *TrueType 1.0 Font Files* document.*

TrueType Collections

An interesting enhancement to the TrueType font format is the concept of *TrueType Collections* (TTCs). This allows TrueType fonts to efficiently share data across similar fonts. In short, TTCs effectively include multiple TrueType fonts within a single physical file whereby many of the glyphs are shared by the individual font instances.

* *http://www.microsoft.com/truetype/*

The following lists some of the ways in which TTCs can be (and, more importantly, have been) used in the context of CJKV fonts:

- A complete Japanese font, along with its matching kana designs (for example, five matching kana designs are associated with each of TypeBank's Japanese typeface designs)
- Two or more Korean fonts sharing a single hanja design (not all Korean fonts include glyphs for hanja, because they are treated somewhat generically)

Of course, there are many other possibilities and uses for TTCs.

TTCs are implemented by including all characters (glyphs) in the 'glyf' table. But, there need to be multiple 'cmap' tables, one for each font that will appear in applications' font menus. These multiple 'cmap' tables refer to a different subset of the glyphs found in the 'glyf' table. This process of creating a TTC is nontrivial, and it requires very precise mappings in the 'cmap' table—each 'cmap' subtable must refer to the appropriate subset of glyphs in the 'glyf' table.

Virtually all TrueType fonts bundled with the various localized versions of Windows Vista are TrueType Collection fonts.

TTC files can be identified by their filename extension *ttc*, as opposed to *ttf* for standard TrueType fonts. TrueType Collections are described in Microsoft's *TrueType 1.0 Font Files* document.

AAT—formerly QuickDraw GX

Apple was the first company to raise the proverbial "bar" with respect to providing advanced typographic features as a component or components of the fonts themselves—their solution was originally called QuickDraw GX, also referred to as TrueType GX in some circles. However, the font portion of QuickDraw GX functionality became known as *Apple Advanced Typography* (AAT), which was used in conjunction with *Apple Type Services for Unicode Imaging* (ATSUI).*

Table 6-19 lists some of the TrueType tables that Apple has created on its own for developing AAT fonts—meaning that you will not find these tables in TrueType fonts that run under Windows. Some of these tables are specific to AAT, and some are not, as noted in the table.

* When treated as a Japanese transliteration, the acronym ATSUI means "hot" (熱い *atsui*) or "thick" (厚い *atsui*).

Table 6-19. AAT tables

Table tag	Description	AAT-specific?
bdat[a]	Bitmapped data	No
bloc[b]	Bitmapped data locations (offsets)	No
bsln	Baseline adjustment information	Yes
fdsc	Font descriptor	No
feat	QuickDraw GX feature list	Yes
mort	QuickDraw GX features	Yes
morx	Extended 'mort' table	Yes
trak	Tracking information	No

a. Microsoft's TrueType fonts may include an 'EBDT' table for the same purpose—see Table 6-18.

b. Microsoft's TrueType fonts may include an 'EBLC' table for the same purpose—see Table 6-18.

More information on AAT is still available in publications from Apple, such as *QuickDraw GX Font Formats: The TrueType Font Format Specification*, *Inside Macintosh—QuickDraw GX Typography*, and *Inside Macintosh—Text*. The latest AAT developments are available at the Apple website.*

The use of AAT fonts has diminished, which is clearly the result of the wide adaptation and acceptance of OpenType fonts, whose format is covered later in this chapter.

TrueType Open

TrueType Open was intended to be Microsoft's answer to Apple's AAT, and its description is included here primarily for historical purposes. Although AAT and TrueType Open attempt to solve the same typographic problems, their approaches were a bit different. TrueType Open fonts are still considered valid TrueType fonts and are fully compatible with existing applications. But, some applications may not be able to access the tables specific to TrueType Open.

TrueType Open defines five additional tables, each of which is indicated and briefly described in Table 6-20. These five tables became part of the OpenType specification, which is described in the section that follows.

Table 6-20. TrueType Open tables

Table tag	Table name	Description
GSUB[a]	Glyph SUBstitution	Substitute glyphs: one to one, one to *n*, one from *n*, or *n* to one
GPOS	Glyph POSitioning	Specific position of glyphs

* *http://developer.apple.com/textfonts/*

Table 6-20. TrueType Open tables

Table tag	Table name	Description
BASE	BASEline	Baseline adjustment information—useful when mixing different scripts, such as Chinese and Latin characters
JSTF	JuSTiFication	Provides additional control over glyph substitution and positioning in justified text—affects spacing
GDEF	Glyph DEFinition	Classifies the font's glyphs, identifies attachment points, and provides positioning data for ligature carets

a. Apple's AAT fonts may include a 'mort' or 'morx' table for the same basic purpose—see Table 6-19.

The 'GPOS' table benefits scripts such as romanized Vietnamese in which the exact positioning of accents is crucial. One of the most powerful tables of TrueType Open is 'GSUB', which is functionally similar to AAT's 'mort' and 'morx' tables. The 'GPOS' and 'GSUB' tables will be discussed in greater depth later in this chapter when we learn about OpenType fonts.

Clearly, building a fully functional TrueType Open font provides a very powerful mechanism for delivering advanced typographic features to applications. Microsoft has published the TrueType Open specification in the *TrueType Open Font Specification* document.* It is designed to be an open specification, hence its name, meaning that developers can create their own typographic features.

As you will learn in the next section, most of what TrueType Open offers effectively became OpenType. This is a good thing, because OpenType effectively ended the so-called font wars, by allowing PostScript and TrueType outlines to be equally represented in the same font format.

OpenType—PostScript and TrueType in Harmony

Finally. The day has arrived. Well, to be accurate, the day arrived several years ago, but it has taken those several years for the infrastructure to mature. In any case, PostScript and TrueType have merged into a single standard called OpenType, sometimes abbreviated as OTF.† This effectively means that it will make absolutely no difference whether the underlying glyphs descriptions, which are sometimes referred to as charstrings, are PostScript Type 1 or Type 2 (though the latter is the more common), or TrueType. Adobe Systems and Microsoft jointly announced this font format in 1996, and its complete specification is available online.‡ Adobe Systems, Apple, Microsoft, and other companies provide tools that enable font developers to more easily build and test OpenType fonts.

* *http://www.microsoft.com/truetype/*

† The three-character filename extension for OpenType fonts is *otf*.

‡ *http://www.microsoft.com/opentype/otspec/*

Compared to legacy font formats, OpenType has several key advantages, the vast majority of which can be enumerated as follows:

- Cross-platform and supported natively by the major OSes
- Completely self-contained
- Default encoding is Unicode
- Supports TrueType or PostScript outlines
- Improved encapsulation and specification of font-level and glyph metrics, which affects vertical writing and the treatment of ideographs
- Can include advanced typographic features, such as glyph substitution

OpenType fonts are supported natively under Windows 2000 and later, but are supported in Windows 95, Windows 98, Windows NT (version 4.0), and Mac OS (versions 8.6 through 9.2 and Classic) through the use of *Adobe Type Manager* (ATM).* Windows 2000, XP, and Vista, along with Mac OS X, support OpenType fonts without the need for ATM. Because of their native status under Windows 2000 and later, OpenType fonts are encoded according to Unicode. That is, they include a Unicode-encoded 'cmap' table.

For all practical purposes, OpenType is an extension to TrueType Open, which means that all of its functionality, such as the ability to define advanced typographic features (metrics and glyph substitutions), is available to OpenType fonts.

Up until now, all legacy font formats, such as the Mac OS 'FOND' resource and the Windows PFM file, did not include information that specified values exclusively for handling vertical writing. Font developers and applications were forced to overload the semantics of available fields, such as those that represent font-level ascent and descent. OpenType provides an elegant solution by including dedicated fields that give applications correct and unambiguous information for vertical writing. The top and bottom of the design space, which are critical for establishing reference points for proper vertical writing, are stored in the 'OS/2' table's *sTypoAscender* and *sTypoDescender* fields. The corresponding fields in the 'hhea' table, specifically *Ascender* and *Descender*, should be set to the same values.

Table 6-21 lists the tables that are absolutely required for an OpenType font to function properly, along with other tables that are for other purposes or specific to some font categories.

* More specifically, ATM version 4.1 or later for Windows, and ATM version 4.5 or later for Mac OS.

Table 6-21. OpenType tables and their purposes

Purpose	Table tags
Required	cmap, head, hhea, hmtx, maxp, name, OS/2, post
For TrueType	cvt, fpgm, glyf, loca, prep
For PostScript	CFF, VORG
For bitmaps	EBDT, EBLC, EBSC
Advanced typography	BASE, GDEF, GPOS, GSUB, JSTF
Vertical	vhea, vmtx
Other	DSIG, gasp, hdmx, kern, LTSH, PCLT, VDMX

OpenType fonts that support vertical writing should be sure to include the 'vhea' and 'vmtx' tables, and should additionally support the 'vert' or 'vrt2' GSUB features for allowing applications to access the vertical variants for glyphs that require them. This effectively means that typical OpenType CJKV fonts should include these two tables, along with the 'vert' and 'vrt2' GSUB features. For those with a 'CFF' table, the 'VORG' table should also be included as it supplements the data that is in the 'vmtx' table.

The overall version number of an OpenType font is specified in its *head.fontRevision* field, meaning the *fontRevision* field of the 'head' table. This version number may be reflected elsewhere in the font, such as part of the *name.ID=5* string, but it is the setting in the 'head' table that should ultimately be trusted. On Mac OS X and Windows, simply double-clicking an OpenType font will either bring up a dialog that displays the version number, or it will launch an application that can be used to display the version number.

OpenType GSUB features

The 'GSUB' table of OpenType is inherited from TrueType Open. There is a very high number of registered OpenType GSUB (and GPOS) features.* But, in the context of CJKV fonts, a much smaller number of GSUB features are applicable.

Table 6-22 lists the four basic types of glyph substitution that are possible through the use of GSUB features, along with examples of their use, the majority of which are tied to specific GSUB features.

Table 6-22. OpenType Glyph SUBstitution types

Substitution type	Example	
One-to-one substitution	Vertical substitution—*vert*	〜 ➡ ﴾
One-to-*n* substitution	Ligature decomposition	㌔ ➡ キログラム

* *http://www.microsoft.com/typography/otspec/featuretags.htm*

Table 6-22. OpenType Glyph SUBstitution types

Substitution type	Example	
One-from-*n* substitution	Glyph variants—*aalt*	辺 ➡ 邊, 邊, 邊, 邊, 邊, 邊, 邊, 邊, 邉, 邉, 邉, 邉, 邉, 邉, 邉, 邉, 邉, 邉, 邉, or 邉
n-to-one substitution	Ligature construction—*dlig*	キログラム ➡ ㌔

These four glyph substitution types differ in the number of entry and exit GIDs that they specify. The simplest, of course, is the one-to-one substitution, which specifies exactly one entry GID and results in exactly one exit GID. The GSUB features that are most applicable to CJKV fonts are listed in Table 6-23.

Table 6-23. GSUB feature tags for OpenType CJKV fonts

GSUB feature tag	Full name and description	Applicable locales
aalt	Access All Alternates	all
afrc	Alternative Fractions	all
ccmp	Glyph Composition/Decomposition	all
cpct	Centered CJK Punctuation	Chinese
dlig	Discretionary Ligatures	all
expt	Expert Forms	Japanese
frac	Fractions	all
fwid	Full Widths	all
hkna	Horizontal Kana Alternates	Japanese
hngl	Hangul	Korean
hojo	Hojo Kanji Forms—JIS X 0212-1990 prototypical glyphs	Japanese
hwid	Half Widths	all
ital	Italic	all
jp78	JIS78 Forms—JIS C 6226-1978 prototypical glyphs	Japanese
jp83	JIS83 Forms—JIS X 0208-1983 prototypical glyphs	Japanese
jp90	JIS90 Forms—JIS X 0208-1990 prototypical glyphs	Japanese
jp04	JIS2004 Forms—JIS X 0213:2004 prototypical glyphs	Japanese
liga	Standard Ligatures	all
locl	Localized Forms	all
nalt	Alternate Annotation Forms	all
nlck	NLC Kanji Forms	Japanese

Table 6-23. GSUB feature tags for OpenType CJKV fonts

GSUB feature tag	Full name and description	Applicable locales
pkna	Proportional Kana	Japanese
pwid	Proportional Widths	all
qwid	Quarter Widths	all
ruby	Ruby Notation Forms	Japanese
smpl	Simplified Forms	Chinese, Japanese
trad	Traditional Forms	Chinese, Japanese
twid	Third Widths	all
vert	Vertical Writing	all
vkna	Vertical Kana Alternates	Japanese
vrt2	Vertical Alternates and Rotation	all

Some GSUB features are expected to be turned on or enabled by default in the applications that use them. The 'ccmp' and 'liga' GSUB features are such examples. Another example is the use of the 'vert' or 'vrt2' GSUB features, one of which should be turned on by default when the writing mode is vertical.

OpenType GPOS features

The 'GPOS' table of an OpenType font specifies features that alter the metrics, in terms of set widths and positioning, of the glyphs to which they refer.

The GPOS features that are most applicable to CJKV fonts are listed in Table 6-24. Be aware that the first three GPOS features are intended for horizontal writing, and that the last three are vertical counterparts of these features, intended for vertical writing.

Table 6-24. GPOS feature tags for OpenType CJKV fonts

GPOS feature tag	Full name and description	Applicable locales
halt	Alternate Half Widths—make full-width forms half-width	all
kern	Kerning	all
palt	Proportional Alternate Widths—make full-width forms proportional	all
vhal	Alternate Vertical Half Metrics—vertical version of 'halt'	all
vkrn	Vertical Kerning—vertical version of 'kern'	all
vpal	Proportional Alternate Vertical Metrics—vertical version of 'palt'	all

Although there is a separate 'kern' table, kerning is best implemented through the use of the 'kern' and 'vkrn' GPOS features.

Other ways to affect the glyph metrics in OpenType fonts can be made via other tables, the most notable of which is the 'vmtx' table. Almost all CJKV character set standards include full-width Latin, Greek, and Cyrillic characters, and the fonts that are based on these same character set standards include full-width glyphs that correspond to the characters. When in horizontal writing mode, these full-width glyphs are expected to use a Latin baseline. But, if the same glyphs, positioned along the Y-axis such that they rest on a Latin baseline, are used in vertical writing mode, the results are less than pleasing. One solution is to include separate glyphs for these characters, for horizontal and vertical writing. A better solution is to make adjustments to the position of these glyphs along the Y-axis, specifically to center them along the Y-axis. This means that the same glyphs can be used for horizontal and vertical writing. When these glyphs are used in horizontal writing mode, they rest on a Latin baseline, but when used in Vertical writing mode, they are centered along the Y-axis. The settings of the 'vmtx' table are considered default behavior.

OpenType 'cmap' tables

Although not specific to OpenType fonts, due to its TrueType heritage, the 'cmap' table is composed of subtables, each of which are tagged by format, platform ID, and language ID. Two types of Unicode encodings are supported through the various 'cmap' table formats, all of which are identified by even integers. Format 4 was originally intended to support UCS-2 encoding, but it is now best thought of as supporting BMP-only UTF-16 encoding. Format 12 was intended to support UCS-4 encoding, but it equally supports UTF-32 encoding thanks to the pure superset/subset relationship of UCS-4 and UTF-32 encodings. Those OpenType fonts that support IVSes will include a Format 14 subtable, in addition to a Format 4 or Format 12 subtable, or both.

Format 8 supports a mixed 16- and 32-bit encoding and is obviously intended to support UTF-16 encoding. However, it is a 'cmap' format that is not supported by Microsoft, and thus best considered to be deprecated.

Note that all 'cmap' table formats are tagged with an even integer. The first 'cmap' table format is 0 (zero), and the highest right now is 14. Thus, Formats 0, 2, 4, 6, 8, 10, 12, and 14 exist. Table 6-25 lists the current 'cmap' table formats, along with the types of encodings they support. Formats 8 and higher are intended to support 32-bit or 4-byte encodings.

Table 6-25. Current 'cmap' table formats

Format	Supported encodings	Examples
0	Single-byte—0x00 through 0xFF	Any single-byte encoding
2	Mixed one- and two-byte	EUC-CN, EUC-KR, GBK, Shift-JIS, and so on
4[a]	16-bit	UCS-2, BMP-only UTF-16
6	16-bit—single code range	UCS-2, BMP-only UTF-16
8	Mixed 16- and 32-bit	UTF-16

Table 6-25. Current 'cmap' table formats

Format	Supported encodings	Examples
10	32-bit—single code range	UTF-32
12[a]	32-bit	UTF-32
14	Unicode Variation Sequences	Ideographic Variation Sequences

a. If a Format 12 subtable is present, its contents must be a superset of the contents of the Format 4 subtable.

More detailed information about the 'cmap' table is available in the OpenType specification.

CMap versus 'cmap'

Font documentation may refer to CMap and 'cmap'. Poorly written font documentation will treat these as equivalent, and to some extent they are, because both map character codes to glyphs in a font. However, they are certainly not equivalent. Well-written documentation will clearly distinguish CMap from 'cmap'.

CMap refers to a PostScript resource and is used for CID-keyed fonts. A CMap resource maps character codes to CIDs. A CID is specific to CID-keyed fonts, and it is how glyphs in a CIDFont resource are identified and accessed.

'cmap' refers to an 'sfnt' table, and is used by OpenType, TrueType, TrueType Open, AAT, and sfnt-wrapped CIDFonts to map character codes to glyphs in the font. A 'cmap' table maps character codes to GIDs. The only exception to this are sfnt-wrapped CIDFonts in which the character codes specified in a 'cmap' table map to CIDs in the 'CID' table. As described in the previous section, a 'cmap' table can include one or more subtables.

The difference between CID and GID is also significant, and how to distinguish them is equally important. The topic of CID versus GID is covered later in this chapter.

OpenType versus TrueType, Type 1, and CID-keyed fonts

OpenType effectively bridges TrueType, Type 1, and CID-keyed fonts, which is a very good thing. For TrueType fonts, the presence of OpenType-specific tables, such as 'GPOS' or 'GSUB', is what distinguishes OpenType fonts from legacy TrueType fonts. To a great extent, not much needs to change in order for TrueType fonts to become genuine OpenType fonts.

For the PostScript font formats, specifically name-keyed (aka Type 1) and CID-keyed fonts, the 'CFF' table supports both of these. In fact, from a font development point of view, one still needs to build Type 1 fonts or CIDFont resources, which serve as the input for the tools that build OpenType fonts. In other words, font developers are still expected to build name-keyed Type 1 fonts or CIDFont resources, but the resulting files are no longer provided to end users as is, but become one of the many source files for building

an OpenType font. In addition, round-trip conversion between CFF and the legacy PostScript font formats is supported, and is part of the CFF specification.

Name-keyed versus CID-keyed

Type 1 and some TrueType fonts are name-keyed, meaning that each glyph is assigned a unique name. The name is a string with limitations and restrictions in terms of what characters can and cannot be used. The first character of the name cannot be a digit, but digits can be used elsewhere in the name. Upper- and lowercase letters can obviously be used, along with a small number of symbols. Spaces are not allowed. What is allowed is a subset of ASCII.

For Type 1 fonts intended to become the 'CFF' table of an OpenType font, there are useful naming conventions. Glyphs for common characters to some extent follow the Type 1 glyph names. Those outside that scope follow other conventions. For example, "u" or "uni" followed by the four or five uppercase hexadecimal digits that represent a Unicode scalar value is one convention. In fact, using such glyph names helps to drive the building of the 'cmap' table that maps the same Unicode scalar values to the glyphs in the 'CFF' table, at least when using the *MakeOTF* tool provided in AFDKO, which is described later in this chapter. AFDKO includes a file called *GlyphOrderAndAliasDB* that provides standard glyph names for name-keyed fonts, and that also serves to map obsolete and deprecated glyph names to the current glyph naming convention.

Name-keyed fonts are limited to one hint dictionary. This means that a single set of hinting parameters, such as alignment zones and stem widths, is applied to every glyph in the font.

CIDFont resources, of course, are CID-keyed. Each glyph is assigned a unique integer value. In accordance to the 64K glyph limit, the maximum number of CIDs is 65,536. Because CIDs begin at zero, the highest possible CID is thus 65535. When building OpenType fonts and using a CIDFont resource for the 'CFF' table, the specification of an appropriate CMap resource is what drives the building of the 'cmap' table. When a CIDFont resource is converted to a 'CFF' table, its CIDs become GIDs, but the mapping between GID and CID is retained.

CID-keyed fonts can include multiple hint dictionaries. Adobe Systems' Adobe-Japan1-6 fonts have 15 or 16 hint dictionaries. Sixteen hint dictionaries are used only when subroutine redistribution is required in order to ensure that the total number of global or local subroutines does not exceed the subroutine limit. Table 6-26 lists each of the 15 standard hint dictionaries used for Adobe-Japan1-6 fonts, along with the number of glyphs specified in each.

Table 6-26. The 15 hint dictionaries of the Adobe-Japan1-6 character collection

Hint dictionary name	Number of glyphs	Description
AlphaNum	1,700	Annotated alphanumeric glyphs
Alphabetic	223	Full-width Greek, Cyrillic, and Latin glyphs
Dingbats	1,053	Miscellaneous glyphs
DingbatsRot	38	Prerotated version of Dingbats
Generic	190	Half- and full-width generic glyphs
GenericRot	78	Prerotated version of Generic
HKana	179	Half-width kana glyphs
HKanaRot	179	Prerotated version of HKana
HRoman	175	Half-width Latin glyphs
HRomanRot	175	Prerotated version of HRoman
Kana	1,769	Kana glyphs, including annotated forms
Kanji[a]	14,943	Kanji glyphs, including annotated forms
Proportional	1,034	Proportional Latin glyphs
ProportionalRot	1,034	Prerotated version of Proportional
Ruby	288	Ruby glyphs

a. When 16 hint dictionaries are used, the Kanji hint dictionary is split into two hint dictionaries: Kanji and KanjiPlus. The Kanji hint dictionary includes 10,216 glyphs, and KanjiPlus includes 4,727 glyphs.

As Table 6-26 demonstrates, the hint dictionary arrangement for Adobe-Japan1-6 effectively separates or divides its glyphs according to glyph class. All of the kana glyphs, for example, are in the Kana hint dictionary. Note how the prerotated glyphs are in their own hint dictionaries, separated from their unrotated counterparts. This is done because the prerotated glyphs require different hinting parameters to be specified.

Of course, it is possible to build CIDFont resources that use a single hint dictionary, and for some fonts, this may be appropriate. Bear in mind that glyphs that do not rest on a baseline, such as kana, hangul, and ideographs, generally require a different set of alignment zones than those glyphs that do rest on a baseline, such as Latin, Greek, and Cyrillic glyphs.

Glyph Sets

Regardless of the font format, and regardless of whether the glyphs are stored as bitmapped patterns or as scalable outlines, every font is a collection of glyphs, typically based on the same typeface style. This collection is referred to as a *glyph set*. These glyph sets have no inherent encoding. The 'cmap' subtables of an OpenType or TrueType font, or CMap resources that are compatible with CIDFont resources, effectively apply an encoding by mapping character codes to the glyphs in a font.

Glyph sets necessarily evolve, usually due to new or expanded character set standards on which they are based. Sometimes, changes to or expansion of glyph sets is due to Taro's Law.*

Static Versus Dynamic Glyph Sets

Now that we have described today's most widely used font formats, it is clearly useful to write about the distinction between static and dynamic glyph sets. Today's fonts generally differ as to whether their glyph sets are static or dynamic, and can be described as follows:

A static glyph set
: Based on a standard, meaning that any font that is based on a static glyph set is predictable and stable. Static glyph sets are advantageous when building large volumes of fonts. The development and testing of such fonts is greatly simplified, in my opinion and experience.

A dynamic glyph set
: Allows font developers to add new glyphs on a whim.

TrueType fonts, by definition, are based on dynamic glyph sets. OpenType fonts can be either. OpenType fonts with name-keyed 'CFF' tables are based on dynamic glyph sets. Only CID-keyed fonts, including CID-keyed 'CFF' tables, are based on static glyph sets. The concept of "character collection," as used by CID-keyed fonts, is a real-world implementation of static glyph sets. Interestingly, and as you will learn in this chapter, it is possible to build OpenType fonts with CID-keyed 'CFF' tables that are based on dynamic glyph sets.

CID Versus GID

The glyphs in an OpenType or TrueType font are identified through the use of GIDs (*Glyph IDs*), and those in a CIDFont resource are identified through the use of CIDs (*Character IDs*). GIDs and CIDs thus perform similar functions, but are different in some contexts.

GIDs must be contiguous. The OpenType file format is based on the 'sfnt' file structure of TrueType fonts, and the dynamic nature of the glyph sets of TrueType fonts necessitated contiguous GIDs. Gaps or missing glyphs are not allowed.

CIDFont resources allow missing glyphs, which are correctly referred to as *empty intervals*. The static nature of the glyph sets of CID-keyed fonts necessitated the use of empty intervals in order to maintain the CID integrity of the static glyph sets on which they are based.

* If a character set is composed of characters whose prototypical glyphs need not and should not be changed, some of the prototypical glyphs will always be changed by the authorities.

For an OpenType font whose 'CFF' table is based on a CIDFont resource, meaning that the CIDFont resource is converted to CFF, the CIDs of the CIDFont resource are converted to GIDs in the resulting 'CFF' table. The real issue is whether the GIDs in the 'CFF' table correspond to the CIDs in the source CIDFont resource. As long as the source CIDFont resource does not include any empty intervals, GID=CID in the resulting OpenType font. If the source CIDFont resource includes any empty intervals, the point where GID≠CID is immediately after the first instance of an empty interval.

Consider a CIDFont resource that is based on the Adobe-Japan1-6 character collection, which defines 23,058 glyphs, from CID+0 through CID+23057. If we were to omit CIDs 15444 through 15448, the resulting CID ranges would be as follows:

0–15443
15449–23057

CIDs 15444 through 15448 would thus be treated as empty intervals. If this CIDFont resource were to be converted to a 'CFF' table, the resulting GID range would be as follows:

0–23053

When a CIDFont resource is converted to a 'CFF' table, the GID→CID mapping is preserved as part of the 'CFF' table. This means that a 'CFF' table can be converted back into a CIDFont resource as long as it is CID-keyed.

Std Versus Pro Designators

Today's fonts frequently use designators that serve to indicate the degree to which their glyph sets cover character set standards, or extend beyond them. For OpenType fonts, the two most common designators are Std and Pro. These two basic designators have variations, at least in the context of CJKV font implementations. These are listed and described in Table 6-27.

Table 6-27. Std and Pro designator variations—Japanese-specific examples

Designator	Base designator	Description
Std	Std	Adobe-Japan1-3
StdN	Std	JIS2004-savvy version of Std
Plus	Std	Equivalent to Std
PlusN	Std	JIS2004-savvy version of Plus
Pro	Pro	Adobe-Japan1-4
ProN	Pro	JIS2004-savvy version of Pro
Pr5	Pro	Adobe-Japan1-5
Pr5N	Pro	JIS2004-savvy version of Pr5

Table 6-27. Std and Pro designator variations—Japanese-specific examples

Designator	Base designator	Description
Pr6	Pro	Adobe-Japan1-6
Pr6N	Pro	JIS2004-savvy version of Pr6

The distinction between the Std and Pro designators, in terms of which is the most appropriate for a given glyph set or glyph complement, is not always easily understood, and requires some explanation and examples.

Some type foundries seem to use the sheer number of glyphs as the basis for distinguishing the use of these designators. Adobe Systems uses a more rigid set of criteria for making the distinction. In terms of Adobe Systems' CJKV fonts, if the glyph complement of a font more or less adheres to character set standards, the Std designator, or a variation thereof, is used. Although the Adobe-GB1-5 character collection, described in the next page or so, includes a rather stunning 30,284 glyphs, the Std designator is nonetheless used for fonts based on it. This is because the Adobe-GB1-5 character collection, more or less, includes only the glyphs that correspond to the characters that are printed in the GB 18030-2000 standard, along with the glyphs for the characters of one of its regional scripts, specifically Yi.

Adobe Systems uses the Pro designator, or a variation thereof, only if the glyph complement significantly extends beyond the characters found in character set standards, and thus includes glyphs that are intended to satisfy most professional or commercial publishing needs. The Adobe-Japan1-4 character collection represents the first glyph set that qualified for the Pro designator. Adobe Systems has not yet defined Chinese or Korean Pro character collections, though there are clearly plans to do so at some point.

In summary, I feel it is prudent to stress that the use of the Pro designator should be used sparingly, and the sheer number of glyphs may not be the best criteria for determining its use.

Glyph Sets for Transliteration and Romanization

Another difficulty that is often faced by users and developers alike is the availability of fonts that include the glyphs necessary for transliteration purposes, or for scripts that are based on Romanization. This may seem like a trivial issue, but I assure you that it is not. Up until OpenType Japanese Pro fonts were available, meaning a glyph set of Adobe-Japan1-4 or greater, typical Japanese fonts didn't include the glyphs necessary to transliterate Japanese text, referring to macroned vowels. Consider the following three examples:

- Common Japanese words, such as the place name *Tōkyō* (東京), require macroned vowels to properly express long vowels when using the Hepburn system, which is the most widely accepted Japanese transliteration system. The Kunrei and Nippon transliteration systems, by contrast, use the more common circumflexed vowels to denote long vowels, such as *Tôkyô* for the same example.

- Chinese transliteration, specifically *Pinyin*, requires not only macroned vowels for the first or "flat" tone, and vowels with the *acute* and *grave* diacritic marks for the second (or "rising") and fourth (or "falling") tones, respectively, but also vowels adorned with the *caron* diacritic mark for the third or "falling-rising" tone. The Chinese words *xìnxī chǔlǐ* (信息处理) and *zīxùn chǔlǐ* (資訊處理), both of which mean "information processing," exemplify three of the four tones that require glyphs adorned with diacritic marks. Interestingly, complete Pinyin transliteration requires additional base forms, specifically *ê* (*e circumflex*) and *ü* (*u umlaut*), which can take on a diacritic mark to indicate tone, along with *m* and *n* that can do the same.
- Vietnamese, specifically its Romanized script called *Quốc ngữ*, is even more complex in its use of multiple diacritic marks and additional base characters that are exemplified by the name of the script itself. Even the Vietnamese word *Việt Nam*, which means "Vietnam," makes use of glyphs that are not found in typical Latin fonts. Vietnamese is also unique in that some of its base forms, such as *ơ* (*horned o*) and *ư* (*horned u*), and one of its diacritic marks, specifically the *hỏi* (also called *hook above* or *curl*), are not used by any other transliteration or Romanization system.

Some fonts, such as Minion Pro and Myriad Pro, both of which are Adobe Originals and bundled with many Adobe applications, include more glyphs than typical fonts, and are suitable for purposes such as transliteration and Romanization. The current versions of these two typeface families include the glyphs necessary for Japanese transliteration and Vietnamese. Unfortunately, they are missing glyphs that correspond to vowels with the caron diacritic mark, required for Pinyin transliteration. This book is using interim versions that include these glyphs, and future versions are naturally expected to include them.

Adobe Systems is in the process of documenting Adobe Latin glyph sets that detail various levels of glyph coverage.* Adobe Latin 4 and greater, with an estimated 616 glyphs, supports the transliteration and Romanization systems just described. I should point out that one of Thomas Phinney's *Typblography* blog entries details some of the ideas and concepts that are expected to be reflected in the Adobe Latin glyph set documentation.†

Character Collections for CID-Keyed Fonts

A character collection represents a standardized set of CIDs and glyphs, meaning that two CIDFont resources that adhere or conform to the same character collection specification assign the same glyphs to the same CIDs. It also means that both CIDFont resources can be used with the same set of CMap resources.

* *http://www.adobe.com/type/browser/info/charsets.html*

† *http://blogs.adobe.com/typblography/2008/08/extended_latin.html*

There are three /CIDSystemInfo dictionary entries that must be present in every CIDFont and CMap resource, indicated by the following code:

```
/CIDSystemInfo 3 dict dup begin
  /Registry (Adobe) def
  /Ordering (CNS1) def
  /Supplement 5 def
end def
```

Yes, this same dictionary is present in the header of both CIDFont *and* CMap resources! In order for a CIDFont and CMap resource to be used together as a valid CID-keyed font, their respective /Registry and /Ordering strings must be identical. This compatibility check prevents the inadvertent or intentional mixing of CIDFonts and CMaps resources of different character collections.

A character collection is often referred to as an *ROS*, which is an abbreviated form of *Registry, Ordering, and Supplement*.

An overview of Adobe Systems' public CJKV character collections

Adobe Systems' CID-keyed font file specification makes it an almost trivial matter to define new or extend existing CJKV fonts. As part of my enjoyable work at Adobe Systems, I have developed the CJKV character collections specifications listed in Table 6-28.

Table 6-28. Adobe Systems' CJKV character collections for CID-keyed fonts

Character collection	CIDs	Previous Supplements	Adobe Systems Technical Note
Adobe-GB1-5	30,284	0 through 4	5079
Adobe-CNS1-5[a]	19,088	0 through 4	5080
Adobe-Japan1-6[b]	23,058	0 through 5	5078
Adobe-Japan2-0—deprecated	6,068	none	5097
Adobe-Korea1-2	18,352	0 and 1	5093
Adobe-Identity-0	n/a	none	n/a

a. Dirk Meyer designed Adobe-CNS1-1 and Adobe-CNS1-2.

b. Leo Cazares and Stephen Amerige designed Adobe-Japan1-0, but I designed all of its subsequent Supplements, specifically Adobe-Japan1-1 through Adobe-Japan1-6.

Specific Supplements of all but one of these character collections—specifically Adobe-GB1-3, Adobe-CNS1-2, Adobe-Japan1-3, and Adobe-Korea1-2—were designed to add only prerotated instances of all proportional- and half-width characters found in earlier Supplements. Their purpose is to significantly improve the vertical handling of such characters in the context of OpenType through the use of the 'vrt2' GSUB feature. As new Supplements were added, any additional non–full-width glyphs were accompanied by prerotated instances in order to maintain this functionality.

The Adobe-Vietnam1-0 character collection, which is intended to eventually support the TCVN 5712:1993, TCVN 5773:1993, and TCVN 6056:1995 character set standards, is

still, after 10 years, in the experimental stages. Its current form still supports only the glyphs for the TCVN 6056:1995 character set standard. For those font developers wishing to implement fonts that include only the glyphs necessary for supporting the Latin-based script, meaning Quốc ngữ, name-keyed fonts based on the Adobe Latin 4 glyph set represent a much better solution than CID-keyed fonts.

Each of these public character collections are associated with one or more CMap resources. As you have learned, a CMap resource imposes or applies an encoding to a CIDFont resource. As of this writing, there are a total of 178 CMap resources that correspond to the six public character collections that are listed in Table 6-28.

Adobe strongly encourages font developers to conform to these character collection specifications when developing their own CJKV font products. They were developed not only for Adobe Systems to use, but also for other font developers to use. So, they are to be treated as public character collections. Also, in order for applications to perform advanced layout or composition in a consistent manner across fonts, they must depend on the assumption that a given CID represents the same glyph, semantically speaking, in all fonts that are identified as sharing the same character collection. This allows advanced layout and composition across multiple fonts using glyphs not yet encoded in Unicode. The Adobe Systems Technical Note references in Table 6-28 can be used to obtain the corresponding character collection specification document from Adobe.* Of course, developers are free to define their own character collections, along with their corresponding CMap resources. They can also develop new CMap resources for these public character collections.

The following sections describe each of these six CJKV character collections in some detail, but please refer to the document entitled *Adobe CJKV Character Collections and CMaps for CID-Keyed Fonts* (Adobe Systems Technical Note #5094), which is the official documentation that describes the character sets and encodings supported by each of these CJKV character collections.

The Adobe-GB1-5 character collection

The Adobe-GB1-5 character collection, which enumerates a staggering 30,284 CIDs, supports the GB 2312-80, GB 1988-89, GB/T 12345-90, GBK, and GB 18030-2005 character sets. It also includes the corrections and additions to GB 2312-80 as specified in GB 6345.1-86. Supported encodings include ISO-2022-CN, EUC-CN, GBK, GB 18030, and Unicode. Table 6-29 lists the six supplements of this character collection, along with the number of CIDs defined in each one.

* *http://www.adobe.com/devnet/font/*

Table 6-29. Adobe-GB1-5 character collection supplements

Character collection	Total CIDs	Additional CIDs
Adobe-GB1-0	7,717	n/a
Adobe-GB1-1	9,897	2,180
Adobe-GB1-2	22,127	12,230
Adobe-GB1-3	22,353	226
Adobe-GB1-4	29,064	6,711
Adobe-GB1-5	30,284	1,220

Table 6-30 lists the CMap resources that are associated with the Adobe-GB1-5 character collection specification.

Table 6-30. Adobe-GB1-5 CMap resources

CMap name	Vertical?	Character set	Encoding
GB-H	Yes	GB 2312-80	ISO-2022-CN
GB-EUC-H	Yes	GB 2312-80	EUC-CN
GBpc-EUC-H	Yes	GB 2312-80 for Mac OS-S	EUC-CN
GBT-H	Yes	GB/T 12345-90	ISO-2022-CN
GBT-EUC-H	Yes	GB/T 12345-90	EUC-CN
GBTpc-EUC-H	Yes	GB/T 12345-90 for Mac OS-S	EUC-CN
GBK-EUC-H	Yes	GBK	GBK
GBKp-EUC-H	Yes	GBK	GBK
GBK2K-H	Yes	GB 18030	GB 18030
UniGB-UCS2-H[a]	Yes	Unicode	UCS-2
UniGB-UTF8-H	Yes	Unicode	UTF-8
UniGB-UTF16-H	Yes	Unicode	UTF-16BE
UniGB-UTF32-H	Yes	Unicode	UTF-32BE
Adobe-GB1-5[b]	No	All CIDs	<00 00> through <76 FF>

a. The UniGB-UCS2-H CMap resource is obsolete and its use is deprecated. The UniGB-UTF16-H CMap resource should be used instead.

b. This CMap resource is called the Identity CMap, and encodes all glyphs at their respective CIDs.

Supplement 0, known as Adobe-GB1-0, supported only the GB 2312-80 and GB 1988-89 character sets by enumerating 7,717 CIDs.

The GB/T 12345-90 character set was supported through an additional 2,180 CIDs that make up Supplement 1, specifically Adobe-GB1-1, and represent traditional hanzi replacements for GB 2312-80 characters plus additional hanzi found in rows 88 and 89.

The complete set of 20,902 CJK Unified Ideographs of Unicode version 1.0.1, known as the *Unified Repertoire and Ordering*, became supported in Supplement 2 by adding 12,230 glyphs, the intention of which was to support the GBK character set.*

Supplement 3 added only 226 prerotated glyphs that are expected to be accessed through the use of the OpenType 'vrt2' GSUB feature.

Supplement 4 added 6,711 glyphs to support the GB 18030-2000 character set, and Supplement 5 added 1,220 more glyphs to support the Yi regional script that is shown in GB 18030-2005. There are no plans to add to this glyph set the glyphs for the five remaining GB 18030-2005 regional scripts, specifically Korean, Mongolian, Tai Le, Tibetan, and Uyghur. It is best to implement support for those regional scripts through the implementation of separate fonts.

The Adobe-GB1-6 character collection is not yet in development, but may include the appropriate glyphs to allow font developers to build genuine OpenType Simplified Chinese Pro fonts.

The Adobe-CNS1-5 character collection

The Adobe-CNS1-5 character collection, which enumerates 19,088 CIDs, supports the CNS 11643-1992 (Planes 1 and 2 only), CNS 5205-1989, Big Five, Hong Kong GCCS, and Hong Kong SCS-2004 character sets. Given its character set coverage, the Adobe-CNS1-5 character collection thus supports the Taiwan and Hong Kong locales. This glyph set also includes the popular ETen extension for the Big Five character set, along with the Hong Kong extensions set forth by DynaComware and Monotype Imaging. Supported encodings include ISO-2022-CN, ISO-2022-CN-EXT, EUC-TW, Big Five, and Unicode. Table 6-31 lists the six supplements of this character collection, along with the number of CIDs defined in each one.

Table 6-31. Adobe-CNS1-5 character collection supplements

Character collection	Total CIDs	Additional CIDs
Adobe-CNS1-0	14,099	n/a
Adobe-CNS1-1	17,408	3,309
Adobe-CNS1-2	17,601	193
Adobe-CNS1-3	18,846	1,245
Adobe-CNS1-4	18,965	119
Adobe-CNS1-5	19,088	123

* This does not, however, mean that an Adobe-GB1-2 (or greater Supplement) CIDFont resource can be used for multiple locales. In fact, its glyphs are designed with the Chinese locale in mind, making them unsuitable for other locales.

Table 6-32 lists the CMap resources that are associated with the Adobe-CNS1-5 character collection specification.

Table 6-32. Adobe-CNS1-5 CMap resources

CMap name	Vertical?	Character set	Encoding
B5-H	Yes	Big Five	Big Five
B5pc-H	Yes	Big Five for Mac OS-T	Big Five
ETen-B5-H	Yes	Big Five with ETen extensions	Big Five
HKgccs-B5-H	Yes	Hong Kong GCCS	Big Five—extended
HKscs-B5-H	Yes	Hong Kong SCS-2004	Big Five—extended
HKdla-B5-H	Yes	DynaComware HK A	Big Five
HKdlb-B5-H	Yes	DynaComware HK B—665 hanzi only	Big Five
HKm471-B5-H	Yes	Monotype Hong Kong—471 set	Big Five
HKm314-B5-H	Yes	Monotype Hong Kong—314 set	Big Five
CNS1-H	Yes	CNS 11643-1992 Plane 1	ISO-2022-CN
CNS2-H	Yes	CNS 11643-1992 Plane 2	ISO-2022-CN-EXT
CNS-EUC-H	Yes	CNS 11643-1992 Planes 1 and 2	EUC-TW
UniCNS-UCS2-H[a]	Yes	Unicode	UCS-2
UniCNS-UTF8-H	Yes	Unicode	UTF-8
UniCNS-UTF16-H	Yes	Unicode	UTF-16BE
UniCNS-UTF32-H	Yes	Unicode	UTF-32BE
Adobe-CNS1-5[b]	No	All CIDs	<00 00> through <4A FF>

a. The UniCNS-UCS2-H CMap resource is obsolete and its use is deprecated. The UniCNS-UTF16-H CMap resource should be used instead.

b. This CMap resource is called the Identity CMap, and encodes all glyphs at their respective CIDs.

Supplement 0 includes glyphs for simultaneously support of the Big Five and CNS 11643-1992 character set, though only Planes 1 and 2 of the latter are supported. The ordering of the hanzi glyphs in Adobe-CNS1-0 favors that which is set forth in CNS 11643-1992, not Big Five. In addition, the two duplicately encoded hanzi in Big Five are not assigned CIDs. Instead, the Big Five CMap resources duplicately encode them.

Supplement 1 added 3,309 glyphs to support the common Hong Kong extensions at the time, specifically Hong Kong GCCS, along with the Hong Kong extensions to Big Five that were defined by DynaComware and Monotype Imaging.

Supplement 2 added only 193 prerotated glyphs that are expected to be accessed through the use of the OpenType ‘vrt2’ GSUB feature.

Supplements 3, 4, and 5 added 1,245, 119, and 123 glyphs, respectively, to support the Hong Kong SCS-1999, -2001, and -2004 character sets, respectively. Supplement 6 is expected to be defined to cover the 68 additional hanzi set forth in Hong Kong SCS-2008.

The Adobe-Japan1-6 character collection

The Adobe-Japan1-6 character collection, which enumerates 23,058 CIDs, supports all vintages of the JIS X 0208:1997 character set, from JIS C 6226-1978 (aka JIS78) to JIS X 0208:1997, along with the JIS X 0201-1997, JIS X 0212-1990, and JIS X 0213:2004 character sets. It also supports the Apple KanjiTalk version 6, Apple KanjiTalk version 7, Microsoft Windows version 3.1J, Microsoft Windows version 95J, Fujitsu FMR, and NEC vendor extensions of JIS X 0208, along with Kyodo News' U-PRESS. Supported encodings include ISO-2022-JP, Shift-JIS, EUC-JP (code sets 0 through 2), and Unicode. Table 6-33 lists the seven supplements of this character collection, along with the number of CIDs defined in each one.

Table 6-33. Adobe-Japan1-6 character collection supplements

Character collection	Total CIDs	Additional CIDs
Adobe-Japan1-0	8,284	n/a
Adobe-Japan1-1	8,359	75
Adobe-Japan1-2	8,720	361
Adobe-Japan1-3	9,354	634
Adobe-Japan1-4	15,444	6,090
Adobe-Japan1-5	20,317	4,873
Adobe-Japan1-6	23,058	2,741

Table 6-34 lists all of the CMap resources that are associated with the Adobe-Japan1-6 character collection specification.

Table 6-34. Adobe-Japan1-6 CMap resources

CMap name	Vertical?	Character set	Encoding
H	Yes	JIS X 0208:1997	ISO-2022-JP
RKSJ-H	Yes	JIS X 0208:1997	Shift-JIS
EUC-H	Yes	JIS X 0208:1997	EUC-JP
78-H[a]	Yes	JIS C 6226-1978	ISO-2022-JP
78-RKSJ-H[a]	Yes	JIS C 6226-1978	Shift-JIS
78-EUC-H[a]	Yes	JIS C 6226-1978	EUC-JP
83pv-RKSJ-H[b]	No	KanjiTalk version 6	Shift-JIS

Table 6-34. Adobe-Japan1-6 CMap resources

CMap name	Vertical?	Character set	Encoding
90pv-RKSJ-H[c]	Yes	KanjiTalk version 7	Shift-JIS
90ms-RKSJ-H	Yes	Windows 3.1J and Windows 95J	Shift-JIS
90msp-RKSJ-H[d]	Yes	Windows 3.1J and Windows 95J	Shift-JIS
78ms-RKSJ-H	Yes	Windows 3.1J and Windows 95J	Shift-JIS
Add-H	Yes	Fujitsu's FMR Japanese	ISO-2022-JP
Add-RKSJ-H	Yes	Fujitsu's FMR Japanese	Shift-JIS
Ext-H	Yes	NEC Japanese	ISO-2022-JP
Ext-RKSJ-H	Yes	NEC Japanese	Shift-JIS
NWP-H	Yes	NEC Word Processor	ISO-2022-JP
Hankaku	No	Half-width Latin and katakana	One-byte
Hiragana	No	Full-width hiragana	One-byte
Katakana	No	Full-width katakana	One-byte
Roman	No	Half-width Latin	One-byte
WP-Symbol	No	Special symbols	One-byte
UniJIS-UCS2-H[e]	Yes	Unicode	UCS-2
UniJIS-UCS2-HW-H[e]	Yes	Unicode	UCS-2
UniJIS-UTF8-H	Yes	Unicode	UTF-8
UniJIS2004-UTF8-H	Yes	Unicode	UTF-8
UniJIS-UTF16-H	Yes	Unicode	UTF-16BE
UniJIS2004-UTF16-H	Yes	Unicode	UTF-16BE
UniJIS-UTF32-H	Yes	Unicode	UTF-32BE
UniJIS2004-UTF32-H	Yes	Unicode	UTF-32BE
UniJISX0213-UTF32-H	Yes	Unicode	UTF-32BE
UniJISX02132004-UTF32-H	Yes	Unicode	UTF-32BE
Adobe-Japan1-6[f]	No	All CIDs	<00 00> through <5A FF>

a. These were once available in OCF fonts, but were dropped starting with many early products. They were "brought back from the dead" under CID-keyed font technology.

b. Unlike other CMap resources that have corresponding vertical versions, the 83pv-RKSJ-H CMap resource does not have a vertical version. In other words, the 83pv-RKSJ-V CMap resource does not exist.

c. I chose to use 90pv-RKSJ-H in the spirit of 83pv-RKSJ-H. Still, no one really knows what the "pv" stands for, and that is officially okay.

d. 90ms-RKSJ-H and 90msp-RKSJ-H differ only in that the former uses half-width Latin characters in the single-byte range (0x20 through 0x7E), and the latter uses proportional.

e. The UniJIS-UCS2-H and UniJIS-UCS2-HW-H CMap resources are obsolete and their use is deprecated. The UniJIS-UTF16-H or UniJIS2004-UTF16-H CMap resource should be used instead.

f. This CMap resource is called the Identity CMap, and encodes all glyphs at their respective CIDs.

Adobe-Japan1-0, the original Supplement, enumerates the same number of glyphs as found in OCF fonts, specifically 8,284 CIDs. Supplement 0 did not support the JIS X 0208-1990, JIS X 0208:1997, Apple KanjiTalk version 7, Microsoft Windows version 3.1J, nor Microsoft Windows 95J character sets.

Adobe-Japan1-1, or Supplement 1, added 75 glyphs to support JIS X 0208-1990 (and thus JIS X 0208:1997 because they are identical in terms of character set and prototypical glyphs) and the Apple KanjiTalk version 7 character set, bringing the total number of CIDs to 8,359.

Adobe-Japan1-2, or Supplement 2, added 361 glyphs to support the Microsoft Windows versions 3.1J and 95J character sets, bringing the total number of CIDs to 8,720. 359 of these 361 glyphs cover the IBM Selected Kanji set.

Adobe-Japan1-3, or Supplement 3, added only 634 prerotated glyphs that are expected to be accessed through the use of the OpenType 'vrt2' GSUB feature.

Supplement 4 added 6,090 glyphs, with the intent to define the first Pro glyph set, in order to satisfy most of the commercial and professional publishing needs in Japan. Among the additional glyphs are various kanji, many of which are variant forms. JIS X 0221-1995's Ideographic Supplement 1, which is a set of 918 kanji from JIS X 0212-1990, are among the 6,090 glyphs that are included in Adobe-Japan1-4.

The 4,873 glyphs added for Supplement 5 were primarily designed to support the JIS X 0213:2000 character set, along with additional kanji variants. Like Supplement 4, this supplement was designed to further satisfy most of the needs of commercial and professional publishing.

Supplement 6, which added 2,741 glyphs, obsoleted the Adobe-Japan2-0 character collection (which is described in the section that follows) by providing complete support for the JIS X 0212-1990 character set. Kyodo News' U-PRESS character set is also covered by Adobe-Japan1-6. Given the coverage of JIS X 0212-1990 in Supplements 4 and 5, the number of additional glyphs that were necessary was relatively low.

Supplement 7 is currently in the very early stages of planning. Glyphs to support the ARIB STD-B24 character set, along with the glyphs for additional U-PRESS characters, are among the glyphs being considered for Adobe-Japan1-7.

The Adobe-Japan2-0 character collection—deprecated

The Adobe-Japan2-0 character collection, which enumerates 6,068 CIDs, supports the entire JIS X 0212-1990 character set. Supported encodings include ISO-2022-JP-2 (and thus ISO-2022-JP-1), EUC-JP (code set 3 only), and Unicode. Table 6-35 lists the CMap resources that are associated with the Adobe-Japan2-0 character collection specification.

Table 6-35. Adobe-Japan2-0 CMap resources

CMap name	Vertical?	Character set	Encoding
Hojo-H	Yes	JIS X 0212-1990	ISO-2022-JP-2
Hojo-EUC-H	Yes	JIS X 0212-1990	EUC-JP
UniHojo-UCS2-H[a]	Yes	Unicode	UCS-2
UniHojo-UTF8-H	Yes	Unicode	UTF-8
UniHojo-UTF16-H	Yes	Unicode	UTF-16BE
UniHojo-UTF32-H	Yes	Unicode	UTF-32BE
Adobe-Japan2-0[b]	No	All CIDs	<00 00> through <17 FF>

a. The UniHojo-UCS2-H CMap resource is obsolete and its use is deprecated. The UniHojo-UTF16-H CMap resource should be used instead. But, given that this entire character collection is obsoleted and its use is deprecated, this table note doesn't mean very much.

b. This CMap resource is called the Identity CMap, and encodes all glyphs at their respective CIDs.

The Adobe-Japan2-0 character collection lacks certain key glyphs, such as the half-width or proportional Latin sets, because it was originally intended to be used in conjunction with Adobe-Japan1-*x* CIDFonts. Now that the Adobe-Japan1-6 character collection has effectively obsoleted this character collection, this point is moot.

The Adobe-Korea1-2 character collection

The Adobe-Korea1-2 character collection, which enumerates 18,352 CIDs, supports the KS X 1001:1992 character set, along with all possible 11,172 hangul. Supported encodings include ISO-2022-KR, EUC-KR, UHC, Johab, and Unicode. The Mac OS extension as used by Mac OS-KH and KLK, which totals to 1,137 additional characters, is also supported. Table 6-36 lists the three supplements of this character collection, along with the number of CIDs defined in each one.

Table 6-36. Adobe-Korea1-2 character collection supplements

Character collection	Total CIDs	Additional CIDs
Adobe-Korea1-0	9,333	n/a
Adobe-Korea1-1	18,155	8,822
Adobe-Korea1-2	18,352	197

Table 6-37 lists the CMap resources that are associated with the Adobe-Korea1-2 character collection specification.

Table 6-37. Adobe-Korea1-2 CMap resources

CMap name	Vertical?	Character set	Encoding
KSC-H	Yes	KS X 1001:1992	ISO-2022-KR
KSC-EUC-H	Yes	KS X 1001:1992	EUC-KR
KSCpc-EUC-H	Yes	KS X 1001:1992 for Mac OS-KH[a]	EUC-KR[b]
KSC-Johab-H	Yes	KS X 1001:1992—Johab[c]	Johab
KSCms-UHC-H	Yes	KS X 1001:1992—Johab[c]	UHC[d]
KSCms-UHC-HW-H	Yes	KS X 1001:1992—Johab[c]	UHC[d]
UniKS-UCS2-H[e]	Yes	Unicode	UCS-2
UniKS-UTF8-H	Yes	Unicode	UTF-8
UniKS-UTF16-H	Yes	Unicode	UTF-16BE
UniKS-UTF32-H	Yes	Unicode	UTF-32BE
Adobe-Korea1-2[f]	No	All CIDs	<00 00> through <47 FF>

a. The Mac OS extension includes approximately 1,137 additional symbols.

b. Apple's EUC-KR encoding includes an expanded second-byte range: 0x41 through 0x7D and 0x81 through 0xFE.

c. Includes all 11,172 hangul.

d. Unified Hangul Code. See Appendix E for more details.

e. The UniKS-UCS2-H CMap resource is obsolete and its use is deprecated. The UniKS-UTF16-H CMap resource should be used instead.

f. This CMap resource is called the Identity CMap, and encodes all glyphs at their respective CIDs.

Supplement 0 includes only 4,620 hanja, though KS X 1001:1992 defines 4,888 hanja. Why is there a discrepancy? 268 of the 4,888 hanja in KS X 1001:1992 are genuine duplicate characters, and they appear in that character set standard multiple times because they have multiple readings. The Adobe-Korea1-2 CMap resources thus multiply-encode hanja as appropriate, by mapping multiple code points to a single CID. There is, of course, a minor file size benefit, but the greatest advantage is that characters that are genuinely duplicated in the KS X 1001:1992 standard are represented by the same glyph in a font based on the Adobe-Korea1-2 character collection.

The 8,822 CIDs that make up Supplement 1 are entirely glyphs for hangul, and represent the difference between all 11,172 possible hangul and the 2,350 defined in KS X 1001:1992.

Supplement 2 added only 197 prerotated glyphs that are expected to be accessed through the use of the OpenType 'vrt2' GSUB feature.

Note that one of the two characters added in KS X 1001:1998, along with the one added in KS X 1001:2002, are not included in Adobe-Korea1-2. When Supplement 3 is defined, glyphs for these two characters shall be included. In the case of the euro currency symbol that was added in KS X 1001:1998, full-width and proportional forms are likely to be included in Adobe-Korea1-3, along with a prerotated form of the latter.

The special-purpose Adobe-Identity-0 character collection

The special-purpose Adobe-Identity-0 character collection was initially used by Acrobat and PDF for representing glyphs by their CIDs, but it can also be used for developing OpenType CJKV fonts with special-purpose dynamic glyph sets. The Adobe-Identity-0 character collection should be used only if the character collections described in the previous sections are unsuitable for the intended glyph set.

Adobe Systems' かづらき Std L (KazurakiStd-Light), which is an Adobe Original typeface designed by Ryoko Nishizuka (西塚涼子 *nishizuka ryōko*), represents the very first OpenType font that was built by taking advantage of the flexibility and benefits of the Adobe-Identity-0 character collection. Its fully proportional nature, along with its vertical-only hiragana ligatures, effectively meant that the Adobe-Japan1-6 character collection was unsuitable.

The document entitled *Special-Purpose OpenType Japanese Font Tutorial: Kazuraki* (Adobe Technical Note #5901) describes the techniques that were used to build the KazurakiStd-Light font, with the specific intention of guiding other font developers in the building of similar fonts. Due to the special nature of such fonts, some applications may experience difficulty using them, or using some of their functionality. It goes without saying that an adequate amount of application-level testing is the prudent thing to do when developing such fonts.

Is there a Pan-CJKV character collection?

Yes and no. Another very appropriate application of the special-purpose Adobe-Identity-0 character collection is for the building of Pan-CJKV fonts, specifically a single font instance that includes the necessary glyphs to satisfy the requirements of some or all CJKV locales. In other words, a large number of Unicode points will be associated with multiple glyphs, and to which glyph they map depends on the selected locale. Some Unicode code points, such as U+4E00, obviously need no more than a single glyph in order to satisfy all CJKV locales. Other Unicode code points may require up to five distinct glyphs.

One would think that yet another static glyph set definition would be appropriate for building OpenType Pan-CJKV fonts. Given the degree to which typeface design directly affects how many glyphs are required per Unicode code point to support the necessary locales, breaking free from a static glyph set is highly desirable. In addition, given the amount of design resources that are necessary to design a Pan-CJKV font, the number of such fonts is likely to be very low, perhaps even countable on one hand.

The biggest hurdle in developing a Pan-CJKV font is, of course, to design one or more glyphs per Unicode code point. The URO includes 20,902 ideographs. Extensions A and B include 6,582 and 42,711 ideographs, respectively. All of these code points potentially require more than one glyph. Once the glyphs are designed, the preferred way in which to implement is through the use of the OpenType 'locl' (*Localized Forms*) GSUB feature. Table 6-38 lists the language and script tags that must be specified for the locale-specific glyph substitutions in the 'locl' GSUB feature.

Table 6-38. Language and script tags for OpenType Pan-CJKV fonts

Locale	Language tag	Script tag
Simplified Chinese	ZHS	hani
Traditional Chinese[a]	ZHT and ZHH	hani
Japanese	JAN	kana
Korean	KOR	hang

a. Because there are separate language tags for Chinese as used in Taiwan (ZHT) and Hong Kong (ZHH), and because both tags are appropriate for Traditional Chinese, both language tags should be specified.

Adobe InDesign CS3 and later, for example, recognizes and makes use of the 'locl' GSUB feature. It is possible to define paragraph and character styles that specify a locale that directly corresponds to the language and script tags that are specified in the 'locl' GSUB feature.

Some fonts, such as Arial Unicode MS, which is bundled with Windows, include the 'locl' GSUB feature, but its script tags are set differently. Compared to what is listed in Table 6-38, this font sets the script to 'hani' for Korean, presumably because the scope of its 'locl' GSUB feature is limited to CJK Unified Ideographs. This is not necessarily a bug in the font or in Adobe InDesign, but rather a reflection of differing philosophies about what constitutes language tagging. Given that this is relatively new functionality, the likelihood of change is high. Again, an adequate amount of application-level testing is the prudent thing to do.

Another perhaps not-so-obvious consideration is which glyphs to use as the default for a Pan-CJKV font. Given that GB 18030-2005, by definition, includes all CJK Unified Ideographs in Unicode, Simplified Chinese serves as a very suitable and appropriate default in terms of the glyph that are directly encoded in the 'cmap' table. Even if the Simplified Chinese glyphs are set as the default, it is still important to define paragraph and character styles that specify Simplified Chinese.

Interestingly, the glyphs shown in Table 3-99, found in Chapter 3, were implemented as a small yet fully functional "proof of concept" Pan-CJKV font and were selected through the use of appropriately set InDesign character styles and the 'locl' GSUB feature.

Character collection Supplements—the past, present, and future

Whenever a CJKV character collection is updated, which is always performed by defining a new Supplement, there is almost always a motivation or trigger for doing so. The character sets used by each locale are based on national standards and common vendor extensions thereof, and there is now influence from international character sets, specifically Unicode and ISO 10646. When these national standards, vendor extensions thereof, or international character sets expand or otherwise change, a new Supplement sometimes becomes necessary. Due to the static nature of character collections, this has to be done in the form of a new Supplement.

As an historical example, there were two motivations for defining the Adobe-Japan1-1 character collection: the publishing of the JIS X 0208-1990 character set, which added two kanji, and Apple's definition of the KanjiTalk7 character set. The result was a somewhat modest Supplement containing only 75 additional glyphs.

Sometimes, however, two sets of characters that are useful within a single locale are intentionally divided or separated by placing them into two separate Supplements, whereby the more frequently used set forms an earlier Supplement, and the other becomes a later Supplement, or is used to define two completely independent character collections.

The Adobe-GB1-1 character collection is an example of the former scenario. Adobe-GB1-0 defines the glyphs for supporting the GB 2312-80 character set, which represented the most basic character set standard for China at the time of its establishment. Every Chinese typeface design (that is, for China, not Taiwan) must minimally conform to the GB 2312-80 character set. However, some typeface designs also include the characters for the GB/T 12345-90 character set, which, as you learned in Chapter 3, is the traditional analog of GB 2312-80. When comparing these two character sets in detail, one finds that there are 2,180 hanzi in GB/T 12345-90 that are not included in GB 2312-80. So, these 2,180 hanzi became Supplement 1, specifically Adobe-GB1-1. Much of this history has been obscured or obliterated by GBK and GB 18030, but Adobe-GB1-1 preserves it to some extent.

The Adobe-Japan2-0 character collection is an example of the latter scenario. The JIS X 0212-1990 character set, as described in Chapter 3, enumerates 6,067 characters, 5,801 of which are kanji. But, very few typeface designs included the glyphs for these characters—FDPC's Heisei Mincho W3 was the only design readily available with the glyphs for the JIS X 0212-1990 for a number of years. So, these 6,067 characters became a separate and independent character collection, specifically Adobe-Japan2-0. The Adobe-Japan1-*x* character collection incrementally started to support the JIS X 0212-1990 character collection, beginning with Supplement 4. Adobe-Japan1-6 served to complete the support for JIS X 0212-1990, which also served to obsolete the Adobe-Japan2-0 character collection. Now, it is considered better practice to implement JIS X 0212-1990 support through the use of the Adobe-Japan1-6 character collection.

Whether to divide collections of glyphs into separate Supplements of a single character collection or to define them as independent character collections is decided by considering glyph availability for the majority of typeface designs. Forcing or obligating typeface designers or type foundries to come up with thousands of additional glyphs per typeface design is not a very reasonable thing to do. By defining separate Supplements, the burden placed on typeface designers and type foundries is eased.

In some cases, the motivation for change was not triggered by any specific character set, whether national, vendor-specific, or international. Adobe-Japan1-4, for example, was specifically designed to leap beyond those standards, and to include the glyphs necessary for professional and commercial publishing in Japan. A wide variety of glyph sets were studied, and there was also close collaboration with several Japanese type foundries. The

end result was the first "Pro" glyph set targeted toward the Japanese market. There are plans to do the same for Chinese and Korean.

Supporting Unicode in CID-keyed fonts

Because the CID-keyed font file specification is extremely flexible and extensible—primarily because it effectively divorces all encoding information from the font file itself—a large number of encodings can be supported simply by building an appropriate CMap file. Because Unicode is intended as a single character set that provides access to all alphabets, including CJKV characters, it is a natural candidate for one such encoding.

One typical question that is asked with regard to developing Unicode fonts is whether to include all characters in Unicode. Given the 64K glyph limit, and the fact that Unicode version 5.1 has surpassed the 100,000-character barrier, it is simply not possible to build such fonts, let alone the cast typeface design resources that would be necessary to pull off such a feat. Here are some points to consider:

- Input methods, which represent the typical means by which users input characters, are already restricted to a subset of the corresponding national character set standard. Using Unicode as the underlying encoding will not magically extend the scope of an input method.
- Many characters that are used in one CJKV locale may be completely useless or superfluous in another—consider the simplified hanzi in China, which are considered useless in Japan.
- Glyph-design principles vary greatly across CJKV locales. Even given the same typeface design, many characters will need to have their glyphs constructed differently depending on the locale.

Of course, having multiple input methods installed, one for each locale, can be useful for users whose work extends beyond a single locale. And, for these same users, having access to characters not normally available in a given locale has advantages in some special-purpose situations.

The next question that people often ask deals with specifying the most useful subset of Unicode for each CJKV locale. To a great extent, this question is already answered by virtue of the existing national character set standards for each CJKV locale, such as GB 2312-80 for China. Simply map these CJKV character set standards to Unicode, and you instantly have a useful font for that CJKV locale. It is really that simple. For CID-keyed fonts, it means issuing a new set of Unicode-encoded CMap resources—the CIDFonts themselves, which can be several megabytes in size, need not be reissued. The glyphs need not change. Only the encoding that is imposed on the glyphs needs to change.

I have developed Unicode—specifically UCS-2–, UTF-8–, UTF-16BE–, and UTF-32BE–encoded—CMap resources for all five of Adobe Systems' public CJKV character

collections, and have made them available for developers or anyone else to use.* In fact, these Unicode CMap resources, with the exception of the UCS-2 ones, are considered the primary CMap resources today, and in general are the only ones that are updated on a regular basis.

There are two types of events that trigger Unicode CMap resource updates. One is obviously the definition of a new Supplement for a character collection. The other is a new version of Unicode. Unicode version 5.1, for example, allowed seven previously unencoded kanji to be encoded, thus triggering seven additional mappings in the Adobe-Japan1-6 Unicode CMap resources.

Table 6-39 lists each of Adobe Systems' CJKV character collections, along with the name of each Unicode-encoded CMap resource.

Table 6-39. Unicode CMap resources

Character collection	UCS-2[a]	UTF-8	UTF-16BE	UTF-32BE
Adobe-GB1-5	UniGB-UCS2-H	UniGB-UTF8-H	UniGB-UTF16-H	UniGB-UTF32-H
Adobe-CNS1-5	UniCNS-UCS2-H	UniCNS-UTF8-H	UniCNS-UTF16-H	UniCNS-UTF32-H
Adobe-Japan1-6[b]	UniJIS-UCS2-H[c]	UniJIS-UTF8-H	UniJIS-UTF16-H	UniJIS-UTF32-H[d]
Adobe-Japan2-0	UniHojo-UCS2-H	UniHojo-UTF8-H	UniHojo-UTF16-H	UniHojo-UTF32-H
Adobe-Korea1-2	UniKS-UCS2-H	UniKS-UTF8-H	UniKS-UTF16-H	UniKS-UTF32-H

a. The UCS-2 CMap resources are obsolete, and their use is deprecated. The corresponding UTF-16BE CMap resource should be used instead.

b. The Adobe-Japan1-6 character collection also includes JIS2004-savvy CMap resources for UTF-8, UTF-16BE, and UTF-32BE encodings, specifically UniJIS2004-UTF8-H, UniJIS2004-UTF16-H, and UniJIS2004-UTF32-H.

c. The UniJIS-UCS2-HW-H CMap resource also exists, but like its sibling, it is obsolete and its use is deprecated.

d. The UniJISX0213-UTF32-H and UniJISX02132004-UTF32-H CMap resources also exist, and differ in that 65 code points that correspond to infrequently used symbols map to glyphs with proportional widths instead of full-width ones.

Of course, all of these Unicode-encoded CMap resources have vertical counterparts. For example, the vertical counterpart of UniKS-UTF8-H is UniKS-UTF8-V. Simply replace the final "H" with a "V" in the CMap resource name.

Ruby Glyphs

Ruby glyphs, commonly referred to as *furigana* outside of publishing circles, are specific to Japanese typography and serve to annotate kanji with readings. How ruby glyphs are used in line-layout is deferred until Chapter 7, but here we focus on the details of the glyphs used for ruby, such as their special design considerations.

A complete set of ruby glyphs, or a special-purpose ruby font, typically consists of hiragana, katakana, and a handful of symbols. When one compares the glyphs of a ruby font,

* *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/*

or the ruby glyphs in a Japanese font, with those of standard Japanese fonts, there are two important differences to note, indicated as follows:

- Small kana, such as ぁ, ぃ, ぅ, ぇ, ぉ, こ, っ, や, ゆ, よ, and わ for hiragana, are sometimes the same size as their standard-sized equivalents. This means that しょう and しよう , specifically the よ versus よ, cannot be distinguished.*
- Small kana, when they are the same size as their standard-sized equivalents, do not need special vertical-use forms.

The Adobe-Japan1-4 character collection includes a complete set of ruby glyphs, and OpenType fonts based on them are expected to use the 'ruby' GSUB feature to allow applications to access these glyphs. Adobe InDesign, for example, supports the 'ruby' GSUB feature in the context of its ruby functionality. The Adobe-Japan1-5 character collection includes additional ruby glyphs that correspond to somewhat more obscure kana characters.

Generic Versus Typeface-Specific Ruby Glyphs

Ruby glyphs can come in several varieties, depending on the typeface and the type foundry that designed them: generic, generic to a font family, and specific to a typeface:

Generic
: Designed to be used with a variety of fonts and font families.

Generic to a particular font family
: Intended for use with all weights of that font family—due to the small size at which ruby glyphs are set, they do not always benefit from differing weights.

Specific to a particular typeface design
: Although not very common, these do serve special purposes. Morisawa's Ryumin font family, for example, includes several ruby glyphs, one for nearly every weight.

Which variety of ruby glyphs you use depends on what is available for the font you are using, or what glyphs are available in the font itself.

Table 6-40 illustrates the standard kana and ruby kana glyphs that are included in Adobe Systems' 小塚明朝 Pr6N B (KozMinPr6N-Bold) typeface design. The Adobe-Japan1-6 character collection, on which this font is based, includes a complete set of glyphs specifically designed for ruby use.

* The distinction between standard and small kana is often critical. For the examples しよう (*shiyō*) and しょう (*shō*), the former represents the reading for the common compounds 使用 (*shiyō*) and 仕様 (*shiyō*), and the latter can represent the reading for any one of a large number of kanji.

Table 6-40. Standard versus ruby kana—KozMinPr6N-Bold

Character type	Sample text
Standard	さるもきからおちる
Ruby	さるもきからおちる

Note how their designs are quite different, although they are included in the same typeface design. In order for ruby glyphs to be legible at small sizes, they are often slightly lighter or heavier than their standard counterparts, depending on the relative weight of the typeface.

Chapter 7 explores the more practical aspects of ruby glyphs, such as how they are used and typeset, and how they differ from their close cousins, pseudo-ruby glyphs.

Host-Installed, Printer-Resident, and Embedded Fonts

In the past, an important user and developer concern was how to map host-installed fonts to printer-resident fonts. The host-installed font either needed to include information that explicitly specified to which printer-resident font it corresponded, or else an OS-level database had to exist that mapped host-installed fonts to printer-resident fonts. The sections that follow describe how this font mapping took place on Mac OS, Windows (versions 3.1, 95, and 98), and to a limited extent, the X Window System.

The large size of typical CJKV fonts gave an advantage when mapping to printer-resident fonts, because downloading such large fonts on a per-job basis was tedious. Subsetting and PDF embedding have effectively resolved this issue.

Now, the preferred way in which to handle fonts in a printing workflow is to embed the glyphs in PDF. This is a good thing, because it guarantees that the same glyphs in host-installed fonts are used to display or print the document. The only thing that would prevent the proper embedding of glyphs in a PDF, besides the document author explicitly choosing not to do so, is if the embedding permissions of the font are set to a value that does not permit embedding.

Installing and Downloading Fonts

While there are plenty of tools and utilities for installing non-CJKV fonts, whether to the host—Mac OS or Windows, for example—or to the printer, those designed for handling CJKV fonts are still rare due to the added complexity. Most font developers end up developing their own font installation software or licensing it from another company.

Font developers who are serious about authoring their own font installation and downloading software should read Adobe Systems Technical Notes #5174 (*CID-Keyed Font*

Installation for PostScript File Systems) and #5175 (*CID-Keyed Font Installation for ATM Software*), which provide the necessary technical details for performing these tasks.

Given that the current PDF-based document workflow removes the font burden from the printer, the need to download the fonts to the printer has been effectively eliminated, or at least significantly minimized. The current workflow paradigm also has the benefit of guaranteeing that the document prints exactly as it is displayed because the same font, and perhaps more importantly the same version of the font, is used for both purposes.

The PostScript Filesystem

PostScript devices use a hierarchical filesystem composed of several directories, along with some subdirectories. Exactly where font components are installed depends on the vintage of the font format. Table 6-41 illustrates how a PostScript Japanese OCF font is distributed across a PostScript filesystem. Note that the base font name, such as "HeiseiMin-W3," has been replaced by an "X" for the sake of brevity and readability.

Table 6-41. OCF font file structure

Directory	Contents
fonts	X-83pv-RKSJ-H, X-83pv-SuppA-H, X-83pv-SuppB-H, X-Add-H, X-Add-RKSJ-H, X-Add-RKSJ-V, X-Add-SuppA-H, X-Add-SuppA-V, X-Add-SuppB-HV, X-Add-V, X-EUC-H, X-EUC-V, X-Ext-H, X-Ext-RKSJ-H, X-Ext-RKSJ-V, X-Ext-SuppA-H, X-Ext-SuppA-V, X-Ext-SuppB-HV, X-Ext-V, X-H, XJIS.zm_23, XJIS.zm_29, X-JIS.zm_2E, X-NWP-H, X-NWP-V, X-PropRoman, X-RKSJ-H, XRKSJ-UserGaiji, X-RKSJ-V, X-SJ.zm_82, X-SJ.zm_82v, X-SJ.zm_85, X-SuppA-H, X-SuppA-V, X-SuppB-HV, X-V, X.Hankaku, X.Hiragana, X.Katakana, X.Oubun, X.Oubun-Add, X.Roman, X.Roman83pv, X.SuppK, X.WP-Symbol
fsupp	X-83pv-SuppA_BDY, X-83pv-SuppB_BDY, X-Add-SuppA_BDY, X-Add-SuppB_BDY, X-Add_BDY, X-EUC_BDY, X-Ext-SuppA_BDY, X-Ext-SuppB_BDY, X-Ext_BDY, X-NWP_BDY, X-SuppA_BDY, X-SuppB_BDY, X_BDY
pgfonts	X::AlphaNum, X::Alphabetic, X::Dingbats, X::HKana, X::HRoman, X::JIS83-1Kanji, X::JIS83-2, X::Kana, X::KanjiSupp
pgfsupp	X::AlphaNum_COD, X::AlphaNum_CSA, X::Alphabetic_COD, X::Alphabetic_CSA, X::Dingbats_COD, X::Dingbats_CSA, X::HKana_COD, X::HKana_CSA, X::HRoman_COD, X::HRoman_CSA, X::JIS83-1Kanji_COD, X::JIS83-1Kanji_CSA, X::JIS83-2_COD, X::JIS83-2_CSA, X::Kana_COD, X::Kana_CSA, X::KanjiSupp_COD, X::KanjiSupp_CSA

That's certainly a lot of files! Eighty-five to be exact. And for a single font! Intimidating? I certainly hope so. Keep in mind that all of these files were present for *every* PostScript Japanese OCF font. Furthermore, the file structure of an OCF font differs depending on its vintage—the OCF font file structure illustrated in Table 6-41 represents the structure found in the very last Japanese OCF fonts produced by Adobe Systems.

Japanese OCF fonts produced by Adobe Systems were also dependent on a handful of generic font files that contained, among other things, glyphs for the half- and full-width line-drawing elements as found in row 8 of JIS X 0208:1997. The file structure for these files is provided in Table 6-42.

Table 6-42. OCF font file structure—generic components

Directory	Contents
pgfonts	Generic::FullWidth, Generic::HalfWidth
pgfsupp	Generic::FullWidth_COD, Generic::FullWidth_CSA, Generic::HalfWidth_COD, Generic::HalfWidth_CSA

In contrast, Table 6-43 illustrates the PostScript file structure for a CIDFont resource and its associated CMap resources. Note the simplicity.

Table 6-43. CID-keyed font file structure

Directory	Contents
Resource/CIDFont	HeiseiMin-W3
Resource/CMap	78-EUC-H, 78-EUC-V, 78-H, 78-RKSJ-H, 78-RKSJ-V, 78-V, 78ms-RKSJ-H, 78ms-RKSJ-V, 83pv-RKSJ-H, 90ms-RKSJ-H, 90ms-RKSJ-V, 90msp-RKSJ-H, 90msp-RKSJ-V, 90pv-RKSJ-H, 90pv-RKSJ-V, Add-H, Add-RKSJ-H, Add-RKSJ-V, Add-V, Adobe-Japan1-0, Adobe-Japan1-1, Adobe-Japan1-2, Adobe-Japan1-3, Adobe-Japan1-4, Adobe-Japan1-5, Adobe-Japan1-6, EUC-H, EUC-V, Ext-H, Ext-RKSJ-H, Ext-RKSJ-V, Ext-V, H, Hankaku, Hiragana, Katakana, NWP-H, NWP-V, RKSJ-H, RKSJ-V, Roman, UniJIS-UCS2-H, UniJIS-UCS2-HW-H, UniJIS-UCS2-HW-V, UniJIS-UCS2-V, UniJIS-UTF16-H, UniJIS-UTF16-V, UniJIS-UTF32-H, UniJIS-UTF32-V, UniJIS-UTF8-H, UniJIS-UTF8-V, UniJIS2004-UTF16-H, UniJIS2004-UTF16-V, UniJIS2004-UTF32-H, UniJIS2004-UTF32-V, UniJIS2004-UTF8-H, UniJIS2004-UTF8-V, UniJISPro-UCS2-HW-V, UniJISPro-UCS2-V, UniJISPro-UTF8-V, UniJISX0213-UTF32-H, UniJISX0213-UTF32-V, UniJISX02132004-UTF32-H, UniJISX02132004-UTF32-V, V, WP-Symbol

Once you install a set of CMap resources that are associated with a CJKV character collection, they need not be installed for subsequent CIDFont resources that share the same character collection—they are common across all CIDFonts resources of the same character collection.

For details on how CID-keyed fonts are installed onto a PostScript filesystem, I again suggest reading the document entitled *CID-Keyed Font Installation for PostScript File Systems* (Adobe Systems Technical Note #5174).

Mac OS X

Mac OS X has greatly simplified the way in which fonts are installed and supported, especially compared to Mac OS 9 and earlier. Fonts that are required by Mac OS X, meaning that they are used for purposes such as for the UI, are installed in a specific directory, as follows:

/System/Library/Fonts/

Because these fonts are required by Mac OS X, they cannot be removed, and users cannot install additional fonts into that directory. Other OS-bundled fonts (which are considered

optional and can thus be removed), along with user-installed fonts, are installed in the following public directory:

/Library/Fonts/

Users can also install fonts locally, to their own account, thus making them private when more than one user account is present on a single machine. For example, I could install fonts into the following directory on my computer:

/Users/lunde/Library/Fonts/

For single-user machines, there is effectively no functional difference between installing fonts into the public or private font directory, though one concern is different versions of the same font, one of which is installed in public directory, while the other is installed in the private one. Of course, when such a conflict exists, the font with the higher version number wins, as it should.

Font Book

Mac OS X includes an application called Font Book that catalogs the fonts that are currently installed, and that can also be used to install and active fonts. Double-clicking on a font launches Font Book.

Font Book is a very convenient way in which to explore what fonts are currently installed, and it allows each font to be previewed. Attributes of the installed fonts, such as the version number, are also displayed. Font Book also has the ability to perform font validation, for any font, regardless of whether it is installed. This is useful for diagnosing font issues.

Mac OS 9 and Earlier

Given the dominance of Mac OS X, the information in this section is more or less for historical purposes, and it serves to demonstrate how the font situation has improved for users who transitioned from Mac OS 9 and earlier to Mac OS X.

Mac OS utilized the style map of a resource named 'FOND', which is a critical part of each font suitcase, in order to provide the mapping to a printer-resident PostScript font. If a font suitcase includes more than one font, there are necessarily multiple instances of the 'FOND' resource. The style map of a 'FOND' resource included two important pieces of information that were highly dependent on one another:

- A string that was a fully qualified PostScript font name, such as STSong-Light-GBpc-EUC-H. This string then became the argument to PostScript's "findfont" procedure when this font is used.
- A byte that indicated the length of this PostScript font name string—this byte appeared immediately before the string itself. This byte, for this particular example, has a value of `0x17` (decimal 23) that corresponds to the byte-length of the string STSong-Light-GBpc-EUC-H.

The 'FOND' resource also included other crucial font-related information, such as the following:

Menu name (such as 华文宋体 for STSong-Light)
: This is actually the name of the 'FOND' resource instance, not part of the 'FOND' resource. This is the string displayed in application font menus.

FOND ID
: An internal identifier that served to indicate the script or locale of the font.

Widths table
: Explicitly listed the widths, expressed in ¼096 units, for glyphs encoded in the range 0x00 through 0xFF, meaning for 256 total code points.

The FOND ID ranges listed in Table 6-44 are important for CJKV font developers because they indicated to the OS how to treat the font in terms of locale and encoding. Note that each of these four CJKV locales was restricted to ranges that each contained 512 FOND IDs.

Table 6-44. FOND ID ranges

FOND ID range	Script	Character set	Encoding
16384–16895	Japanese	JIS X 0208:1997	Shift-JIS
16896–17407	Traditional Chinese	Big Five	Big Five
17408–17919	Korean	KS X 1001:1992	EUC-KR
28672–29183	Simplified Chinese	GB 2312-80	EUC-CN

The 'FOND' resource, although typically small in size, provided to the OS the information necessary to make a font available in application font menus, the font's locale, and the mapping to a printer-resident PostScript font.

The contents of the 'FOND' resource were also used by ATM (described in Chapter 8) to decide which host-based outline font to access. It was also possible for two or more different 'FOND' resources to reference the same PostScript font name. Table 6-45 illustrates this phenomenon.

Table 6-45. Multiple 'FOND' resources with identical PostScript font names

Menu name	Source	PostScript font name
細明朝体	Mac OS-J	Ryumin-Light-83pv-RKSJ-H
L リュウミン L-KL	Adobe Systems	Ryumin-Light-83pv-RKSJ-H
中ゴシック体	Mac OS-J	GothicBBB-Medium-83pv-RKSJ-H
M 中ゴシック BBB	Adobe Systems	GothicBBB-Medium-83pv-RKSJ-H

Although outline font data can be shared by more than one 'FOND' resource, quite often there are differing global metrics information—specified in the 'NFNT' resource—that alter the exact placement of glyphs. One, of course, cannot install two 'FOND' resources that share the same menu name.

Also of potential interest is the fact that many of the commonly used Mac OS-J fonts were based on the same typeface designs, but were either in a different format—PostScript versus TrueType—or used a different design space. Table 6-46 lists such Mac OS-J fonts.

Table 6-46. Identical typeface designs in different Mac OS font formats

Menu name	Format	Design space	Source
L リュウミン L-KL	PostScript	1000×1000	Morisawa's Ryumin Light
リュウミンライト-KL	TrueType	2048×2048	Morisawa's Ryumin Light
M 中ゴシック BBB	PostScript	1000×1000	Morisawa's Gothic BBB Medium
中ゴシック BBB	TrueType	2048×2048	Morisawa's Gothic BBB Medium
平成明朝 W3	PostScript	1000×1000	FDPC's Heisei Mincho W3
平成明朝	TrueType	2048×2048	FDPC's Heisei Mincho W3
平成角ゴシック W5	PostScript	1000×1000	FDPC's Heisei Kaku Gothic W5
平成角ゴシック	TrueType	2048×2048	FDPC's Heisei Kaku Gothic W5
Osaka	TrueType	256×256	FDPC's Heisei Kaku Gothic W5

However, these identical designs in different font formats are not compatible due to metrics differences in the proportional Latin glyphs that are encoded in their one-byte ranges.

Where are the bitmaps?

Latin bitmapped fonts for Mac OS are encapsulated in a font suitcase (so named because the icon looks like a suitcase). CJKV bitmapped fonts for Mac OS are separated into two basic components:

- A font suitcase, so named because its icon looks like a suitcase
- One or more 'fbit' files, so named because its file type is 'fbit'*

Legacy fonts (that is, older fonts) typically consist of a rather small—in terms of byte-size—font suitcase plus one or two large 'fbit' files. Contemporary fonts consist of a large font suitcase plus one small 'fbit' file. Table 6-47 provides information about legacy and contemporary bitmapped fonts for Mac OS.

* The 'fbit' files are not necessary when running under QuickDraw GX or with Mac OS version 8.5 or greater.

Table 6-47. Legacy Mac OS bitmapped fonts

Vintage	Font suitcase resources[a]	'fbit' resources[a]	Comments
Older	FOND, NFNT	fbit	One-byte bitmaps in 'NFNT' resource; two-byte bitmaps in 'fbit' file's data fork
Old	FOND, NFNT, sfnt	fbit, fdef, fdnm	One-byte bitmaps in 'NFNT' resource; all bitmaps in 'sfnt' resource

a. This file may also contain other resources, such as 'vers' (version).

The real difference between legacy and contemporary Mac OS bitmapped fonts is the appearance of the 'sfnt' resource. Contemporary fonts can store bitmapped data in the 'bdat' table (also known as the 'sbit' table) of the 'sfnt' resource. The 'fbit' file's 'fbit' resource contains pointers into the data fork of the 'fbit' file (for legacy fonts) or into the 'bdat' table of the font suitcase (for contemporary fonts). The 'sfnt' resource can also store a complete outline font, either TrueType, Type 1, or CIDFont. The 'fdef' resource allows Mac OS to understand the contemporary format. The 'fdnm' resource is described in the following section.

Whereas the font suitcases appear the same regardless of CJKV script, the 'fbit' files appear differently because they have a different creator. Table 6-48 lists the four CJKV scripts supported by Mac OS, along with the file type and creator for their 'fbit' files.

Table 6-48. File type and creators for 'fbit' files

Script	File type	Creator
Simplified Chinese	fbit	CSYS
Traditional Chinese	fbit	TSYS
Japanese	fbit	KSYS
Korean	fbit	hfon

Are 512 FOND IDs enough?

512 FOND IDs may seem like plenty at first glance (and, apparently to the designers of Mac OS), but as the result of a past boom in available CJKV fonts, it was not nearly enough. For example, Adobe Systems—as of the writing of the first edition—had over 100 different Japanese fonts in its type library, which consumes approximately one-fifth of the entire Japanese FOND ID range. Many Korean type foundries also have well over 100 individual fonts in their type libraries. The lack of FOND IDs was indeed a problem.

Apple was forced to devise a system to handle this situation of too few FOND IDs for non-Latin scripts, and the result was the birth of the *Asian Font Arbitrator* and the 'fdnm' resource, whose function was to dynamically harmonize FOND IDs of CJKV fonts in situations when a user installs a new font whose FOND ID conflicts with one that is already installed.

Mac OS power users know that FOND ID harmonization is something that was introduced in System7. However, CJKV fonts do not consist of only a font suitcase—they also use the infamous 'fbit' files. While the font suitcase includes single-byte bitmapped data in the 'NFNT' resource, and there is one 'NFNT' resource instance per bitmapped size, the 'fbit' files contain two-byte bitmapped data, or, for newer fonts, contain pointers into the font suitcase's 'bdat' table, which is inside the 'sfnt' resource. The 'fbit' files contain back pointers to the font suitcase (actually, to its FOND ID). Thus, when standard System7 FOND ID harmonization kicks in, only the FOND ID in the font suitcase becomes harmonized, and the back pointers in the 'fbit' files now point off to a nonexistent font suitcase.

The Asian Font Arbitrator requires that an 'fdnm' resource exist in the 'fbit' files, and that an 'fdnm' table exist in the corresponding 'FOND' resource. This "resource plus table" combination is linked by a common attribute, which cannot be broken even if FOND ID harmonization takes place. This common attribute is the font's menu name. All CJKV fonts for use on Mac OS should include a well-formed 'fdnm' resource and table. Contact Apple for more information on the Asian Font Arbitrator.

Where, oh where have my vertical variants gone?

Of the CJKV locales, only Japanese had adequate vertical support on Mac OS. Most other OSes, such as Windows, used a separate font resource to act as the vertical instance of a font. In other words, the vertical version of a font appeared as a separate font instance in application font menus. Unlike other OSes, Mac OS-J and JLK encoded the vertical variants within the horizontal font instance at standard offsets from their horizontal counterparts.

KanjiTalk6 and Mac OS-J both supported the same set of 53 vertical variants for Japanese, but at different offsets. Thirty-one of these had their horizontal versions in row 1 (punctuation and symbols) of JIS X 0208:1997, 10 in row 4 (small hiragana), and 12 in row 5 (small katakana).

Under KanjiTalk6, these 53 vertical variants were at a 10-row offset from their horizontal code points, that is, in rows 11, 14, and 15. For example, the small hiragana "a" character (あ) is normally encoded at Shift-JIS <82 9F> (Row-Cell 04-01), but its vertical version was encoded at Shift-JIS <87 9F> (Row-Cell 14-01). As far as Shift-JIS encoding is concerned, the shift value was 0x05, which is the difference between the first-byte values, specifically 0x87 minus 0x82.

Under Mac OS-J or JLK, these same 53 vertical variants were at an 84-row offset, that is, in rows 85, 88, and 89. Using the same example as before, small hiragana "a" (あ) is normally encoded at Shift-JIS <82 9F> (Row-Cell 04-01), but its vertical version was encoded at Shift-JIS <EC 9F> (Row-Cell 88-01). As far as Shift-JIS encoding is concerned, the shift value is 0x6A, which is the difference between the first-byte values, specifically 0xEC minus 0x82.

PostScript Japanese fonts have always had these 53 vertical variants at the 84-row offset. Under KanjiTalk6, Mac OS had to further offset the vertical variants—which were already offset by 10 rows—by 74 rows when printing to PostScript printers.

Table 6-49 summarizes the two types of offsets for handling vertical variants under KanjiTalk, Mac OS-J, and JLK.

Table 6-49. Vertical variant row offsets

Mac OS version	Vertical offset	Vertical rows	Value for 'tate' table
KanjiTalk6	10-row	11, 14, and 15	0x05
Mac OS-J and JLK	84-row	85, 88, and 89	0x6A

Japanese 'FOND' resources for use under Mac OS-J or JLK normally include a 'tate' (縦 *tate*, meaning "vertical") table, which explicitly indicates which offset should be used for vertical variants. If no 'tate' table exists, Mac OS assumes that a 10-row offset is desired. PostScript Japanese fonts, therefore, must explicitly indicate an 84-row offset by using the 'tate' table and setting its value to 0x6A.

FONDedit, a utility provided to developers by Apple, was able to properly create and set a 'tate' table in 'FOND' resources.

Microsoft Windows—2000, XP, and Vista

Windows 2000, XP, and Vista support OpenType fonts natively. And, compared to earlier versions of Windows OS, the overall user experience with regard to installing and handling fonts is much better.

Installing one or more fonts into these contemporary versions of Windows OS is simply a matter of selecting the Fonts control panel and following the instructions. Removing fonts is performed through the use of the same control panel. Be sure to quit all applications before installing fonts, particularly those applications in which you intend to immediately use the newly installed fonts.

Microsoft Windows—Versions 3.1, 95, 98, ME, and NT4

Versions of Microsoft Windows prior to 2000, XP, and Vista, and unlike Mac OS, typically provided two instances per CJKV font: one is horizontal, and the other is the corresponding vertical instance. But, how are these font instances differentiated for the user? Have you ever heard of the at (@, 0x40) symbol? It is prefixed to the menu name of the vertical font instance. Table 6-50 illustrates this, and for the sake of completeness, I also included the equivalent PostScript font name.

Table 6-50. Windows horizontal and vertical fonts

Writing direction	Menu name	PostScript font name
Horizontal	HY 중고딕	HYGoThic-Medium--KSCms-UHC-H
Vertical	@HY 중고딕	HYGoThic-Medium--KSCms-UHC-V

The critical Windows file that maps host-based fonts to printer-resident fonts is called the *Printer Font Metrics* (PFM) file. This file provides similar information to that found in the 'FOND' resource on Mac OS. Two PFM files are typically required for each CJKV font: one for horizontal and the other for vertical.

Instead of using a range of FOND IDs whereby specific ranges are reserved for each locale, the PFM includes a single-integer identifier, known as the *lfCharset* value, to indicate script or locale. Table 6-51 lists the lfCharset values that are important for CJKV font development.

Table 6-51. CJKV lfCharset values

lfCharset	Script	Character set	Encoding
128	Japanese	JIS X 0208:1997	Shift-JIS
129	Korean	KS X 1001:1992	EUC-KR or UHC
131	Korean	KS X 1001:1992	Johab
134	Simplified Chinese	GB 2312-80	EUC-CN or GBK
136	Traditional Chinese	Big Five	Big Five

Extreme care must be taken when creating PFMs due to byte order issues. I know this from personal experience, mainly because I developed a CJKV PFM generator in Perl—this tool runs on big- or little-endian machines and creates little-endian output. There are many fields within a PFM file that specify two- and four-byte numeric values, such as metrics and other integer-like values. The values must be represented in little-endian byte order. Practical information on building PFM files for CJKV fonts, including this Perl utility, can still be found in the document entitled *Building PFM Files for PostScript-Language CJK Fonts* (Adobe Systems Technical Note #5178).

Installing and registering PFMs

Once you have built a well-formed PFM file, it needs to be installed and registered to the OS. Installation of a PFM file can be anywhere on the filesystem, but is typically in the same location as other PFMs.

Registering the PFM to the OS is a matter of visiting and editing a system-level control file called *WIN.INI*. This file is where you must specify the full path to the PFM file (which is why a PFM can be stored anywhere on the filesystem). Let's consider that we installed two

PFM files in the following paths, the first of which corresponds to the horizontal PFM file, and the second of which corresponds to its vertical counterpart:

```
C:\PSFONTS\STSONGLI\METRICS\GKEUC_H.PFM
C:\PSFONTS\STSONGLI\METRICS\GKEUC_V.PFM
```

The following is an example `[PostScript]` section from a *WIN.INI* file, and includes the lines added for the preceding two PFM files:

```
[PostScript, \\SJ TW 7\panda]
softfonts=2
softfont1=C:\PSFONTS\STSONGLI\METRICS\GKEUC_H.PFM
softfont2=C:\PSFONTS\STSONGLI\METRICS\GKEUC_V.PFM
```

Note that the keyword "softfonts" indicates the number of softfont entries, which, in this case, is two. Note that when you register the PFMs by editing the *WIN.INI* file, you are informing the PostScript driver that the fonts are resident on the printer hard disk or otherwise available on the PostScript printer. More detailed information on installing PFM files can be found in the document entitled *CID-Keyed Font Compatibility with ATM Software* (Adobe Systems Technical Note #5175).

Of course, when dealing with OpenType fonts, and given the fact that they are well supported in Windows XP and Windows Vista, the need to work with PFM files has significantly diminished.

Unix and Linux

Unix (and Linux) have historically been plagued with relatively poor support for outline font formats, especially when compared to Mac OS, Mac OS X, and the many flavors of Windows (2000, XP, and Vista). FreeType has changed this.* Now, the preferred way in which to handle fonts in Unix and Linux OSes is through the use of a portable font-rendering engine called FreeType 2. FreeType 2 supports a wide variety of font formats, and most importantly, it supports OpenType. It is included with virtually all Linux distributions. The name "FreeType" suggests that it is free, and its usage falls under the terms of your choice of two different open source licenses. To a great extent, FreeType is expected to be used by application developers, and so it offers font-rendering services.

In terms of the advanced typographic functionality offered in OpenType fonts, such as through the use of 'GPOS' and 'GSUB' tables, FreeType 2 provides no such services, but does correctly render the glyphs. Pango is essential for text layout on modern Linux distributions due to its extensive support for OpenType features, and because it has an emphasis toward internationalization.†

* *http://freetype.sourceforge.net/index2.html*

† *http://www.pango.org/*

X Window System

The X Window System* natively uses one of two font formats, both bitmapped, and both derivable from BDF, specifically SNF and PCF. SNF (*Server Natural Format*) is used on X11R4 and earlier, and PCF (*Portable Compiled Format*) is used on X11R5 and later. The current version is X11R7. In the extremely unlikely event that you are running DPS (*Display PostScript*), you can also use PostScript CJKV fonts.† FreeType 2 is used in the X Windows client-side XFT Font Library, which is itself used in the Qt and GTK toolkits.

X Window System font names are unique in that they are abnormally long, mainly because the name itself includes almost all of its global information. The following is an example taken from a JIS X 0208:1997 font:

```
-Misc-Fixed-Medium-R-Normal--16-150-75-75-C-160-JISX0208.1997-0
```

Hyphens are used to separate the information specified in the font name. Table 6-52 lists each of these fields.

Table 6-52. X Window System font names

Field name	Sample value	Description
Foundry	Misc	The font developer.
Font family	Fixed	The font's family name.
Weight	Medium	The font's relative weight.
Slant	R	The font's slant angle.
Set width	Normal	The font's proportionate width.
Additional style		Additional style information.
Pixels	16	The font's size measured in pixels.
Points	150	The font's size measured in 10ths of a point (thus, 15 points).
Horizontal resolution—dpi	75	The horizontal resolution at which the bitmapped pattern was designed.
Vertical resolution—dpi	75	The vertical resolution at which the bitmapped pattern was designed.
Spacing	C	The font's spacing: P for proportional, M for monospaced, and C for character cell (such as a box).
Average width	160	The font's average width measured in 10ths of a pixel (thus, 16 pixels).
Character set registry	JISX0208.1997	The character set designation.
Character set encoding	0	For CJKV fonts, 0 usually means ISO-2022 encoding, and 1 means EUC encoding; for CNS 11643-1992, it is used to indicate plane number.

Luckily, the X Window System also has a convenient method for aliasing these excessively long font names to shorter ones, by using a special file called *fonts.alias* in a font

* *http://www.x.org/*

† Unix systems offered by Sun and SGI were often bundled with DPS.

directory. An example line from this file is as follows (using the example font name used previously):

```
jis97 -Misc-Fixed-Medium-R-Normal--16-150-75-75-C-160-JISX0208.1997-0
```

For more information on X Window System font formats, I suggest Chapter 6, "Font Specification," in O'Reilly Media's *X Window System User's Guide* by Valerie Quercia and Tim O'Reilly. The article entitled "The X Administrator: Font Formats and Utilities" (*The X Resource*, Issue 2, Spring 1992, pages 14–34) by Dinah McNutt and Miles O'Neal provides detailed information about X Window System font formats, including BDF, SNF, and PCF.

Installing and registering X Window System fonts

There are several relatively easy steps that must be followed in order to install and register X Window System fonts. These steps are outlined as follows, though the strings that will change due to different filenames or directories have been emboldened:

1. Convert a BDF file to SNF or PCF, depending on what version of X Window System you are running:

   ```
   % bdftopcf jiskan16-1997.bdf > k16-97.pcf
   % bdftosnf jiskan16-1997.bdf > k16-97.snf
   ```

2. Copy the resulting SNF or PCF file to a font directory, then run the `mkfontdir` command as follows (assuming that you are creating a directory called fonts in your home directory) in order to create a *fonts.dir* file:

   ```
   % mkfontdir ~/fonts
   ```

3. Add this font directory to your font search path by running `xset` with the `+fp` option:

   ```
   % xset +fp ~/fonts
   ```

4. Create and edit the *fonts.alias* file—in the same location as the corresponding *fonts.dir* file. You must run `xset` with the `fp` option in order to make the X Font Server aware of your newly created alias:

   ```
   % xset fp rehash
   ```

5. You can now display the font using `xfd` (*X Font Displayer*) with the `-fn` option, as follows:

   ```
   % xfd -fn k16-97
   ```

You can now use this font in other X Window System applications, such as *Emacs*. It is really that simple.

PostScript extensions for X Window System fonts

Mac OS and Windows supported a mechanism whereby the OS is made aware of the fully qualified PostScript font name that corresponds to the OS-specific font resources. For Mac OS, it was the style map of the font suitcase's 'FOND' resource that provides this

information; for Windows, it was a field in the PFM file. The following is the BDF header that corresponds to a PostScript font—the PostScript extensions are emboldened:

```
STARTFONT 2.1
FONT -Adobe-HeiseiMin-W3-R-Normal--12-120-72-72-P-115-Adobe.Japan1-3
SIZE 12 72 72
FONTBOUNDINGBOX 13 12 -2 -3
STARTPROPERTIES 23
FONTNAME_REGISTRY "Adobe"
FOUNDRY "Adobe"
FAMILY_NAME "HeiseiMin"
WEIGHT_NAME "W3"
SLANT "R"
SETWIDTH_NAME "Normal"
ADD_STYLE_NAME ""
PIXEL_SIZE 12
POINT_SIZE 120
RESOLUTION_X 72
RESOLUTION_Y 72
SPACING "P"
AVERAGE_WIDTH 115
CHARSET_REGISTRY "Adobe.Japan1"
CHARSET_ENCODING "3"
CHARSET_COLLECTIONS "Adobe-Japan1"
FONT_ASCENT 9
DEFAULT_CHAR 0
_ADOBE_PSFONT "HeiseiMin-W3-EUC-H"
_ADOBE_XFONT "-Adobe-HeiseiMin-W3-R-Normal--12-120-72-72-P-115-Adobe.Japan1-2"
_DEC_DEVICE_FONTNAMES "PS=HeiseiMin-W3-EUC-H"
COPYRIGHT "Copyright 1996 Adobe Systems Incorporated. All Rights Reserved."
ENDPROPERTIES
CHARS 8720
```

The fully qualified PostScript font name, HeiseiMin-W3-EUC-H, is provided in two places. PostScript drivers can extract the fully qualified PostScript font name from the arguments of the two fields `_ADOBE_PSFONT` or `_DEC_DEVICE_FONTNAMES`. But, in the case of the `_DEC_DEVICE_FONTNAMES` field, its argument must be split so that the initial `PS=` does not become part of the fully qualified PostScript font name.

Font and Glyph Embedding

The most common way in which fonts, or the glyphs of fonts, are embedded is in the context of Acrobat and *Portable Document Format* (PDF). It is also possible to embed fonts into SWF, which is the Adobe Flash file format. Due to the typical large size of CJKV fonts, and the fact that only a small fraction of their glyphs are referenced in a normal document, entire CJKV fonts are not embedded. Instead, only the glyphs that are referenced in the document in which the font is to be embedded are. This is merely common sense, is practical, and is referred to as *subsetted embedding*.

Some fonts, however, include information about the vendor's desired embedding behavior. The 'OS/2' table, specifically the *fsType* field, specifies the embedding levels of OpenType and TrueType fonts.* Table 6-53 lists common *OS/2.fsType* values.

Table 6-53. Common OS/2.fsType values—embedding levels

Value	Meaning
0	Installable Embedding—this is the most liberal level of embedding
2	Restricted License Embedding—no embedding is permitted
4	Preview and Print Embedding—this is the minimum level of embedding
8	Editable Embedding

Back to Acrobat and PDF, as long as all of the fonts that are used are not set to Restricted License Embedding, everything that is needed to print a document—meaning the fonts, graphics, and other page elements—can be encapsulated into a PDF file. This PDF file subsequently serves as a complete, robust, and reliable digital representation of the document.

Cross-Platform Issues

For purely historical reasons, seven of Adobe Systems' PostScript Japanese fonts have different menu names (that is, the localized font name that appears in applications' font menus) between Mac OS and Windows, as far as OCF fonts are concerned. Table 6-54 illustrates these differences (note that the presence or absence of a "space" is sometimes the only difference).

Developers who plan to support PostScript Japanese fonts and are developing cross-platform applications need to be aware of such issues.

Table 6-54. Mac OS and Windows font menu names

Base PostScript font name	Mac OS	Windows
Ryumin-Light	L リュウミン L-KL	リュウミン L-KL
GothicBBB-Medium	M 中ゴシック BBB	中ゴシック BBB
FutoMinA101-Bold	B 太ミン A101	太ミン A101
FutoGoB101-Bold	B 太ゴ B101	太ゴ B101
Jun101-Light	L じゅん 101	じゅん 101
HeiseiMin-W3	平成明朝 W3	平成明朝W3
HeiseiKakuGo-W5	平成角ゴシック W5	平成角ゴシックW5

* *http://www.microsoft.com/typography/otspec/os2.htm*

To a significant extent, OpenType has effectively solved the cross-platform issues that have plagued the various font formats over the years. For example, an OpenType font has identical binary content regardless of platform or OS. In my experience, the only thing that could potentially affect the cross-platform nature of an OpenType font in a negative fashion would be to use non-ASCII characters for its filename. To the extent that Unicode is supported today, because OpenType fonts are files, and because files are easily moved, copied, or transferred, it takes only a single non-Unicode client to mangle a non-ASCII filename. Given that fonts are meant to be installed in specific directories, away from users' eyes, using ASCII characters for their filenames is appropriate, and for the reasons just described, also prudent.

Font Development Tools

If you are interested in creating your own typefaces, by all means endeavor to do so. Be aware, however, that designing typefaces requires special skills and a lot of time, especially for a CJKV typeface that contains thousands, if not tens of thousands, of glyphs. If you are a font developer, the information in the sections that follow are sure to be helpful in learning about what font development tools are available.

Designing a CJKV typeface in outline font format is almost always never an individual task, but the result of a dedicated group effort. It would literally take several man years for an individual to design a complete CJKV typeface of adequate quality and glyph set coverage. This should not stop anyone from trying, though. Sometimes, you may simply need to add a few glyphs to an existing typeface. It is also not enough to be able to design individual glyphs: all the glyphs that you design for a typeface must also match each other in terms of both style and weight.

Compared to 10 years ago, when the first edition of this book was published, there are now many more font development tools available, and the vast majority are Unicode-enabled and support OpenType. The following sections attempt to list and describe the tools that I find the most appealing, due to their functionality and overall level of support.

In addition to the font development tools that I describe in the sections that follow, Apple* and Microsoft† also make many of their own font development tools available. VOLT (*Visual OpenType Layout Tool*) and MS Font Validator, both developed by Microsoft, deserve special mention and should be explored. In any case, I strongly encourage you to explore and use these font development tools. That's why they are made available.

Bitmapped Font Editors

While nearly all outline font editors (described in the sections that follow) include the ability to create and edit bitmapped fonts, even if it is by merely generating them from

* *http://developer.apple.com/textfonts/Fonttools/*

† *http://www.microsoft.com/typography/DevToolsOverview.mspx*

their corresponding outlines, there are also dedicated bitmapped font editors. Because outline fonts are now very common on Mac OS X and Windows and are really the preferred format for fonts, the demand or need for dedicated bitmapped font editors has diminished to a significant extent. However, bitmapped fonts are still a big part of Unix, Linux, and the X Windows System. Mark Leisher's *XmBDFEditor* was considered a very good bitmapped font editor for Unix, Linux, and related OSes, but has since been replaced by his *gbdfed* tool.*

Outline Font Editors

Several high-quality font editors are commercially available and are quite useful for font development purposes, whether it is at the personal or business level.

Historically, one of the most prominent font editors was *Fontographer*, originally developed by Altsys and available for Mac OS and Windows.† It then became an Aldus product, but changed hands as part of the Adobe Systems merger, and subsequently became a Macromedia product. It is now a FontLab product, which was partially due to Macromedia's merge with Adobe Systems in late 2005.

Fontographer allows its users to create fonts for virtually any platform—Mac OS, Mac OS X, the various flavors of Windows, Unix, Linux, and so on—even if the version of Fontographer that is running is not designed for that specific platform or OS.‡ Fontographer also supports TrueType fonts, and it is available in a Japanese version with a Japanese-language interface, but it has no Japanese-specific functionality. Fontographer allows its users to build Type 1 or TrueType fonts from the same outline data.

The outline font editor that I highly recommend is FontLab Studio, available for Windows and Mac OS X and used by the majority of professional and commercial font developers.§ FontLab Studio surpasses Fontographer in all respects, such as offering four-axis multiple master support and native TrueType editing, meaning that it allows the user or developer to specify TrueType hints and quadratic splines. FontLab Studio also includes a macro language supported by Python and provides the ability to define a subroutine library for commonly used "parts" of glyphs. Although the number of glyphs in these font formats cannot exceed 64K, as described earlier in this chapter, FontLab Studio limits the number of glyphs to 6,400. This number of glyphs is more than sufficient for most non-CJKV needs. If you need to build a font that includes more than 6,400 glyphs, consider using AsiaFont Studio, which is described in the following section.

* *http://www.math.nmsu.edu/~mleisher/Software/gbdfed/*

† *http://www.fontlab.com/font-editor/fontographer/*

‡ The sole exceptions to this are multiple master and legacy Mac OS font support. Only the Mac OS version allows users to create these complex Type 1 fonts, and it is limited to two axes, meaning up to four master designs per glyph. Also, because legacy Mac OS fonts, called *font suitcases*, are not flat files, and include the Mac OS–specific resource fork, only the Mac OS version of Fontographer can generate such fonts.

§ *http://www.fontlab.com/font-editor/fontlab-studio/*

FontLab Studio provides full support for OpenType layout features, specifically those specified in the 'GPOS' and 'GSUB' tables. This allows developers to build feature-rich OpenType fonts. This is yet another reason why FontLab Studio is considered one of the best, if not the best, outline font editors available today.

Outline Font Editors for Larger Fonts

The outline font editors described in the previous section are limited in that they can create only single-byte–encoded fonts, or have a severe limit in terms of the maximum number of glyphs. As glyph design tools, they are of commercial quality. In fact, I had been a devoted Fontographer user for many years, and now use FontLab Studio for much of my work. Do not, however, underestimate the usefulness of these outline font editors: glyph design is completely independent from the ability to create multiple-byte– or Unicode-encoded fonts. Luckily, at least one such font editor is finally available: AsiaFont Studio.* TTEdit is a comparable TrueType editor for Windows, and the same software developer also make OTEdit for Mac OS X and Windows.†

AsiaFont Studio was originally called FontLab Composer, and allowed users to build CID-keyed fonts; that is, fonts that contained thousands of characters, and were genuine multiple-byte–encoded CJKV fonts! This was a big feat. AsiaFont Studio now supports the building of OpenType fonts and still represents the only reasonably priced, commercially available application that allows end users and developers alike to build CID-keyed fonts, whether the result is simply a CIDFont resource or a complete OpenType font. AsiaFont Studio is currently available for Windows and Mac OS X. The Windows version generates the following files when building a CIDFont resource:

- CIDFont resource
- CID-keyed AFM file‡
- Two PFM files: one horizontal and one vertical

AsiaFont Studio is an outline font editor that was specifically designed for creating and manipulating multiple-byte fonts. It provides most of the capabilities found in FontLab Studio, such as the ability to edit the outlines of the glyphs, and it adds the ability to organize and arrange huge glyph sets, such as those required for building CJKV fonts.

As an outline font editor, AsiaFont Studio provides all the usual tools and features for working with glyphs and outlines. The glyph editing window provides layers for illustrating grids, nodes, control points, guides, hints, bitmapped templates, and glyph previews. In addition, AsiaFont Studio includes a special palette of vector paint tools that give the user the ability to create and edit glyphs with standard bitmap-manipulation tools that result in vector forms.

* *http://www.fontlab.com/font-editor/asiafont-studio/*

† *http://musashi.or.tv/ttedit.htm*

‡ Built according to the latest *Adobe Font Metrics* (AFM) specification. See Adobe Technical Note #5004.

AsiaFont Studio eases the pain of manipulating fonts with thousands or tens of thousands of glyphs by subdividing them into subfonts, each of which can be viewed independently, and its glyphs can be freely moved and rearranged. AsiaFont Studio also provides the user with the big picture, or a bird's eye view, called the *CMap view*, which displays the entire font in a single table.

To assist in the process of building CIDFont resources or corresponding OpenType 'CFF' tables that conform to public glyph sets, such as those published by Adobe Systems, AsiaFont Studio is bundled with several bitmap-based templates that correspond to each of the five CJKV character collections that Adobe Systems has developed for building CIDFont resources (see Table 6-28). Adobe Systems' public CMap resources, as listed in Tables 6-30, 6-32, 6-34, 6-35, and 6-37, can be used with such fonts. Of course, users are free to create their own character collections, if desired.

FontForge is another outline font editor that is quite capable, in terms of supporting a very large number of glyphs up to the maximum that the formats allow, and in supporting a wide variety of outline font formats.*

It is important to know that using an outline font editor that supports up to 64K glyphs is not necessary in order to build such fonts. Such fonts can be managed through the building of smaller fonts in a directory structure, and put together or assembled into a single font through the use of other tools, such as those available in AFDKO, which is described in the next section.

AFDKO—Adobe Font Development Kit for OpenType

Adobe Systems released a suite of command-line–driven font development tools, called AFDKO (*Adobe Font Development Kit for OpenType*), which are specifically designed to ease or simplify the development of OpenType fonts.† These tools are updated on a regular basis, in order to address any bugs and to provide enhancements or new functionality, with new options for existing tools, and sometimes in the form of new tools. It is important to note that the tools provided in AFDKO are the same tools that Adobe Systems uses to develop its own OpenType fonts, meaning that I and other members of Type Development at Adobe Systems use them to develop and test OpenType fonts; because they are command-line tools, they are suitable for batch processing. The tools provided in AFDKO are thus industrial-strength, and have a high level of support commitment by Adobe Systems. In case it is not painfully obvious, I highly recommend their use. In addition, FontLab and DTL licensed portions of AFDKO as the basis for the OpenType support for their font production applications, such as FontLab Studio and DTL FontMaster.‡

* *http://fontforge.sourceforge.net/*

† *http://www.adobe.com/devnet/opentype/afdko/*

‡ *http://www.fonttools.org/*

Table 6-55 lists the majority of the font development and testing tools that are included in AFDKO, along with a brief description of their use and whether it is intended or suitable for development, testing, or both.

Table 6-55. AFDKO tools, their purposes, and descriptions

Tool name	Purpose	Description
autohint	Development	Apply hinting to Type 1 or Type 2 charstrings
checkOutlines	Development and testing	Detect and fix outline issues, such as overlapping or reversed paths
compareFamily	Testing	Compare family-level attributes of multiple fonts
detype1	Development	Decompiles a Type 1 font into human-readable form—see type1
MakeOTF	Development	OpenType font compiler
mergeFonts	Development	Merge multiple fonts, add/remove glyphs, and change glyph names
rotateFont	Development	Rotate/shift glyphs and change glyph names
sfntdiff	Testing	Compare OpenType (aka 'sfnt') tables
sfntedit	Development and testing	Add, remove, replace, or delete OpenType (aka 'sfnt') tables
spot	Testing	'sfnt' resource analysis
tx[a]	Development and testing	Font conversion and analysis
type1	Development	Compiles a Type 1 font from human-readable form—see detype1

a. Short for *Type eXchange*.

Careful and effective use of the AFDKO tools can be used to establish a complete font production workflow. The mergeFonts tool, for example, is a very rudimentary CIDFont resource compiler, able to merge multiple name-keyed Type 1 fonts into a single CIDFont resource. Precise use of this tool, specifically through the use of carefully written mapping files that drive the specification of hint dictionaries and the mapping of glyph names to CIDs, can result in a truly industrial-strength CIDFont resource compiler, as opposed to a rudimentary one.

The document entitled *AFDKO Version 2.0 Tutorial: mergeFonts, rotateFont & autohint* (Adobe Technical Note #5900), which is included in AFDKO as part of its documentation, provides insights into making more effective use of some of its more powerful tools.

AFDKO and subroutinization

I have already pointed out that CFF has a size advantage over TrueType, given its use of Type 2 charstrings. Given the same number of glyphs and the same typeface design, the CFF representation is usually half the size of the equivalent TrueType representation. The MakeOTF tool that is included in AFDKO takes this size advantage a step further and allows a CFF to be subroutinized. Common elements, whether they are complete paths or path segments, are detected and stored as subroutines. The algorithm to detect and collect subroutines is recursive, meaning that as the number of glyphs increase, so does the

amount of time that subroutinization takes. Subroutinization is also memory-intensive, meaning that the more RAM you have, the better. For CJKV fonts with thousands of glyphs, I suggest a minimum of 1 GB of RAM. More is better. Depending on the design, subroutinization can further reduce the size of the CFF by 5 to 50%. In my experience, Korean fonts, with thousands of glyphs for hangul, get the best subroutinization results, close to 50% reduction in size. For ideographs, it depends on the design. Some designs achieve only 5% reduction, and some as much as 35%.

The subroutinization that MakeOTF performs has limitations. First, there are two types of subroutines: *global* and *local*. The global subroutines apply to the entire CFF. The local ones are specific to each hint dictionary. For name-keyed CFFs, there is no difference between global and local subroutines, because they specify only one hint dictionary. CID-keyed CFFs can have multiple hint dictionaries. The subroutine limit is 64K – 3, meaning 65,533 subroutines. This limit applies to the global subroutines, and also to each set of local subroutines in the case of CID-keyed CFFs. If the number of global or local subroutines exceeds the subroutine limit, there are techniques for reducing the number of subroutines, at least for CID-keyed fonts with multiple hint dictionaries. I have found through experience that splitting the largest hint dictionary into two smaller hint dictionaries can result in redistribution of the global and local subroutines. As an example, for the longest time, I was unable to subroutinize a font called AdobeSongStd-Light. I was perplexed as to why. This is an Adobe-GB1-5 font with 30,284 glyphs. Its Hanzi hint dictionary included 27,701 glyphs. I separated 18,686 of its glyphs into a new hint dictionary called HanziPlus. I was able to subroutinize this font only after splitting the Hanzi hint dictionary. Table 6-56 details the number of subroutines that resulted, showing only the two relevant hint dictionaries.

Table 6-56. Global versus local subroutines—AdobeSongStd-Light

Subroutine type	Number of glyphs	Number of subroutines
Global	30,284	62,515
Local	9,015—Hanzi	3,502
Local	18,686—HanziPlus	35,270

As you can see, the number of global subroutines for this particular font is barely under the 64K – 3 limit. I'd like to emphasize that without splitting the Hanzi hint dictionary into two separate hint dictionaries, which resulted in a redistribution of the global versus local subroutines, it was simply not possible to subroutinize this particular font.

Perhaps of interest, some earlier software cannot handle fonts with more than 32K – 3 (32,765) global or local subroutines, and some base the limit on the number of global subroutines plus those from any one hint dictionary, meaning that any "global plus local" subroutine pair cannot exceed the 64K – 3 limit. For this reason, there may be times when it is best to keep the number of global and local subroutines to a minimum.

In order to keep the number of global and local subroutines under the 32K – 3 limit, a similar technique was applied to the Kozuka Mincho Pr6N fonts. The hint dictionary structure for Adobe-Japan1-6 was described earlier in this chapter, specifically in Table 6-26, and the default structure assigns 14,943 glyphs to the Kanji hint dictionary. In order to redistribute the global and local subroutines, this hint dictionary was split into two, keeping 10,216 glyphs in the Kanji hint dictionary and assigning 4,727 to a new hint dictionary called KanjiPlus. Table 6-57 lists the results of this effort for each of the six weights of Kozuka Mincho Pr6N.

Table 6-57. Global versus local subroutines—all six weights of KozMinPr6N

	Version	ExtraLight	Light	Regular	Medium	Bold	Heavy
CFF size	6.001	5116775	5769975	5794548	5902144	6017616	5891456
	6.002	5108840	5761185	5793722	5898303	6014263	5884416
Global	6.001	1,714	1,582	1,573	1,540	1,557	1,708
	6.002	28,412	30,814	29,929	28,991	27,119	25,360
Kanji	6.001	31,869	35,151	34,070	33,279	31,206	29,271
	6.002	7,189	7,418	7,147	7,117	6,892	7,124
KanjiPlus	6.002	1,862	1,876	1,787	1,713	1,695	1,801

Note how the splitting of the Kanji hint dictionary for version 6.002 significantly altered the distribution of global and local subroutines, and moderately affected the size of the 'CFF' table, all in a positive way. The number of global subroutines climbed considerably, but they managed to stay below the lower 32K – 3 limit. Likewise, the number of local subroutines for the now-smaller Kanji hint dictionary dropped significantly.

To check for the presence of subroutines in an OpenType font with a 'CFF' table and to display the number of global and local subroutines on a per–hint dictionary basis, the following `tx` command line can be used (substitute the `<font>` portion of the command line with the name of a file that corresponds to an OpenType font with a CFF table or a stand-alone CFF table instantiated as a file):

```
% tx -dcf -1 -T gl <font> | grep '^--- FD|^count'
```

Of course, it is possible to disable subroutinization by using the MakeOTF tool's `-nS` option. Unfortunately, subroutinization is a binary setting for this tool. In other words, it is either enabled or disabled, and there are no adjustments or fine-tuning possible. Because CFF already provides a significant size advantage, my advice is to disable subroutinization when in doubt.

TTX/FontTools

Another incredibly useful font tool to which I would like to draw your attention is called TTX.* Its name suggests a relationship to the *tx* tool that is included with Adobe Systems' AFDKO, but they are entirely different. They share one thing in common, though. Both are intended to manipulate fonts, but in completely different ways. TTX is written in Python, which is an incredibly powerful scripting language, and it is open source. TTX was developed and is maintained by Just von Rossum of the company Letraset.

TTX allows one to convert OpenType or TrueType fonts to and from an XML representation, the purpose of which is obviously to modify the contents of such fonts. Needless to say, this tool allows one to wield incredible power due to the richness made possible through an XML representation, and the ability to perform round-trip conversion. In addition to the ability to modify OpenType or TrueType fonts, TTX obviously allows font developers to create new fonts.

Font Format Conversion

There are utilities, such as FontLab's TransType, that allow users to convert fonts from one format to another†—for example, from TrueType to Type 1 format and vice versa. TrueKeys, developed by Xiaolin Allen Zhao (赵小麟 *zhào xiǎolín*), is a shareware program for changing the encoding of TrueType fonts.‡ For example, TrueKeys can change a Shift-JIS–encoded TrueType Japanese font to Unicode encoding by modifying its 'cmap' table accordingly. Most font editors, such as FontLab Studio and AsiaFont Studio, can also perform font format conversion tasks. Which tool is best for you honestly depends on the nature of your needs.

There are two important issues to bear in mind before performing any level of font format conversion:

- The license agreement included with most commercial font software may state that the data must not be converted to other formats—this is a legal issue.
- The font format conversion process almost always strips out (or renders ineffective) the "hinting" information that allows the font data to rasterize better at smaller point sizes and at low resolutions—this is a quality issue.

These issues may be reasons for not modifying font data. Outline (as opposed to bitmapped) fonts are considered executable software. In other words, fonts are considered programs—for example, the PostScript language, or a subset thereof, constitutes a complete programming language—and are therefore copyrightable, at least in the U.S. So why do these font format conversion utilities exist? Well, not all fonts include restrictions that

* *http://sourceforge.net/projects/fonttools/*

† *http://www.fontlab.com/font-converter/transtype/*

‡ *http://www.unidocsys.com/TrueKeys.shtml*

prevent or preclude conversion to other formats, and there are plenty of fonts in the public domain.

Admittedly and thankfully, the need to convert fonts from one format to another has diminished over the years. Part of this is due to OpenType and the trend toward building such fonts. My own font format conversion needs typically fall into the desire to convert a font based on a legacy font format into an OpenType font.

Gaiji Handling

Gaiji,* sometimes referred to as *external characters*,† are best described as glyphs that the user cannot enter. Gaiji is a Japanese word, written 外字 (*gaiji*), and like the Japanese word *sushi*,‡ it has become somewhat internationalized due to its broad and sweeping implications. The prototypical gaiji is an ideograph, but a gaiji can be a generic symbol, a corporate logo, or a new currency symbol. It may or may not be in Unicode. It may be in a known glyph set. It may be in a font, but simply not accessible due to IME limitations or because it is simply not encoded. What is key is that the desired glyph is not available in an installed font (meaning that it is not in the selected font) or not available in its typeface style.

In Appendix E we will explore the vendor-specific characters that are defined within vendor-specific character sets. I state there that these characters do not convert very well between different encodings. These fall outside the realm of standard character sets, which we can call *system-defined characters* (SDCs). Gaiji are typically used in proper names, for logos, and for historical or technical purposes, and can be separated into two distinct categories:

- User-defined characters (UDCs)
- System-specific characters (SSCs)

Gaiji are not used very frequently, but when they are needed, their support becomes critical. Because support for gaiji is not yet widespread, various solutions or techniques have been fielded or employed over the years. The expansion of the Adobe-Japan1-*x* character collection, which has become the foundation or basis for OpenType Japanese fonts, has diminished the need for gaiji to some extent. But, given the open-ended nature of the gaiji problem, even Adobe-Japan1-6 with its 23,058 glyphs cannot hope to solve it.

UDCs are those characters that a single user or entity creates for personal or private use. This class of gaiji requires a solution no matter what platform is used.

* 外 (*gai*) means "external," and 字 (*ji*) means "character."

† What's the opposite of external characters? You guessed it, internal characters! We can call these critters *naiji* (内字 *naiji*) in Japanese. In case it is not clear, this footnote is intended to be mildly anecdotal.

‡ Sushi can be represented by すし, 寿司, 鮨, or even 鮨, all of which are read *sushi*. Interestingly, and very appropriate to this discussion, the last representation, 鮨, is often treated as a gaiji. Furthermore, the word *sushi*, like the word *gaiji*, has no plural form.

SSCs are those characters that are considered standard on one OS, but not across multiple OSes. In a closed system or environment, there is effectively no difference between SDCs and SSCs, but the moment one attempts to interoperate between OSes, they become very different entities, with very different results. For example, the traditional-form kanji 黒 (meaning "black") was considered an SDC under Windows 3.1J and Windows 95J, but because it is also an SSC, and specific to Windows, it was not available under other OSes, such as KanjiTalk or JLK (Mac OS). This particular kanji was a gaiji, specifically a UDC, as far as Mac OS is concerned. This has changed with Mac OS X, and this character is now considered standard. And, to be absolutely clear, gaiji are not limited to ideographs—many are symbols or logos.

Both types of gaiji pose problems when information interchange is necessary. The target encoding or character set may not support certain characters that are used in a document. This is especially true for user-defined characters, which are usually specific to a single person's environment. Even JIS X 0212-1990 characters can be considered gaiji if the target system does not support their use.* Some might imagine that Unicode is a solution to this problem. While this appears to be true at first glance (after all, there are now over 70,000 ideographs from which to pick and choose!), there are tens of thousands of CNS 11643-1992 hanzi that were not part of Unicode's URO. But now, thanks to CJK Unified Ideographs Extensions A and B, CNS 11643-1992 is more or less fully supported through the use of Unicode.

The success of printing or displaying gaiji depends on the fonts you are using. If the font that you have chosen includes a glyph for the gaiji, you should get proper output. If you need to use a vendor character set on that vendor's operating system, you most likely have access to fonts that correspond to it.

A problem arises when printing user-defined characters. In the case of bitmapped fonts, it is usually possible to create a new bitmapped character, then add it to the repertoire of characters in the font. However, creating a new outline character is a bit more tedious as it requires much more design skill (you may even have to create a corresponding bitmapped character!). Font design software, such as FontLab Studio and Fontographer, are excellent tools for creating your own bitmapped and outline fonts.

The Gaiji Problem

Solving the persistent problem of gaiji is not so trivial, primarily because the problem involves a very broad set of issues, such as the following:

- How to design or create gaiji
- How to encode gaiji, if at all

* For example, in Shift-JIS encoding, which does not support the JIS X 0212-1990 character set, they are considered gaiji. Adobe Systems once released a Mac OS font product called 平成明朝 W3 外字 (*heisei minchō W3 gaiji*) that included glyphs for all of the characters in JIS X 0212-1990 as a series of one-byte–encoded Type 1 fonts.

- How to enter gaiji into documents or text
- How to display or print gaiji
- How to exchange documents or text that contain gaiji
- How to search for gaiji in text

So, what can one do about solving these problems? Unfortunately, coming up with an elegant and working solution is tough and nontrivial. The main problem is one of portability, and a solution would need to somehow allow for the successful transmission of UDCs and SSCs to systems or environments that do not support them. Even large character sets do not have complete coverage of all SSCs, and that doesn't even touch upon the problem of supporting UDCs. A necessary step in finding a solution is to embed glyph data, in the form of outlines that can be scaled for different point sizes and resolutions, into the documents that use them, so when they are transmitted, they come along for the ride. Such a solution must include a mechanism for detecting which characters are user-defined so that it is unambiguous as to which ones need to be embedded.

The first person or company to implement a platform-independent solution to the gaiji problem will be well rewarded by the computer industry. In my opinion, Adobe Acrobat and OpenType were steps in this direction, but SING, which is described in the following section, represents the one true gaiji solution, because it effectively breaks away from the characteristics that have always bound legacy gaiji solutions to closed environments. More information about Adobe Acrobat and PDF is available in Chapter 12. OpenType fonts were covered earlier in this chapter.

Thanks to Unicode, which now includes over 100,000 characters, the need for gaiji has diminished to some extent; for a small class of users, the need has been erased. But, the open-ended or dynamic nature of the gaiji problem means that a solution is still necessary, no matter how many characters are included in Unicode.

In some cases, the selected font include the desired glyph, but there may be reasons why it cannot be entered. One obvious reason is because it may not be encoded. It may be a variant form of character. Specific OpenType GSUB features can be used to enter such glyphs, but only if the application is OpenType-savvy. Another not-so-obvious reason is that the input method simply does not support the desired character. The desired glyph may correspond to an obscure character, and the more obscure a character is, the less chance that an input method will allow its entering, at least through convenient means such as by reading.

Gaiji have been historically implemented as single-byte Type 1 or TrueType fonts, because for the longest time, the commonly available font editors were able to generate only single-byte fonts. Composite font technologies then allowed these single-byte fonts to become part of an existing font through the definition of a virtual font. Now, Unicode offers even more possibilities. Still, all of these font-based solutions share one common characteristic: the glyphs must encoded, by hook or by crook. Gaiji can be encoded in the user-defined region of legacy encodings, or in the PUA of Unicode—*by hook*. Gaiji can

also be encoded at code points that have already been assigned to other characters, which is a technique often referred to as *poaching—by crook*. Liberating gaiji from the encoding requirement is thus key in developing a genuine solution.

SING—Smart INdependent Glyphlets

SING (Smart INdependent Glyphlets) is an initiative and technology developed by Adobe Systems that is specifically designed to solve the gaiji problem.* SING's approach to solving the gaiji problem is to distribute and use small font-like objects that are referred to as glyphlets. A SING glyphlet is effectively a single-glyph font that "looks and feels" much like an OpenType font, but intentionally lacks key 'sfnt' tables, such as 'name' and 'OS/2', that would otherwise enable it to appear as selectable font in application font menus. In addition, the file extension that is used, *.gai*, also helps to explicitly identify SING glyphlets.

A SING glyphlet includes only one meaningful glyph, at GID+1, along with the obligatory *.notdef* glyph, required by today's font formats, at GID+0. SING glyphlets that correspond to characters that are expected to have a vertical variant should include a second functional glyph, at GID+2, which is subsequently assumed to be the vertical variant.

Because a SING glyphlet includes only one meaningful glyph, it is small and lightweight. A typical SING glyphlet, including a rich collection of metadata, is less than 5K in size.

Because a SING glyphlet is small and lightweight, it can be transmitted easily and quickly. A SING glyphlet is thus intended to be portable, and embedded into the documents in which it is used. In essence, SING glyphlets are sticky to the documents in which they are used. Portability is an issue that plagues legacy gaiji solutions.

I strongly feel that SING will succeed where other legacy gaiji solutions have failed, because one of the most troubling aspects of legacy gaiji solutions happens to not be a requirement for SING: *the requirement or need to somehow encode the glyphs*. If the glyph that corresponds to a SING glyphlet has an appropriate code point in Unicode, but non-PUA, it can be encoded. But, SING does not require that a glyphlet be encoded. Instead, the metadata that is defined and encapsulated in its 'META' table is intended to identify the glyph through means other than encoding. Some of these means are intended to be used by IMEs for the purpose of easing or trivializing keyboard entry, such as reading data and variant relationships.

The following bullet points effectively summarize the key benefits of SING:

- A SING glyphlet includes only one meaningful glyph, and is therefore lightweight.
- A SING glyphlet is portable and is designed to be embedded in the documents in which it is used.

* *http://www.adobe.com/products/indesign/sing_gaiji.html*

- A SING glyphlet does not require an encoding to be used.
- A SING glyphlet can include a rich collection of metadata to better identify its glyph and to allow users to more easily input them into documents.

Adobe InDesign CS2 (aka InDesign version 4.0) was the very first application to support SING, and does so through the use of a library called Tin, which augments installed fonts with glyphlets that are designed to work with them.* There is also an application called Adobe SING Glyphlet Manager (ASGM) that installs and manages SING glyphlets and makes use of the SING Library.

There are two ways for applications to implement support for SING. The first and perhaps the easiest way to implement SING is what Adobe InDesign has done, specifically to augment installed fonts with the appropriate glyphlets, which means that the application thinks that the SING glyphlets are genuinely included in the installed fonts and treats them no differently. Of course, the SING glyphlets, although augmented to installed fonts in memory, are sticky to the document and travel with it to ensure portability. The second way does not involve font augmentation, and obviously requires that the application support a new class or type of text object. Depending on the nature and complexity of the application, along with available resources, one of these two ways of supporting SING is best.

In addition to the obvious application of employing SING as a powerful cross-platform gaiji solution, SING can also be used as a mechanism for delivering updated or corrected glyphs for existing glyphs in fonts that are already in users' hands. This glyph update mechanism can be effected at the glyph (GID or CID) or encoding (Unicode) levels, or both.

'META' and 'SING' tables

SING glyphlets include two tables that are not present in OpenType fonts, specifically the 'META' and 'SING' tables, briefly described as follows:

- The 'META' table is designed to encapsulate a very rich collection of metadata that can be used to identify, classify, and categorize the glyph of the SING glyphlet. For ideographs, the metadata can include the number of strokes, the indexing radical, dictionary references, and so on. A lot of thought went into SING, and although the 'META' table includes a large number of predefined fields, it has a mechanism that allows user-defined metadata to be included.
- The 'SING' table specifies the version number of the SING glyphlet, embedding permissions (equivalent to the *OS/2.fsType* field, and thus should be set to `0`, the most liberal setting) and other higher-level attributes that are not considered to be metadata per se.

* Note that "Tin" is neither an acronym nor abbreviation.

Table 6-58 lists some of the more important *META.ID* fields, along with a brief description of their purpose.

Table 6-58. SING META.ID fields and descriptions

META.ID	Name	Description
0	*MojikumiX4051*	JIS X 4051 composition class
1	*UNIUnifiedBaseChars*	Unicode value, if appropriate—"a" and "v" specifiers indicate actual or variant relationship
2	*BaseFontName*	The font with which the glyph is to be used
3	*Language*	The language and locale of the glyph—"ja-JP" is used for Japanese
10	*StrokeCount*	If an ideograph, the total number of strokes
11	*IndexingRadical*	If an ideograph, the indexing radical expressed as a Unicode character in the range U+2F00 through U+2FD5
12	*RemStrokeCount*	If an ideograph, the number of remaining strokes
20	*CIDBaseChars*	Registry, Ordering, and CID value, if appropriate—"a" and "v" specifiers indicate actual or variant relationship
21	*IsUpdatedGlyph*	Indicates whether the glyph is a replacement at the Unicode or CID level
24	*LocalizedInfo*	Localized equivalents of name.IDs 0, 3, 7–14, and 19
25	*LocalizedBaseFontName*	Localized form of META.ID=2
100–114	*ReadingString*	Readings, whether transliterated or otherwise, along with glyph names
200–202	*AltIndexingRadical*	Alternate indexing radical systems—see META.ID=11
250–257	*AltLookup*	Specific ideographic dictionary references
500–505	*UNICODEproperty*	Unicode character properties, such as General Category, Bidirectional Category, Combining Class, and so on
602	*UNIDerivedByOpenType*	The Unicode value and OpenType GSUB feature that results in this glyph
603	*UNIOpenTypeYields*	The Unicode value that results from this glyph when the OpenType GSUB feature is specified
604	*CIDDerivedByOpenType*	The Registry, Ordering, and CID value and OpenType GSUB feature that results in this glyph
605	*CIDOpenTypeYields*	The Registry, Ordering, and CID value that results from this glyph when the OpenType GSUB feature is specified
20000–32767	*VendorSpecific*	Vendor-specific metadata

Table 6-59 provides some examples, expressed as XML, for specifying *META.ID* fields and their contents.

Table 6-59. SING META.ID fields expressed as XML

XML	Interpretation
<SING:MojikumiX4051> 12 </SING:MojikumiX4051>	JIS X 4051 Class #12
<SING:UNIUnifiedBaseChars> 佪󠄀;a </SING:UNIUnifiedBaseChars>	U+4F6A and <U+4F6A, U+E0100>
<SING:UNIUnifiedBaseChars> 徊;v </SING:UNIUnifiedBaseChars>	Variant of U+5F8A
<SING:BaseFontName> KozMinStd-Bold </SING:BaseFontName>	Augment KozMinStd-Bold
<SING:BaseFontName> KozMinPro-Bold </SING:BaseFontName>	Augment KozMinPro-Bold
<SING:BaseFontName> RyoDispPlusN-SemiBold </SING:BaseFontName>	Augment RyoDispPlusN-SemiBold
<SING:BaseFontName> RyoDispPlusN-Bold </SING:BaseFontName>	Augment RyoDispPlusN-Bold
<SING:StrokeCount> 8 </SING:StrokeCount>	Eight total strokes
<SING:IndexingRadical> ⼈;2 </SING:IndexingRadical>	Radical #9—two strokes
<SING:RemStrokeCount> 6;⼈ </SING:RemStrokeCount>	Six remaining strokes—Radical #9
<SING:CIDBaseChars> Adobe-Japan1;16781;a </SING:CIDBaseChars>	Adobe-Japan1 CID+16781
<SING:CIDBaseChars> Adobe-Japan1;4790;v </SING:CIDBaseChars>	Variant of Adobe-Japan1 CID+4790
<SING:IsUpdatedGlyph> 0 </SING:IsUpdatedGlyph>	0—not an updated glyph

The examples provided in Table 6-59 serve to demonstrate that it is possible to include, in a single SING glyphlet, multiple instances of some *META.ID* fields. In fact, including multiple instances of some *META.ID* fields is considered good practice and leads to richer metadata.

Building SING glyphlets

There are two types or classifications of SING glyphlets: typeface-specific and generic. Typeface-specific glyphlets can be associated with more than one font instance through the use of multiple *META.ID=2* (*BaseFontName*) and *META.ID=25* (*LocalizedBaseFontName*) fields. Generic glyphlets, defined by including a single *META.ID=2* field that is set to the string Generic, are amalgamated into a single font instance, at least in Adobe InDesign CS2 and later. In the Japanese version of InDesign CS2 and later, this amalgamated font instance is called 外字 (*gaiji*). For all other languages, this font instance is called ExtraGlyphlets.

In terms of tools for building SING glyphlets, Adobe Systems provides in their *Glyphlet Development Kit*[*] (GDK) a command-line tool called *cvt2sing* that serves as a very robust and industrial-strength SING glyphlet compiler, which uses a source font and an XML file that encapsulates the metadata and other information for the 'META' and 'SING' tables. Adobe Illustrator CS2 (aka Adobe Illustrator version 12.0) and higher includes a plug-in called the Glyphlet Creation Tool (GCT) for creating SING glyphlets. Interestingly,

* *http://www.adobe.com/devnet/opentype/gdk/topic.html*

although GCT is used by Adobe Illustrator to create SING glyphlets, the SING glyphlets that it creates cannot be used by the application (at least, up through and including Adobe Illustrator CS4). One of the first third-party SING glyphlet creation tools is FontLab's SigMaker, specifically version 3.0 and higher.* It runs on Mac OS X and Windows. Another is a Windows application called SINGEdit, but the SING glyphlets that it creates can be used on Mac OS X or Windows, and are effectively cross-platform.†

As soon as the Tin Library that is used by Adobe InDesign is enhanced in terms of performance and its use is propagated to other applications, such as Adobe Acrobat, Adobe Flash, and Adobe Illustrator, I predict that SING will become the preferred solution to the gaiji problem.

Interestingly, Adobe Acrobat provides minimal SING support, which we can refer to as SING-*compatible*, from its very early versions. When InDesign CS2 or higher is used to create a PDF file from a document that includes one or more SING glyphlets, the glyphs that correspond to the SING glyphlets properly embed into the PDF file. They display and print correctly, but their metadata is lost. At some point, Acrobat clearly needs to become SING-*savvy*, which means that the SING glyphlets, in their entirety, are embedded into PDF files.

More information about building SING glyphlets can be found in the document entitled *SING Glyphlet Production: Tips, Tricks & Techniques* (Adobe Technical Note #5148).

All Your PUA Are Belong To Us

One very important part of SING's specification and its philosophy is that PUA code points are not only discouraged, they're not allowed. All legacy gaiji solutions share one key characteristic, specifically that their glyphs need to be encoded, which means that in the context of Unicode that they are encoded in its PUA. Currently occupied code points can also be poached, which is a technique that is deemed far worse than PUA usage.‡

No one can argue that legacy gaiji solutions do not work, but they tend to work better in closed environments. The more closed an environment is, the more reliably legacy gaiji solutions function. The more open an environment is, the more functionality issues that are exposed.

One important goal of SING is complete functionality in open environments. The nature of PUA code points necessitates closed environments. Of course, supporting SING doesn't come for free. Applications that intend to support SING must be enhanced to either 1) support SING glyphlets as a new type of text-like object, or 2) learn to deal with fonts that do not have static glyph sets.

* *http://www.fontlab.com/font-utility/sigmaker/*

† *http://musashi.or.tv/singedit.htm*

‡ *Poaching* is a word that refers to illegal hunting or taking of game. In the context of gaiji, poaching is not illegal, but rather inappropriate.

Ideographic Variation Sequences

To some extent, Ideographic Variation Sequences (IVSes, described at length in Chapter 3) handle what some believe to be gaiji by virtue of the fact that the IVS mechanism allows otherwise unencoded, but somewhat standardized or common, variant forms of ideographs to be encoded through the use of a sequence of two Unicode code points, specifically a Base Character followed by a Variation Selector. For some class of applications, if a glyph is unencoded, it cannot be represented in the documents that it creates. As long as the glyph in question is an ideograph, considered a variant form of an encoded character, and falls under the ISO 10646 "Annex S" unification principles, the possibility of handling it through the use of the IVS mechanism exists. Otherwise, the IVS mechanism would be inappropriate.

For applications and environments that require a plain-text representation or environments in which only plain-text data can persist, such as the form fields of a PDF Form or web browsers, IVSes can handle glyphs that would otherwise require gaiji treatment in such situations. Also, IVSes provide a means of representing the identity of glyphs in plain text, but fall short in solving the gaiji problem, because they do not add new glyphs to existing fonts. Some applications, such as Adobe InDesign, allow users to use and enter *any* glyph in a font, regardless of whether it is encoded or not.

XKP, A Gaiji Handling Initiative—Obsolete

XKP, short for Extended Kanji Processing, is a specification developed by a Microsoft-sponsored consortium for extending operating systems so that developers can access, define, create, display, and print user-defined characters in a standardized way. Moving from a Shift-JIS to Unicode (UCS-2 encoding) architecture effectively increases the total number of available UDCs from 1,880 to 6,400.*

While XKP was originally developed for Japanese-specific purposes, its techniques can be easily adapted for use with other CJKV locales. XKP doesn't simply provide a method to encode user-defined characters, but provides other information to aid input and searches. Table 6-60 lists the sort of information that can be associated with each UDC.†

Table 6-60. XKP's user-defined character information

Field name	Example	Description
Yomi	Yomi=ケン　つるぎ	Readings
Busyu	Busyu=18	Radical

* If the code points beyond the BMP were to be taken into account, this 6,400 PUA code point figure would become 137,468, because Planes 15 and 16 each provide an additional 65,534 PUA code points . See Chapter 3 for more details.

† Bonus points (or extra credit) will be awarded to those who can figure out what the glyph is for the character described in Table 6-60.

Table 6-60. XKP's user-defined character information

Field name	Example	Description
Kakusu	Kakusu=9	Number of strokes
URP	URP=0x5263	Unified Representative Point[a]
FontPath	FontPath=STKAIREG.TTF	The filename of the UDC font
FontName	FontName=华文楷体	The name of the UDC font
FontCodePoint	FontCodePoint=0xBDA3	The code point in the UDC font

a. A code point reference that represents the "parent" or standard version of the UDC. It effectively means that the UDC will be treated as a variant of the parent character for purposes such as searching.

Once you have properly registered your UDCs, you can then start making use of them in your documents. There are two ways in which UDCs can be referenced:

- As a user-defined code point, in UCS-2 encoding's Private Use Area (PUA), such as the value U+E000
- As an ampersand-prefixed, eight-digit hexadecimal string, such as "&00000001"

XKP also defines three character sets—Basic, Extended, and Compatibility—and three implementation levels—1, 2, and 3—for Japanese. These are referred to as SDCs in XKP. Two of the character sets refer to Japanese subrepertoires specified in JIS X 0221-1995. Table 6-61 lists these implementation levels, along with references to the Japanese subrepertoires specified in JIS X 0221-1995.

Table 6-61. XKP's JIS X 0221-1995 implementation levels

Implementation level	Character set	JIS X 0221-1995 reference
1	Basic	1, 6, and 7
2	Basic, Compatibility	1, 6, 7, and Windows 95J-specific characters
3	Basic, Compatibility, Extended	1, 2, 3, 4, 6, and 7

The only Japanese subrepertoire that is not covered by XKP is number 5, which represents the 8,745 remaining CJK Unified Ideographs (aka the URO) in Unicode.

More information about XKP is available in the form of documentation and an SDK (*Software Developer Kit*).*

Adobe Type Composer (ATC)—Obsolete

Although this is primarily of historical interest, Adobe Type Composer (ATC) is software that allowed end users to rearrange the glyphs of Japanese typefaces, and also let them add

* *http://www.xkp.or.jp/*

new glyphs to the same typefaces in the user-defined region of Shift-JIS encoding. Adobe Type Composer was specifically designed to work only with PostScript Japanese fonts, and included three basic functionalities:

- Add new glyphs to an existing Japanese font in the user-defined region of Shift-JIS encoding—<F0 41> through <FB FC>.
- Substitute the glyphs that correspond to specific character classes from one Japanese font for another Japanese font, such as kana and symbols.
- Adjust the baseline for the built-in or substituted proportional Latin glyphs so that they are better aligned with the glyphs that correspond to the kana and kanji.

The resulting font can be considered a "virtual" font that is installed at the system level, meaning that it is available to the OS and all applications. This font is virtual in the sense that it contains absolutely no glyphs of its own, but is simply a recipe for combining parts of other fonts into a new font resource.

The following are descriptions of what each of Adobe Type Composer's three primary functions were intended to perform:

- First, you could add up to a maximum of 2,256 glyphs, defined as 12 fonts containing up to 188 glyphs each, to installed PostScript Japanese fonts. In other words, you could either create or purchase additional glyphs, in the form of Type 1 fonts that include up to 188 glyphs, specifically encoded in the ranges 0x40 through 0x7E and 0x80 through 0xFC, and then add them to a font, specifically in the user-defined region of Shift-JIS encoding.*
- Second, you could substitute the kana glyphs or some select symbols with those of a different typeface style. Of course, this mixing of typefaces could be performed by most word-processing software simply by selecting the kana and changing them to another font, but this clearly is a very tedious task for longer documents. But what about software, such as simple text editors, that allows only a single font selection per document? That is one of those circumstances when this functionality became useful.
- Third, the relative baseline of the proportional Latin glyphs could be shifted, whether they are the glyphs that are built-in or referenced from another font. The proportional Latin glyph coverage was limited to ASCII, which was in turn due to the limitations imposed by Shift-JIS encoding itself. The shifting of the baseline was intended to improve the aesthetics when a different set of proportional Latin glyphs were used.

Adobe Type Composer clearly made it more feasible for type foundries, along with individual typeface designers, to design kana-only fonts that were intended to be mixed with kanji and other symbols from a different Japanese typeface. In fact, a large number of kana-only fonts were released by a variety of Japanese type foundries, some of which

* Note that the encoding ranges directly correspond to the second-byte encoding ranges of Shift-JIS encoding.

included prebuilt ATC fonts that serve to simplify the overall user experience. Of course, kana-only fonts can be used standalone, but the user experience is much better if they are combined with a font that includes a complete set of kanji, given that typical Japanese text includes both kana and kanji.

Adobe Type Composer rearranged only 90pv-RKSJ-H– and 83pv-RKSJ-H–based PostScript fonts, both of which were accessible only on Mac OS. This means that the user-defined character area corresponds to only Shift-JIS–encoded (aka "RKSJ") fonts.

Note that Adobe Type Composer was simply a frontend to a very powerful font rearrangement technology available on all PostScript devices capable of supporting CID-keyed fonts. A PostScript ProcSet (Procedure Set—a fancy name for a PostScript program) called AdobeTypeComposer is made available on these PostScript devices. Later in this chapter you will learn the structure of a rearranged font resource and how to create your own.

Although Adobe Type Composer is no longer used, due to its limitations, comparable rearranged font functionality, referred to as Composite Fonts, is now available in higher-end applications. This is covered in the next section.

Composite Font Functionality Within Applications

Some applications provide functionality, comparable to what Adobe Type Composer offered, called *Composite Fonts*, which are amalgamated fonts that are implemented as font instances that are accessible only within the specific application. These Composite Fonts appear as though they were genuine font instances, but they are virtual fonts that are defined through the use of a recipe that specifies a base, primary, or parent font, along with additional fonts.

As an example, consider that kana characters constitute over 70% of typical Japanese text, meaning that one can dramatically change the "look and feel" of a document simply by using an alternate set of kana glyphs from another font, and leaving the kanji as is. Many Japanese type foundries provide kana-only font products that are specifically designed to be used in the context of a Composite Font. This technique is a relatively economical way to enhance the functionality of a smaller Japanese type library. Carefully observe the Japanese text in Table 6-62. The block of text on the left is set in a single Japanese typeface, whereas the same block of text on the right substitutes a different typeface for the kana and punctuation glyphs. Interestingly, both blocks of text are set using proportional metrics. The proportional nature of the kana glyphs is clearly more striking for the text block on the right.

Table 6-62. The effect of changing the kana design

KozMinPr6N-Regular	KozMinPr6N-Regular and TBRyokanMStd-M
普通の和文フォントは明朝体とゴシック体ですが、スペシャルなフォントもあります。例えば、丸ゴシック体、楷書体、毛筆体、および教科書体というフォントに人気があります。	普通の和文フォントは明朝体とゴシック体ですが、スペシャルなフォントもあります。例えば、丸ゴシック体、楷書体、毛筆体、および教科書体というフォントに人気があります。

See what I mean? The abundance of kana clearly allows you to change the "look and feel" of a Japanese document. Of course, implementing this functionality through the definition of a Composite Font is merely a convenience mechanism. It is obviously possible, though a somewhat tedious task, to individually change the font for each character as appropriate.

Adobe FrameMaker, Adobe Illustrator, Adobe InDesign, Adobe PageMaker, Canon EDICOLOR, and QuarkXPress are examples of applications that provide users with the ability to create Composite Fonts whose recipes begin by selecting a base font, and to allow substitution by character class, such as substituting the Latin and kana glyphs of the base font with those of a different font. For some of these applications, the Composite Font definitions can be exported to a file, which can then be provided to another user.

Canon EDICOLOR, perhaps as an exception, also allows the user to specify fonts for two categories of gaiji: system and user. See the description of EDICOLOR in Chapter 7 for more details.

Gaiji Handling Techniques and Tricks

When one needs to enter into documents some glyphs that are outside a supported character set or glyph set, there are several proven techniques. None of these techniques provide a high degree of information interchange, but do result in a correctly displayed or printed page. Following are the techniques that I used to produce the first edition of this book:

- Supporting special glyphs not found in common character sets
- Supporting multiple character sets on what is effectively a single-encoding system

The first technique, for supporting special glyphs, involves producing one or more standard one-byte–encoded Type 1—or TrueType, if that's your preference—fonts containing all the special glyphs you require. This can include modifying existing fonts, such as my modification of the ITC Garamond family to make the glyphs that correspond to

macroned vowels available. Once you have built such fonts, you then need to decide how to access their glyphs. There are effectively two choices available to you:

- Use the Type 1 (or TrueType) fonts in a standalone manner
- Add the Type 1 (or TrueType) fonts to the user-defined region of an existing CJKV font

The first choice is by far the easiest to implement, because fonts function as standalone by default. The second choice requires a tool for adding these special characters to existing fonts. Adobe Type Composer is such a tool designed to interact with PostScript fonts, but is currently limited to Japanese and Mac OS. It was discussed in an earlier section of this chapter.

The second technique, for supporting multiple character sets, offers some degree of information interchange. Suppose that you prefer to work in a single-locale environment, such as Mac OS-J or Mac OS with JLK, but you need to produce a rather large, say book-length, document that includes glyphs that correspond to Chinese, Korean, and Vietnamese character sets.* You could obtain and install the appropriate system resources, but for some character sets, such as JIS X 0212-1990, none exist for Mac OS. What should you do? You should have learned in Chapter 3 that almost all legacy CJKV character sets are fundamentally based on a 94×94 matrix with some exceptions. My solution for producing the first edition of this book was to first acquire CIDFont resources for the desired CJKV character sets. I then created Shift-JIS–encoded CMap resources for each character set, sometimes separating character sets into separate planes when necessary, to be compatible with Shift-JIS encoding. Table 6-63 details the character sets I needed to support the first edition of this book, along with the CIDFont and CMap resources that I used.

Table 6-63. Shift-JIS–encoded CJKV fonts

Character set	CIDFont resource	CMap resource
GB 2312-80	STSong-Light	GB-RKSJ-H
GB/T 12345-90	STSong-Light	GBT-RKSJ-H
CNS 11643-1992 Plane 1	MingTiEG-Medium	CNS01-RKSJ-H
CNS 11643-1992 Plane 2	MingTiEG-Medium	CNS02-RKSJ-H
CNS 11643-1992 Plane 3	MingTiEG-Medium	CNS03-RKSJ-H
CNS 11643-1992 Plane 4	MingTiEG-Medium	CNS04-RKSJ-H
CNS 11643-1992 Plane 5	MingTiEG-Medium	CNS05-RKSJ-H
CNS 11643-1992 Plane 6	MingTiEG-Medium	CNS06-RKSJ-H
CNS 11643-1992 Plane 7	MingTiEG-Medium	CNS07-RKSJ-H

* Say, er, uh, like the first edition of this book.

Table 6-63. Shift-JIS–encoded CJKV fonts

Character set	CIDFont resource	CMap resource
CNS 11643-1986 Plane 15	MingTiEG-Medium	CNS15-RKSJ-H
Hong Kong GCCS	HKSong-Regular	HK-RKSJ-H
JIS X 0212-1990	HeiseiMin-W3H	Hojo-RKSJ-H
KS X 1001:1992	HYSMyeongJo-Medium	KS-RKSJ-H
KS X 1002:1991	HYSMyeongJo-SMedium	KS2-RKSJ-H
TCVN 6056:1995	MingTiEGV-Medium	TCVN-RKSJ-H

Although this technique did not allow easy input of characters because the Mac OS-J input methods are geared toward the JIS X 0208:1997 character set, it did help by greatly simplifying and trivializing the building of the character set tables, such as those used in the appendixes of the first edition. It provided for the occasional use of CJKV characters outside of JIS X 0208:1997 throughout that book. In other words, this setup was perfect for my needs. If you feel that this technique would be useful for your own publishing needs, I have made these special-purpose Shift-JIS–encoded CMap resources available for public use.* But, please be aware that these specialized CMap resources are not officially supported nor endorsed by Adobe Systems. And, given the broad extent to which they make use of the poaching technique, I advise against using them unless your environment necessitates their use.

Regardless of which gaiji handling technique you choose to use, you must consider how to input the additional or nonstandard glyphs. If you prefer to enter the glyphs through reading, whether as individual characters or as a compound, the dictionaries that are used by conventional CJKV input methods need to be extended, which can be a tedious task. Otherwise, code input is mandated. But, for some special purposes, code input is completely adequate.

Creating Your Own Rearranged Fonts

Although this section is included primarily for historical purposes, in order to demonstrate how things were done in the past, there are still some environments that can take advantage of, or leverage, rearranged fonts. Although the Adobe Type Composer application is effectively dead, given that its use is limited in terms of supported OS, supported font formats, and supported encodings, one can still craft rearranged fonts. After all, Adobe Type Composer is merely a frontend to rearranged font technology.

In order to use a newly defined rearranged font, one must ensure that the rearranged font resource itself, along with all font components to which it refers, are installed onto the PostScript device, whether it is a PostScript printer or an application that makes use of

* *ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/rksj-cmaps.tar.Z*

a PostScript interpreter, such as Adobe Acrobat Distiller. This effectively means that the following scenarios are possible, and will result in proper execution of a rearranged font resource:

- Explicitly and forcibly download all referenced font components to the PostScript device, either to its hard disk (permanently) or to RAM (temporarily).*
- Download the referenced fonts, followed by the rearranged font resource, within the PostScript stream—the referenced fonts and the rearranged font are available only for the duration of the job, and vanish from memory once it is complete.

Anything else will result in failure to properly execute the rearranged font resource.

The following example is a complete and valid rearranged font resource. Note that it depends on the presence of the PostScript ProcSet called *AdobeTypeComposer*. All CID-capable PostScript devices include this ProcSet.

In a nutshell, this rearranged font resource defines a new PostScript font name called HeiseiMin-W3-Fugu-RKSJ-H. This uses the HeiseiMin-W3-90ms-RKSJ-H CID-keyed font as its template font, then adds the contents of two Type 1 fonts, named MyUDC1 and MyUDC2. Each of these fonts contains 94 glyphs in the range 0x21 through 0x7E, and are added to the first row of the Shift-JIS user-defined range, specifically <F0 40> through <F0 FC>.† I have emboldened nonboilerplate information.

```
%%BeginResource: Font (HeiseiMin-W3-Fugu-RKSJ-H)
%%DocumentNeededResources: ProcSet (AdobeTypeComposer)
%%+ Font (HeiseiMin-W3-90ms-RKSJ-H)
%%+ Font (MyUDC1)
%%+ Font (MyUDC2)
%%IncludeResource: ProcSet (AdobeTypeComposer)
%%IncludeResource: Font (HeiseiMin-W3-90ms-RKSJ-H)
%%IncludeResource: Font (MyUDC1)
%%IncludeResource: Font (MyUDC2)
%%Version: 1.000

1 dict begin /FontName /HeiseiMin-W3-Fugu-RKSJ-H def end

/languagelevel where { pop languagelevel 2 ge } { false } ifelse

{ /CIDInit /ProcSet resourcestatus
```

* In the case of Distiller, the rearranged font resource, along with the font components that it references, must be installed into its appropriate subdirectories.

† PostScript fonts, such as Type 1, typically have three names associated with them: *FontName*, *FullName*, and *FamilyName*. The only one that matters to PostScript (and thus to rearranged font files) is FontName. While FullName and FamilyName are used only for informational purposes, and contain strings as their values (PostScript strings are delimited by parentheses and can contain spaces), FontName contains a font object as its value (PostScript objects are prefixed with a slash, and cannot contain any spaces). Please refer to the following examples taken from a valid Type 1 font:

```
/FullName (Jeffrey Special) readonly def
/FamilyName (Jeffrey) readonly def
/FontName /JeffreySpecial def
```

```
  { pop pop /CIDInit /ProcSet findresource }
  { /AdobeTypeComposer /ProcSet findresource }
  ifelse
}
{ AdobeTypeComposer }

ifelse

begin
%ADOStartRearrangedFont
/HeiseiMin-W3-Fugu-RKSJ-H [ /HeiseiMin-W3-90ms-RKSJ-H
                            /MyUDC1 /MyUDC2 ]
beginrearrangedfont
  1 usefont
    2 beginbfrange
    <f040> <f07e> <21>
    <f080> <f09e> <60>
    endbfrange
  2 usefont
    1 beginbfrange
    <f09f> <f0fc> <21>
    endbfrange
endrearrangedfont
end

%%EndResource
%%EOF
```

Note that the syntax of this resource is quite simple, and it effectively defines the following information:

- A new PostScript font name: HeiseiMin-W3-Fugu-RKSJ-H
- A template font: HeiseiMin-W3-90ms-RKSJ-H
- One or more component fonts: MyUDC1 and MyUDC2*
- The code points in the template font at which the component fonts are to be encoded, also specifying at what code point in the component fonts to begin referencing glyphs

The template and component fonts must be valid font instances, such as a Type 1 (or Type 42, if a TrueType rasterizer is available), OCF, or CID-keyed font. Note that a CIDFont resource by itself, without a CMap resource with which to impose an encoding, is not considered a valid component within the context of a rearranged font file.

* These are typically Type 1 fonts, but can be another composite font, such as a CID-keyed font (that is, a CIDFont plus CMap combination that forms a valid font instance). You can even use the same font that was specified as the template font in order to perform cute tricks such as really rearranging the characters within the font.

If Type 1 fonts are to be used as the component fonts, there are some special considerations. Typical Type 1 fonts use what is known as *StandardEncoding*. The presence of the following line in a Type 1 font header confirms that StandardEncoding is used:

```
/Encoding StandardEncoding def
```

Type 1 fonts that specify *StandardEncoding* can be affected by what is referred to as *font re-encoding*. Font re-encoding typically affects glyphs in the extended ASCII range, specifically those encoded at 0x7F and higher, which is the encoding range used to accommodate the minor (or not so minor, in some cases) difference in encoding for some accented glyphs and symbols on different OSes. In other words, font re-encoding can effectively mangle Type 1 fonts that were not intended to be used as typical Type 1 fonts.

To avoid font re-encoding, most font-editing applications allows fonts to be built using custom encodings. But, one must bear in mind that some OSes and font clients try to be smart, ignore the fact that a font specifies a custom encoding, and base the decision whether to re-encode by inspecting the glyph names.* This undesirable behavior can almost always be circumvented by using nonstandard glyph names. A convention that I prefer to use is to name each glyph according to its encoding, or a portion of its encoding. I do this with the character "c" (meaning "character" or "cell") followed by a two-digit hexadecimal code, such as "*20*" for 0x*20*. This results in glyph names that can range from "c*00*" (for 0x*00*) through "cFF" (for 0xFF).

Let's break apart some of these sections for further analysis and explanation. First, the section that specifies the name of the rearranged font, its template font, and its component fonts:

```
/HeiseiMin-W3-Fugu-RKSJ-H [ /HeiseiMin-W3-90ms-RKSJ-H
                            /MyUDC1 /MyUDC2 ]
```

The string `HeiseiMin-W3-Fugu-RKSJ-H`, which represents the name of the rearranged font, is followed by an array, consisting of three elements, and delimited by brackets. The first array element, `HeiseiMin-W3-90ms-RKSJ-H`, represents the template font. The rearranged font, thus, becomes a copy of this font, and inherits any and all of its properties and attributes, such as glyph complement, writing direction, and so on. The remaining elements, specifically `MyUDC1` and `MyUDC2`, represent the additional font components used for rearrangement, also know as component fonts.

Between the two keywords, `beginrearrangedfont` and `endrearrangedfont`, is the code that performs the actual rearrangement. There is one subsection for each of the component fonts specified in the preceding array—the first element of the array, which represents the template font, is considered the 0th element. Thus, the following rearrangement code refers to the MyUDC1 font, which is the 1st element of the array:

```
1 usefont
  2 beginbfrange
```

* Such OSes often peek at the names of the glyphs within the font to determine whether or not to re-encode them.

```
<f040> <f07e> <21>
<f080> <f09e> <60>
endbfrange
```

The numeral that appears before the keyword `usefont` refers to an element of the font array, which means MyUDC1 in the case of the value `1`. The lines that follow describe the rearrangement in terms of an encoding range within the template font—that is, the specific encoding range to modify in the template font—followed by a beginning code point from which to begin referencing characters from the component font. In other words, the encoding range <F0 40> through <F0 7E> in HeiseiMin-W3-90ms-RKSJ-H is replaced by glyphs from MyUDC1 encoded in the range 0x21 through 0x5F. The integer value that appears before the `beginbfrange` keyword specifies the number of lines of rearrangement code that a client should expect, and it must agree with the actual number of lines of rearrangement code.

As you can see, once you understand the simplistic syntax of a rearranged font resource, it becomes a trivial task to create rearranged fonts of your own, or, if you are a developer, to build your own front-end to this somewhat powerful PostScript-based technology. However, given today's widespread use of OpenType fonts, along with a host-based printing model that makes extensive use of PDF, this specific implementation of rearranged fonts has limited value.

Acquiring Gaiji Glyphs and Gaiji Fonts

While it is certainly possible to design the glyphs for your own gaiji, and to build your own gaiji fonts, it is likely to be easier for most users to simply purchase them as commercial products.* Nearly all type foundries sell gaiji packages, and some even include what can be considered as gaiji in proprietary extensions of their font products. Some type foundries specialize in gaiji products. Biblos Font† and Enfour Media‡ in particular are companies that market Japanese gaiji fonts. Appendix E of this book goes into more detail with regard to these somewhat standard gaiji character sets.

If you use Adobe InDesign, or if you are considering doing so, I strongly urge you to explore SING as a way to handle your gaiji needs. There are now plenty of tools with which to create SING glyphlets, and any InDesign document that uses SING glyphlets can be exported to PDF, which will retain the glyphs for display and printing on any environment.

For those who wish to pursue creating their own gaiji by designing the glyphs from scratch or by referencing existing glyphs in one or more installed fonts, any one of many tools can be used. Earlier sections of this chapter provided the names and descriptions of suitable tools, one or more of which are certainly up to the task, though my strong recommendation is to use FontLab Studio for these needs because it has the richest set of features.

* After all, most users, even power users, are not professional type designers.

† *http://www.biblosfont.co.jp/*

‡ *http://www.enfour.com/*

Advice to Developers

Many application developers and users who explore what is included with OSes and applications may discover that an adequate collection of CJKV fonts may be right at their fingertips, in the form of the fonts (almost always outline fonts these days) that were bundled with the OS or with one or more specific applications or application suites. Mac OS X, for example, is bundled with a rich variety of OpenType Japanese fonts. Users that require typeface designs or glyph complements beyond the basic fonts provided by the OS will obviously look to font developers for additional offerings.

For application developers, I have the following two points to recommend:

- Take advantage of the fonts that are provided by the OS. Some are bundled for the purpose of displaying UIs and are thus tuned for that, either by the nature of their design—serif versus sans serif—or relative weight. Some are bundled for application use. Even the OS includes a small number of applications, such as text editors and the like.
- When taking advantage of the fonts provided by the OS, bear in mind that they can change over time—specifically their names, their glyph complements, and even the fonts themselves. In lieu of referencing specific fonts by name, OSes often provide generic APIs for this. Mac OS X, for example, provides APIs that effectively mean "get system font" and "get application font" that serve this purpose, and are highly recommended for application developers to use.

For font developers, or those developers who also build font products, there are two things that I cannot recommend highly enough:

Embrace OpenType by building OpenType fonts.
: OpenType is the most widely supported font format today, and has more benefits than any other font format currently in use. The fact that the latest versions of the major OSes support OpenType fonts natively should make this an easy decision.

Embrace Unicode by building OpenType fonts.
: Supporting OpenType effectively forces you embrace Unicode, because the default encoding of OpenType 'cmap' tables is Unicode. This is a good thing. You will not regret making these decisions, and more importantly, your customers will thank you for doing so.

In the end, developers need to become aware of the pros and cons of each font format so that they can decide for themselves when it is appropriate to implement one format versus another. Bundling an outline font with a product, for example, will require that an appropriate rasterizer be present in order to render bitmaps from its outlines. Given the state of today's OSes and their capabilities in terms of supported font formats, along with the broad extent to which Unicode is now supported, OpenType is clearly the best choice in terms of font format. Remember, an OpenType font can include PostScript or TrueType outlines, and effectively merges these formerly competing font formats.

Finally, when developing fonts, one of the most critical tasks, which is often neglected or forgotten, is that fonts must undergo thorough and rigorous testing before they are released to users or provided to OS and application developers for bundling. Unlike applications, fonts tend to be updated less frequently, and to some extent, are treated as static software. For this reason, it is considered best practice to detect and correct any and all known problems prior to their release.

CHAPTER 7
Typography

The foundation for producing printed material in any language—no matter how basic—involves typography. Typography refers to the use of type, specifically page- and line-layout techniques, rules, and principles. It is about page composition. This chapter is where we are finally able to apply what we have learned in earlier chapters: Chapter 3 illustrated the tens of thousands of characters at our disposal; Chapter 4 described how these characters can be encoded; Chapter 5 provided information about the input of these characters through software called input methods, using hardware called keyboards; and Chapter 6 provided details about fonts and font formats. To a great extent, everything comes together here, in this chapter. After all, the ultimate goal of processing or manipulating CJKV text is to produce documents, whether printed or in electronic form.

You may have observed, perhaps even at a somewhat subconscious level, that poor typography sticks out and is easily noticed, but that good typography is almost always overlooked and seemingly invisible. In other words, when text is poorly typeset or laid out, the focus of the reader is on the typography, not its content. Furthermore, when text is typeset well, the words are more directly conveyed to the reader, with less interference. Thus, for typographers, and to some extent for the applications that perform typography, the better they do their job, the less their efforts are noticed by those who use the end result of their labors.

People who read this book have a fundamental understanding that CJKV text can be writen or typeset horizontally and vertically, but there is clearly much more to CJKV typography than simply these two writing modes. There are also many formatting considerations, such as line breaking, justification, inter-glyph spacing, alternate metrics, kerning, glyphs substitution, and so on. You will soon have a deep appreciation for the extent to which CJKV text does not follow all Western-language–style composition rules, and for good reason, given the properties of their writing systems.

Excellent references on typography are readily available, and one of the overall best, in my opinion and experience, is Robert Bringhurst's *The Elements of Typographic Style*, now in its Third Edition (Hartley & Marks, 2004). Seybold Publications' *The Seybold Report* sometimes contains information of a CJKV nature. More detailed information on

Japanese line-layout can be found in the Japanese standard designated JIS X 4051:2004, whose contents are described throughout this chapter. A discussion about compliance with this standard is in the next section, and for those who read Japanese, in Mitsuo Fukawa's (府川充男 *fukawa mitsuo*) book entitled 組版原論—タイポグラフィと活字・写植・DTP (*kumihan genron—taipogurafi to katsuji, shashoku, DTP*; 太田出版, 1996). Other books about typography, some of them written in languages other than English, can be found in this book's bibliography.

Note that while the vast majority of the examples provided in this chapter are for Japanese, almost all of the rules and principles that are described are applicable to other CJKV locales, sometimes with minor modification. Also, most of the typographic examples provided in this chapter were created directly in Adobe InDesign, as part of the text of this book.

Rules, Principles, and Techniques

When one reads information about CJKV line-layout, sets of rules and principles are often presented, such as those outlined in the standard designated JIS X 4051:2004. And, many page- and line-layout programs allow the user to adhere to these rules and principles, sometimes to varying degrees, and with some amount of control of the parameters. While page- and line-layout rules are often based on sound and proven typographic principles, they do not always need to be followed to the letter—they can be broken when the situation or context permits. Typography is part science and part art. It is its science part that follows the rules, and its art part that wants to break them as the situation dictates.

The most important aspect of page- and line-layout is consistency. The formation of and adhering to page- and line-layout rules and principles results in consistency. Consistency is a good thing, because it results in good typography. In other words, whatever rules and principles you choose to apply, they need to be applied in a consistent manner. If a set of predefined rules happen to work well with the document that you are typesetting, by all means follow them. This book, for example, follows the book design rules set forth by the book designers of O'Reilly Media.

Keep in mind while reading this chapter that typography is a genuine art form. Providing an unskilled painter with a set of painting rules will not result in an aesthetically pleasing piece of art. Likewise, putting a top-of-the-line digital SLR camera in the hands of a photography novice does not guarantee that a good photo will result, and providing a rifle to an unskilled hunter does not guarantee that game will be harvested. Paint brushes, cameras, firearms, and the typographic controls of an application are merely tools. What matters most, in terms of achieving an end result, is how these tools are used. All artists have their own set of rules and principles—many unwritten—and they obey or break them as the situation dictates.

JIS X 4051:2004 Compliance

JIS X 4051:2004, *Formatting rules for Japanese documents* (日本語文書の組版方法 *nihongo bunshō no kumihan hōhō*), is the standard that specifies the rules, principles, and techniques that are necessary for typesetting Japanese documents.* Several publishing-quality applications, such as Adobe FrameMaker, Adobe Illustrator, Adobe InDesign, Adobe PageMaker-J, FreeHand MX, QuarkXPress-J, and Canon EDICOLOR now provide full or partial JIS X 4051:2004 compliance.

The JIS X 4051 standard has grown considerably since its inception in 1993. The 1993 version has 28 pages, the 1995 version has 69 pages, and the 2004 version has 206 pages with 6 pages of frontmatter. Needless to say, JIS X 4051 is now a fully matured standard, as evidenced by its second revision, along with the high level of acceptance throughout the industry.

One of the most important aspects of JIS X 4051:2004 to remember is that it is the first national standard that attempts to document the page- and line-layout rules for any type of CJKV text. Many of the page- and line-layout rules for Japanese were either poorly or inconsistently described in proprietary documentation, or else not documented at all. While I wrote in the beginning of this chapter that consistency is more important than a set of rules, you must realize that consistency arises from adhering to a set of rules and principles. JIS X 4051:2004 provides application developers and typographers with a starting point from which to define their own set of typographic rules. The guidance that this standard offers is invaluable.

Many of the typographic rules described in JIS X 4051:2004 are loosely described in this chapter. If your work, research, or interests involve Japanese typography, I strongly encourage you to more deeply explore what JIS X 4051:2004 has to offer by obtaining and studying the standard.

In addition, if you are a member of a standards committee in China, Korea, Taiwan, or Vietnam, or related organization, or have influence on such committees and organizations, I strongly encourage you to consider establishing and publishing a standard that describes a basic set of your locale's page- and line-layout rules and principles, similar to Japan's JIS X 4051:2004 standard. The publishing of JIS X 4051:2004 exerted a very positive influence on the Japanese publishing industry, and clearly served to lift application development to the next level.

* This standard was originally designated JIS X 4051-1993, and was subsequently revised in 1995 and 2004. The 1993 and 1995 versions shared a slightly different title, in both English and Japanese: *Line composition rules for Japanese documents* and 日本語文書の行組版方法 *nihongo bunshō no gyō kumihan hōhō*.

GB/T 15834-1995 and GB/T 15835-1995

China has established two standards that provide some amount of line-layout guidance, as follows:

- GB/T 15834-1995, *Use of Punctuation Marks* (标点符号用法 *biāodiǎn fúhào yòngfǎ*)
- GB/T 15835-1995, *General Rules for Writing Numerals in Publications* (出版物上数字用法的规定 *chūbǎn wùshàng shùzì yòngfǎ de guīdìng*)

Although these two GB standards are not even close to being as comprehensive as Japan's JIS X 4051:2004, they do provide general principles for composing Chinese text, in both horizontal and vertical writing modes.

Typographic Units and Measurements

Before one begins to construct lines of text for a document, there must be a consistent and established set of units and measurements in place for the purpose of specifying the size of text and spacing. If one is to consider only nonproprietary typesetting systems, there is little benefit to discussing typographic units other than points. But, many proprietary typesetting systems are still in use today, and many typographic guides still make use of other typographic units. Knowing how to convert between them is useful.

Before the days of easily scaled type, names were associated with various and specific type sizes. Table 7-1 lists some of these type size names, along with their corresponding sizes expressed in points.

Table 7-1. Historical names for type sizes

Name	Size in points
Diamond	4
Pearl	4.5
Ruby[a]	5
Nonpareil	6
Emerald	6.5
Minion	7
Brevier	7.5
Bourgeois	8
Long primer	9
Elite	10
Small pica	11
Pica	12

Table 7-1. Historical names for type sizes

Name	Size in points
English	14
Great primer	16
Paragon	18
Two-line small pica	22
Two-line pica	24
Two-line English	28
Two-line great primer	32
Three-line pica	36
Four-line pica	48
Four-line English	56
Five-line pica	60
Six-line pica	72

a. In case you haven't figured it out yet, this is the origin of the typographic term "ruby" that was introduced in Chapter 6. Remember that ruby glyphs are typically set at half the size of their parent glyphs. Its corresponding point size, specifically 5 points, is half the size of a common text size, specifically the elite at 10 points.

Some of these historical names are still in use today, such as pica. Some of them are used in somewhat specific contexts, such as ruby, which has seemingly lost its absolute type size attribute, and instead is used for its relative type size. Because most typographic units of measurement have long and sometimes misunderstood histories, such discussions are obviously beyond the scope of this book. As previously written, what is important in the context of this book is to know that there are many typographic units, and that you can convert between them.

Two Important Points—Literally

Quite literally, there are two types of points used in typography today. The point is a unit of measurement used to specify the size of characters, and also for specifying types of spaces, such as leading and letter spacing. Which point is being used—the *DTP* or *Didot*—is important because they represent different sizes.

It is important to realize that the DTP point has become the predominant typographic unit of measurement used by PostScript and related technologies.

The DTP point

The DTP point (or simply "point") is the most commonly used typographic unit today, at least for American and British systems. The point is usually considered to be 1/72 of one inch, which also means 0.3515 millimeters, 0.01383 inches, or 1/12 of one pica.* To be exact, there are 72.27 points per inch—the result of the mathematical operation 12 ÷ 0.166044 (the number of points per pica divided by the length of one pica in inches).

Some applications allow the user to alter the definition of the point. QuarkXPress, for example, allows the user to change the definition of a point—the default is 72 points per inch according to the PostScript imaging model. This is useful if you have been using a different typographic scale and want to stick with it.

The Didot point

The Didot point is still sometimes used in continental Europe, and is approximately 7% larger than the DTP point. It represents 0.37597 millimeters, 0.01483 inches, or 1/12 of one cicero. The cicero is similar to a pica, but like the DTP versus Didot point, it is approximately 7% larger than the pica.

Other Typographic Units

The two types of points are Western typographic units, but surely there must be typographic units developed in non-Western cultures. Indeed there are, and three other typographic units are still used in some proprietary typesetting systems in CJKV locales, written as shown in Table 7-2 in the respective CJKV locales.

Table 7-2. Other typographic units

Abbreviated name	Chinese	Japanese	Korean
Q	级数/級數 *jíshù*	級数 *kyūsū*	급수/級數 *geupsu*
H	齿数/齒數 *chǐshù*	歯数 *hasū*	치수/齒數 *chisu*
G	号数/號數 *hàoshù*	号数 *gōsū*	호수/號數 *hosu*

As suggested in Table 7-2, I will henceforth refer to these three typographic units as Q,† H, and G, respectively. The use of Q and H as abbreviations for the first two typographic units is actually common practice in Japan. The Q typographic unit is defined in JIS X 0207-1979‡ and in GB 3937-83.

* A pica is traditionally 4.22 millimeters or 0.166 inches, but the convention today is to consider a pica to be the same as 1/6 of one inch.

† I feel obligated to point out that there is absolutely no relation between Q (the unit of typographic measurement), "Q" (an omnipotent being that appeared in the *Star Trek: The Next Generation* series), and "Q" (007's personal spy-gadget technician).

‡ Previously designated JIS C 6225-1979

Q and H are equivalent to one-fourth of one millimeter (0.25 mm)—although both units represent the same size, they are used for completely different purposes, as follows:

- Q are used solely for specifying type size
- H are used solely for specifying leading, escapement, line length, and spaces—everything but type size

One may wonder about where these typographic units originated. The Q typographic unit roughly corresponds to *lens number* in reference to the lenses that were used by older photo typesetting machines, whereby each lens number corresponded to a specific font size on the film. The H typographic unit refers to how many gear "teeth" the film is advanced on the drum, thus being used solely for distances, and not for type size. The ideograph 歯 (pronounced *ha* in Japanese), which represents the H typographic unit, literally means "tooth" or "teeth."

Adobe Illustrator, Adobe InDesign, Adobe FrameMaker, Founder FIT, FreeHand MX, QuarkXPress, and Canon EDICOLOR are the few nonproprietary page-layout systems that allow the user to specify Q for units of typographic measurement. Providing support for Q and H in nonproprietary page-layout systems makes it easier for users of proprietary equipment to transition to nonproprietary software.

So, you may be wondering why there is a need for both Q and H—time for a little Q and A (*Questions and Answers*). When one specifies how text is to be laid out into lines, it is common to specify both type size and leading. The text of this book, for example, is laid out as 10.2/12.5, which means 10.2-point type and 12.5-point leading. The leading here refers to the distance from the baseline of one line of text to another: 10.2 points for the characters themselves plus 2.3 points of virtual "lead" as extra space. There are other conventions that specify only the additional space instead of the point size plus the additional space. This could result in 10.2/2.3. Japanese layout is specified in terms of the spacing between objects, and not the distance from one baseline to another. Needless to say, all of this can become quite confusing, because it is not always clear which value represents the type size, and which represents the leading or spacing. To some extent, the use of Q and H helps to make this obvious, because one value is clearly indicating type size, and the other indicates spacing of some sort. Text laid out as 10Q/13H uses 10Q (2.5 mm or 7.112-point) type size plus 13H (3.25 mm or 9.246-point) leading.

The other typographic unit, which I am calling G, is unique in that its scale is the reverse of what one finds in the other typographic units. That is, the higher the value of G, the smaller the size. And, like the Q unit, G is used strictly for type size. This typographic unit, as used in Japan, was developed by Shozo Motoki (本木昌造 *motoki shōzō*) in 1933, and was supposedly based on a traditional Japanese metric system called Kujira-Jaku (鯨尺 *kujira jaku*). The Chinese developed a comparable system with the same name, but it differs in that fractional, referred to as "small" (小 in Chinese), values are included within its scale. For example, in Chinese, the 0G value is referred to as 初号, and the fractional or "small" version is referred to as 小初.

Various sizes on the G scale relate to one another in terms of being either half or double the size of other sizes in the same scale. The following sizes on the G scale relate to one another in terms of being pure or close multiples of one another:

- 1G and 4G
- 0G (Initial G), 2G, 5G, and 7G
- 3G, 6G, and 8G

Table 7-3 provides these same G units together, as used in Japan and China, along with the equivalent sizes in points so that it is clear how they are related. For the values as used in China, the fractional or "small" values are provided in parentheses.

Table 7-3. The G typographic unit—Japan and China

Country	Set A	Set B	Set C
Japan	n/a	0G—42 points	n/a
	1G—27.5 points	2G—21 points	3G—16 points
	4G—13.75 points	5G—10.5 points	6G—8 points
	n/a	7G—5.25 points	8G—4 points
China	n/a	0G—42 points (small = 36)	n/a
	1G—26 points (small = 24)	2G—22 points (small = 18)	3G—16 points (small = 15)
	4G—14 points (small = 12)	5G—10.5 points (small = 9)	6G—7.5 points (small = 6.5)
	n/a	7G—5.5 points	8G—5 points

Needless to say, the G is a typographic unit of measurement whose scale limits the maximum size of a character to 42 points. It is also no longer used.

Some page-layout systems also use the notion of "character" as a typographic unit. Founder FIT and Canon EDICOLOR, for example, allow the user to specify dimensions in terms of number of characters, for both the horizontal and vertical direction. For writing systems whose characters are uniform in size, which effectively means that they can be set on a rigid grid, this has many advantages.

Horizontal and Vertical Layout

CJKV text, from a traditional point of view, is set vertically. Columns begin at the right side of the page and work their way to the left. Also, books are read beginning from what in the West is considered the back of the book. Fortunately, it is also acceptable to set CJKV text horizontally and to read books in "Western" direction. There are, however, a few punctuation marks and other characters that require special handling when set vertically, such as their positioning within the design space or being placed in 90° clockwise rotation.

PostScript's flexible text-handling capabilities allow you to set CJKV text vertically. This is accomplished through the use of a vertical font instance.* Whether or not you can actually typeset CJKV text vertically depends greatly upon the OSes and applications you are using—the underlying PostScript CJKV fonts have inherent vertical support. Mac OS, for example, did not have any built-in vertical support, so application developers who wished to provide their users with the ability to set text vertically had to implement vertical support themselves. Genuine vertical support was introduced in Mac OS X, as part of ATSUI APIs. CoreText, which replaces ATSUI, also supports vertical text.

One critical aspect of vertical layout is the relative position of the vertical origin, normally centered over the top of the design space. As you will see in Figure 7-1, the typical design space found in CJKV fonts is negatively offset from the baseline (Y coordinate `0`) anywhere from 120 to 200 units. This effectively means that the relative position from the horizontal baseline—Y coordinate `0`—depends on the design space of the font. Interestingly, legacy font resources, such as the 'FOND' resource for Mac OS and the PFM file for Windows, did not have any field in which to encapsulate such information. This is information specific to and crucial for vertical writing, but almost all legacy font formats were not designed with vertical writing in mind. The fields that contain values for font-level ascent and descent are sometimes—incorrectly and dangerously, in my opinion—overloaded or changed to encapsulate the design space rather than the true font-level ascent and descent values. OpenType has improved this situation, and in a cross-platform manner, by providing tables and fields that are used to specify vertical writing parameters.

Another aspect of vertical layout is *escapement*, that is, the amount of space from one character to the next, also referred to as a glyph's *set width*. If a glyph's set width is 1000 units—a typical CJKV glyph—then its escapement, whether in horizontal or vertical writing mode, is the same as its point size. That is, the glyph for a character set at 10 points uses 10-point escapement. Typical Japanese typeface designs that are used to compose newspapers, however, are set in a nonsquare design space, such as 1000×800, that is, 1000 units wide and 800 units high. The subject of nonsquare designs is covered in the next section.

Table 7-4 provides some examples of Japanese text set horizontally and vertically, including some of the characters that require special vertical handling.

Applications that provide the capability for vertically set CJKV text more often than not allow users to edit text vertically—this can be a new experience for those not accustomed to it. The cursor is usually a horizontal bar, as opposed to a vertical bar, and moves top to bottom, and then right to left. Keyboard cursor keys, for example, behave according to the logical direction of the text, not the physical direction. Some simpler applications permit users to enter and edit text horizontally only, but may permit vertical printing. This is not a particularly fine example of a WYSIWYG situation.

* That is, PostScript fonts whose `/WMode` value is set to 1.

Table 7-4. Horizontal and vertical layout—Japanese

Horizontal	Vertical
普通の「DTPシステム」は縦書きレイアウトをサポートしていますが、簡単なワープロやテキストエディターはサポートしません。縦書きのサポートの為にはフォントも必要です。全てのポストスクリプト日中韓越フォントには縦書きフォントも含まれています。	普通の「DTP システム」は縦書きレイアウトをサポートしていますが、簡単なワープロやテキストエディターはサポートしません。縦書きのサポートの為にはフォントも必要です。全てのポストスクリプト日中韓越フォントには縦書きフォントも含まれています。

Nonsquare Design Space

As mentioned in the previous section, there are some fonts whose design space is not square, in particular those fonts that are used for printing newspapers. It is also possible, using today's page-layout systems, to artificially scale square designs so that they fit in a nonsquare design space, but the results from such an operation are far from being aesthetically pleasing.

Table 7-5 provides an example of text that was set using a font with a square design space, using Morisawa's A-OTF リュウミン Pr6N L-KL (RyuminPr6N-Light) typeface design, and one set with a nonsquare design space, using Morisawa's A-OTF 毎日新聞明朝 Pro L (MNewsMPro-Light) typeface design.

Newspaper publishers prefer to use these "compressed" typeface designs because they can fit more text in the same amount of space. I included the generic line-drawing characters because they illustrate that simple scaling is not used to create these designs—they are designed in a nonsquare design space. As clearly specified on Morisawa's website, their A-OTF 毎日新聞明朝 Pro L (MNewsMPro-Light) typeface design is set in a 1000×800 design space, but because the font itself is implemented using the conventional 1000×1000 design space, meaning that its design has been stretched along the Y-axis by 25%, the glyphs should be compressed to 80% when used in applications.

Although a typeface design set in a 1000×800 design space can be used for horizontal writing, it is not very common, and the results are not very pleasing (but may be appropriate under some circumstances). Some Korean typeface designs, in particular their hangul, are set in a 800×1000 design space, giving them a compressed effect, but these are intended for horizontal writing.

Table 7-5. Square and nonsquare design spaces

The Character Grid

An essential element in CJKV typography is the ability for the user to establish a character-based grid, if desired. Some applications, such as Adobe InDesign, Canon EDICOLOR, and QuarkXPress, allow the user to establish a character grid, and also allow the user to use "character" as a unit of measurement for determining line lengths and so on.

Because most CJKV characters, such as zhuyin, kana, hangul, and especially ideographs, are typically set in a uniform design space, usually square, it is somewhat natural to want to set the glyphs in a character-based grid. However, the inclusion of some punctuation or Latin glyphs can cause this grid to break down and cease to be rigid.

Table 7-6 provides an example of a character-based grid. Note how punctuation is still allowed to dangle, such as at the end of the fourth line. Details about dangling punctuation can be found later in this chapter in the section entitled "Line Breaking and Word Wrapping."

Morisawa's Fuzzy完全箱組 (*fajī kanzen hakogumi*, meaning "fuzzy perfect-box layout") software, an Adobe Illustrator plug-in, performs various calculations that enable grid-like behavior for Adobe Illustrator, but doesn't actually set up a grid as shown in Table 7-6. Its purpose was to do the necessary calculations in order to fill the selected text box with the text that is inside (which may need to be stretched or otherwise enlarged in order to fill it).

Table 7-6. Character grid example

Glyph string
普通の「ＤＴＰシステム」は縦書きレイアウトはサポートしています。簡単なワープロやテキストエディターはサポートしません。縦書きのサポートの為にはフォントも必要です。全てのポストスクリプト中日韓越フォントには縦書きフォントも含まれています。

Vertical Character Variants

While the majority of characters appear the same regardless of writing direction, some characters, due to their orientation or position within the design space, must somehow change to accommodate different writing directions.

Table 7-7 illustrates how the glyphs for some characters change their orientation or positioning within their design space depending on whether they are being set horizontally or vertically. These glyphs' design spaces have been highlighted through the use of registration marks for much easier comparison, and to better understand the relative position of the glyphs in their design space. Additionally, there are some glyphs that must undergo more than one transformation in order to change from the horizontal form to its appropriate vertical form. Consider the glyphs for the following two characters: 〜 (a wave or swung dash) and ー (long vowel mark, Japanese-specific). The corresponding vertical forms of these characters are ｜ and 丨, respectively. Note how these two characters are rotated 90˚ and flipped in order to become the proper vertical variants. Some font developers often forget to flip the glyphs for these characters when creating their vertical variants—it is a very easy mistake to make, unfortunately.

Table 7-7. Sample characters that require special vertical handling—Japanese

Description	Horizontal	Vertical
Ideographic period	。	。
Ideographic comma	、	、

Table 7-7. Sample characters that require special vertical handling—Japanese

Description	Horizontal	Vertical
Long vowel symbol	ー	
Opening bracket	「	
Closing bracket	」	
Small katakana i	ィ	ィ
Small katakana o	ォ	ォ

Table 7-8 provides a much more complete list of characters that are known to have vertical variants in at least one locale or implementation, listed in the order in which they appear in Unicode. Included for reference are the Row-Cell values from the GB 2312-80, CNS 11643-2007 (Plane 1), JIS X 0213:2004, and KS X 1001:2004 character set standards, for China, Taiwan, Japan, and Korea, respectively.

Table 7-8. Characters and their vertical variants

Unicode	China	Taiwan	Japan	Korea[a]
U+00AB			1-09-08	
U+00B0	01-67	22-78	01-75	01-38
U+00BB			1-09-18	
U+2010	03-13		01-30	01-09
U+2015	01-10[b]	01-25	01-29[c]	01-10
U+2016	01-12	02-61	01-34	01-11
U+2018	01-14	01-68	01-38[c]	01-14
U+2019	01-15	01-69	01-39[c]	01-15
U+201C	01-16	01-70	01-40[c]	01-16
U+201D	01-17	01-71	01-41[c]	01-17
U+2025		01-13	01-37[c]	01-05
U+2026	01-13[b]	01-12	01-36[d]	01-06

Table 7-8. Characters and their vertical variants

Unicode	China	Taiwan	Japan	Korea[a]
U+2032	01-68	01-75	01-76	01-39
U+2033	01-69	01-73	01-77	01-40
U+3001	01-02[b]	01-03	01-02[d]	01-02[e]
U+3002	01-03[b]	01-04	01-03[d]	01-03[e]
U+3008	01-20[b]	01-50[f]	01-50[d]	01-20
U+3009	01-21[b]	01-51[f]	01-51[d]	01-21
U+300A	01-22[b]	01-46[f]	01-52[d]	01-22
U+300B	01-23[b]	01-47[f]	01-53[d]	01-23
U+300C	01-24[b]	01-54[f]	01-54[d]	01-24
U+300D	01-25[b]	01-55[f]	01-55[d]	01-25
U+300E	01-26[b]	01-58[f]	01-56[d]	01-26
U+300F	01-27[b]	01-59[f]	01-57[d]	01-27
U+3010	01-30[b]	01-42[f]	01-58[d]	01-28
U+3011	01-31[b]	01-43[f]	01-59[d]	01-29
U+3013	01-94		02-14	01-75
U+3014	01-18[b]	01-38[f]	01-44[d]	01-18
U+3015	01-19[b]	01-39[f]	01-45[d]	01-19
U+3016	01-28[b]		1-02-58	
U+3017	01-29[b]		1-02-59	
U+3018			1-02-56	
U+3019			1-02-57	
U+301C	01-11	02-36	01-33[d]	01-13
U+3041	04-01	09-50	04-01[c]	10-01

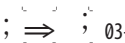
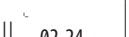
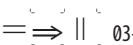
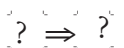
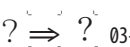
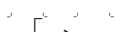
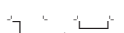
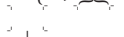
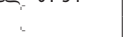
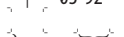
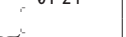
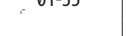
Table 7-8. Characters and their vertical variants

Unicode	China	Taiwan	Japan	Korea[a]
U+3043	い 04-03	い 09-52	い⇒い 04-03[c]	い 10-03
U+3045	う 04-05	う 09-54	う⇒う 04-05[c]	う 10-05
U+3047	え 04-07	え 09-56	え⇒え 04-07[c]	え 10-07
U+3049	お 04-09	お 09-58	お⇒お 04-09[c]	お 10-09
U+3063	つ 04-35	つ 09-84	つ⇒つ 04-35[c]	つ 10-35
U+3083	や 04-67	や 10-22	や⇒や 04-67[c]	や 10-67
U+3085	ゆ 04-69	ゆ 10-24	ゆ⇒ゆ 04-69[c]	ゆ 10-69
U+3087	よ 04-71	よ 10-26	よ⇒よ 04-71[c]	よ 10-71
U+308E	わ 04-78	わ 10-33	わ⇒わ 04-78[c]	わ 10-78
U+3095			か⇒か 1-04-85	
U+3096			け⇒け 1-04-86	
U+30A0			=⇒= 1-03-91	
U+30A1	ア 05-01	ア 10-46	ア⇒ア 05-01[c]	ア 11-01
U+30A3	イ 05-03	イ 10-48	イ⇒イ 05-03[c]	イ 11-03
U+30A5	ウ 05-05	ウ 10-50	ウ⇒ウ 05-05[c]	ウ 11-05
U+30A7	エ 05-07	エ 10-52	エ⇒エ 05-07[c]	エ 11-07
U+30A9	オ 05-09	オ 10-54	オ⇒オ 05-09[c]	オ 11-09
U+30C3	ツ 05-35	ツ 10-80	ツ⇒ツ 05-35[c]	ツ 11-35
U+30E3	ヤ 05-67	ヤ 11-18	ヤ⇒ヤ 05-67[c]	ヤ 11-67
U+30E5	ユ 05-69	ユ 11-20	ユ⇒ユ 05-69[c]	ユ 11-69
U+30E7	ヨ 05-71	ヨ 11-22	ヨ⇒ヨ 05-71[c]	ヨ 11-71
U+30EE	ワ 05-78	ワ 11-29	ワ⇒ワ 05-78[c]	ワ 11-78
U+30F5	カ 05-85	カ 11-36	カ⇒カ 05-85[c]	カ 11-85

Table 7-8. Characters and their vertical variants

Unicode	China	Taiwan	Japan	Korea[a]
U+30F6	05-86	11-37	05-86[c]	11-86
U+30FC		11-43	01-28[d]	
U+31F0			1-06-78	
U+31F1			1-06-79	
U+31F2			1-06-80	
U+31F3			1-06-81	
U+31F4			1-06-82	
U+31F5			1-06-83	
U+31F6			1-06-84	
U+31F7			1-06-85	
U+31F8			1-06-86	
U+31F9			1-06-87	
U+31FA			1-04-85	
U+31FB			1-04-85	
U+31FC			1-04-85	
U+31FD			1-04-85	
U+31FE			1-04-85	
U+31FF			1-04-85	
U+FF01	03-01[b]	01-10	01-10	03-01
U+FF08	03-08[b]	01-30[f]	01-42[d]	03-08
U+FF09	03-09[b]	01-31[f]	01-43[d]	03-09
U+FF0C	03-12[b]	01-02	01-04	03-12
U+FF0E	03-14	01-05[g]	01-05	03-14

Table 7-8. Characters and their vertical variants

Unicode	China	Taiwan	Japan	Korea[a]
U+FF1A	 03-26[b]	 01-08	 01-07[c]	 03-26
U+FF1B	 03-27[b]	 01-07	 01-08	 03-27
U+FF1D	 03-29	 02-24	 01-65[d]	 03-29
U+FF1F	 03-31[b]	 01-09	 01-09	 03-31
U+FF3B	 03-59		01-46[d]	 03-59
U+FF3D	 03-61		01-47[d]	 03-61
U+FF3F	 03-63[b]	 02-05	01-18	 03-63
U+FF5B	 03-91[b]	 01-34[f]	 01-48[d]	 03-91
U+FF5C	 03-92	 01-24	 01-35	 03-92
U+FF5D	 03-93[b]	 01-35[f]	 01-49[d]	 03-93
U+FF5F			 1-02-54	
U+FF60			1-02-55	
U+FFE3	03-94	02-03	01-17	03-94

a. Some implementations that support vertically set Korean text make use of Western-style punctuation that is simply rotated 90°. This affects the vertical variants for ‘ (U+2018), ’ (U+2019), “ (U+201C), ” (U+201D), ！ (U+FF01), ， (U+FF0C), ． (U+FF0E), ： (U+FF1A), ； (U+FF1B), and ？ (U+FF1F). Sometimes, the half- or proportional-width forms of these characters are simply rotated 90° clockwise.

b. Specified in GB/T 12345-90—vertical variants are encoded from 06-57 through 06-85.

c. Specified beginning in JIS X 0208:1997.

d. Specified beginning in JIS C 6226-1978.

e. For some implementations, the vertical form is the standard form.

f. The vertical variant of this character is encoded exactly two code points forward. For example, the vertical variant of 01-50 is encoded at 01-52.

g. One must wonder how ・ (a period) is differentiated from a centered dot—contrast ・ with ・.

Note how some characters' vertical forms, such as Unicode U+FF1A (full-width colon), are arranged differently among CJKV locales—shifted to the upper-right corner in China and Korea, as is in Taiwan, and rotated 90° clockwise in Japan. Ligatures that are composed of ideographs, katakana, or hangul also have vertical variants. Although few of these ligatures are included in national character set standards, they are commonplace in vendor extensions (see Appendix E). Tables 7-45 and 7-46 provides some examples of these ligatures that require vertical variants.

As illustrated in Table 7-8, some CJKV locales use different conventions for handling vertical characters. Taiwan, for example, uses centered punctuation instead of left- or top-justified ones. Contrast ﹒ with ． (an ideographic period) and ﹑ with 、 (an ideographic comma). This technique permits the same glyph to be used for both horizontal and vertical writing.

There are some differences between how vendors have defined vertical variants and how they have been defined in national character sets. Using JIS X 0208:1997 as an example, we find that the way they defined vertical variants are, by and large, the same. Apple's extensions to JIS X 0208:1997—both versions, used on different vintages of Mac OS-J—share the same set of 53 vertical variants. JIS X 0208:1997 also defines a set of 53 vertical variants. Yet, they are not the same. Table 7-9 lists the characters that are common and specific to these two vertical variant definitions.

Table 7-9. Common and implementation-specific vertical variants—Japanese

	Category	Glyphs
Common	26 symbols	﹑ ﹒ ﹘ ﹘ ﹏ ⋯ ‥ ﹙ ﹚ ﹝ ﹞ ﹛ ﹜ ﹤ ﹥ ﹦ ﹦ ﹁ ﹂ ﹃ ﹄ 【 】 ＝
	10 small hiragana	ぁ ぃ ぅ ぇ ぉ っ ゃ ゅ ょ ゎ
	12 small katakana	ァ ィ ゥ ェ ォ ッ ャ ュ ョ ヮ カ ケ
Specific	JIS X 0208:1997	： ‘’ “”
	Mac OS-J	― ― – ‖ |

All versions of Mac OS-J actually encode the vertical variants—very early versions of Mac OS-J (KanjiTalk version 6 and earlier) encode them 10 rows after their horizontal versions, and current versions (KanjiTalk version 7 and later) encode them 84 rows after their horizontal versions. This effectively means that the vertical version of Row-Cell 01-02 (ideographic comma) is encoded at either 11-02 or 85-02, depending on the vintage of Mac OS-J.

Other implementations, such as the character set used for the Japanese versions of Microsoft Windows (versions 3.1J, 95J, 98J, and NT-J), define additional vertical variants, including some of those listed in the Table 7-8. In addition to the *identical* set of 53 vertical variants currently specified by Mac OS-J, the Japanese versions of Microsoft Windows specify the use of vertical versions of the following characters: full-width comma (01-04), full-width period (01-05), "geta" mark (02-14), and 16 katakana ligatures (not part of JIS X 0208:1997). Furthermore, it specifies that the 4 arrows and 30 of the 32 line-drawing elements in JIS X 0208:1997 be rearranged as shown in Table 7-10 (their corresponding Unicode code points are provided in parentheses).

Table 7-10. Rearranging horizontal characters for vertical use

Character code	Horizontal	Vertical
02-10 (U+2192)	→	↓
02-11 (U+2190)	←	↑
02-12 (U+2191)	↑	→
02-13 (U+2193)	↓	←
08-01 (U+2500)	─	│
08-02 (U+2502)	│	─
08-03 (U+250C)	┌	┐
08-04 (U+2510)	┐	└
08-05 (U+2518)	└	┘
08-06 (U+2514)	┘	┌
08-07 (U+251C)	├	┬
08-08 (U+252C)	┬	┤
08-09 (U+2524)	┤	┴
08-10 (U+2534)	┴	├
08-12 (U+2501)	━	┃
08-13 (U+2503)	┃	━
08-14 (U+250F)	┏	┓
08-15 (U+2513)	┓	┗
08-16 (U+251B)	┛	┛
08-17 (U+2517)	┗	┏
08-18 (U+2523)	┣	┳
08-19 (U+2533)	┳	┫
08-20 (U+252B)	┫	┻
08-21 (U+253B)	┻	┣
08-23 (U+2520)	┣	┳
08-24 (U+252F)	┳	┫
08-25 (U+2528)	┫	┻

Table 7-10. Rearranging horizontal characters for vertical use

Character code	Horizontal	Vertical
08-26 (U+2537)	┷	┠
08-27 (U+253F)	┿	╂
08-28 (U+251D)	┝	┰
08-29 (U+2530)	┰	┥
08-30 (U+2525)	┥	┸
08-31 (U+2538)	┸	┝
08-32 (U+2542)	╂	┿

Note that all of the vertical forms are a result of 90° clockwise rotation. The line-drawing elements at Row-Cell 08-11 (┼) and 08-22 (╋) do not require substitution for vertical use because the same form results from 90° clockwise rotation. These map to Unicode code points U+253C and U+254B, respectively.

Dedicated Vertical Characters

There is another class of vertical characters, specifically those for which there is no horizontal form. I refer to these as dedicated vertical characters. A small number of such characters are used in Japanese. Three of them are specifically designed to be used together and fused to one another. In other words, tracking or full-justification must not increase their spacing. Table 7-11 lists these vertical characters and demonstrates how some of them are used together.

Table 7-11. Dedicated vertical characters—Japanese

	U+303B	U+3033	U+3034	U+3035	<U+3033, U+3035>	<U+3034, U+3035>
Glyphs						

While the usual vertical forms are entered as their corresponding horizontal form, and the invocation of a vertical substitution feature results in the appropriate glyphs, there is no pathway for these dedicated vertical characters.

Some specialized fonts may include additional dedicated vertical glyphs. Table 7-48 provides examples from one such font, which includes a small set of two- and three-character

hiragana ligatures that are intended only for vertical use, and for which there are no horizontal forms.

Vertical Latin Text

While the transformation from horizontal to vertical layout is somewhat straightforward for most CJKV characters and involves mainly punctuation and symbols (and small kana in the case of Japanese), handling Latin text in vertical writing mode requires special considerations and has more than one option. As they say about Perl: TIMTOWTDI.*

PostScript CJKV fonts, by default, treat Latin glyphs the same as everything else when it comes to vertical writing mode—their orientation remains the same. The preferred way in which to vertically set Latin text, however, involves 90˚ clockwise rotation.

The following are the ways in which Latin glyphs can be set vertically, in order of relative preference:

- Rotated 90˚ clockwise.
- Converted to full-width forms, and then set as is—most Latin glyphs used in CJKV text are half- or proportional-width.
- Set together horizontally in the same cell using half- or third-width forms if they consist of only two or three characters, respectively—this is sometimes referred to as *tatechuyoko* (縦中横 *tatechūyoko*) in Japanese, which means "horizontal in vertical."†
- Set as is—this is rarely desirable unless you are writing a document that needs to illustrate the various ways to set Latin text vertically.‡

Table 7-12 illustrates these four methods for setting Latin text vertically using the sample Japanese text "これはJPテキストだ" (meaning "This is JP text"). Only the two uppercase Latin characters "JP" change in the four examples.

As I wrote earlier, you will find that most users will prefer these methods in the order presented in Table 7-12. Exactly which one is preferable may depend on the length of the embedded Latin text. Some applications, by default, rotate these characters 90˚ clockwise. Any other setting may require explicit direction from the user.

* The classic Perl slogan: *There Is More Than One Way To Do It*.

† Adobe InDesign, Adobe Illustrator (version 8.0 or greater), Founder FIT, FreeHand MX, and QuarkXPress use this term, but Canon EDICOLOR uses 連文字 (*ren moji*). Adobe Illustrator version 7.0 used 組み文字 (*kumi moji*).

‡ Like this book, or any book for that matter that includes information about CJKV typography. It is also the default way in which PostScript images such glyphs.

Table 7-12. Vertically set Latin text

90° clockwise	Full-width	Horizontal in vertical	As is
これはJPテキストだ	これはＪＰテキストだ	これはJPテキストだ	これはＪＰテキストだ

Horizontal in vertical mode—Tatechuyoko

The *tatechuyoko* ("horizontal in vertical") method of setting Latin text deserves further explanation and coverage. Given the limited amount of space in which to set Latin text within the relative width of a typical ideograph or other Japanese character, there are limitations in terms of what can and cannot be done when taking advantage of this functionality.

Many contemporary fonts, such as those based on the Adobe-Japan1-4 character collection, or greater Supplement, provide special-purpose third- and quarter-width glyphs that are very effective when using *tatechuyoko* mode.

Table 7-13 provides examples of half-, third-, and quarter-width digits being set in *tatechuyoko* mode. The glyphs that are used in Table 7-13 are not artificially compressed or synthesized, but are instead deliberately designed to be those specific widths.

Note how the half-, third-, and quarter-width glyphs fit perfectly in the same space in which typical Japanese glyphs are set, when two, three, and four digits are used, respectively. When more digits are used, the glyphs necessarily protrude on both sides, as illustrated when using six quarter-width glyphs.

Table 7-13. Additional tatechuyoko examples

Half-width	Third-width	Quarter-width	Quarter-width (six digits)
値段は10円だ	値段は100円だ	値段は1000円だ	値段は100000円だ

Full-width Latin, Greek, and Cyrillic glyph issues

When full-width Latin, Greek, or Cyrillic glyphs are set vertically, an adjustment to the relative glyph baselines is necessary. These same glyphs, when set horizontally, rest comfortably on a baseline. If these glyphs were to be set vertically, there would be uneven spacing between the glyphs, as illustrated in Table 7-14, along with some glyphs that overlap.

Table 7-14. Horizontal versus vertical Latin, Greek, and Cyrillic text

Metrics	Latin	Greek	Cyrillic
Default	photography	φωτογραφία	фотография

Table 7-14. Horizontal versus vertical Latin, Greek, and Cyrillic text

Metrics	Latin	Greek	Cyrillic
Adjusted	p h o t o g r a p h y	*ϕ* *ω* *τ* *ο* *γ* *ρ* *α* *ϕ* *ί* *α*	ф о т о г р а ф и я

Prior to OpenType, some fonts would provide separate glyphs for both purposes. Now, OpenType allows the same glyphs to be used for both purposes, and 'vmtx' and 'VORG' table settings allow the desired behavior to take place. Each glyph, instead of resting on the Latin baseline at Y coordinate 0, is centered along the Y-axis. The full-width "p" glyph of Adobe Systems' 小塚明朝 Pr6N L (KozMinPr6N-Light) font, for example, is adjusted such that its vertical origin is set at 628 instead of the default value of 880. This has the effect of raising the glyph. Likewise, the full-width "h" glyph of the same font is adjusted such that its vertical origin is set at 885, which lowers the glyph ever so slightly. Both of these adjustments help to ensure that these two glyphs do not collide when set vertically.

Clearly, the only way in which to further improve the result would be to apply proportional metrics to the glyphs. And, some fonts provide such information.

Line Breaking and Word Wrapping

Most CJKV text requires special handling for the beginning and ends of lines, which is commonly called line breaking, word wrapping, or sometimes hyphenation. In Japanese, this is referred to as 禁則処理 (*kinsoku shori*), which literally means "prohibited (character) processing." There are some characters, usually punctuation and enclosing characters, that should not begin a new line, and likewise, there are characters that should not terminate a line. There are similar rules in English, but they are much more important for CJKV text because there are no spaces between words—punctuation marks are treated like any other character.*

Table 7-15 lists the characters that should not begin a new line. These include characters such as punctuation marks, closing quotes, closing bracket-like characters, small kana

* Korean and any transliterated CJKV text are considered to be exceptions to this general rule—spaces are used to delimit words.

(Japanese-specific), and some symbols. The characters in the first rank have priority in processing, at least in Japanese. Some applications handle only some ranks. In general, the more advanced or better the software, the more of these characters are supported. And, the more characters that are supported, the stronger the treatment is considered, and some applications, such as Adobe InDesign, allow the user to set or adjust the strength of this feature by manipulating the set of characters that are treated in this way.

From a semantic or syntactic point of view, most of these characters act as modifiers for or appear immediately after text. In Japanese, these characters are called 行頭禁則文字 (*gyōtō kinsoku moji*).

Table 7-15. Characters prohibited from beginning lines

Rank	Characters
1	、。，．：；？！’ ” ） 〕 ］ ｝ 〉 》 」 』 】
2[a]	々ー あ い う え お つ や ゆ よ わ ア イ ウ エ オ ツ ヤ ユ ヨ ワ カ ケ
3	゛ ゜ ヽ ヾ ゝ ゞ — ‐ ゜ ′ ″ ℃ ℉ ¢ ％ ‰

a. Of course, almost all of the characters in this rank are Japanese-specific.

Table 7-16 lists the characters that should not terminate a line. These are basically opening quotes, opening bracket-like characters, and some symbols, including some currency symbols. They are ranked into two groups. From a syntactic point of view, most of these characters appear immediately before text that they modify. In Japanese, these characters are called 行末禁則文字 (*gyōmatsu kinsoku moji*).

Table 7-16. Characters prohibited from terminating lines

Rank	Characters
1	‘ “ （ 〔 ［ ｛ 〈 《 「 『 【
2	¥ $ £ € @ § 〒 ＃

Of course, vertical variants of characters in Tables 7-15 and 7-16 are to be handled accordingly. Also, whether a character is full-width or not should not affect its treatment during line breaking—the characters illustrated in Tables 7-15 and 7-16 have been rendered as full-width, but their non–full-width counterparts, whether they are half-width or proportional, require the same treatment.

Still another category to consider are those characters that are not allowed to be broken on either side, meaning that they cannot begin nor terminate lines, and are thus inseparable, meaning that they cannot be spaced out by tracking or full-justification. In Japanese, these characters are called 分離禁止文字 (*bunri kinshi moji*). Long dashes and dot leaders are such characters. Table 7-17 lists the inseparable characters.

Table 7-17. Inseparable characters

Characters
——・・・・

There are three fundamental methods used to line-break or word-wrap CJKV text. Which one you use often depends upon the program you are using, and most high-end line-layout–capable applications allow the user to choose which style to use in any particular situation. Adobe FrameMaker, for example, automatically uses push-in-first or push-out-first line breaking in order to maintain optimal spacing as the user dictated in the paragraph designer.

The first line-breaking method is known as *push-in-first* (in Japanese, 追い込み禁則処理 *oikomi kinsoku shori*). Push-in-first line breaking works by moving characters that are prohibited from beginning a new line back up to the end of the previous line. It can also shift up a character from the following line such that characters that are prohibited from terminating a line do not. Table 7-18 provides two example texts before any line breaking is applied.

Table 7-18. Before line breaking

Glyph strings
□□□□□□□□□□□「□□□□□□□」□□□□□□□□□□□□□□□□□□、□□□□□□□□□□□、□□□□□□□□。
□□□□□□□□□□□□□□□□□□□□□□□□□□□「□□□□□□□」、「□□□□□□□」□□□□□□□□□□□・□□□□□□□□□□□、□□□□□□□□□□□□□□□。

Table 7-19 illustrates what happens after push-in-first line breaking is applied to the two texts in Table 7-18.

Table 7-19. After push-in-first line breaking

Glyph strings
□□□□□□□□□□□「□□□□□□□」□□□□□□□□□□□□□□□□□□、□□□□□□□□□□□、□□□□□□□□。
□□□□□□□□□□□□□□□□□□□□□□□□□□□「□□□□□□□」、「□□□□□□□」□□□□□□□□□□□・□□□□□□□□□□□、□□□□□□□□□□□□□□□。

The first text in Table 7-19 involved an ideographic comma shifting up to the end of the previous line, and the second text involved shifting up a nonwrapping character.

A second line-breaking method is called *push-out-first* (in Japanese, 追い出し禁則処理 *oidashi kinsoku shori*). Push-out-first line breaking works by forcing characters that are prohibited from terminating a line to shift down to the next line. It is also possible for a character to shift down to the next line so that it precedes a character that is prohibited from beginning a line. Table 7-20 illustrates push-out-first line breaking being applied to the texts from Table 7-18.

Table 7-20. After push-out-first line breaking

Glyph strings
□□□□□□□□□□□「□□□□□□□」□□□□□□□□□□□□□□□□□、 □□□□□□□□□□□、□□□□□□□□。
□□□□□□□□□□□□□□□□□□□□□□□□□□□「□□□□□□□」、「□ □□□□□□」□□□□□□□□□□□・□□□□□□□□□□□、□□□□□□□ □□□□□□□□□□。

Note how push-out-first line breaking can result in lines that appear to prematurely end. While the characters used in the example text provided in Tables 7-18 through 7-20 are set using equal set widths (full-width), other aspects of typography, specifically spacing and alternate metrics, can help to adjust the result.

A third line-breaking technique is called *push-out-only* (in Japanese, it is called ぶら下がり禁則処理 *burasagari kinsoku shori*). This method employs a strategy whereby a character—almost always punctuation, such as a period or comma (Western or ideographic)—is left hanging or otherwise dangling on the right margin, or on the bottom margin if the writing mode is vertical.* These characters appear to hang or dangle off the end of the line, and are sometimes referred to as *dangling* or *hanging punctuation*. Some implementations of this line-breaking method can push out more than one character. Push-out-only is often provided as an option for push-in-first line breaking. Table 7-21 provides text in a state before push-out-only line breaking is applied. Note how the right side of the text block is flush, so even if only part of the punctuation character needs to hang out to fit, the characters are spaced out so that the punctuation will hang out by its full half-em width.

* Unjustified text is considered to have a "ragged" right side.

Table 7-21. Before push-out-only line breaking

Glyph string
□□□□□□□□□□「□□□□□□□」、「□□□□□□□」□□□□□□□ 、□□□・□□□□□□□□□□□、□□□□□□□□□□□□□□□□□。□□ □□□□□□□□「□□□□□□□」□□□□□□□□□□□□□□□□、□□□ □□□□□□□、□□□□□□□□。□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□、□□□□□□□□□□□、□□□□□□□□。

Table 7-22 illustrates the result of applying push-out-only line breaking to the text in Table 7-21.

Table 7-22. After push-out-only line breaking

Glyph string
□□□□□□□□□□「□□□□□□□」、「□□□□□□□」□□□□□□□、 □□□・□□□□□□□□□□□、□□□□□□□□□□□□□□□□□。□□□ □□□□□□□「□□□□□□□」□□□□□□□□□□□□□□□□、□□□□ □□□□□□、□□□□□□□□。□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□、□□□□□□□□□□□、□□□□□□□□。

Very few characters are allowed to dangle, and the JIS X 4051:2004 standard specifies only periods and commas, as shown in Table 7-23.

Table 7-23. Push-out-only characters

Characters
、。，．

Proper application of line breaking is absolutely crucial if a program is to succeed in the market. Some developers have implemented all the line-breaking methods just described, and permit the user to select which one or ones to use.

After CJKV text has been properly line-broken using one or more of the methods described in this section, there still remains the issue of how to right-justify—or bottom-justify if in vertical writing mode—the text in case a ragged edge is not desired. This is discussed later in this chapter in the section entitled "Line Length Issues."

Character Spanning

In the West, we usually think of tabs in terms of the whitespace that serves as an alternate to the use of spaces.* Common tabs include left, right, centered, and decimal. A spanning tab is unique in that it adjusts the spacing between every character that appears before it.

Most CJKV words, with the exception of those for Korean and Vietnamese, are strung together with no intervening spaces, so adjusting the inter-character spacing becomes mandatory. Under most implementations and conditions, CJKV characters—kana, hangul, and ideographs—have equal set widths (that is, they are considered full-width), which effectively means that every character occupies the same amount of typographic space.† Latin characters are typically spaced proportionally, meaning that the widths of characters differ depending on their shape and typeface design. The Latin typeface used for the text of this book, Adobe Minion Pro Regular, is proportionally spaced, but you will occasionally find a mono-spaced font, Courier, used for code samples or other purposes.

In addition to properly span lines of text, there are special ways to span CJKV text on a much smaller scale than the text of entire paragraphs. In Japanese, this is known as 均等割 (*kintō wari*) or 均等割付 (*kintō waritsuke*). This technique is most often used when listing names. Table 7-24 provides an example of a list of Japanese names justified in various ways.

Table 7-24. Examples of character spanning

Default	Narrow spanning	Wide spanning
久保田久美子	久保田久美子	久 保 田 久 美 子
藤本みどり	藤 本 み ど り	藤 本 み ど り
山本太郎	山 本 太 郎	山 本 太 郎
小林劍	小 林 劍	小 林 劍
泉均	泉 均	泉 均

Note how the justification takes place within units such as small text blocks, and that all the characters are adjusted such that they line up equally on both sides. You can think of this as text-level justification as opposed to the usual page-level justification. Nisus Writer, a CJKV-capable word processor for Mac OS X, has provided this type of text-level justification for quite some time through the use of a special type of tab stop. Adobe InDesign

* The golden rule is that you should use a tab whenever you feel the urge to type more than one space.

† It is true that kana and a handful of ideographs benefit from proportional metrics, and typesetting systems are moving in that direction. Proprietary typesetting systems already have this capability.

refers to this functionality as *jidori* (字取り *jidori*), and it is used in conjunction with the character grid.

Alternate Metrics

Most users of CJKV-capable word processors and other text-processing applications are all too familiar with the concept of full-width characters. To a great extent, it represents a limitation. You may have experienced a feeling of being trapped in the mind set that CJKV-specific characters—zhuyin, kana, hangul, and ideographs—must always use equal set widths. While this is certainly typical, it is not always true. However, bear in mind that some CJKV locales, such as China and Taiwan, still follow the convention of setting all characters on a very rigid grid, except perhaps punctuation and bracket-like characters.

A new twist to desktop CJKV type technology is the ability to supplement CJKV fonts with alternate metrics—alternate in the sense that the default metrics are still preserved as full-width, but that the user can now choose alternate widths. Alternate metrics can encompass character classes such as punctuation, symbols, kana, and even ideographs. Some fonts have implemented alternate metrics through additional font instances that appear in applications' font menus—this means that two fonts now appear in applications' font menus where before there was only one. One is typically the fixed-width font, and the other is the one with half-width and proportional metrics. An example of this can be found when exploring Microsoft's Windows (95J, 98J, NT-J, XP-J, or Vista) OS. It includes two basic fonts, MS 明朝 (*MS minchō*) and MS ゴシック (*MS goshikku*). The versions of these fonts that include alternate metrics—proportional kana, half-width symbols and punctuation, and condensed katakana—are called MS P明朝 (*MS P minchō*) and MS Pゴシック (*MS P goshikku*), respectively. The "P" is clearly an abbreviation for "proportional."

The examples that I provide in the following sections will clearly illustrate the use of alternate metrics through contrast with the default (full-width) metrics. And, all of these examples use the alternate metrics that are built into the commercially available versions of the fonts—sfnt-wrapped CIDFonts for Mac OS. Some Mac OS QuickDraw GX fonts and some Microsoft Windows TrueType Open fonts also provided alternate metrics.

Applications, such as Adobe FrameMaker and early versions of Adobe Illustrator, refer to alternate or proportional metrics as *tsume* (詰め *tsume*), which is a Japanese word meaning "squeezing" or "stuffing." In almost all cases, alternate metrics result in either proportional or half-width metrics. Adobe Illustrator now makes use of *mojikumi* tables, such as Adobe InDesign.

Half-Width Symbols and Punctuation

It turns out that most punctuation and enclosing (bracket-like) characters do not completely fill a full-width cell—they almost always occupy less than half of the cell, leaving a lot of unused whitespace on one side, or on all sides.

Table 7-25 lists six classes of traditionally full-width characters that can be set with half-width alternate metrics.

Table 7-25. Half-width symbols and punctuation classes

Half-width classes[a]	Example characters
50% left	（ 〔 ［ ｛ 〈 《 「 『 【 ‘ “
50% right	、 。 ， ． ） 〕 ］ ｝ 〉 》 」 』 】 ’ ”
50% top[b]	⌒ ⌐ ⌐ ⌒ ∧ ⩔ ¬ ⌐ ⌐ ′ ″
50% bottom[b]	、 。 ， ． ◡ ⌣ ⌣ ⌣ ∨ ⩒ ⌞ ⌞ ⌣ ′ ″
25% left and right	： ； ！
25% all sides	・

a. The percentages listed in this column refer to the amount of empty space that is trimmed from the full-width design space in order to end up with half-width metrics.

b. These are vertical variants of characters two rows above—note that some do not have vertical variants.

Note how some characters are specific to vertical writing, but that some characters, such as ・ (centered dot), are the same regardless of writing direction.

Table 7-26 provides an example of Japanese text that uses half-width symbols and punctuation in horizontal writing mode. Whether all of these modes are possible depends on the line- or page-layout application you are using.

Table 7-26. Full-width versus half-width symbols and punctuation—horizontal

Metrics	Glyph String
Full-width	あっ。剣、劍、及び「ふぐ・河豚本」だ。
Half-width	あっ｡剣､劍､及び｢ふぐ･河豚本｣だ｡
Combination[a]	あっ。剣､劍､及び「ふぐ･河豚本」だ。

a. The ideographic periods and commas, along with the center dot, are half-width, but the brackets are full-width.

As a side note, if the Japanese line-layout rules set forth in JIS X 4051:2004 were to be applied together with half-width metrics for the symbols and punctuation, the half-width example Japanese text in Table 7-26 would look identical to the full-width example.

Table 7-27 provides the same example of Japanese text as found in Table 7-26, but this time in vertical writing mode.

Table 7-27. Full-width versus half-width symbols and punctuation—vertical

Full-width	Half-width	Combination
あっ。剣、劍、及び「ふぐ・河豚本」だ。	あっ。剣、劍、及び「ふぐ・河豚本」だ。	あっ。剣、劍、及び「ふぐ・河豚本」だ。

According to the JIS X 4051:2004 standard, the glyphs for all of these punctuation characters, by default, should use half-width metrics. But, when they come into contact with other character classes, such as kana or kanji, they should be given additional space—usually this is a half-width space, which in a way makes them full-width again. These characters themselves are separated into two classes: those that are considered "opening" (see the first line of characters in Table 7-25), and those that are considered "closing" (see the second line of characters in Table 7-25).

If a closing character is immediately followed by an opening character—such as in the text 字典」「新漢—then half an *em* of space is added after the closing character, effectively making the closing character full-width and the opening character half-width. However, if two closing or opening characters come together—such as in the text とは秘密）、及び—there is no additional space inserted. Table 7-28 illustrates these same two texts set horizontally with and without the application of these rules. Adobe Systems' 小塚明朝 Pr6N R (KozMinPr6N-Regular) typeface is used for the example.

Table 7-28. The application of spacing rules

Rule application	Glyph string
No rules	字典」「新漢
With rules	字典」「新漢
No rules	とは秘密）、及び
With rules	とは秘密)、及び

In case it is not obvious, the glyph spacing that results from the application of spacing rules looks much better than the glyph spacing without spacing rules. These spacing rules were formulated for good reason.

The JIS X 4051:2004 standard includes much more detailed information about such spacing rules.

Proportional Symbols and Punctuation

Some of the same punctuation and symbols listed in the previous section (see Table 7-25) can also be set with proportional widths—as opposed to fixed half-width metrics. In fact, there are some characters that must use proportional widths because their design is too large for half-width metrics. A prime example is the full-width Latin characters that are typical of all CJKV fonts.

Table 7-29 illustrates the word "Typography" set horizontally using the full-width Latin glyphs of Morisawa's A-OTF リュウミン Pr6N L-KL (RyuminPr6N-Light) typeface design. When the same glyphs are set using proportional metrics, using the OpenType 'palt' (*Proportional Alternate Widths*) GPOS feature, the inter-glyph spacing is reduced. But, the glyphs themselves are identical. It is also possible to substitute full-width glyphs with those that were specifically designed to be proportional, through the use of the OpenType 'pwid' (*Proportional Widths*) GSUB feature. Depending on the typeface design, the glyphs and relative spacing that results from applying GPOS versus GSUB features may be the same or different. In the case of Morisawa's A-OTF リュウミン Pr6N L-KL (RyuminPr6N-Light) typeface design, as used in Table 7-29, the results look identical, though the underlying GIDs are different.

Table 7-29. Full-width and proportional Latin characters—horizontal

Metrics	Glyph string
Full-width	Ｔｙｐｏｇｒａｐｈｙ
Proportional—GPOS	Typography
Proportional—GSUB	Typography

Table 7-30 illustrates the same word as Table 7-29 using the same typeface design, but this time set vertically.

Table 7-30. Full-width and proportional Latin characters—vertical

Full-width	Proportional—GPOS	Proportional—GSUB
		

There are, of course, other characters that require proportional rather than half-width metrics. Depending on the typeface design, they may include full-width versions of the question mark (？) and kana iteration marks (ヽ, ヾ, ゝ, and ゞ).

Proportional Kana

While most Japanese users are accustomed to using full-width kana in their documents, those who have been using proprietary typesetting equipment know that most kana designs are inherently conducive to being set with proportional widths.

Table 7-31 illustrates a short Japanese sentence, using both full-width and alternate (proportional, in this case) metrics for the kana. TypeBank's タイプバンク明朝 Std M (TypeBankMStd-M) is used in this example.

Table 7-31. Full-width and proportional kana

Metrics	Glyph string
Full-width	きょう、本を買った。
Proportional	きょう、本を買った。

Note how the small kana—the characters ょ and っ in this example—have undergone the most drastic reduction in set width, which results in improved readability. While this is typical for standard kana designs, it is by no means a steadfast rule. In other words, the amount of reduction in set widths varies from design to design.

Table 7-32 illustrates the same Japanese sentence, but this time using an alternate kana design, TypeBank's TB良寛M Std M (TBRyokanMStd-M) combined with TypeBank's タイプバンク明朝 Std M (TypeBankMStd-M) for the kanji.

Table 7-32. Full-width and proportional kana—alternate kana

Metrics	Glyph string
Full-width	きょう、本を買った。
Proportional	きょう、本を買った。

Note how the alternate kana design illustrated in Table 7-32 results in a more drastic difference in metrics between full-width and proportional. Even the standard kana (that is, not only the small kana) have undergone a drastic reduction in set width.

Table 7-33 includes the same examples as provided in Tables 7-31 and 7-32, but set vertically. This clearly demonstrates that proportional metrics also benefit kana when set vertically.

Table 7-33. Full-width and proportional kana—vertical

Full-width	Proportional	Full-width	Proportional
きょう、本を買った。	きょう、本を買った。	きょう、本を買った。	きょう、本を買った。

Note how the alternate kana design provides less of a contrast than with the horizontally set example.

Proportional Ideographs

It is a generally accepted notion that ideographs fit within a square design space. While this notion is true most of the time, there are, of course, exceptions.

Table 7-34 illustrates a short string of ideographs, 中日韓越 ("CJKV"), set horizontally. Note how the second character, 日 (the "J" of "CJKV"), is no longer full-width when proportional metrics are applied. Morisawa's A-OTF リュウミン Pr6N L-KL (RyuminPr6N-Light) is used for the example.

Table 7-34. Full-width and proportional ideographs—horizontal

Metrics	Glyph string
Full-width	中日韓越
Proportional	中日韓越

Similarly, there are other ideographs whose shapes benefit from proportional metrics when set vertically. Table 7-35 illustrates a short ideograph string, 第一勧業 (*dai-ichi kangyō*, the name of a famous Japanese bank), set vertically. Note how the second ideograph, 一 (*ichi*, meaning "one"), has undergone a drastic reduction in overall set width. Again, Morisawa's A-OTF リュウミン Pr6N L-KL (RyuminPr6N-Light) typeface design is used for the example.

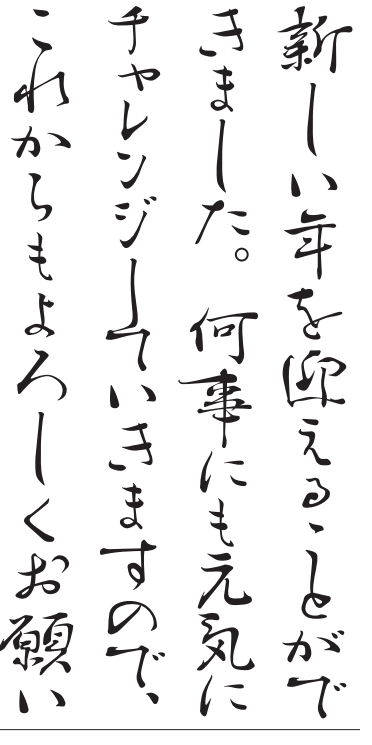
Table 7-35. Full-width and proportional ideographs—vertical

Full-width	Proportional
第 一 勧 業	第 一 勧 業

The proportional ideographs shown in Tables 7-34 and 7-35 are pseudo-proportional in that their default metrics are full-width, but that their shapes allow proportional metrics to be applied, for horizontal or vertical use, but generally not for both. Interestingly, there

now exists a Japanese font that includes genuine proportional ideographs, specifically that the default metrics are proportional for both horizontal and vertical use. Additionally, their shapes do not suggest that they are confined to the prototypical "square" design space. This font in question is Adobe Systems' かづらき Std L (KazurakiStd-Light), and Table 7-36 provides some example text set with this font, in both writing directions.

Table 7-36. Horizontal and vertical layout—proportional Japanese

Horizontal	Vertical
新しい年を迎えることができました。何事にも元気にチャレンジしていきますので、これからもよろしくお願い申し上げます。	

Believe it or not, there are times when applying proportional metrics to ideographs does more harm than good. That is when kerning comes to the rescue….*

Kerning

While alternate metrics provide a way to escape from the trap or mind set of using only full-width characters, it sometimes introduces new problems. Luckily, such problems are solved through the use of kerning, which is the process of adjusting inter-glyph spacing

* This is obviously different from Kern coming to the rescue. My brother's name is Kern. It is quite unfortunate that he knows absolutely nothing about kerning.

according to context. The context in the case of kerning happens to be the proximity of two glyphs. These are called *kerning pairs*. In Western typography using Latin characters, the most common kerning pairs are in the uppercase set. "A" and "V" (such as in the pair-kerned instance "JAVA," and "JAVA" is an example of an instance of the same string that is not pair-kerned) represent one kerning pair—when set together they appear to be too far apart, and are thus kerned.

Kerning, like alternate metrics, is a typeface-dependent attribute. This means that the actual kerning values change depending on the typeface design.

Table 7-37 illustrates a short kana phrase, どうして (*dōshite*, meaning "why"), set using full-width and proportional metrics with Adobe Systems' 小塚明朝 Pr6N R (KozMin-Pr6N-Regular) typeface design, and also with kerning enabled. When this high-frequency phrase is set using proportional metrics, the inter-glyph spacing between し (*shi*) and て (*te*) appears to be too great. This problem is addressed through the proper application of kerning.

Table 7-37. Kerning kana characters

Metrics	Glyph string
Full-width	どうして
Proportional	どうして
Kerned	どうして

There are also times when negative kerning, meaning an increase in inter-glyph spacing, is required for CJKV text. For example, consider the two-ideograph string 一二, which represents the glyphs for the ideographs meaning "one" and "two," respectively. When set vertically, both of these ideographs are clear candidates for proportional metrics. However, if these two ideographs are in proximity, and in the sequence just given, they require negative kerning. Otherwise, they may resemble 三 (the ideograph meaning "three") or the result may simply look bad, depending on the typeface design or the glyphs' default metrics. Table 7-38 illustrates this phenomenon using kanji from Morisawa's A-OTF リュウミン Pr6N L-KL (RyuminPr6N-Light).

Table 7-38. Kerning ideographs

Full-width	Proportional	Kerned
三 年 二 月	三 年 二 月	三 年 二 月

As you can see, the introduction of alternate metrics, along with proper kerning, can make a world of difference in the overall appearance and quality of CJKV documents.

Morisawa & Company* developed an Adobe Illustrator plug-in called Fuzzyカーニング (*fajī kāningu*, meaning "Fuzzy Kerning") that provided the ability to kern Japanese text. They have also developed a QuarkXPress XTension (plug-in) called Dr.カーニング ("Dr. Kerning"). Because of the capabilities of OpenType fonts, which more easily allow kerning information to be included through the use of the 'kern' (*Kerning*) and 'vkrn' (*Vertical Kerning*) GPOS features, these plug-ins are no longer being developed. For users who are satisfied with a pseudo-kerned look, Adobe InDesign includes the ability to adjust glyph side bearings by a specified percentage.

Line-Length Issues

Once you begin applying alternate metrics or adjusting other spacing aspects of CJKV text, you are then left with the problem of how to calculate line length to perform common typographic tasks, such as right-justification. When dealing with text that is set on a rigid grid, such as Chinese, calculating line length becomes a bit more trivial.

The inclusion of Latin text—even one character—is the typical culprit for causing line-length calculation problems in CJKV text. Another culprit is the application of line-breaking rules—adjusting text so that certain characters do not begin or terminate lines. The following sections provide methods for adjusting spacing so that right-justification can be performed.

I need to emphasize the importance of this section. There is a balance, which can be described as the proper handling of two opposing forces. One of these forces wants all glyphs to be set onto a rigid grid. The other force introduces issues, usually in the form of glyphs

* *http://www.morisawa.co.jp/*

that are not full-width, and thus nonconformant to grid-like behavior. Applications must restore the balance by performing minute calculations and adjusting inter-glyph spacing as appropriate, taking advantage of punctuation marks and the spacing between character classes. It is an extremely complex subject, far beyond the scope of this book. This book merely serves as a vehicle to make developers aware of this important issue.

Manipulating Symbol and Punctuation Metrics

As described earlier in this chapter, many symbols and most punctuation marks have forms that lend themselves to being set with half-width or proportional metrics. One of the first methods for adjusting spacing is to apply alternate metrics to one or more symbols or punctuation marks. The preference is usually to apply such metrics to punctuation marks, then symbols.

Table 7-39 illustrates an example three-line text that requires the use of half-width symbols or punctuation marks due to the application of line breaking.

Table 7-39. Adjusting symbol and punctuation metrics

	Glyph string
Before	□□□□□□□□□□□、□□□□□□□、□□□□□□□□□□□□□□□□□「□□□□□□□□□□□」□□□□□□□□□□。□□□□□□□「□□□□□」□□□□□□□□□□、□□□□□□□□□□□□□□□□□□。
After	□□□□□□□□□□□、□□□□□□□、□□□□□□□□□□□□□□□□□「□□□□□□□□□□□」□□□□□□□□□□。□□□□□□□「□□□□□」□□□□□□□□□□、□□□□□□□□□□□□□□□□□□。

The two ideographic commas in the first line are converted to half-width forms, which has the side-effect of reversing the effect of line breaking by allowing another character to terminate the second line instead of the opening bracket.

Manipulating Inter-Glyph Spacing

Texts that include only full-width characters with a few punctuation, and symbols thrown in can be set fully justified by manipulating the spacing of punctuation and symbols—as illustrated in the previous section. However, when any proportional glyphs come into play, inter-glyph spacing often needs to be adjusted in order to achieve right-justification.

Table 7-40 illustrates an example of inter-glyph spacing being used to help right-justify text that includes glyphs with proportional widths.

Table 7-40. Adjusting inter-glyph spacing

	Glyph string
Before	Japan、California、□□□ South Dakota □□□□□□□□□□□□□□□ □□□「□□□□□□□□」□□□□□□□□□□。□□□□□「□□□□ □□□□」□□□□□□□□□、□□□□□□□□□□□□□□□□□□。
After	Japan、California、□□□ South Dakota □□□□□□□□□□□□□□□ □□□「□□□□□□□□□」□□□□□□□□□□□。□□□□□「□□□□ □□□□」□□□□□□□□□、□□□□□□□□□□□□□□□□□□。

Note how the inter-glyph spacing of the characters in the first line of the text is adjusted so that it can be properly right-justified.

JIS X 4051:2004 Character Classes

The JIS X 4051:2004 standard establishes character classes for the purpose of properly specifying inter-glyph spacing. Inter-glyph spacing is all about context, specifically when glyphs from different character classes come into contact or are in proximity. These character classes are often referred to as *mojikumi* (文字組み *mojikumi* in Japanese) classes. *Mojikumi* is the Japanese word that means "text composition," and obviously refers to typography.

Interestingly, the three versions of the JIS X 4051 standard specify slightly different sets of character classes. Table 7-41 lists the character classes that are specified by the JIS X 4051:2004 standard, and additionally indicates which ones were present in the 1993 and 1995 versions of the standard through the use of the integer identifier that are used in the standard itself.

Table 7-41. JIS X 4051 character classes

2004 character class	1993 equivalent	1995 equivalent
1—Opening parentheses and quotation marks	same	same
2—Closing parentheses and quotation marks	same	same
3—Japanese characters prohibited from starting lines	3[a]	same
4—Hyphens and hyphen-like characters	n/a	n/a
5—Question and exclamation marks	4	4
6—Bullets, colons, and semicolons	5	5
7—Periods	6	6
8—Inseparable characters	7	7
9—Prefixed abbreviated symbols	8	8
10—Suffixed abbreviated symbols	9	9

Table 7-41. JIS X 4051 character classes

2004 character class	1993 equivalent	1995 equivalent
11—Ideographic space	10	10
12—Hiragana	n/a	11
13—Japanese characters other than those in classes 1–12	11[b]	12[c]
14—Characters used in note references	n/a	n/a
15—Body characters of an attached sequence	n/a	13
16—Body characters of an attached ruby other than a compound ruby	n/a	14[d]
17—Body characters of an attached compound ruby	n/a	n/a
18—Characters used in numeric sequences	12	15
19—Unit symbols	13	16
20—Latin space	14	17
21—Latin characters other than a space	15	18
22—Opening parentheses for inline notes	n/a	19
23—Closing parentheses for inline notes	n/a	20

a. Full-width characters prohibited from starting lines.

b. Japanese characters other than those above.

c. Japanese characters other than those in classes 1–11.

d. Body characters of attached ruby.

Table 7-41 indicates that the names of some of the character classes have changed, along with the integers associated with them. The integer identifiers are important, because some applications refer to these JIS X 4051 character classes through the use of these integer identifiers. This effectively means that the character class integer identifiers must be tightly bound to a specific version or year of the JIS X 4051 standard.

In terms of the changes that took place in this standard, specifically referring to the assignment of integer values to the character classes, ideographs are assigned to JIS X 4051:2004 Character Class 13. The same character class in the 1993 and 1995 versions of the standard are assigned the integer values 11 and 12, respectively.

In any case, once these character classes have been established, which constitutes character properties, there are two additional properties or behaviors that need to be defined, specifically the following:

Default glyph metrics
: As an example, commas, periods, bullets, colons, semicolons, and quotation marks are treated as though they have half-width metrics. If the glyphs that correspond to these characters do not use half-width metrics, the application must compensate appropriately. In the vast majority of Japanese fonts, these glyphs use full-width metrics.

Inter-glyph spacing according to character class

When the glyphs that correspond to characters of different character classes come together in a run of text, there is spacing behavior. In other words, extra space, measured using a fraction of an *em*, is introduced depending on which two character classes are in proximity.* Typical values are one-fourth and one-half of an *em*.

There is much more to the JIS X 4051:2004 standard than what is described in this section, and in this chapter for that matter. I strongly encourage you to acquire and explore the JIS X 4051:2004 standard, along with applications that implement its functionality.

Multilingual Typography

Mixing together the glyphs that correspond to Latin and CJKV characters in a single document, which can be referred to as multilingual text, requires that one take into consideration many typographic issues, such as the Latin baseline, the spacing between Latin and CJKV glyphs, and selection of appropriate typeface designs. Let us explore each of these issues in the following sections.

Latin Baseline Adjustment

The glyphs for CJKV-specific characters, such as zhuyin, kana, hangul, and ideographs, do not rest on a baseline as is the case for the glyphs for Latin characters. Instead, these glyphs are optically or visually centered within the character cell.† Correct or consistent use of a baseline is the first obstacle you are likely to encounter when mixing Latin and CJKV text.

All of Adobe Systems' CJKV fonts include CJKV-specific characters that are set in a 1000-unit design space that extends from the coordinates 0,−120 to 1000,880. That is, a 1000×1000 cell that is lowered 120 units in relation to the baseline at Y coordinate 0. Figure 7-1 illustrates this design space using glyphs from Adobe Systems' Adobe 宋体 Std L (AdobeSongStd-Light) typeface.

Note how the glyph for the full-width Latin character "A" is resting comfortably on the baseline located at Y coordinate 0 (commonly referred to as the Latin baseline), but that the hanzi 剑 (*jiàn*) is optically centered within the entire 1000×1000 cell without regard to the baseline at Y coordinate 0.

* In Japanese, this extra space is referred to as *aki* (空き *aki*), which simply means "space."

† The glyphs for some CJKV-specific characters, such as the small kana used in Japanese, are exceptions to this general rule.

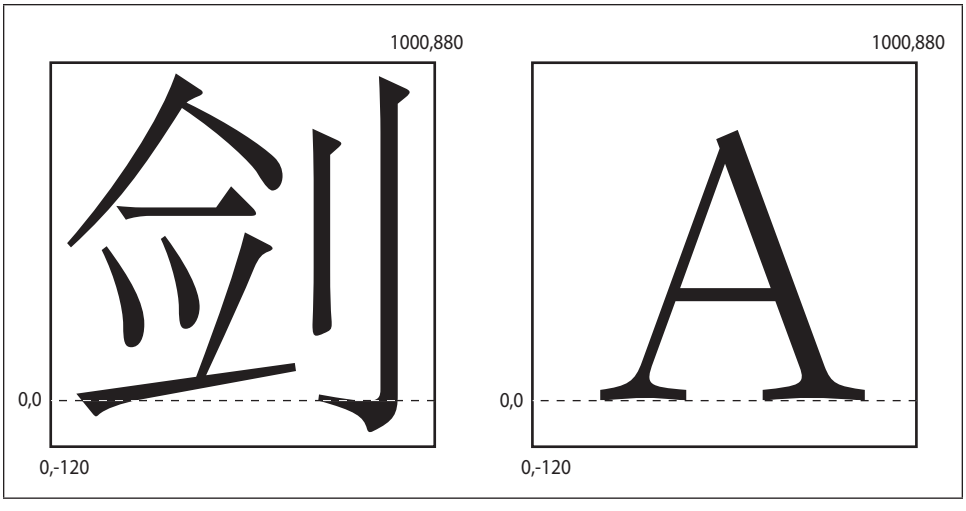


Figure 7-1. 1000×1000 character design space

Other type foundries, however, often use a different design space for CJKV-specific glyphs. 0,–200 to 1000,800 has been widely used by many CJKV type foundries, such as Changzhou SinoType Technology, Seoul Systems, SoftMagic, and Yoon Design Institute. I have also encountered font data that used the following design spaces:

- 0,–110 to 1000,890—Monotype Imaging
- 0,–130 to 1000,870—EulHae
- 0,–160 to 1000,840—Fontworks
- 0,–166 to 1000,834—Hanyang Information & Communications

High-end, page-layout applications, such as Adobe InDesign, automatically calculate the ideographic em-box of all fonts so that glyphs will be placed correctly relative to one another on the line, and on a character grid. Such applications also allow the user to adjust the baseline and size ratios in order to better adapt Latin fonts for use with CJKV characters, which is done in the context of defining composite fonts.

So, when does a different baseline adversely affect typography? When mixing fonts whose glyphs are set in different design spaces. This is when it is critical that an OpenType font includes a well-specified 'BASE' table that explicitly specifies the relative position of design space. Without a well-specified 'BASE' table, applications are forced to use heuristics.

Proper Spacing of Latin and CJKV Characters

Exactly how you space, or adjust the spacing between, Latin and CJKV glyphs that are in proximity—that is, adjacent—very much depends on the nature of the document that you are creating. Different types of documents have different relative quantities of these

glyphs, which is an important aspect in the decision-making process with regard to spacing and similar typographic adjustments.

If the document is primarily composed of CJKV glyphs with some Latin glyphs sprinkled throughout, then the convention or principle is to use extra space to separate these two classes of glyphs. Conversely, if the document is primarily composed of Latin glyphs with CJKV glyphs sprinkled around, such as this book, then conventional Latin spaces suffice, given the extent to which spaces are important in Western typography. Table 7-42 provides example text that is primarily Japanese, with Latin glyphs included, using Adobe Systems' 小塚明朝 Pr6N R (KozMinPr6N-Regular) typeface design.

Table 7-42. Spacing between Latin and CJKV glyphs

Spacing	Glyph string
Solid	誕生日は1965.08.12です。
Extra space	誕生日は 1965.08.12 です。

Note the thin spaces that are used between the Latin and CJKV glyphs—this serves to improve overall readability, making the transition between glyph classes far less abrupt. The use of standard spaces results in too much space between these two glyph classes. Most CJKV-capable page-layout applications either provide the ability to use thin spaces—which is not desirable, because it becomes a manual operation—or else the ability to automatically insert extra space when appropriate, based on line-layout rules and principles, such as those set forth in JIS X 4051:2004.

According to the JIS X 4051:2004 standard, there should be a quarter-width space between Latin and CJKV glyphs, but sophisticated line-layout applications allow the user to adjust this value as necessary. In fact, many Japanese magazine publisher "house rules" specify a much tighter spacing amount, and thus regard the JIS standard as being too loose. The bottom line is that some amount of space should be used to separate Latin and CJKV glyphs so that they are not set solid. While the JIS X 4051:2004 standard provides guidance in this regard, you are free to use whatever amount of space you see fit.

Interestingly, because Korean text is primarily composed of hangul and uses conventional spaces as a word separator, Korean typography has more in common with Western typography than with Chinese or Japanese typography. Korean text, for example, typically uses Western punctuation. However, like Chinese and Japanese text, Korean words consisting of more than one hangul can be broken across lines.

Mixing Latin and CJKV Typeface Designs

Every CJKV font, perhaps with the exception of Vietnamese, typically includes a minimally functional set of glyphs for Latin characters, usually encoded in the one-byte range for legacy font formats. On Mac OS, these Latin glyphs were almost always proportional-width. Although these Latin glyphs look well with the rest of the glyphs of the font, there are times when a different Latin font should to be used. This book, for example, uses a variety of CJKV fonts for its non-Latin content, but the Latin portion is consistently the same: Adobe Minion Pro for body text, and Adobe Myriad Pro for headlines and inside tables. Although the first edition of this book, which used the ITC Garamond family, required me to develop edited—meaning modified or hacked—versions of ITC Garamond to accommodate certain typographic aspects of the book (such as the macroned vowels used for transliterated Japanese text, for Chinese Pinyin, and for Vietnamese), the fonts used for this edition support a much larger glyph set, and editing was not necessary.

Okay, so when do the built-in Latin glyphs suffice, and when is a separate Latin font required? It depends on the nature of the text. Documents that consist primarily of CJKV glyphs with a few Latin glyphs sprinkled throughout typically do not require the so-called "typographically correct" glyphs that are typically found in Latin fonts. These include smart (or curly) quotes, the various dashes, and ligatures. CJKV fonts either have these glyphs available—usually as full-width forms—or else have no need for them. This is why the built-in Latin glyphs of CJKV fonts usually suffice for these types of documents. Text that is composed primarily of Latin glyphs requires Latin fonts in order to provide typographically pleasing results.

Another consideration is whether cross-platform compatibility is an issue. Many CJKV-capable applications these days are available for multiple OSes. Adobe FrameMaker, at one point, was available for Mac OS, Unix, and Windows.* However, the built-in Latin glyphs of most CJKV fonts sometimes have different metrics depending on the OS on which they are installed. To a great extent, OpenType has solved this problem, because the same font file is truly cross-platform, and thus provides the same metrics regardless of the platform. However, in the past, PostScript Japanese fonts installed onto Mac OS typically used proportional-width Latin glyphs, but the same fonts instantiated as different font files and installed onto Windows typically used half-width Latin glyphs. Simply selecting a separate Latin font for Latin glyphs, besides providing access to typographically correct characters, such as the various dashes, smart quotes, and Latin ligatures, was a way to better ensure some degree of cross-platform compatibility. The first edition of this book, for example, was set in ITC Garamond Light, an independent Latin font. The CJKV glyphs come from a variety of CJKV fonts, as listed in that book's colophon.

* Unfortunately, there is no Mac OS X version of Adobe FrameMaker. There has been a petition in the works since 2004 to convince Adobe Systems' management of the error of their ways. See *http://www.fm4osx.org/*.

Vietnamese typeface issues

A fully functional Vietnamese font, one that includes glyphs for both Latin characters and ideographs, is an example that brings together the complexities from both worlds of typography. While readers of this book should now be well aware of the issues that surround the use of ideographs in typography, Latin characters used in Vietnamese, which are called Quốc ngữ, require extensive use of diacritic marks for base characters and tones. Chinese, Japanese, and Korean text, when transliterated, requires diacritic marks to indicate vowel length, tone, or other phonetic attributes. However, these transliteration systems are not considered the primary orthography for these locales, so their fonts do not require these additional glyphs. Vietnamese or Vietnamese-enabled fonts, on the contrary, require that these additional glyphs to be minimally functional.

It is safe to conclude that the ideal environment or basis for a fully functional Vietnamese font is Unicode, in which all necessary characters are encoded, and thus easily accessible. Table 7-43 illustrates the lowercase simple vowels, including "y," along with the required permutations for Vietnamese Quốc ngữ.

Table 7-43. Simple vowels and their Quốc ngữ permutations—lowercase

Simple vowel	Required Quốc ngữ permutations
a	à ả ã á ạ ă ằ ẳ ẵ ắ ặ â ầ ẩ ẫ ấ ậ
e	è ẻ ẽ é ẹ ê ề ể ễ ế ệ
i	ì ỉ ĩ í ị
o	ò ỏ õ ó ọ ô ồ ổ ỗ ố ộ ơ ờ ở ỡ ớ ợ
u	ù ủ ũ ú ụ ư ừ ử ữ ứ ự
y	ỳ ỷ ỹ ý ỵ

That's right, there are anywhere from 5 to 17 required permutations for a single unadorned vowel. Uppercase requires the same set of permutations.

Glyph Substitution

Glyph substitution, the act of substituting one glyph for another, is an incredibly useful typographic feature. Given the extent to which OpenType is supported in today's applications, glyph substitution is no longer a trick or gimmick, but rather a fairly standardized

feature—at least for higher-end applications, many of which are described at the end of this chapter.

There are four basic types of glyph substitution that benefit CJKV fonts, many of which equally apply to *any* font, regardless of language or script:

- Vertical substitution
- Variant substitution
- Ligature construction
- Ligature decomposition

Vertical substitution is a form of glyph substitution that is expected to take effect when text is typeset in vertical writing mode. It is almost always considered a functionality that is invoked automatically when a document is being typeset in vertical mode. With the exception of standard ligatures, such as *fi* and *fl*, the remaining types of glyph substitution are explicitly invoked by the user.

Character and Glyph Variants

When considering the various types of characters found in CJKV text, one of the most common types of substitution involves the ability to select different forms of a character, such as traditional, simplified, or variant (异体字/異體字 *yìtǐzì* in Chinese, 異体字 *itaiji* in Japanese, and 이체자/異體字 *icheja* in Korean) forms. Some variants can be defined more specifically, such as JIS78 (JIS C 6226-1978) for Japanese. And, some variants do not necessarily involve ideographs, such as annotated forms.

Table 7-44 lists several classes of glyph variants, along with appropriate examples for each using a variety of typeface designs. When appropriate, the corresponding OpenType GSUB feature tag is indicated.

Table 7-44. Glyph variant examples

Glyph substitution	Default glyph	Glyph substitution results
To traditional—trad	台	臺, 颱, 檯
To simplified—smpl	國	国
To variant—aalt	辺	邊, 邊
To JIS78—jp78	唖	啞
To hangul—hngl	樂	낙, 락, 악, 요
To annotated—nalt	あ	(あ), ㋐, 🅐, 🄐, 🅰, 🄰, 🅰

Note the numerous cases of one-from-*n* substitutions, especially the Japanese-specific example, whose entry point to the 'aalt' (*Access All Alternates*) GSUB feature is the kanji 辺. Yes, there exist Japanese fonts that include 23 variant forms that are associated with that kanji. In any case, you are likely to encounter one-to-one and one-from-*n* substitutions, depending on the class of variant substitution, the font you are using, and the application's ability to make use of glyph substitution features. Luckily, given the broad support for OpenType, the number of applications that support glyph substitution is increasing, not decreasing.

A good example of a Japanese phrase that clearly illustrates the benefits of applying glyph substitution is 学校の桜並木に黒い虫. This phrase was coined by Kazuo Koike (小池和夫 *koike kazuo*) on the Font-G mailing list over 10 years ago, and means "black bugs stick on the cherry trees in the schoolyard." This is pronounced *gakkō-no sakura namiki-ni kuroi mushi*. When this phrase is rendered using alternate or variant forms made accessible through glyph substitution features, the results are as shown in Table 7-45.

Table 7-45. Glyph substitution example—Japanese

Glyph substitution	Glyph string
None	学校の桜並木に黒い虫
JIS83 forms	学校の桜並木に黒い虫
Traditional forms	學校の櫻竝木に黑い蟲
JIS83 and traditional forms	學校の櫻竝木に黑い蟲

Interestingly, on Mac OS, meaning before Mac OS X, some of these alternate forms were not easily accessible simply because they were not available in the character set that was supported. These included the kanji 校 (the JIS83 form of 校) and 黑 (the traditional form of 黒). Now, through the use of the OpenType 'jp83' (*JIS83 Forms*) and 'trad' (*Traditional Forms*) GSUB features, these glyphs are easily accessed. The other traditional forms are simply encoded elsewhere, and the 'trad' GSUB feature merely serves as a convenience mechanism for accessing the traditional forms.

Ligatures

Ligatures, single characters whose underlying glyphs are those of other characters, are also very common, especially in Japanese.* This is also known as *n*-to-one substitution. The most common types of CJKV ligatures include those composed of kana, hangul, ideographs, and Latin characters. Variants of ligatures—but not all ligatures—include abbreviated and vertical forms. Table 7-46 illustrates examples of ligatures for CJKV text.

Table 7-46. Ligature examples

Ligature type	Original text	Ligature form	Vertical variant
Kana ligature	キログラム[a]	キロ グラム	グキ ムロ
Hangul ligature	주식회사[b]	주식 회사	회주 사식
Ideograph ligature	株式会社[c]	株式 会社	会株 社式
Latin ligature	FAX[d]	FAX	n/a

a. Read *kiroguramu*, meaning "kilogram."

b. Read *jusik hoesa*, meaning "incorporated."

c. Read *kabushiki gaisha*, meaning "incorporated."

d. An abbreviation for "facsimile," in case you haven't kept up with technology.

Some ligatures have abbreviated forms. That is, not all of the underlying glyphs that serve to compose the ligature need to be in the abbreviated form—they are also enclosed. The abbreviated ligature form of the four ideographs 株式会社, for example, is ㈱. It is also considered to be a form of the ideograph 株 annotated with parentheses.

There have historically been two styles of katakana ligatures used for Japanese, both of which are found in character set standards and glyph sets: *justified* and *centered*. These two styles are distinguished only for those ligatures that consist of three elements. The Adobe-Japan1-6 character collection includes both styles for those katakana ligatures that have legacy issues. The preferred style, from a typographic perspective, is the justified style. Specifically, the third element is left-justified for its horizontal form, and top-justified for its corresponding vertical form. Table 7-47 provides several examples of justified and centered katakana ligatures.

* From one perspective, ligatures are the most common in Korean, whose hangul are ligatures based on jamo. Does this mean that the hangul ligature example in Table 7-46 is actually a nested ligature?

Table 7-47. Justified and centered katakana ligature examples

Style	Original katakana text	Ligature form	Vertical variant
Justified	グラム[a]	グラム	ムグラ
	ページ[b]	ページ	ジペー
	ワット[c]	ワット	トワッ
Centered	グラム	グラム	ムグラ
	ページ	ページ	ジペー
	ワット	ワット	トワッ

a. Pronounced *guramu*, meaning "gram."

b. Pronounced *pēji*, meaning "page."

c. Pronounced *watto*, meaning "watt."

I should point out that the Adobe-Japan1-6 character collection includes a large number of katakana ligatures, and the centered style is found only in Supplements 0 and 1. Those found in Supplements 4 and 6 are only in their proper justified form.

Some fonts, such as かづらき Std L (KazurakiStd-Light), are special-purpose, and has additional ligature forms not found in typical Japanese fonts. This font includes a small number of hiragana ligatures that are effected only in vertical writing mode. Table 7-48 provides examples of the two- and three-character hiragana ligatures that are provided in this font.

Table 7-48. Vertical hiragana ligatures—examples

Two-character—off[a]	Two-character—on	Three-character—off[b]	Three-character—on
			

Table 7-48. Vertical hiragana ligatures—examples

Two-character—off[a]	Two-character—on	Three-character—off[b]	Three-character—on
			
			

a. These three two-characters hiragana strings are ある (*aru*), こと (*koto*), and して (*shite*).

b. These three three-characters hiragana strings are しかし (*shikashi*), として (*toshite*), and などの (*nadono*).

Ligatures are generally made accessible through the use of the 'liga' (*Standard Ligatures*) or 'dlig' (*Discretionary Ligatures*) GSUB feature of OpenType fonts. For most applications, the 'liga' GSUB feature is turned on by default, and ligatures that are expected to be used by default, such as *fi* and *fl* for Latin fonts, or the hiragana ligatures found in the special-purpose かづらき Std L font, should be included in this GSUB feature. Other ligatures are more appropriate for including as part of the 'dlig' GSUB feature definition.

Annotations

While many of the typographic features and functionalities described thus far may have seemed Japanese-specific due to the frequent use of Japanese examples, most apply equally well to Chinese, Korean, and Vietnamese text. And, most of the features make perfect typographic sense, such as rules that forbid certain classes of characters from beginning or terminating lines. But, there are some other aspects of typography that are very much Japanese-specific. This includes compliance with the JIS X 4051:2004 standard, the use of ruby glyphs and other annotations, and inline notes.

Ruby Glyphs

Ruby (ルビ *rubi*) glyphs, quite simply, are reduced-size kana glyphs that appear above (or sometimes below) one or more kanji, and act to annotate characters by indicating their reading. This often helps readers, both native and nonnative, to understand the meaning

or reading of rarely used characters and words.* When in vertical writing mode, ruby glyphs typically appear to the right of the kanji to which they are associated. Ruby glyphs are usually referred to as *furigana* (振り仮名 *furigana*) or as *glosses* in the academic world, but in typographic circles are referred to as *ruby*.

All the genuine Japanese ruby glyphs used in the sections of this chapter that follow are from Adobe Systems' 小塚明朝 Pr6N R (KozMinPr6N-Regular) design, and as explained in Chapter 6, are genuine ruby designs.

Applying ruby

There are two contexts in which ruby glyphs are used. One context is global in the sense that all kanji are annotated with ruby to indicate their readings. Children's books use ruby glyphs quite extensively—Japanese children first learn kana, and then kanji in a highly incremental fashion. These small annotations written using reduced-size kana glyphs allow Japanese children—and foreigners who are learning to read Japanese—to learn kanji readings.

The other context are typical documents in which there may appear rarely used or nonstandard kanji—only words containing these kanji are annotated with ruby glyphs. Native Japanese speakers, other than children, are expected to be able to read the unannotated kanji, at least for frequently used ones.

Mono ruby

The simplest type of ruby are called *mono ruby* (モノルビ *mono rubi* in Japanese), aptly named because one or more ruby glyphs serve to annotate only a single parent glyph. Because readings for kanji can require up to as many as five kana, the number of required ruby glyphs must match. Table 7-49 provides examples of mono ruby that consist of one to five ruby glyphs.

Table 7-49. Mono ruby examples—Japanese

One ruby	Two ruby	Three ruby	Four ruby	Five ruby
こ 小	けん 劍	はやし 林	あらかじ 予	こころざし 志

This brings us to another issue with regard to ruby glyphs, specifically how they are aligned with respect to the characters that they annotate. The examples provided in Table 7-49 were all center-aligned, but Table 7-50 illustrates examples of left-, center-, and right-alignment.

* No one can read all kanji. Well, at least those of us who are mere mortals.

Table 7-50. Mono ruby alignment variations examples—Japanese

Number of ruby glyphs	Left-alignment	Center-alignment	Right-alignment
One	こ 小	こ 小	こ 小
Three	はやし 林	はやし 林	はやし 林

There is little benefit in discussing the alignment of mono ruby consisting of only two ruby glyphs, because the glyphs obviously align in one way only.

Of course, all of these alignment principles apply equally well when the text is set vertically—the ruby glyphs appear on the right side of the glyphs that they serve to annotate.

Group ruby

One of the most common types of ruby are called group ruby (グループルビ *gurūpu rubi* in Japanese). That is, they are ruby glyphs that serve to annotate two or more characters, usually kanji. Exactly how they are aligned becomes more of an issue than with mono ruby.

Table 7-51 provides some examples of kanji compounds annotated with ruby glyphs according to group ruby principles. I should point out that the examples in Table 7-51 are somewhat contrived, because using orthodox readings in group ruby violates common "house rules."

Table 7-51. Group ruby examples—Japanese

Example	Example	Example
かんじ 漢字	かぶしきがいしゃ 株式会社	はんちゅう 範 疇
kanji	*kabushiki gaisha*	*hanchū*
kanji	incorporated	category

It is also possible to combine the two ruby principles for strings of kanji that have logical separations, such as personal names in which there is precedent to distinguish family from given names. Table 7-52 provides two examples of applying both types of ruby principles to Japanese names.

Table 7-52. Combining mono and group ruby principles—Japanese

Group and group	Mono and group
やまもとたろう 山本太郎	はやしともこ 林 朝子
yamamoto tarō	*hayashi tomoko*

The complexity of typesetting ruby glyphs goes well beyond the simplistic examples that I have provided. To learn more about how to typeset ruby glyphs, I suggest that you study the JIS X 4051:2004 standard, and also explore the rich ruby settings made available by Adobe InDesign.

Pseudo ruby

There are also special-purpose ruby glyphs, which basically can mean one of two things—or both:

- The ruby glyphs are not part of a special-purpose ruby font
- The use of glyphs for characters other than hiragana and katakana (and a few specialized ruby-specific symbols), such as glyphs for Latin characters or kanji

These are called *pseudo ruby* (擬似ルビ *giji rubi* in Japanese) glyphs. For example, words written using katakana can be annotated with ruby glyphs that are actually reduced-size kanji—these are most often used to indicate the Japanese equivalent of loan words. Kanji compounds can also be annotated with ruby glyphs that are merely reduced-size Latin glyphs. Table 7-53 provides some examples of pseudo ruby.

Table 7-53. Pseudo ruby examples—Japanese

Kanji	Kanji	Latin characters
拳 銃 ピストル	計 算 機 コンピュータ	N E C 日本電気
pisutoru	*konpyūta*	*nippon denki*
pistol	computer	NEC

You may find that some Japanese word-processing applications support the typesetting of ruby glyphs, but it is a much more common feature of page-layout applications. Adobe FrameMaker version 5.5 and later, for example, includes support for ruby glyphs. Adobe InDesign's support for ruby is perhaps the richest among capable applications, and takes advantage of the 'ruby' (*Ruby Notation Forms*) GSUB feature that is included in OpenType Japanese fonts that are based on Adobe-Japan1-4 or greater Supplement.

One can imagine how ruby glyphs can be applied to Chinese, Korean, and Vietnamese. Chinese text can be annotated with Pinyin or zhuyin, Korean text (that is, hanja) can be annotated with hangul, and Vietnamese text (that is, chữ Hán and chữ Nôm) can be annotated with Quốc ngữ. In fact, Chinese text annotated with zhuyin characters—as readings—is quite common.

Inline Notes—Warichu

Japanese text can be set as inline or inset notes, called *warichu* (割注 *warichū*) in Japanese. The text for inline notes is the same as ruby glyphs (that is, half-size), but are set within the text using two lines enclosed by a set of parentheses. Table 7-54 provides an example of Japanese text that includes an inline note, set using Adobe Systems' 小塚明朝 Pr6N R (KozMinPr6N-Regular) typeface.

Table 7-54. Inline note example

Glyph string
情報処理（この本のテーマは日中韓越情報処理）です。

While it is theoretically possible to fake kana, hangul, and ideograph ligatures using the principles of inline notes, the results are far from being typographically pleasing. While the glyphs used for inline notes are purposely and intentionally reduced in size, applying the same technique for constructing ligatures will result in glyphs that do not match the surrounding text, and will look more like inline notes, because that is precisely what they are. Table 7-55 provides an example of genuine versus faked katakana ligatures in text set using FDPC's 平成角ゴシック Std W5 (HeiseiKakuGoStd-W5) typeface.

Table 7-55. Genuine versus faked katakana ligatures

Ligature type	Glyph string
Genuine	身長は 188 ㌢です。
Faked	身長は 188 センチです。

Pay close attention to how the vertical spacing and relative weight of the two types of ligatures differ. The ligature form in question is ㌢ (*senchi*, short for and meaning "centimeter"). The genuine ligature blends with the surrounding text while the faked one does not. The faked ligature's vertical spacing is such that it looks like two separate lines of text—this is precisely what inline notes entail.

More information about inline note composition can be found in the JIS X 4051:2004 standard. Adobe Illustrator version 7.0 and later supports inline notes.

Other Annotations

In addition to annotating ideographs or other characters with readings or meanings, it is also possible to add annotations that simply serve to add emphasis, comparable to the use of an underline in Western text. In Japanese, these marks are called *boten* (傍点 *bōten*), and appear above, in horizontal writing, or to the right, in vertical writing, of the characters to which they add emphasis. These marks are sometimes referred to as *kenten* (圏点 *kenten*).

Table 7-56 illustrates the most common type of *boten* mark used in Japanese, which is a simple black dot, along with an alternate type, which is sometimes called a *sesame dot*, both of which are typeset using Adobe Systems' 小塚明朝 Pr6N H (KozMinPr6N-Heavy) typeface.

Table 7-56. Standard and alternate Japanese boten marks

Boten style	Glyph string
Standard	このサンプルは重要です。
Alternate	このサンプルは重要です。

Some page-composition systems, such as QuarkXPress, minimally support these two types of *boten* marks. Adobe InDesign and Canon EDICOLOR provide nearly a dozen types of these glyph annotations, and also permit the user to define and use their own in case the supplied set is deemed to be too limited for the intended use. Table 7-57 illustrates the 10 *boten* marks that are made available by Adobe InDesign, which are called *kenten* marks in the context of that application.

Table 7-57. InDesign kenten marks

Name	Example
Sesame dot	このサンプルは重要です。
White sesame dot	このサンプルは重要です。

Table 7-57. InDesign kenten marks

Name	Example
Snake eye	
Black circle	
Small black circle	
Double circle	
Black triangle	
White triangle	
White circle	
Small white circle	

Typographic Applications

Although typical word-processing applications allow the user to adequately typeset most documents, there are often circumstances when a significantly greater degree of typographic control is necessary, such as when composing books or other complex documents.

Today, page-layout and some graphics applications provide the user with a high level of typographic functionality—many also offer CJKV-specific features. What I describe next are what I consider to be the most widely used typographic applications available today. Some are locale-specific in that their UI is tailored for a specific language, and some are available with a more diverse UI.

Page-Layout Applications

Page-layout applications are considered the most complex CJKV text-processing tools available, because they allow users to typeset text in a variety of ways, and with much greater precision than tools such as text editors and word processors. There are many dedicated CJKV-capable page-layout systems available, but they tend to be proprietary in nature, and are very expensive and not easily upgraded.

Most page-layout systems—with the exception of Adobe InDesign, Adobe FrameMaker, and perhaps QuarkXPress—are not very efficient for direct text entry. It is usually necessary to first enter the text using conventional word-processing applications, such as Microsoft Word, then import the text into the page-layout application for further manipulation. Adobe InCopy, the text-editing module for Adobe InDesign, is especially suited for text entry. There are usually filters available that allow you to retain certain attributes of the text during the import process, such as the fonts used (names, point sizes, styles, and so on), tab settings, and line spacing. Some page-layout systems, such as Adobe FrameMaker, have always been effective for text entry, and for such applications, there is absolutely no need to use a word processor as part of the document workflow.

You can expect to find in these applications features such as a vertical writing mode (except for the current version of Adobe FrameMaker), multiple columns, full control over glyph, word, and line spacing, the ability to construct tables, and rudimentary graphics functionality.*

While my personal preference for page layout is Adobe InDesign, you will find that most applications—not only page-layout applications—have extremely loyal followings. As the complexity of a program increases, so does its learning curve. This is especially true of page-layout systems that use a slightly different paradigm for laying out text. While fierce competition in this market has led to competitive upgrades and import filters among these applications, completely transitioning from one page-layout system to another is not so simple and can literally take months of learning and experience. Sticking with the system that you currently use has an advantage in that you are more likely to be aware of its strengths and weaknesses, and more importantly, know how to compensate for its weaknesses and take best advantage of its strengths. Learning the strengths and weaknesses of a new page-layout system can be a time-consuming process.

I must also point out that some word processors, such as the Japanese version of Microsoft Word, have evolved to the point where they provide very sophisticated page-layout facilities, rivaling some of those described in this section.

* Although better results typically come from creating graphics through the use of a dedicated graphics application (such as Adobe Illustrator or Adobe Photoshop), which are subsequently imported as an EPS or PDF file, or if supported by the application, as a native file.

Adobe InDesign

In my opinion and experience, Adobe InDesign is the premier page-composition application available today, with the richest and most diverse collection of text composition and typographic controls.* Its support for high-end Japanese typography is simply unmatched. Although Adobe InDesign is considered by many to be an application that was originally designed for the U.S. market, and merely localized and enhanced for Japanese and other CJKV locales, its support for Japanese began from version 1.0.† The current version, at least as I write this book, is version 5.0, which is better known as Adobe InDesign CS3. As usual, fully functional trial versions of this application are available.

Composition and layout of text in Adobe InDesign is performed through the use of single- and multiple-line text composers. In other words, the decisions for line breaking and spacing can use varying degrees of context. For the single-line text composer, the context is a single line. In the case of the multiple-line text composer, the entire paragraph serves as its context and as a useful default. And, for both types of text composers, there exist Western and Japanese versions. To enable Japanese functionality, such as vertical writing mode behavior and JIS X 4051:2004 compliance, the Japanese composer must be enabled for the text being manipulated, which is at the paragraph level.

One of the more powerful features of Adobe InDesign is its Glyph Panel, which allows the user to access any glyph in any font, sorted by *Glyph ID* (GID) or by Unicode. When the selected font is CID-keyed, although the glyphs are sorted by GID, the CID of a glyph is displayed when the cursor moves over it. The Glyph Panel also represents a way in which to access many OpenType GSUB features, many of which are specific to CJKV fonts. The general-purpose 'aalt' GSUB feature is accessed directly through the glyphs that are displayed in the Glyph Panel, indicated by a small black triangle at the bottom-right corner of glyphs for which variant forms are available. Other GSUB features are accessible through the Glyph Panel's flyout menu.

Adobe InDesign supports a large number of OpenType GSUB and GPOS features that are applicable to CJKV text and are included in CJKV fonts. Each subsequent version of Adobe InDesign generally adds support for additional GSUB or GPOS features, though the set of such features supported by Adobe InDesign CS3 is quite rich. Table 7-58 lists the OpenType GSUB and GPOS features—at least the ones applicable to CJKV fonts—that are made accessible by Adobe InDesign CS3, which is the current version while writing this book.

* *http://www.adobe.com/products/indesign/*

† Much of Adobe InDesign's Japanese functionalities and capabilities are due to the efforts of a development team led by Nat McCully, a senior engineer at Adobe Systems, whom I have known for nearly 20 years. We knew each other while at school, mainly due to my original *JAPAN.INF* document, which spawned *Understanding Japanese Information Processing* (O'Reilly Media, 1993). Quite coincidentally, we both moved from the Midwest, where we were studying, to California at about the same time, in mid-1991. I started to work at Adobe Systems, and Nat started at Claris, a division of Apple. Nat subsequently joined Adobe Systems over 10 years ago, in 1998, specifically to work on InDesign from its very early planning stages, and because it represented a unique opportunity to revolutionize DTP in Japan. Nat has been a trailblazer ever since then.

Table 7-58. OpenType GSUB and GPOS features supported in Adobe InDesign CS3

	Feature tag[a]	OpenType flyout	Glyph Panel	Glyph Panel flyout	Elsewhere
GSUB	aalt		Yes		
	trad			Yes	
	expt			Yes	
	jp78			Yes	
	jp83			Yes	
	jp90			Yes	
	jp04			Yes	
	nlck			Yes	
	hwid			Yes	
	twid			Yes	
	qwid			Yes	
	pwid			Yes	
	fwid			Yes	
	dlig	Yes			
	frac	Yes			
	hkna/vkna	Yes			
	ital	Yes			
	zero	Yes			
	liga				"Character" flyout
	locl				Language settings[b]
	vert				In vertical text
	ruby				Ruby feature
GPOS	palt/vpal	Yes			
	kern/vkrn				"Character" panel

a. Some GSUB features have horizontal and vertical versions, shown in this column together, separated by a slash. The writing direction of the text determines which one is used for the affected text.

b. The language settings as specified by character and paragraph tags are what control the use of the 'locl' GSUB feature.

I should point out that the Glyph Panel has the ability to enumerate all OpenType GSUB features that are included in the selected font, in the form of a filter for what the Glyph Panel displays. In other words, when an OpenType GSUB feature is selected, only the glyphs that result from applying the selected GSUB feature are displayed.

As Table 7-58 indicates in one of its table notes, Adobe InDesign allows users to specify a language as part of character and paragraph tag definitions, which can be directly

linked to behavior in OpenType features. Setting the language in InDesign actually sets the OpenType language and script. For example, setting a character or paragraph tag to Japanese sets the OpenType language and script tags to 'JAN' and 'kana', respectively. In addition to obviously benefitting GSUB features, such as 'locl' (*Localized Forms*), Adobe InDesign language settings can benefit virtually any OpenType feature, including GPOS features such as 'kern' (*Kerning*) and 'vkrn' (*Vertical Kerning*).

One of the strengths of Adobe InDesign is its ability to use any glyph in any font, through the use of its Glyph Panel. Although this works great for the occasional difficult-to-enter glyph, when one needs to enter dozens, hundreds, or even thousands of specific glyphs, the Glyph Panel suddenly becomes cumbersome. Luckily, Adobe InDesign supports a rich Tagged Text format that allows users to enter specific glyphs through the use of tagged text and by specifying CIDs. The following is an example of a complete tagged text file that serves to enter into Adobe InDesign the glyph specified by CID+14106:

```
<SJIS-MAC>
<ParaStyle:><cSpecialGlyph:14106><0001a><cSpecialGlyph:>
```

The first tag, `<SJIS-MAC>`, specifies the encoding format and the platform. Supported encoding formats include *ASCII*, *ANSI*, *UNICODE*, *SJIS*, *GB18030*, *BIG5*, and *KSC5601*. Supported platforms include *MAC* and *WIN*.

The somewhat undocumented `<cSpecialGlyph:>` tag serves to specify glyphs and is the focus of this discussion. Note the use of `<001A>` between the `<cSpecialGlyph:>` tags, which serves to specify a Unicode code point (U+001A). In order to enter arbitrary glyphs using this tag, this Unicode code point must be present, and it must be set to either U+001A (`<001A>`) or U+FFFD (`<FFFD>`). If it is set to any other value, the glyph associated with the Unicode code point will be used in lieu of the specified glyph, meaning that the glyph that is specified by the `<cSpecialGlyph:>` tag will be ignored. The former Unicode value (U+001A) is intended to be used for non-Latin glyphs, and the latter value (U+FFFD) is for Latin or Latin-like glyphs. I should also point out that while the `<cSpecialGlyph:>` tag allows arbitrary glyphs to be entered into Adobe InDesign, its use should be balanced with its limitation that the glyphs are associated with the Unicode value that is specified, meaning U+001A or U+FFFD. This peculiar behavior propagates to PDF files that are generated. This effectively means that such glyphs are not searchable in InDesign and in the resulting PDFs. For more information about Adobe InDesign Tagged Text, please consult the Adobe InDesign documentation.

Another incredibly useful feature of Adobe InDesign is the ability to use a notation for specifying Unicode values when performing search and search/replace operations. UTF-16BE values must be used; the notation requires UTF-16BE code units enclosed in angled brackets and can be used in search and replace strings. For example, to search for the ideograph 吉 (U+20BB7) or to use it as replacement text, <D842><DFB7> must be used. This is the UTF-16BE encoding form.

As a side note, this book was typeset using Adobe InDesign CS3, specifically the Japanese version running on Mac OS X.

Adobe FrameMaker

Adobe FrameMaker is an extremely powerful page-layout system that is popular in technical and book-making circles. It has been developed for Mac OS, Unix, and Windows (95, 98, NT, XP, and Vista).* Unfortunately, there is no Mac OS X version of Adobe FrameMaker.†

Adobe FrameMaker was designed to handle complex page layout for highly structured documents, such as books and technical reports. But, many have found that it can also serve well as a word processor. This is a feature not shared by other popular page-layout applications, such as Adobe PageMaker. For many page-layout systems, one usually composes the text in a word processor, and then imports that text for further page layout refinements.

Two of the most outstanding features of Adobe FrameMaker, in my opinion, are its ability to create and edit tables and to gracefully handle footnotes, including table notes. Some chapters in this book have more tables than number of pages, so table support was a critical aspect in the production. Footnote support was also important.

For those who develop documents based on *Standard Generalized Markup Language* (SGML), Adobe FrameMaker+SGML provides SGML facilities.

Adobe FrameMaker version 5.1 and greater for Mac OS is WorldScript-II–compatible, which effectively means that it is CJKV-capable. The first edition of this book was created, in terms of both text entry and page layout, using Adobe FrameMaker version 5.5, which provided additional CJKV enhancements, such as ruby support, alternate metrics, and nominal JIS X 4051 compliance.

Adobe PageMaker

Originally created by Aldus, Adobe Systems continued to develop a very popular page-layout system called Adobe PageMaker, which has been enhanced to handle CJKV text as fully localized Chinese, Japanese, and Korean versions.‡ Although no longer being developed, it is still available for both Mac OS X and Windows. One of the main enhancements to the localized versions of this application, besides the obvious handling of Unicode-encoded characters, is the ability to set CJKV text vertically. An example of one of its Japanese-specific features is support for ruby glyphs.

Adobe PageMaker, in my opinion, is ideally suited for creating complex documents that have potentially different layouts from page to page. The latest version of Adobe PageMaker provides text-manipulation tools, such as a search/replace function and spell-checker—these functions are made accessible in the context of its *story editor*.

* Adobe FrameMaker was formerly known as simply *FrameMaker*—Frame Technologies merged with Adobe Systems in late 1995. See *http://www.adobe.com/products/framemaker/*.

† *http://www.fm4osx.org/*

‡ Adobe PageMaker was formerly known as *Aldus PageMaker*—Aldus merged with Adobe Systems in 1994. See *http://www.adobe.com/products/pagemaker/*.

As an historical side note, Adobe PageMaker versions 4.0J and 4.5J were used to typeset both printings of *Understanding Japanese Information Processing* (O'Reilly Media, 1993).

Founder FIT

FIT (Focus on Integrated Typesetting or 飞腾 *fēiténg*, meaning "to soar") is a page-layout system developed by Peking University Founder Group (北大方正集团公司 *běidà fāngzhèng jítuán gōngsī*), available for Mac OS X and Windows.* It is worth noting that it is not based on a page-layout system originally developed for English-speaking users.

Although the original version of FIT was designed for use in China (Simplified Chinese), it has since been made available for use in Taiwan (Traditional Chinese) and Japan (called Founder FIT),† and there are plans for a Korean version.

Some of the more noteworthy features of FIT include the following:

- A ruby feature that supports zhuyin and Pinyin—the positioning options are quite rich (top, bottom, left, or right of parent characters)
- Various options for writing direction
- Various options for punctuation marks: full-width, proportional, and even centered
- Observation of line-breaking rules

Also of interest is that it provides an equation editor and table tool. Like Canon EDICOLOR (described later in this chapter), it allows the use of "character" as a unit of measurement. Of course, it also supports the Q typographic unit. Some interesting details about FIT are available online.‡

QuarkXPress

QuarkXPress, developed by Quark Incorporated, is another very popular and sophisticated page-layout system available for Mac OS X and Windows.§ The latest localized versions have the option to use English menus and dialogs.

QuarkXPress allows the user to use a wide variety of typographic units, such as the Q. It also allows users to redefine typographic measurements such as the point (the PostScript imaging model defines there to be 72 points per inch, but that can be easily changed in QuarkXPress). Adobe InDesign provides comparable functionality. Other significant features of CJKV versions of QuarkXPress include:

- Ruby support.
- CJKV support in kerning table editor, in horizontal and vertical writing modes.

* *http://www.founderpku.com/*

† *http://www.founder.co.jp/*

‡ *http://www.jagat.or.jp/asia/report/China3.htm*

§ *http://www.quark.com/*

- Modification of the line-breaking character set—there are predefined weak and strong sets, and the user can change their definitions or create new ones.
- Definition of font sets that allow users to select a different font for different types of scripts, such as ideographs, kana, Latin characters, numerals, and symbols—the font specified for each script can be baseline-shifted or scaled at the time of the definition.

Although not CJKV-specific, the QuarkXPress "Text to Box" feature allows the user to convert text into an editable text box in the shape of the glyphs themselves—similar to the text-to-outline feature available in Adobe Illustrator and FreeHand MX (both of which are described shortly). This feature works with both PostScript and TrueType fonts, and ATM (*Adobe Type Manager*) is required. Of course, this feature does not work for fonts whose outlines are not available or are protected.

QuarkXPress provides extended functionality through third-party plug-ins called XTensions. Quark maintains a list of XTension developers.* Note that some XTensions may not work properly with CJKV versions of QuarkXPress. Quark's Japanese office also maintains a list of XTensions that can be used with QuarkXPress-J (and perhaps other CJKV versions as well).†

Demo versions of QuarkXPress, including CJKV-capable versions, are available from Quark.‡ This provides the potential user the ability to test drive the application before making a purchase.

Canon EDICOLOR

Sumitomo Metal Industries' (住友金属工業株式会社 *sumitomo kinzoku kōgyō kabushiki gaisha*) Publishing Systems Division (パブリッシングシステムズ事業室 *paburisshingu shisutemu jigyōshitsu*) originally developed a sophisticated page-layout application called EDICOLOR (エディカラー *edikarā*), available for Mac OS and Windows. Canon now owns EDICOLOR, and it is available for Mac OS X and Windows.§ EDICOLOR stands for *EDItors' COLOR Layout Software*. Fully functional trial versions of Canon EDICOLOR are available at Canon's website.

Canon EDICOLOR is unique from other common line- and page-layout systems, such as Adobe InDesign and QuarkXPress, in two important ways:

- It is not thought of as a localized version of a line- and page-layout system that was originally intended for Western languages—as is the case for Adobe FrameMaker, Adobe PageMaker, and QuarkXPress.¶

* *http://www.quark.com/products/xpress/xtensions/*

† *http://japan.quark.com/products/xtensions/*

‡ *http://www.quark.com/service/desktop/downloads/* or *http://japan.quark.com/service/desktop/download/*

§ *http://ps.canon-sol.jp/ec/*

¶ Perhaps of historical interest, EDICOLOR's line-layout engine was originally developed by Stonehand in the U.S., and subsequently licensed and adapted to support Japanese typographic features.

- Given that Canon EDICOLOR is targeted toward the Japanese market, it is unlikely that it will ever support languages other than Japanese, such as Chinese and Korean, although it is within the realm of extreme possibilities.

Canon EDICOLOR provides the user with a wide variety of typographic controls, including many Japanese-specific features—such as ruby support, a character-based layout grid, and the use of the Q typographic unit—in order to create the desired page design. Earlier versions also included a special-purpose font, called SMI外字 (*SMI gaiji*), that included many symbol glyphs that are found in typical Gaiji font products, such as those developed by Biblos and Enfour Media. A well-integrated table editor is also part of Canon EDICOLOR.

Canon EDICOLOR's かな詰め (*kana tsume*, which is best translated as "proportional kana") feature makes use of glyph-level bounding box information of AFM (*Adobe Font Metrics*) files to create proportional metrics for kana and a handful of additional characters. AFM files for a large number of Japanese fonts are included with Canon EDICOLOR, which allows users to compose documents using Japanese fonts that are not actually installed onto their system. Table 7-59 illustrates four types of proportional metrics for a short string of katakana glyphs that are set in Adobe Systems' 小塚明朝 Pr6N H (KozMinPr6N-Heavy) typeface design: the hand-tuned proportional metrics developed by Adobe Systems that are included in the font itself, along with three levels or degrees of proportional metrics made available in Canon EDICOLOR, each mathematically calculated from its AFM file.

Canon EDICOLOR also allows the user to define composite fonts—that is, the ability to specify what font to use for what character class, as well as the ability to specify different X- and Y-axis scales, along with baseline adjustment. In addition to being able to specify separate fonts for standard character classes—kanji, kana, Latin, and numerals—Canon EDICOLOR also allows the user to specify fonts for what it refers to as system- and user-defined characters.* By default, the system-defined characters can be set to one of the following bundled OpenType fonts: Edisys-OTF-Gaiji or Edisys-OTF-KAZAR. Note that the original SMI外字 font was renamed to "SMI外字 plus" in version 6.0, presumably with additional glyphs. It is now equivalent to the Edisys-OTF-Gaiji font. Also, the default user-defined characters can be set to one of the following bundled OpenType fonts: Edigai-OTF-GoIwata or Edigai-OTF-MinIwata.

* It is possible to create fonts so that they are recognized by Canon EDICOLOR as system- or user-gaiji. If a font's menu name begins with ＳＭＩ外字, システム外字, or Edisys, then it is treated as a system-gaiji font. If a font's menu name begins with ユーザー外字, ユーザー創作, or Edigai, then it is treated as a user-gaiji font.

Table 7-59. Handtuned and AFM-derived proportional metrics

Metrics	Example
Full-width	トラッキング＆カーニング
Adobe Systems	トラッキング＆カーニング
EDICOLOR Loose	トラッキング＆カーニング
EDICOLOR Standard	トラッキング＆カーニング
EDICOLOR Tight	トラッキング＆カーニング

While QuarkXPress allows the user to freely define or redefine the line-breaking character sets, Canon EDICOLOR additionally permits the user to modify what punctuation marks are allowed to dangle, as does Adobe InDesign.

Like Adobe InDesign, Canon EDICOLOR includes a Character Palette that allows the user to select a glyph by Unicode encoding, Shift-JIS encoding, and GID. When accessing glyphs by GID, it means any glyph in the font, regardless of whether it is encoded, can be entered into documents.

Graphics Applications

Another class of application that often includes advanced typographic capabilities is more graphics-oriented, rather than line- or page-layout–oriented. These programs are known as graphics applications. While most word processors and page-layout applications include a very basic set of built-in graphics tools, they are not nearly as powerful as dedicated graphics applications. Nearly all figures in this book, along with the contents of some of its tables, required intervention by one or more graphics applications. This is typical for most books.

Adobe Dimensions

Adobe Dimensions, although no longer available, provided users with the ability to create three-dimensional objects that can be completely constructed using the application itself,

or it can be exported to other applications, such as Adobe Illustrator or Adobe Photoshop, for further editing or manipulation.*

Although it is possible to emulate three-dimensional objects using standard illustration tools, one cannot rotate or otherwise change the orientation of such objects and still expect their perspective to appear correctly. This is what makes Adobe Dimensions useful. Version 3.0 or later (including the standard or unlocalized version) provides the ability to manipulate CJKV text using three dimensions. Figure 7-2 illustrates Korean text—specifically, the characters 김치 (*gimci*, meaning "kimchi") set using hangul from Adobe Systems' Adobe 명조 Std M (AdobeMyungjoStd-Medium) typeface design—extruded to show three dimensions. Needless to say, Adobe Dimensions was able to provide documents with some very interesting effects.

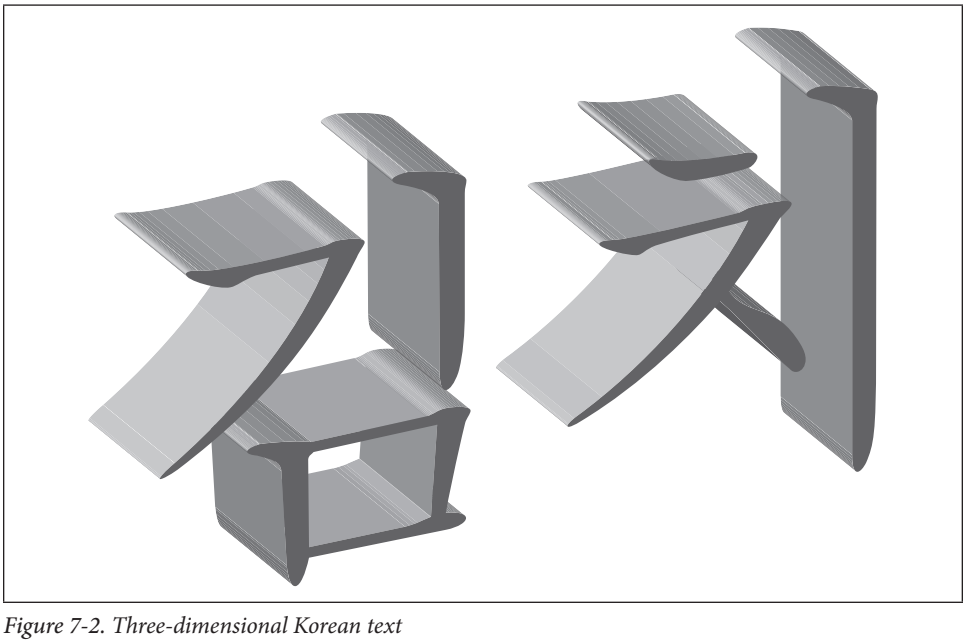


Figure 7-2. Three-dimensional Korean text

Adobe Illustrator

One of the most well-known illustration programs is Adobe Illustrator.† It allows extremely precise layout of text and graphics objects within the context of a single page. Although Adobe Illustrator does not constitute a full-featured page-layout or composition system, it is very useful for short works, such as one-page advertisements, fliers, and so on. It is also useful for creating and editing illustrations that are used in books and other documents,

* *http://www.adobe.com/products/dimensions/*

† *http://www.adobe.com/products/illustrator/*

meaning that the results are used in a genuine page-layout or composition system, such as Adobe InDesign.

Adobe Illustrator provides the user with the ability to set text in both horizontal and vertical mode—for both CJKV and Latin text. All of the variations of setting Latin text vertically, as illustrated in Table 7-12, are fully supported features of Adobe Illustrator. Also of significance is that as of version 7.0 (which is now over 10 years old), the standard—the unlocalized or U.S.—version of Adobe Illustrator is CJKV-capable. And, believe it or not, you are not even required to have an underlying CJKV-capable OS to bring out this functionality![*]

Using Adobe Illustrator is very much like having the entire PostScript programming language at your disposal through a convenient graphical interface, but its capabilities extend far beyond what PostScript offers. Almost every attribute of every text or graphic object you create can be modified to your heart's content. You can even set text along a curve, and convert text—set using either PostScript or TrueType fonts—into editable outlines.[†]

Like other Adobe applications, Adobe Illustrator provides the user with the ability to adhere to the Japanese line- and page-layout rules and principles that have been set forth in the JIS X 4051:2004 standard. Also of importance is the ability to use a font's built-in alternate metrics and certain classes of glyph substitution. The latest version of Adobe Illustrator has excellent support for OpenType glyph substitution features, and it also includes a Glyph Panel comparable to what Adobe InDesign provides.

Nearly all of the typographic examples provided in the first edition of this chapter were produced with Adobe Illustrator using the fonts' built-in typographic features. This was necessary due to limitations of Adobe FrameMaker, such as its inability to support vertical writing. For the second edition, nearly all of the examples were produced directly in Adobe InDesign.

Adobe Photoshop

The most widely used bitmap-graphics–manipulation application is clearly Adobe Photoshop.[‡] Every aspect of a scanned or created image can be modified using this. But, you may be wondering why it is being mentioned in this book.

Adobe Photoshop does not currently provide the user with advanced typographic functionality, but does offer the ability to include CJKV text in images. This can become

* Before you get overly excited, let me warn you that one critical piece of software provided by CJKV-capable OSes is one or more input methods. Without an input method, one cannot easily enter CJKV text, but you can obviously display and print it. Also, you need to identify and install CJKV fonts, many of which are bundled as part of CJKV-capable OSes. Given the mature state of today's OSes, in terms of multilingual support, this is no longer a concern.

† You cannot, however, convert text into editable outlines if the selected font has its outlines protected. Morisawa's Japanese fonts, for example, at one time imposed this restriction, but this has thankfully changed since the first edition of this book was published.

‡ It is also the most widely pirated program. See *http://www.adobe.com/products/photoshop/photoshop/*.

especially useful if the client of the images does not have a CJKV-capable OS, or if the fonts are protected in such a way that prevents access to outlines. A bitmap image of a glyph, rendered at the final output device's resolution, can be easily printed from any computer running Adobe Photoshop. Adobe Photoshop version 5.0 and later provides much better text services than earlier versions, and from version 6.0, includes a very sophisticated text engine, which is the same one used by Adobe Illustrator.

Adobe Photoshop has eased the way in which text is handled by automatically maintaining two layers. One layer is the editable text, which is not rendered into bitmaps. The other layer contains the rendered bitmaps. This feature makes the editing of text in Photoshop documents a much more pleasant experience. Before this functionality was available, users either had to completely re-enter the text or manually maintain these separate layers.

Adobe Photoshop is actually a family of applications. What has been described in this section is the flagship application, the most current version of which is Adobe Photoshop CS3. There are also Extended, Elements, and Lightroom variations of Photoshop available, all of which are described on the Adobe website. All have fully functional 30-day trial versions available.

FreeHand MX

FreeHand MX, now owned by Adobe Systems, includes features and functionality comparable to those provided by Adobe Illustrator, and like Adobe Illustrator is available for Mac OS X and Windows.* In fact, these applications were competitors for many years, and both have loyal followings still to this day.

Additional details and information about FreeHand MX is available, including fully functional 30-day trial versions.† Given the significant functionality overlap of FreeHand MX and Adobe Illustrator, Adobe Systems has no plans to further develop FreeHand MX. FreeHand MX users are thus strongly encouraged to migrate to Adobe Illustrator.

Advice to Developers

For application developers, specifically for those who develop Japanese applications, I strongly recommend that you adhere to the rules and principles set forth in the JIS X 4051:2004 standard. If necessary, explore existing applications that already provide this functionality, which will serve as an example to guide your own efforts. Adobe InDesign is an excellent example of an application that provides a high degree of JIS X 4051:2004 support.

* FreeHand MX was originally known as Aldus FreeHand—Altsys Corporation originally developed FreeHand, but Aldus marketed it. When Aldus merged with Adobe Systems in 1994, FreeHand was relinquished to Altsys, but soon after Macromedia bought Altsys. Macromedia merged with Adobe Systems in 2005, and FreeHand became property of Adobe Systems.

† *http://www.adobe.com/products/freehand/*

Regardless of what language is being manipulated in terms of line and page layout, it is prudent to support vertical writing, proper line breaking, and proper glyph spacing. If the rules and principles of JIS X 4051:2004 are not applicable, do your best to formulate your own rules and principles, and adhere to them as much as possible.

Bear in mind the following guiding principle: *if the result doesn't look right, odds are it isn't.* In other words, if something doesn't appear to be typeset well, chances are it was not, either due to improper settings or inadequate support on the part of the application. For improper settings, the solution is to invoke the proper settings. For situations that arise due to an application limitation or shortcoming, explore ways to work around the limitation. For nearly all seemingly impossible situations, there is almost always a solution. Sometimes it presents itself easily, and sometimes more effort is required for it to be discovered.

CHAPTER 8
Output Methods

If you consider the extent to which character sets, encoding methods, font formats, and typography have been discussed thus far, it is clearly an appropriate time to discuss how CJKV text can be output or displayed through the use of devices such as printers and displays. You learned about font formats and related matters in Chapter 6, then about typography and composition in Chapter 7, but now comes the time when you must finally display, print, or otherwise output the CJKV text that you so carefully composed into documents or smaller text fragments. Printers can range from low-resolution dot-matrix printers to high-resolution photo imagesetters costing tens of thousands of dollars.* The vast majority of advances seem to be in the area of displays. The resolution has become quite high, to include HDTV-like quality, even for mobile devices such as cell phones.

Regardless of what output device is being used, whether it is a computer monitor, a mobile device display, a dot-matrix printer, or even a high-resolution photo imagesetter, the most basic unit of output is either a pixel (in the case of computer monitors or other displays) or a dot (in the case of printers). The resolution of both types of output devices is usually expressed in units called DPI, short for *dots-per-inch*. The most commonly used printers today are 600-dpi or greater laser printers and color inkjet printers, although inexpensive 1200-dpi and greater plain-paper printers have already emerged. For many years, the most commonly-used computer displays had 72-dpi or greater resolution, specifically 72-dpi for Mac OS, and 96- or 120-dpi for Windows. The resolution of displays are increasing, and advances in rendering technology are leading to improvements without the need to increase display resolution.

Given the wide variety of output devices that are available today, exactly what is considered high- or low-resolution is relative, and clearly changes over time. To people who use a 1270-dpi photo imagesetter, 600-dpi and below is considered low-resolution, but those who recently upgraded from a dot-matrix printer to a 600-dpi laser printer may think

* Or yuan, yen, or won as the case may be—actual amounts may depend on currency fluctuations and exchange rates.

that they are now the proud owner of a high-resolution printer. As you can see, it is all relative….

Where Can Fonts Live?

Before we discuss how to print or display CJKV text, let's discuss for a moment where fonts live in relation to the host—meaning your computer—and the printer. Bear in mind that the printing workflow has changed since the first edition of this book was published, and much more of the font burden is now on the host rather than on the printer. This is a very good thing, which will be explained in a moment.

Years ago, PostScript fonts felt at home only when resident on a printer, either in its VM (*Virtual Memory*, also known as RAM), in ROM, or in a built-in or attached hard disk. With the introduction of ATM, PostScript fonts also felt at home on the host. It was only with the introduction of OpenType that PostScript fonts were able to share many of the same advantages of TrueType fonts in a host environment, such as having only a single file per font.

TrueType fonts, although they can be downloaded to and rendered on most contemporary PostScript devices, feel most at home on a host. This is because their format was specifically designed with the host environment in mind.

Non-CJKV PostScript fonts, usually in Type 1 format, could be explicitly downloaded to a PostScript device on a permanent basis. This is still possible today, and is known as a *static download*. These same fonts can also be automatically downloaded when the PostScript printer driver senses that the PostScript device does not have one or more fonts available. This is known as a *dynamic download*. It is also possible to download only those glyphs that are referenced in the document to be printed. This is known as *subsetting*, or as *partial* or *incremental download*. In the absence of incremental downloading capability, it is also possible to send bitmaps of the appropriate resolution to PostScript devices. In fact, sending bitmaps is the only course of action for some classes of printers, such as most ink-jet devices.

Typical CJKV fonts, because they are several megabytes in size, do not lend themselves to dynamic downloading because of the download time and the amount of printer memory required. Instead, they have been installed through a static download. However, contemporary PostScript printer drivers have made it possible to incrementally download CJKV fonts. A typical Japanese document is composed of 70% kana, so the number of kanji will be limited for most documents, leading to a subsetted font that is considerably smaller than the original when incrementally downloaded.*

Now that computers and the communication channel between host and printer are considerably faster than only a few years ago, the advantage of having fonts resident on the

* Of course, if you are printing complete character set tables, most if not all of a CJKV font will be incrementally downloaded.

PostScript device is decreasing. In addition, even with the ability to incrementally download fonts, some CJKV fonts may be restricted in such a way that prevents incremental downloading. Static downloading may still be required.

As alluded to early on in this section, the printing workflow has changed, and the font burden is now on the host rather than the printer. This new printing workflow uses Adobe Acrobat and *Portable Document Format* (PDF) as a reliable way to create a complete digital image of a document, meaning that all of the glyphs and other page elements are embedded into the PDF file. When printed, the embedded glyphs become part of the data stream that is sent to the printer. Because the printer no longer has the font burden, the chance of mismatching host- and printer-resident fonts is minimized, and to some extent, eliminated.

Output via Printing

Many years ago, one could print CJKV text using only bitmapped fonts. In fact, some printers required that the bitmapped characters be resident in the printer hardware itself, meaning in its ROM. As discussed in Chapter 6, bitmapped fonts are not ideal—and quite frankly, an extremely poor choice—for printing high-quality CJKV text. Fortunately, there are now many ways you can improve the quality of CJKV text when output to a wide variety of devices.

Printing devices range from low-resolution dot-matrix printers to high-resolution photo imagesetters. Some of these devices, usually printers and photo imagesetters, usually support PostScript, which means that they have a built-in PostScript interpreter—we discuss why this is a "good thing" later in this section.

At the beginning of this chapter, I stated that any printing or displaying eventually results in the rasterization or rendering of a font into dots or pixels. Every font, no matter what its format, is ultimately resolved into a bitmap. For outline fonts, printing speed and performance are heavily dependent on where the outlines are scaled and subsequently rendered into bitmapped patterns:

- The glyphs, represented as outlines, can be rendered into bitmaps on the computer, and then sent to the printer or display.
- The instructions for rendering the outlines into bitmaps can be sent to the printer, which subsequently renders the outlines into bitmaps of the appropriate size and resolution.

The latter is usually faster, but both result in the same printed page as long as the software performing the rendering is the same or at least similar. Hardware and software for both methods are described throughout this chapter.

The way in which printing is performed has changed over the years, in the form of improvements. Improvements in technology, such as the speed at which data can be sent to the printer, has effectively diminished the need to have the fonts resident on the printer.

In addition, Adobe Acrobat and PDF have improved the document workflow, by placing much more of the burden on the authoring computer, where it belongs, rather than on the printer. This ensures that the result, whether viewed on another computer or output to a printer, is the same.

PostScript CJKV Printers

One of the very first solutions available for obtaining high-quality CJKV output was to acquire a PostScript Japanese printer. One of the most common models was the Apple LaserWriter II NTX-J, a PostScript Level 1 printer with built-in composite font support. The Apple LaserWriter II NTX-J came with a 40 MB hard disk containing two PostScript Japanese fonts, Morisawa's Ryumin-Light and GothicBBB-Medium. There are now many PostScript Japanese printers that come bundled with up to 26 PostScript Japanese fonts. These fonts include the two just listed (sometimes baked into ROM rather than stored on an internal or external hard disk—accessing ROM-based fonts is much faster) plus additional fonts from Morisawa, the most common ones being FutoMinA101-Bold, FutoGoB101-Bold, and Jun101-Light.

Companies such as Apple, Canon, Dainippon Printing, Digital, Electronics for Imaging (EFI), Epson, Fuji-Xerox, Hewlett-Packard, Linotype-Hell, Oki Electric, Varityper, Xerox, and others have manufactured PostScript CJKV printers. The products developed by these companies ranged from laser printers to high-resolution photo imagesetters. Other companies, such as LaserMaster Corporation and Harlequin, manufactured PostScript-compatible CJKV printers, also called PostScript clones.

The definitive guide to PostScript is Adobe Systems' *PostScript Language Reference*, Third Edition (Addison-Wesley, 1999).*

Genuine PostScript

Adobe Systems released PostScript Level 2 to the marketplace during the latter part of 1991. PostScript Level 2 has built-in composite font support, meaning support for Type 0 fonts. Specifically, PostScript version 2011 and greater includes the complete composite font extensions—a handful of PostScript version 2010 printers exist, such as Apple's LaserWriter IIf and IIg, which do not have a complete implementation of the composite font extensions.† Contrary to popular belief (or myth), having the composite font extensions does not automatically give you the ability to use Japanese or other CJKV fonts. This is because special system files must be resident in ROM or on the printer hard disk. Having these composite font extensions does, however, make it easier for printer

* *http://www.adobe.com/devnet/postscript/*

† The following two-line PostScript program, when sent to any PostScript device, will echo back the specific version number of the PostScript interpreter that is resident on the device:

```
%!
version ==
```

manufacturers to produce PostScript CJKV printers by licensing the necessary system files from Adobe Systems. It also makes it possible for users to CJKV-enable such PostScript devices by downloading a contemporary CJKV font using Adobe Systems' installer.*

Versions of PostScript prior to PostScript Level 2, now called PostScript Level 1, did not have composite font support, exceptions being PostScript Japanese Level 1 printers, such as Apple's LaserWriter II NTX-J just mentioned.

No matter which version of PostScript you have in your printer, if the font with which you are attempting to print is resident on the printer ("resident" here refers to being baked into ROM or on the printer's hard disk, whether internal or external), the font is rendered on the printer. To give you an example of the size of a PostScript file compared to sending bitmapped data to the printer from the computer, see the following code, which represents a complete PostScript file for printing my Japanese penname (小林劍) set vertically at 200-point:

```
%!
/Ryumin-Light-V findfont
200 scalefont
setfont
306 720 moveto
<3E2E 4E53 3775> show
showpage
```

This code example provides the necessary rendering instructions to the PostScript interpreter resident on the printer. The printer then renders the characters per the instructions. Compare that with a file that contains bitmapped data for three 200-point characters, which may be a file that is more than 100 times as large. It should be rather obvious which method is faster for sending from the computer (host) to the printer.

The latest version of PostScript is PostScript 3.†

Clone PostScript

Not everyone, especially individual users, has enough money to purchase a CJKV printer equipped with genuine PostScript from Adobe Systems, so this opened the door for the development of PostScript clones.

One of the most prominent—and easily obtainable—clone PostScript implementations is called *Ghostscript*, developed by Aladdin Enterprises.‡ Some of the more interesting

* That is, one very useful side effect of downloading a CID-keyed font from Windows or Mac OS to a PostScript device that is not yet CJKV-enabled (and has all the other requirements, such as PostScript version 2011 or greater, enough RAM, and a hard disk) was the installation of the necessary system files that make the PostScript device CJKV-enabled.

† Note that I did not write "PostScript Level 3" here—Adobe has explicitly stated that the official designation for this post-PostScript Level 2 is "PostScript 3."

‡ *http://www.cs.wisc.edu/~ghost/*

features (interesting, at least in the context of this book) include support for PDF, CFF, and CID-keyed fonts.

Another PostScript clone is called *Jaws*,* originally developed by 5D Solutions Limited, but now available from Global Graphics† (only as an OEM product). The current implementation of Jaws supports CID-keyed fonts, as well as PDF. Jaws is very similar in concept to Adobe Systems' CPSI (*Configurable PostScript Interpreter*), which means that it runs on a standard computer (Mac OS, Unix, and Windows) that is attached to a printing engine, such as an imagesetter.

Keep in mind that some PostScript clone implementations may lack features and functionality that are required to support CJKV fonts, in particular CID-keyed fonts. Ghostscript and Jaws, as exceptions to this general rule, do not appear to have this problem.

Using CID-keyed fonts in Ghostscript

It is not always obvious how to use CID-keyed fonts in Ghostscript, so the following is a detailed outline that clearly demonstrates how to properly install and use them. Although the following steps are for a Korean CID-keyed font, the procedure itself is generic.

1. Obtain a CIDFont resource. For example, you can obtain the *Munhwa-Regular* CIDFont resource from the following URL:

   ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/samples/

2. Obtain the CMap resources that correspond to the character collection of the CIDFont, which in this case is Adobe-Korea1. For example, the following URL contains the Adobe-Korea1-2 CMap resources:

   ftp://ftp.oreilly.com/pub/examples/nutshell/cjkv/adobe/ak12.tar.Z

3. Create a "font-stub" for the CID-keyed font, Munhwa-Regular-KSC-EUC-H, by making a file with the following name:

   Munhwa-Regular-KSC-EUC-H.gsf

   The following represents the complete contents of the newly created *Munhwa-Regular-KSC-EUC-H.gsf* file:

```
/Munhwa-Regular-KSC-EUC-H
/Munhwa-Regular (Munhwa-Regular)
/KSC-EUC-H (KSC-EUC-H)

1 index /CMap resourcestatus
{pop pop pop}
{runlibfile} ifelse
/CMap findresource

3 1 roll
1 index /CIDFont resourcestatus
```

* Stands for Just Another Window Server.

† *http://www.globalgraphics.com/*

```
{pop pop pop}
{runlibfile} ifelse
/CIDFont findresource

[ exch ] composefont pop
```

4. Place the *Munhwa-Regular-KSC-EUC-H.gsf* file and *KSC-EUC-H* CMap resource in the font path (such as */usr/lib/ghostscript/fonts/*).
5. Add a "fontmap" entry for Munhwa-Regular to the *Fontmap* file:

```
% CID-Keyed font
% Korean(Adobe)
/Munhwa-Regular-KSC-EUC-H (Munhwa-Regular-KSC-EUC-H.gsf);
```

6. With all of these steps completed, you can simply use PostScript commands to use the CID-keyed font. What follows demonstrates how this Korean CID-keyed font can be used in Ghostscript (user-entered commands are emboldened):

```
GS> /Munhwa-Regular-KSC-EUC-H findfont 100 scalefont setfont
Loading Munhwa-Regular-KSC-EUC-H font from ./Munhwa-Regular-KSC-EUC-H.gsf... 411
7896 2702441 4104584 2791079 0 done.
GS> 100 100 moveto <B0A1 B0A2> show showpage
```

Passing Characters to PostScript

There is more than one method for passing characters to PostScript as an argument to the *show* operator. The *show* operator is quite important because it is the only method to display text using fonts. The *findfont* and *selectfont* operators are used to indicate what font to use for the characters passed to the *show* operator. When handling normal ASCII text, the standard convention is to simply pass characters as is to the *show* operator using parentheses, as illustrated by the following PostScript program:

```
%!
/Courier findfont 12 scalefont setfont
100 100 moveto
(This is ASCII text) show
showpage
```

But when you are dealing with characters whose values go beyond seven-bit ASCII encoding, some care must be taken to do things correctly. Remember that most CJKV encodings make generous use of eight-bit characters. The following PostScript program illustrates two acceptable methods for passing eight-bit characters to the *show* operator (using EUC-CN encoding for the three-character string 小林剑 *xiǎolín jiàn*):

```
%!
/STFangsong-Light-GB-EUC-H findfont 12 scalefont setfont
100 100 moveto
<D0A1 C1D6 BDA3> show
(\320\241\301\326\275\243) show
showpage
```

The first method (the fourth line of code) uses hexadecimal codes in angled brackets—spaces are ignored, and hexadecimal digits "a" through "f" can be upper- or lowercase.

The second method (the fifth line of code) uses octal codes delimited by standard parentheses. If we use the same text, but specify an encoding that does not use eight-bit characters, such as ISO-2022-CN, there is a third possibility: pass the characters as is. See the sixth line of the following code:

```
%!
/STFangsong-Light-GB-H findfont 12 scalefont setfont
100 100 moveto
<5021 4156 3D23> show
(\120\041\101\126\075\043) show
(P!AV=#) show
showpage
```

While the sixth line of the above PostScript program illustrates how seven-bit CJKV character codes can be passed to the *show* operator as is, there are some characters that need to be escaped with a single backslash character (\) when this is done: left parenthesis, right parenthesis, and the backslash character itself.

Output via Display

Being able to output CJKV text onto a computer monitor or other display device is a very basic requirement in order to successfully input or otherwise manipulate text. Without the ability to display text, imagine for a moment how you would go about processing CJKV text. You'd be virtually blind.

A computer monitor or display screen image consists of pixels, with each pixel representing the most fundamental unit of display. In the case of monochrome displays, each pixel can be either white or black. These days there are better display technologies that offer grayscale, color, and even anti-aliasing options. *Anti-aliasing* is a technique that allows complex glyphs to be displayed at small sizes and still be legible.

As we will discuss in the next section, *Adobe Type Manager* (ATM) and TrueType had been used to provide high-quality output for display devices. They were also useful for printing to environments that did not have the fonts resident, such as non-PostScript devices, or PostScript devices that simply did not have the necessary fonts installed.* There was also Display PostScript (DPS), which was much like ATM but had virtually the entire range of PostScript available. DPS provided what is known as full WYSIWYG (*What You See Is What You Get*), meaning that what was displayed on the screen was what you would get when printed. In the past, Digital, IBM, SGI, Sun, and other workstations supported DPS. Now, one would be hard-pressed to find a modern OS that uses either ATM or DPS.

The X Window System for Unix is a *Graphical User Interface* (GUI), and as such, handles the display of a variety of fonts. It is usually necessary to use a special window for CJKV output, such as a *kterm* (Japanese-specific), *exterm*, or equivalent. Some X Window

* There are some printers that include all of their CJKV fonts in ROM, and have no hard disk nor hard disk port. There is no permanent download option for such printers.

System applications, such as GNU Emacs (version 20 or greater), provide their own CJKV-capable windows.

Adobe Type Manager—ATM

In 1989, Adobe Systems developed a font-rendering program called *Adobe Type Manager* (ATM) for Mac OS, which could be thought of as the font-rendering software used in a PostScript interpreter, but instead resided on the computer (often called the *host*), which allowed users to effectively install printer fonts on the computer and use them with ATM for high-quality computer monitor output. A version for Windows was released in 1990. As mentioned previously, ATM was also used for printing to printers on which the fonts are not resident. ATM worked with non-PostScript printers, too. In many ways, it was revolutionary and was developed in response to TrueType. Many people claimed that the best thing that happened to PostScript was TrueType, because it encouraged the development of ATM, effectively putting them at the same level.

The very first versions of ATM supported only single-byte PostScript fonts, but late in 1991, Adobe Systems released ATM-J, the Japanese version of ATM. This software package was bundled with two PostScript Japanese fonts, Morisawa's Ryumin-Light and GothicBBB-Medium, which the user installed onto their computer. ATM, although not advertised, included full CJKV font support.

If the font was resident on both the computer and printer, ATM did not render the glyphs, but instead let the printer do it. In fact, it was the printer driver that must request bitmaps from ATM—ATM itself did not communicate with the printer. It could not, as it didn't include such facilities. And for display purposes, ATM was used to render glyphs at point sizes for which a bitmapped font was not available.

Figure 8-1 illustrates the ideograph 劍 printed at 216-point (scaled) in three ways: using a 12-point bitmapped font (ugh!), on a screen display with ATM turned on (72-dpi), and finally printer output with ATM turned on (300-dpi). Notice the difference ATM makes in the output quality of the character. The same figure applies to TrueType and OpenType, both of which are covered later in this chapter.

At one point, one was able to purchase over several hundred different PostScript Japanese fonts that worked with ATM.

ATM was designed to run on Mac OS Windows, and there was even a deluxe version, called ATM Deluxe, that provided font management features, such as auto-activation and the ability to group fonts into functional sets. Now, thanks to efforts by Adobe Systems, Apple, and Microsoft, the functionality that was made available by ATM is now integrated into Mac OS X and in Windows 2000, XP, and Vista. Adobe Systems licensed to Apple and Microsoft its core rasterizer, the same used by ATM.

Figure 8-1. The effect of ATM on a 216-point character

SuperATM

SuperATM was an extension to ATM, and made use of the multiple master technology described in Chapter 6. Well, to be a bit more clear, SuperATM offers font substitution functionality, something that is part of Adobe Acrobat. So, you may ask, what problem does SuperATM attempt to solve? Consider a case in which someone provides you with a document, but you do not have the typefaces which were used to create it. What usually happens is that the computer substitutes Courier, a well-known monospaced font ("`this is Courier`"), which completely interrupts the layout and line-breaks of most documents. SuperATM attempted to solve this problem through the use of multiple master font technology. Glyphs with identical metrics were synthesized on an as-needed basis, and were otherwise very close in style and appearance to those used in the original document.

SuperATM used two generic multiple master fonts to accomplish this feat, both of which used unusually creative names. One is a serif font, called Adobe Serif MM, and the other is sans serif, called Adobe Sans MM.

So how are SuperATM and multiple master different? When you select a different weight, width, optical size, or even style for a multiple master font, the interpolation ratios are applied equally to every glyph in the typeface. In the case of SuperATM, however, the matching of metrics is done on a per-glyph, not per-font, level. This means that every glyph in a substituted font may have a unique interpolation ratio. SuperATM, and Adobe Acrobat, discussed next, used a database of font metrics information to accomplish this.

Although SuperATM is dead as a separate product, its functionality was folded into ATM Deluxe. In addition, Adobe Acrobat still makes use of SuperATM functionality, and to accomplish this, still uses the same Adobe Serif MM and Adobe Sans MM fonts for font substitution purposes.

Adobe Acrobat and PDF

Adobe Acrobat and *Portable Document Format* (PDF) have clearly revolutionized the publishing industry by significantly enhancing the document production workflow. A PDF file represents a reliable and complete digital master for a document, whether it is a single-page letter or a 900-page tome.* Everything that is needed to display or print the document can and should be embedded.†

The primary goal and function of Adobe Acrobat and PDF is to achieve true document portability, which explains *Portable Document Format* as a suitable name. Its code name was Carousel while it was under development. This is more or less the same as information interchange, but carried one step further, specifically to include highly stylized text and graphics. This further step functions to preserve the "look and feel" of a document across platforms. This includes the typefaces, graphics, and even color. No longer will you need the application that created the original document, and no longer will Courier be used to replaced missing fonts. You may have guessed that Adobe Acrobat uses Super-ATM technology's font metrics database to accomplish part of its goals. Although Super-ATM itself is dead, as an independent product, its functionality is alive and well even in the latest version of Acrobat (version 9.0 as of this writing).

Adobe Acrobat, specifically its Distiller application, works by first interpreting a PostScript file, which can come from a variety of sources. Acrobat Distiller, which is the work-horse application that converts PostScript files to PDF, then outputs a new file that conforms to PDF specifications. Many applications can create PDF files without the need to use Acrobat Distiller. Adobe applications that provide this functionality do so through the use of a library called *PDFL*.

Adobe Reader, which is freely available for OSes such as Mac OS X, Windows, and even Linux, is used to display and print PDF files.‡ Adobe Acrobat and Adobe Reader are not the only applications that can view and print PDF files. On Mac OS X, for example, the Preview application provides this ability. Single-page PDF files also display inline in the Mac OS X Mail application, when an email with a PDF attachment is open.

More information on Adobe Acrobat and PDF is provided in Chapter 12, where both are described in the context of the Web and printing publishing. The complete PDF specification is available in the form of a document entitled *PDF Reference*, which, oddly enough, is available as a PDF file that can be downloaded from Adobe Systems' website.§

* Hint, hint….

† A note of family trivia here. My beloved wife worked on the Acrobat development team from version 3.0 through version 8.0.

‡ *http://www.adobe.com/products/acrobat/*

§ *http://www.adobe.com/devnet/pdf/pdf_reference.html*

Ghostscript

Ghostscript, which was touched upon earlier in this chapter, is thought by many to simply be clone PostScript. It is far more than that. Those who use Mac OS X or Windows as their OS take it for granted that support for many things are built-in at the OS level, such as support for OpenType fonts and the ability to run a wide variety of applications that author, view, and print PDF files. Some OSes, such as Linux, do not have such capabilities built in.

Ghostscript is an interpreter for the PostScript page-description language, with the ability to view and print PostScript files. Its latest versions include facilities for creating PDF files from PostScript files, much like Acrobat Distiller, and to subsequently view and print them. Ghostscript also supports a wide variety of font formats.

If you use Linux or are considering Linux as your OS, I strongly encourage you to explore what features and functionality Ghostscript offers.*

OpenType and TrueType

Thanks to cooperation and collaboration by Adobe Systems, Apple, and Microsoft, the latest OSes, specifically Mac OS X and Windows, provide equal footing for OpenType and TrueType fonts. It is now possible to build a single font file that serves both platforms, and potentially other platforms, such as Linux and mobile devices.

In Chapter 6 we discussed TrueType as a font format, but here we discuss TrueType as font-rendering machinery. TrueType font-rendering technology is fundamentally the same as used by ATM in that it renders character descriptions (glyphs or outlines) on the computer, then uses the resulting bitmapped fonts for both display and printing.† The TrueType font-rendering machinery has been an integral part of Mac OS since System 7, and likewise since Microsoft Windows version 3.1. The latest versions of these OSes include ATM functionality, which effectively allows PostScript fonts to be used, but the practical result is that OpenType fonts can be used.

Although I did not like history class during my high school years, and my grades clearly reflected this attitude, I do enjoy historical trivia that relates to this book. Go figure. In any case, like ATM, TrueType began as a Latin-only font-rendering technology. In mid-1992, Apple released a package called *Kanji TrueType*, which included two TrueType Japanese fonts: Ryobi's 本明朝-M (*hon minchō M*) and FDPC's 平成角ゴシック (*heisei kaku goshikku*). Then, late in 1992, Apple subsequently released KanjiTalk version 7.1, which was bundled with seven TrueType Japanese fonts—the two just listed plus Osaka (identical to the Kanji TrueType font 平成角ゴシック, but instead set in a 256×256 design space, as opposed to the more typical 2048×2048 design space of TrueType fonts), FDPC's 平成

* *http://pages.cs.wisc.edu/~ghost/*

† Of course, bitmaps of different resolutions are generated to accommodate screen display and printing, both of which require wildly different resolutions.

明朝 (*heisei minchō*), Ryobi's 丸ゴシック-M (*maru goshikku M*), Morisawa's 中ゴシック BBB (*chū goshikku BBB*), and Morisawa's リュウミンライト-KL (*ryūmin raito KL*).* Chinese and Korean versions of Mac OS were also bundled with TrueType fonts appropriate for the locale.

TrueType fonts do have many merits, such as their use of an excellent font-caching mechanism that makes subsequent displaying—well, to be more precise, redisplaying—of glyphs extremely fast.

Microsoft and Apple had further developed their (at the time, incompatible) TrueType formats in order to incorporate what most people refer to as advanced typographic functionality. These technologies were known as TrueType Open and QuickDraw GX, respectively. As noted in Chapter 6, QuickDraw GX subsequently became known as AAT. Also as discussed in Chapter 6, TrueType Open effectively transitioned into OpenType.

Other Printing Methods

A small number of you may not have access to the printing methods described up until now, most likely due to the fact that you do not use an OS with built-in font rendering, or do not own or otherwise have access to a PostScript CJKV printer. Fortunately, you will usually find that the CJKV text-processing software you are using comes with at least a bare-bones method for outputting CJKV text to a printer, whether it uses outline or bitmapped fonts.

There are freely available and dedicated CJKV printing kits, such as CNPRINT, available for MS-DOS, Unix, and VMS, and developed by Yidao Cai (蔡依道 *cài yīdào*).† These programs accept CJKV text files as input, then format the text for printing. These printing kits often come bundled with at least a minimal set of bitmapped fonts, but you are not always limited to using those. Some word processors that do not require an underlying CJKV-capable OS almost always have similar printing facilities.

Besides PostScript and these printing kits, there are other typesetting and page-description languages in use today. Examples include *ditroff*, *triroff*, *troff*, TEX, and LATEX. Some of these, such as TEX and LATEX, have had CJKV-capable versions available for a long time. In fact, there are many slightly different versions of TEX available, such as those made available by NTT and ASCII. Japanese TEX, for example, makes use of fonts from Dainippon Printing. There are also quite a few Korean-enabled versions of TEX and LATEX

* As discussed in Chapter 6, the localized menu names for these two Morisawa fonts were not the same as their PostScript counterparts, and the proportional Latin characters were metrically incompatible with the same PostScript counterparts (although some contemporary PostScript printers include a solution that allows the TrueType version to effectively alias to the PostScript version resident on the printer, and compensates for the metrics incompatibility).

† *http://www.ywpw.com/cai/software/*

available.* For more general information on TEX, I suggest exploring the many books written by its creator, Donald Knuth (高德纳).†

As sort of a section summary, the material in this chapter concentrates on PostScript because it produces a high-quality printed page on virtually any output device, and because the most commonly available outline fonts are in PostScript format. Perhaps of more importance is the fact that these other typesetting and page-description languages have the ability to generate PostScript.

The Role of Printer Drivers

Whenever a document is to be printed, there is mandatory interaction with software called a printer driver. The printer driver communicates with the printer, and whether the printer driver can query the printer—that is, it asks the printer a question about its configuration, such as supported paper sizes or what fonts are resident in its memory or otherwise available for use, then expects an answer—depends on the OS on which it is running, and perhaps the vintage of the OS. For many applications, usually simpler ones, it is the printer driver that is responsible for generating the PostScript that is ultimately sent to the printer for printing. For what are considered high-end applications, such as the graphics or page-composition applications that are described at the end of Chapter 7, the printer driver does not typically generate PostScript, but rather acts as a pass-through agent. In other words, the printer driver accepts or receives the PostScript that is generated by an application, then merely passes it along, without modification, to the printer so that it can be printed.

Printer drivers can also play other critical roles, such as font subsetting and embedding. Although it is common practice to embed Type 1 fonts in PostScript files, primarily to ensure that the printer will use the correct fonts, or the correct version of the fonts, doing the same for CJKV fonts is more problematic for a variety of reasons, such as the following:

- CJKV fonts are typically several megabytes in terms of file size, which often exceeds the RAM capacity of most printers and imagesetters. These fonts have historically been downloaded to the printer's hard disk.‡
- Only a small fraction of the glyphs in a CJKV font are used in a typical document, which makes font subsetting extremely attractive—consider that 70% of typical Japanese text consists of kana, which represent less than 200 glyphs in a font when totalled.
- While the average size of the CJKV fonts I used to produce this book is approximately 5 MB, the largest is over 40 MB!

* *http://www.ktug.or.kr/*

† *http://www-cs-staff.stanford.edu/~knuth/*

‡ CJKV fonts typically require additional resources and infrastructure that may or may not be present on a given printer.

These are all good reasons why embedding entire CJKV fonts has been difficult to implement and is simply not practical. Some printer driver are responsible for performing font subsetting, which consists of three basic steps:

- Determine what glyphs in a given font are used in the requested document.
- Create a new version of the font that contains only the glyphs for the characters referenced in the document.
- Embed the newly built subsetted font into the PostScript file.

Adobe Acrobat, as a medium for creating, displaying, and printing PDF files, also supports font subsetting. When a PDF file is created, only the glyphs that are referenced in the document are embedded.

Microsoft Windows Printer Drivers

In the past, the availability of appropriate printer drivers on Windows was critical in getting your documents to print correctly. Windows had traditionally been a platform that did not allow the printer driver to query the printer. This is why the infamous *PostScript Printer Description* (PPD) files were absolutely necessary.

PPDs contain the printer's configuration information in a convenient (and local) form that can be easily understood (that is, parsed) by printer drivers. PPDs are, in general, read-only documents, but some applications retain their own set of writable PPDs in order to manage fonts that have been subsequently added (that is, aftermarket fonts that have been downloaded) to the printer.

At one point, there were three suitable printer drivers available for Windows, listed in Table 8-1.

Table 8-1. Windows printer drivers—historical

Printer driver	Developer
AdobePS	Adobe Systems
PS Print	EAST
PScript	Microsoft

Although no longer developed nor available, Adobe Systems' *AdobePS* printer driver represented effort to improve the printing architecture for Windows, and provided users and developers alike with plug-in functionality.

Also no longer developed nor available, EAST's *PS Print* printer driver was offered in a variety of versions, depending on what version of Windows you were using, and what

resolution of printer you expected to use.* One feature unique to PS Print was its crop-mark editor.† Unfortunately, the PS Print series was available only for Japanese.

Finally, *PScript* is Microsoft's version of a PostScript printer driver, codeveloped with Adobe Systems.‡ At one point, Adobe Systems enhanced PScript's features, and made it available as AdobePS (previously described). PScript does not provide any sort of plug-in functionality. The version of PScript included in Windows NT version 4.0 and earlier was developed by Microsoft alone, but the version that is included in Windows 2000 and later, to include Windows XP and Vista, was codeveloped with Adobe Systems once again. For printing to non-PostScript devices, *Unidrv* is used instead.§

So, when considering the latest versions of Windows OS, specifically XP and Vista, there is effectively one choice for the printer drive. In other words, no choice. The simplicity of this is a good thing. The printer driver works well.

Mac OS X Printer Drivers

Now, comparable to what happened on Windows, printing on Mac OS X has become simplified. To be more accurate, greatly simplified, when compared to the situation prior to Mac OS X. There are no longer any choices, and thus no confusion, which is precisely why the printing situation has become simplified.

However, prior to Mac OS X, Mac OS users were faced with the prospect of choosing among the following PostScript printer drivers, depending on what other software happened to be installed:

- AdobePS
- LaserWriter
- LaserWriter 8
- PSPrinter

How were each different? Who developed them? When should one be chosen over another? These were all good questions, because it was never clear when it was best to use one over the other.

Both LaserWriter and LaserWriter 8 were maintained by Apple—they are different only in that LaserWriter 8 was of more recent vintage than LaserWriter. LaserWriter 8 was originally developed by Adobe Systems as *PSPrinter*.

PSPrinter, which was renamed to become AdobePS, was developed by Adobe Systems and bundled with virtually all Adobe Systems' applications.

* *http://www.est.co.jp/psprint/* and *http://www.est.co.jp/prndrv/*

† Crop marks are called トンボ (*tombo*) in Japanese.

‡ *http://msdn.microsoft.com/en-us/library/aa506075.aspx*

§ *http://msdn.microsoft.com/en-us/library/ms802044.aspx*

Historical Korean printing issues

PostScript printing of Korean text on Mac OS (meaning prior to Mac OS X), unlike for Chinese and Japanese text, took one of two forms:

- One-byte printing
- Two-byte printing

The *one-byte printing* method assumed that the Korean fonts resident on the PostScript printer were constructed as a series of one-byte–encoded Type 1 fonts—in other words, not a genuine composite font. The *two-byte printing* method is more along the lines of Chinese and Japanese printing, in which the Korean fonts that are resident on the PostScript printer are genuine composite fonts. A Control Panel, included with Mac OS-KH, called *Hangul Jojung* (한글조중 *hangeul jojung*), controlled what printing method was to be used. As we discovered at Adobe Systems shortly after Apple's Korean Language Kit was released at the end of 1996, this Control Panel was not included. The result was that only one-byte–encoded Korean fonts on PostScript printers would work. Apple subsequently released this Control Panel for KLK users.

Output Tips and Tricks

Printing CJKV documents can be problematic for some users, depending on their hardware and software environment. Here I present some tips and tricks that may help some readers to produce CJKV documents in unusual environments.*

Creating CJKV Documents for Non-CJKV Systems

One of the most common questions I get asked is how to create a document that includes CJKV text, but can be exported for use on non-CJKV systems, either for displaying or printing. Fortunately, there are several ways to create these types of documents, such as the following:

- Convert CJKV text into outline graphics
- Convert CJKV text into bitmapped graphics
- Use CJKV-capable Adobe Acrobat

A very brief introduction to Adobe Acrobat was provided in earlier sections of this chapter, and other aspects of Adobe Acrobat are covered in Chapter 12. The following sections detail methods for converting CJKV text into graphic objects, either outline objects or bitmapped graphics.

The latest versions of many applications now support a wide variety of formats, in terms of importing graphics for use in documents. Adobe InDesign is an excellent example. Most

* The use of the term "unusual" here usually refers to non-CJKV–capable environments.

users expect that word processors and document composition software will support EPS, JPEG, and TIFF in terms of importable graphics formats. Adobe InDesign additionally supports PDF and the native formats of some applications, most notably those of Adobe Illustrator and Adobe Photoshop. The ability to import the native formats of applications serves to reduce the number of files that need to be maintained. Instead of maintaining the original file, in native format, along with an EPS, JPEG, or TIFF file for importing into another application, only the single native-format file needs to be maintained. Simple is almost always better, especially if it is coupled with no loss of functionality.

Converting CJKV text into outline graphics

Converting CJKV text into outline graphics requires an application that can perform this functionality. Adobe Illustrator is a suitable application for this task.

This operation is extremely simple. First, you compose and lay out the text as desired. Second, you select the command for converting text into editable outlines. Explore the menus, or consult the manual to determine exactly where in the UI this command is provided. Finally, you save the file as an EPS (*Encapsulated PostScript*) file, unless you plan to import the file into an application that supports the native file format. As already explained, Adobe InDesign can import native Adobe Illustrator files. In any case, the end result of this operation is an EPS or native file that can be imported into other applications, such as those that perform page composition, as graphics.

For some graphic designers and users, this functionality is necessary whether or not they are using a CJKV-capable OS or have a CJKV-capable printer—they require access to glyphs' outlines so that they can make changes, sometimes quite subtle. Graphic designers may simply want to "fill" the character's outline with some kind of interesting pattern, or change its shape in subtle ways for use in advertisement.

The following are some time-saving tips and tricks that are very much worth mentioning at this time:

- Creating outline graphics, as opposed to bitmapped graphics, has the advantage that the EPS or native file can be scaled to almost any size.
- Converting text to outline graphics results in a form that no longer contains hinting information that benefits small point sizes at low resolution. However, when these EPS or native files are output at high-resolution, one would be hard-pressed to tell the difference.
- Because text that has been converted into editable outlines can no longer be edited as text, you are encouraged to save the text for possible future editing. The preferred way in which to do this is to include two identical layers in your document whereby one layer contains the actual text (flagged so that it does not print) and the other layer contains the text after conversion to editable outlines (this is the layer that should be flagged to print). Trust me, this tip alone can save you plenty of time!

- In the past, some Japanese fonts, such as those from Morisawa, were protected in such a way that their outlines were not accessible through this feature.

Keep these points in mind when creating EPS or native files containing text converted into editable outline graphics. Fortunately, there are still ways to work with fonts that do not allow access to their outlines, as described in the following section.

Converting CJKV text into bitmapped graphics

A technique that turns text into bitmapped graphics is desirable only under circumstances when a font's outlines are protected in such a way that prevents applications from accessing their outlines, or when you must use text in an application that supports only bitmapped graphics, such as earlier versions of Adobe Photoshop. Thankfully, the number of fonts that have their outlines protected in this fashion have diminished, and are not likely to increase. For example, I know of no OpenType fonts that have their outlines protected this way.

Converting text into bitmapped graphics first and foremost requires an application suited for the task—Adobe Photoshop is an appropriate tool. All of the steps outlined previously for converting text into editable outlines apply here, except that the conversion to bitmapped graphics is more or less an automatic process as you input text. The handling of text in Adobe Photoshop has also improved, in more than one way, as follows:

- The text engine that is used by Adobe Photoshop is more advanced and produces much better results than before, in terms of text composition.
- When entering text, two layers are automatically maintained, one of which preserves the editable text for the purpose of future editing, and one which is the rendering layer. These layers are saved as part of the Photoshop document file.

In addition, you need to determine what the final output resolution of the document is so that you can set the resolution of the document workspace.*

Advice to Developers

This section presents some of my opinions and suggestions regarding the acquisition of a CJKV-capable publishing system, including hardware and software. Thankfully, much of what is needed is no longer found in dedicated hardware or software.

CJKV-Capable Publishing Systems

After reading the material in this and earlier chapters, you should be well convinced that using outline fonts for generating high-quality CJKV output is indeed the best choice all

* Note, however, that creating a full-page Adobe Photoshop document whose resolution is equivalent to that of a photo imagesetter (1270-dpi or greater) will require several megabytes of disk space. It is not a decision to make lightly, especially if you need to produce a large number of pages.

around. Investing in a publishing system is something that should be done with great care in order to ensure the best possible results.

Only a few years ago, you would have needed to spend tens of thousands of dollars to produce high-quality CJKV documents. But, believe it or not, you can now purchase an entire hardware and software system for producing high-quality CJKV documents for well under $10,000 (U.S.). The actual price may fluctuate depending on how many CJKV typefaces you decide to purchase, and on other factors (for example, you may already own some of the hardware or software).

My recommendations for a basic CJKV-capable publishing system, at least when I wrote this chapter of this book, are provided in Table 8-2.

Table 8-2. CJKV publishing system hardware and software recommendations

Hardware/Software	Description	Estimated cost
Printer	Color PostScript 3 with 600-dpi or greater resolution	$500–$1,000
Typefaces	Serif, sans serif, and script typeface designs in text and display weights	$2,000
CPU	The greatest speed, memory, and capacity that you can afford	$2,000
OS	Mac OS X or Windows Vista	$130–$320
PDF authoring	Adobe Acrobat Pro[a]	$450–$700
Document composition	Adobe InDesign[a]	$700
Graphics	Adobe Photoshop and Adobe Illustrator[a]	$1,300–$1,600

a. One way to save money is to buy the Adobe Creative Suite, either Design Standard or Design Premium, which includes this and other applications for $1,400 and $1,800, respectively.

Given the state of contemporary OSes—such as Mac OS X and Windows Vista—and the large number of bundled fonts, it is possible to use only OS-bundled fonts, as long as your font needs are not complex. Some applications, such as those offered by Adobe Systems, are bundled with CJK fonts. Mac OS X, as an example of an OS, is bundled with a half-dozen high-quality OpenType Japanese fonts, most of which are based on the Adobe-Japan1-5 character collection.

Some Practical Advice

You may be wondering, even after reading this chapter and earlier chapters, which outline font format to use: PostScript or TrueType. Most of what I covered here and in Chapter 6 may have seemed focused on or biased toward PostScript fonts. The correct answer is simple: use both. In fact, OpenType is useful in that it effectively blurs the distinction between PostScript and TrueType fonts. An OpenType font can use PostScript or TrueType outlines, in the 'CFF' and 'glyf' tables, respectively. In other words, a TrueType font is an OpenType font.

I strongly suggest working with OpenType fonts. There are a number of reasons for this. First, OpenType fonts work very well with Adobe Systems' entire line of applications. Second, OpenType fonts are cross-platform, meaning that they work with the latest versions of OSes developed by Apple and Microsoft, and also with most Linux distributions as long as the application uses FreeType 2 or an equivalent OpenType-*savvy* font rasterization library. Third, OpenType has become the preferred font format by type foundries across the globe. Fourth, the font development tools that support OpenType have proliferated much broader than those tools that supported legacy font formats, which explains why OpenType has become the preferred font format by type foundries.

Interestingly, in *Understanding Japanese Information Processing*, which was written way back in 1993, I speculated that PostScript and TrueType font technologies may eventually merge or at least become somewhat indistinguishable.* When the first edition of this book was written, OpenType was at the very early stages of development, as a cooperative effort between Adobe Systems and Microsoft, and its infrastructure was still being developed. Clearly, what was speculated in 1993 has taken place in the context of OpenType, which was discussed in Chapter 6. As already stated, OpenType has become the preferred font format, and is well supported by the major OSes and the applications that run on them.

Finally, I strongly suggest that you embrace PDF as your preferred document format for printing and publishing. The genuinely self-contained nature of PDF is a true blessing, and it can be used to print and display the simplest and most complex documents on a wide variety of devices.†

* For this sort of technology, 1993 is considered a long time ago, but not in a galaxy far, far away.

† As a tidbit of historical trivia related to this book, the first printing of the first edition of this book was done the old-fashioned way, by sending PostScript files to a photo imagesetter and outputting photographic paper. That was at the very end of 1998. In the latter part of 2002, Adobe Acrobat, PDF, and the publishing industry had matured to the point that the second printing of the first edition could be published via PDF, which also paved the way to making the first edition of this book available in PDF at the end of 2006.

CHAPTER 9
Information Processing Techniques

Remember what you have learned in earlier chapters, specifically that CJKV character set standards and encoding methods have many characteristics that make each one unique and distinct. But within a given locale, there is comfort in knowing that at least some effort was made to keep its various encoding methods somewhat compatible. More importantly, the various encodings for Unicode are entirely and completely compatible. In addition, compatibility with Unicode has perhaps become even more critical, given its widespread use in today's OSes and application. What we are discussing, of course, is the issue of interoperability.

Understanding that interoperability issues have become increasingly important—even critical—when one is dealing with CJKV information processing on multiple platforms is a prudent mind set to maintain. Not all computer systems use the same encoding method. Using Japanese as an historic example, Shift-JIS encoding was typically used on Windows- and Mac OS–based machines, EUC-JP encoding was used on Unix-based machines, and ISO-2022-JP encoding was used for electronic transmission, such as email and news. Even with Unicode, there are three basic encodings, and the possibility of needing to deal with all three of them, in addition to interoperating with legacy encoding, is relatively high.

Because I was faced with the difficulties of converting, manipulating, and generating Japanese text, I decided to develop a suite of tools for performing such tasks. Some of these tools have been extended or enhanced to accommodate other CJKV character sets and encoding methods, including Unicode.*

You will find that this chapter first discusses locale issues, then programming languages, followed by algorithms for actual byte-value conversion, which represents the heart of the CJKV code conversion process. However, that is not all that is required. Next, we move on to text stream handling, which serves as the wrapper for the code conversion routines.

* Two of these Japanese-specific tools have been extended, but have taken on different forms: *JChar* is now the *CJKV Character Set Server*, as described at the end of this chapter, and some of the functionality found in *JConv* is available in *CJKVConv.pl*, as described in the section entitled "Code Conversion Across CJKV Locales," in Chapter 4.

As you'll soon realize, the latest Java APIs simplify this task immensely. Programming languages such as Java can perform much of the necessary code conversion through the use of built-in methods, which saves programmers significant time, effort, and energy. There are, however, specialized algorithms that may or may not be available as built-in methods or functions, such as half- to full-width katakana conversion (Japanese-specific) and automatic code detection.* But, even implementing these algorithms in Java provides much simplification due to the use of Unicode—the UTF-16 encoding form—internally. This chapter continues with information about handling multiple bytes as a single unit for operations such as text insertion, deletion, and searching. CJKV implications for sorting, parsing, and regular expressions are covered at the end of the chapter.

In most cases, workable C or Java source code, along with an explanation of the algorithm is given (Appendix C provides Perl equivalents of some of these algorithms). Feel free to use these code fragments in your own programs—that is why they are included in this book. The code samples that I provide here may not be the most efficient code, but they do work.† Feel free to adapt what you find here to suit your own programming style or taste. The entire source code for these algorithms and examples is available in machine-readable form.‡

A not-so-new aspect of programming and OSes is the ability to use what is known as the *locale model*. The locale model is a system that predefines many attributes that are language- and locale-specific, such as the number of bytes per character, date formats, time formats, currency formats, and so on. The actual attributes are located in a library or locale object file, and are loaded when required. The locale model as originally defined by X/Open's XPG4 (*X/Open Portability Guide 4*) and IEEE's POSIX (*Portable Operating System Interface*) contained several categories of features: code set information, time and date formats, numeric formatting, collation information, and so on. Now, the preferred way in which to encapsulate and manage these locale properties is through the use of the CLDR (*Common Locale Data Repository*).

Language, Country, and Script Codes

A typical locale identifier is composed of two components, separated by an underscore. The first part specifies a language identifier ("en" for English, "ja" for Japanese, "ko" for Korean, "zh" for Chinese, "vi" for Vietnamese, and so on). ISO 639-1:2002, *Code for the Representation of Names of Languages, Alpha-2 Code*, is the ISO standard that serves to register these and other two-letter language identifiers.

The second part of a locale specifies a county or region. The standard designated ISO 3166-1:2006, *Codes for the Representation of Names of Countries and Their Subdivisions—*

* After all, before you can convert from one encoding to another, you need to know what the original encoding is.

† If they do not work, I want to know about it!

‡ *http://examples.oreilly.com/9780596514471/Ch9/*

Part 1: Country Codes, specifies (and thus registers) country codes, such as "US" for the U.S., "CN" for China, "TW" for Taiwan, "HK" for Hong Kong, "SG" for Singapore, "JP" for Japan, "KR" for South Korean (Republic of Korea), "KP" for North Korea (Democratic People's Republic of Korea), "VN" for Vietnam, and so on. These are the same two-letter country codes found in many email addresses and URLs.

Note the use of upper- versus lowercase—it is intentional. Also be sure to see RFC 3066, *Tags for the Identification of Languages*, for more information.*

There are now three-letter language codes as defined in ISO 639-2:1998, *Codes for the Representation of Names of Languages—Part 2: Alpha-3 Code*.† Table 9-1 lists ISO languages codes that are of interest to readers of this book.

Table 9-1. ISO 639 two- and three-letter language codes

Language	ISO 639	Terminological (ISO 639-2/T)	Bibliographic (ISO 639-2/B)
English	en	eng	eng
Chinese	zh	zho	chi
Japanese	ja	jpn	jpn
Korean	ko	kor	kor
Vietnamese	vi	vie	vie

Note how the two types of three-letter language codes are identical except for Chinese. Three-letter country codes have also been established. Table 9-2 lists some of them, along with their two-letter counterparts.

Table 9-2. Two- and three-letter country codes

Country	Two-letter	Three-letter
United States	US	USA
China	CN	CHN
Taiwan	TW	TWN
Hong Kong	HK	HKG
Singapore	SG	SGP
Japan	JP	JPN
South Korea	KR	KOR
North Korea	KP	PRK
Vietnam	VN	VNM

* *http://www.ietf.org/rfc/rfc3066.txt*

† *http://www.loc.gov/standards/iso639-2/*

As the world has become more complex, by the mere act of supporting more locales in OSes and applications, the need to more explicitly and consistently declare locales has increased. Thus, moving from two to three-letter identifiers makes perfect sense.

The locale "zh_CN," for example, refers to Chinese as used in China, and would include a Chinese code set (such as GB 2312-80 or GB 18030-2005), a yuan (￥) for the currency symbol, and so on. The locale "zh_TW" refers to Chinese as used in Taiwan, and would use appropriate locale attributes specific to Taiwan. For more information on the locale model, I suggest that you obtain three *X/Open CAE Specifications* books (CAE stands for *Common Applications Environment*) and the *X/Open Guide: Internationalisation Guide*. See this book's bibliography for more information.

Another useful technique is to be able to identify the script. A language can use multiple scripts (for example, Korean uses Latin characters, hangul, and hanja), so this technique is useful for identifying blocks of text. Table 9-3 lists scripts, along with their ISO 15924:2004 (*Codes for the Representation of Names of Scripts*) two-letter codes and ISO 10179:1996 (*Information Technology—Processing Languages—Document Style Semantics and Specification Language (DSSSL)*) public identifiers. Note the intentional use of upper- then lowercase for the two-letter ISO 15924 codes.

Table 9-3. Script codes and identifiers

Script	ISO 15924:2004	DSSSL
Bopomofo	Bp	Script::Bopomofo
Chữ Nôm	Cu	none
Ideographs	Hn	Script::Han
Hangul	Hg	Script::Hangul
Hiragana	Hr	Script::Hiragana
Katakana	Kk	Script::Katakana
Latin	La	Script::Latin
Undetermined script	Zy	none
Unwritten languages	Zx	none

Although fonts, particularly OpenType fonts, declare and use scripts, they use a different system as described in the OpenType specification.* OpenType fonts additionally declare and use three-letter language tags, which are different from the three-letter ones specified by ISO 639-2.†

When used and applied wisely, country, language, and script codes can be used to enhance the performance and reliability of software when handling multilingual information. But

* *http://www.microsoft.com/typography/otspec/scripttags.htm*

† *http://www.microsoft.com/typography/otspec/languagetags.htm*

there is much more to locales than country, language, and script codes. As previously noted, there is some inconsistency in that a common system has yet to be described. This is the perfect time to discuss CLDR.

CLDR—Common Locale Data Repository

The *Common Locale Data Repository* (CLDR), put simply, is a repository or database of locale data that helps to ensure consistent use of such data across OSes and applications.* Given the extent to which languages and scripts are now implemented in today's OSes and applications, a framework for ensuring consistency is absolutely critical, and the CLDR fills this important need. Why is this important? Today's OSes and applications do not function in a vacuum, and instead interoperate to varying degrees. When dealing with a multitude of languages and scripts, the ability to uses a consistent set of locale data across these OSes and applications is clearly a good thing, and leads to less frustration on the part of the software developer and a much better experience for the end users.

CLDR is, of course, based on Unicode, and is written in XML according to LDML (*Locale Data Markup Language*).† The contents of each CLDR release are based upon contributions, often by member companies and individuals of The Unicode Consortium.

Programming Languages

A short discussion of several popular programming languages—C/C++, Java, Perl, Python, Ruby, and Tcl—is necessary before we can meaningfully discuss the information processing techniques presented in the remainder of this chapter. While what I write in the following sections is by no means a complete description or treatment of these programming languages, it does provide information about their salient features and virtues as they relate to CJKV programming.

While it is easy—and, quite frankly, bordering on childish—to explore and argue about the strengths and weaknesses of programming languages, it is ultimately the method of deployment that usually determines what programming language is chosen for a specific task. Like all tools, particularly those that are complex, half the battle involves knowing their strengths and weaknesses, and the other half is subsequently knowing how to take best advantage of their strengths and working around their weaknesses.

A programming trend that is catching on are variables and data structures that use multiple-byte or wide characters.‡ Although C and C++ were enhanced to support these variables and data structures, widespread support has never been achieved. That has since changed with the introduction of the Java programming language, created by James Gosling and Bill Joy of Sun Microsystems.

* *http://www.unicode.org/cldr/*

† *http://www.unicode.org/reports/tr35/*

‡ The issue of multiple-byte versus wide characters was first discussed in Chapter 1.

Most of the algorithms and techniques provided in this chapter make use of the Java APIs, which provides programmers with exceptional internationalization support. Providing C or C++ examples that make use of the locale model or multiple-byte and wide character data structures is not very useful because many compilers still do not support them (ensuring nonportability from day one). Programmers who use C or C++ generally resort to writing their own input and output mechanisms for handling issues such as representing two or more bytes as a single character. If you are a programmer, I encourage you to explore the Java programming language.

C/C++

Most CJKV-capable programs today are still written in C or C++. The wide availability of C/C++ compilers has made this possible, along with the still-stellar performance of the executables that result. However, because of the weak or nonexistent support for internationalization in such compilers, C/C++ may no longer the best choice for some purposes and applications.

It is important to understand, or at least appreciate, the programming paradigms and structures offered by C/C++ because they have been instrumental in forming the foundations for the seemingly more contemporary programming languages, covered next. Many of their programming constructs, such as conditional statements and loops, have been "borrowed" into languages such as Java, Perl, Python, and Ruby. This makes it a much easier task for C/C++ programmers to learn and subsequently master these other programming languages.

One pitfall of the C programming language is that it has the concept of "signed" and "unsigned" characters (type *char*), and in most common implementations the default is signed. For all meaningful string comparisons using multiple-byte encodings, the programmer must set everything to unsigned *char*, which can cause compiler warnings because prototypes no longer agree. And, things won't work correctly if unsigned *char* is used. Some compiler developers, bless their hearts, give programmers the ability to change the default to unsigned *char*.

Java

The first programming language that experts considered to be fully suitable for CJKV programming is Java. Java was touted as being the first programming language to include support for Unicode, but it was not until the version 1.1 release of the language that this feature or capability was fully realized.

The standard Java I/O (*Input/Output*) package, instantiated as *java.io*, provides built-in support for converting between Unicode (the UTF-16 or UTF-8 encoding forms) and numerous non-Unicode encoding methods, such as those covered in Chapter 4 of this book. Java's NIO (*New Input/Output*) package, instantiated as *java.nio*, provides even more elaborate code conversion facilities. This fact alone could effectively render most of

this chapter as being no longer necessary, but luckily for us, there are still some tips, tricks, pitfalls, and caveats that developers need to be aware of before diving headfirst into Java.

A lot of thought has obviously been put into the design of Java as a programming language. One problem that is almost always encountered when writing programs in any language is portability. This usually results from differing data type sizes for different architectures. Table 9-4 lists data types and their sizes as specified by Java.

Table 9-4. Java data types and sizes

Data type	Size (in bits)
boolean	1
char[a]	16
byte	8
short	16
int[b]	32
long	64
float	32
double	64

a. Also known as a UTF-16 code unit.

b. Also known as a UTF-32 code unit.

With almost any other programming language (with C and C++ being as good examples as any), the data type sizes listed in Table 9-4 differ depending on the underlying architecture. Java is effectively a programmer's dream come true, because it implements consistent and cross-platform data sizes in which programmers prefer to think (with the possible exception of type *char*).

Unicode characters can be directly specified in Java as UTF-16 code units through the use of the 16-bit *char* data type and the `\u`*XXXX* notation, whereby the `XXXX` is a placeholder for four hexadecimal digits. Characters outside the BMP, meaning UTF-16 Surrogate Pairs, must therefore be directly represented, so that U+20000 is represented using Java *char* data type as `\uD840\uDC00`. The 32-bit *int* data type is used when specifying characters outside the BMP without using Surrogate Pairs.

An excellent guide to the Java programming language is David Flanagan's *Java in a Nutshell*, Fifth Edition (O'Reilly Media, 2005).*

* *http://oreilly.com/catalog/9780596007737/*

Perl

Usually described as a scripting language, Perl, developed by Larry Wall, is much, much more than that. Perl's main strengths include rapid development, regular expressions (described later in this chapter),* and hashes (associative arrays). It is not so much these individual features that provide Perl with extraordinary text-manipulation capabilities, but rather how these features are intertwined with one another. Other programming languages offer similar features, but there is often no convenient way for them to function together. In Perl, for example, a regular expression can be used to parse text, and at the same time it can be used to store the resulting items into a hash for subsequent lookup.

Perl has been the programming language of choice for those who write CGI programs or do other web-related programming (a topic that is discussed in Chapter 12), because it is well-suited for the task.

The extent to which Perl has become Unicode-enabled is now well-documented.† However, it is strongly advised that the excellent *Perl Unicode Tutorial* be read before exploring that documentation.‡ A lot of the Unicode-enabled portions are exposed when you use the *Encode* module, such as including the following in your Perl program:

```
use Encode;
```

Be sure to read the documentation for the Encode module for more information about its use and limitations.§

Ignoring the Unicode-enabling efforts for Perl, there are still clever ways to use Perl for handling multiple-byte data, most of which make use of regular expression tricks and techniques. The Perl code examples provided in Appendix C should be studied by any serious Perl programmer.

Kazumasa Utashiro (歌代和正 *utashiro kazumasa*) has developed a useful Japanese-enabling Perl library called *jcode.pl*, which includes Japanese code conversion routines.¶ Some may find the Japanese version of Perl, called *JPerl*,** to be useful, although I suggest using programming techniques that avoid JPerl for optimal portability. JPerl adds Japanese support to the following features: regular expressions, formats, some built-in functions (*chop* and *split*), and the *tr*/// operator.

Unicode characters can be directly specified in Perl as scalar values through the use of the `\x{`*`XXXX`*`}` notation, whereby the `XXXX` is a placeholder for up to six hexadecimal digits.

* Tom Christiansen and Nathan Torkington, in *Perl Cookbook*, Second Edition (O'Reilly Media, 2003), describe Perl's regular expression implementation in the following sentence: "It's more like string searching with mutant wildcards on steroids."

† *http://perldoc.perl.org/perlunicode.html*

‡ *http://perldoc.perl.org/perlunitut.html*

§ *http://perldoc.perl.org/Encode.html*

¶ *http://srekcah.org/jcode/*

** *http://www.perl.com/CPAN/authors/Hirofumi_Watanabe/*

The definitive guide to Perl is *Programming Perl*, Third Edition, by Larry Wall et al. (O'Reilly Media, 2000).* Tom Christiansen and Nathan Torkington's *Perl Cookbook*, Second Edition (O'Reilly Media, 2003) is also highly recommended as a companion volume to *Programming Perl*.† The best place to find Perl and Perl-related resources is at CPAN (*Comprehensive Perl Archive Network*).‡

Python

Like Perl, Python is also sometimes described as a scripting language. Python was developed by Guido van Rossum, and is a high-level programming language that provides valuable programming features such as hashes and regular expressions.

Python uses a similar notation as Java for directly specifying Unicode scalar values, specifically `\u`*XXXX*, which also expects exactly four hexadecimal digits. If you want to specify characters outside the BMP, you must instead use the `\U`*XXXXXXXX* notation that expects eight hexadecimal digits. Python also makes use of the *unicode()* constructor for specifying Unicode strings, and also the built-in *unichr()* function for specifying single Unicode characters through the use of integers.

An excellent guide to Python is Mark Lutz's *Programming Python*, Third Edition (O'Reilly Media, 2006).§ There is also a Python website from which Python itself is available.¶

Ruby

Ruby, which is a programming language created by Yukihiro "Matz" Matsumoto (松本行弘 *matsumoto yukihiro*), has become very popular, and for many reasons.** Work on Ruby began in 1993, and was first released in 1995. By the year 2000 it became popular, not only in Japan, but worldwide.††

Ruby can be characterized as a purely object-oriented scripting language that is highly productive, hence its popularity. Ruby was invented in Japan, so naturally some basic internationalization features are built-in, but there is still work to do in that area. The best way in which to make use of Ruby in the context of Unicode is through the UTF-8 encoding form.

* *http://oreilly.com/catalog/9780596000271/*

† *http://oreilly.com/catalog/9780596003135/*

‡ *http://www.perl.com/CPAN/*

§ *http://oreilly.com/catalog/9780596009250/*

¶ *http://www.python.org/*

** *http://www.rubyist.net/~matz/*

†† On a more personal note, the year 2000 is also significant in that my daughter, who happens to be named Ruby, was born then. Guess what programming language she is likely to learn when she is older? (Where, oh where, is a *smiley* when I need one….)

I suggest that you explore the main Ruby programming language website,* which provides an interactive tutorial so that beginners can come up to speed more quickly.† In terms of books, I suggest *The Ruby Programming Language* (O'Reilly Media, 2008) by David Flanagan and Yukihiro Matsumoto.‡

Tcl

Tcl, which stands for *Tool Command Language*, and often pronounced as "tickle," is a programming language that was originally developed by John Ousterhout while a professor at UC Berkeley.§ Like Perl, Python, and Ruby, Tcl is considered a high-level scripting language that provides built-in facilities for hashes and regular expressions. John later founded Scriptics Corporation where Tcl continued to be advanced.

Some important milestones in Tcl's history include its byte-code compiler introduced for version 8.0, and support for Unicode, in the form of the UTF-8 and UTF-16 encoding forms, that began with version 8.1. Tcl now includes a regex package comparable to that used by Perl. Interestingly, the lack of a byte-code compiler had always kept Tcl slower than Perl.

Tcl is rarely used alone, but rather with its GUI (*Graphical User Interface*) component called *Tk* (standing for *Tool Kit*), which is referred to as Tcl/Tk.

Other Programming Environments

Although it is possible to write multiple-byte–enabled programs using all of the programming languages just mentioned, there are some programming environments that have done all this work for you, meaning that you need not worry about multiple-byte enabling your own source code because you depend on a module to do it for you. This may not sound terribly exciting for companies with sufficient resources and multiple-byte expertise, but may be a savior for smaller companies with limited resources.

An example of one such a programming environment is Visix's *Galaxy Global*, a multilingual product based on their *Galaxy* product. (Visix Software has since gone out of business, well over 10 years ago.)

Perhaps of greater interest is Basis Technology's *Rosette Core Library for Unicode*, which is a compact, general-purpose Unicode-based source code library.¶ When embedded into an application, this library adds Unicode text-processing capabilities that are robust and efficient across a variety of platforms, such as Windows, along with various flavors of Linux and Unix. Its functions adhere to the latest Unicode specifications. Important

* *http://www.ruby-lang.org/*

† *http://www.ruby-lang.org/en/documentation/quickstart/*

‡ *http://oreilly.com/catalog/9780596516178/*

§ *http://www.tcl.tk/*

¶ *http://www.basistech.com/unicode/*

functions include code conversion between major legacy encodings and Unicode encodings, character classification (identification of a character), and character property conversion (such as half- to full-width katakana conversion). Basis Technology also offers a general-purpose code conversion utility, called *Uniconv*, built using this library.

Another well-made and well-established globalization library that deserves exploration and strong consideration is ICU (*International Components for Unicode*), which is portable across many platforms, including Mac OS X.* ICU includes superb support for Unicode and works well with C/C++ and Java.

Code Conversion Algorithms

It is very important to understand that only the encoding methods for the national character sets are mutually compatible, and work quite well for round-trip conversion.† The vendor-defined character sets often include characters that do not map to anything meaningful in the national character set standards. When dealing with the Japanese ISO-2022-JP, Shift-JIS, and EUC-JP encodings, for example, algorithms are used to perform code conversion—this involves mathematical operations that are applied equally to every character represented under an encoding method. This is known as *algorithmic conversion*.

However, dealing with Unicode encoding forms such as UTF-8, UTF-16, and UTF-32, and when mapping from one locale to another, requires the use of mapping tables.‡ (Mapping tables are necessary when no code conversion algorithm exists, which usually means that character ordering is different.) This is known as *table-driven*, *tabular*, or *hardcoded conversion*. Table-driven conversion deals with every character on a case-by-case basis. Table 9-5 provides examples that illustrate algorithmic versus table-driven conversion, specifically the first four kanji in JIS X 0208:1997 (for brevity, all code points are expressed in hexadecimal notation).

Table 9-5. Algorithmic versus table-driven conversion

Character	Algorithmic (ISO-2022-JP to EUC-JP)[a]	Table-driven (Unicode to EUC-JP)
亜	<30 21> becomes <B0 A1>	U+4E9C maps to <B0 A1>
唖	<30 22> becomes <B0 A2>	U+5516 maps to <B0 A2>
娃	<30 23> becomes <B0 A3>	U+5A03 maps to <B0 A3>
阿	<30 24> becomes <B0 A4>	U+963F maps to <B0 A4>

a. The algorithm used here is simply "add 0x80 to each byte."

* *http://www.icu-project.org/*

† The only possible exceptions lie in user-defined regions, which do not exist in all encodings of a given locale, and the JIS X 0212-1990 character set, which is not supported by Shift-JIS encoding.

‡ The conversion between Unicode and ASCII/ISO 8859-1:1998, as one exception, is algorithmic.

Figure 9-1 illustrates the difference between algorithmic and table-driven conversion, using the information presented in Table 9-5. Note how algorithmic conversion alters every character in the same way—they are in the same relative position in the new encoding. However, table-driven conversion introduces apparent randomness—each character code is treated as a special case.

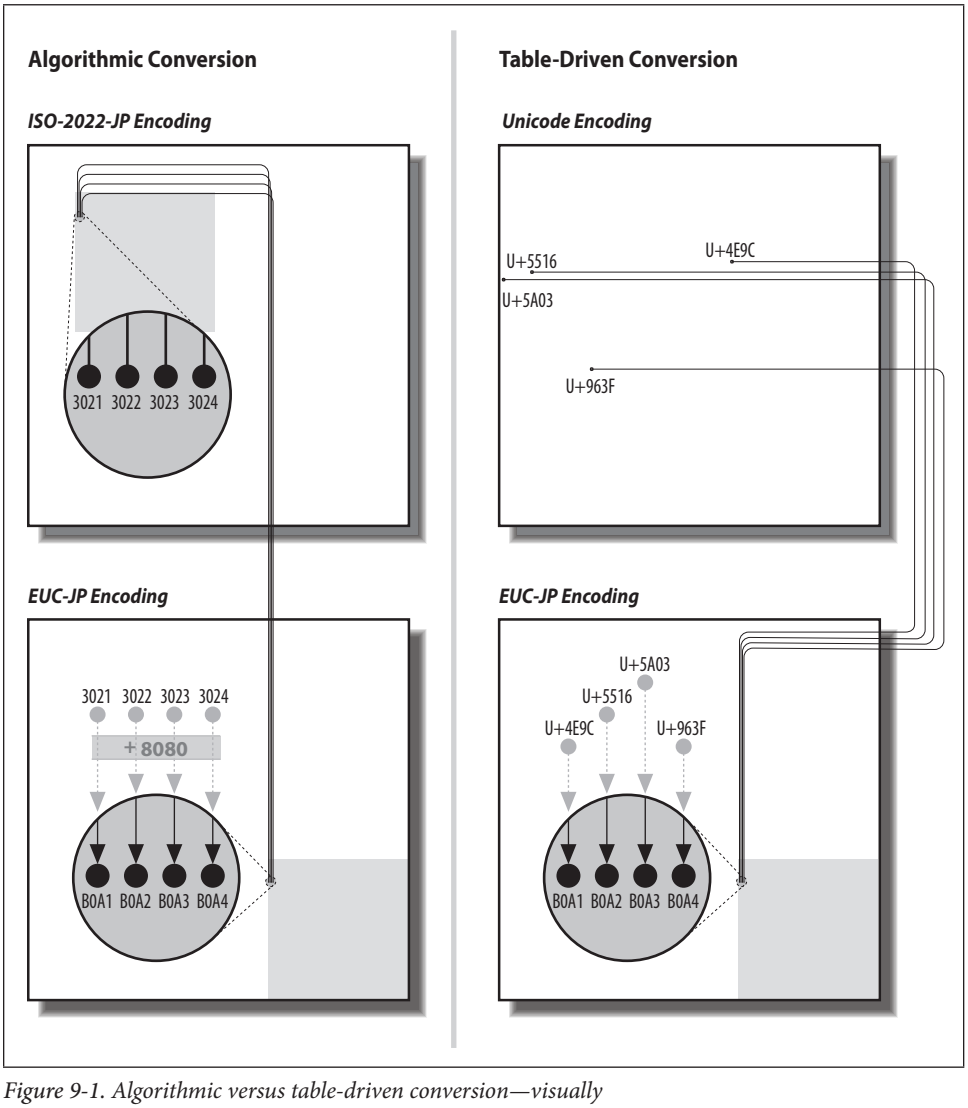


Figure 9-1. Algorithmic versus table-driven conversion—visually

One advantage of converting between legacy encodings and Unicode is that the redundancy in Unicode, in the form of its CJK Compatibility Ideographs, allows for round-trip (that is, one-to-one) conversion. In fact, this is a fundamental design feature of Unicode.

However, when dealing with conversion between character sets of different locales, such as between GB 2312-80 and Big Five, the relationship is not always one-to-one, thus round-trip conversion is not always possible. For this reason, and given the extent to which Unicode is supported in today's OSes and applications, it makes sense to keep your data in Unicode, unless you have a very good reason not to do so.

The code conversion techniques in this section cover Unicode, along with two CJKV-specific encoding methods: ISO-2022 and EUC. The Japanese-specific Shift-JIS encoding method is also covered, as is conversion to and from Row-Cell notation. These techniques can be easily applied to any CJKV locale as long as their character sets are based on these encoding methods or notations.

It is best to treat the vendor encoding methods, as described in Appendix F, as exceptional cases. It is also best to avoid using such encoding methods and character sets if your software requires the maximum amount of flexibility and information interchange—this is a portability issue.

The following sections contain more detailed information about dealing with the conversion of these and other encoding methods. Two of the conversion algorithms require the use of somewhat complex functions for maximum efficiency (at least, when writing code in a language other than Java). The other types of conversion make use of these functions, or perform simple assignments.

Conversion Between UTF-8, UTF-16, and UTF-32

The Unicode Consortium supplies a series of C routines that perform code conversion between the UTF-8, UTF-16, and UTF-32 encoding forms.* It is important to note that conversion between these encoding forms is purely algorithmic. Mapping tables are not necessary. As described previously, mapping tables are necessary only when converting between these encoding forms and non-Unicode encodings.

In my own work in establishing a series of UTF-8, UTF-16, and UTF-32 code conversion routines, written in Perl, I have found that the UTF-32 encoding form serves as an excellent middle ground, meaning that conversion from UTF-8 to UTF-16, and vice versa, is performed by first converting them to the UTF-32 encoding form.

As already noted in Chapter 4, regardless of the encoding, whether it is UTF-8, UTF-16, or UTF-32, any Unicode character that is outside the BMP is represented by four effective bytes. For UTF-8, it means four code units, each of which is one byte. For UTF-16, it means two code units, each of which is 16 bits or 2 bytes. And, for UTF-32, it means one 32-bit code unit, which is equivalent to 4 bytes.

The primary pitfall when dealing with conversion between the UTF-8, UTF-16, and UTF-32 encoding forms is the proper handling of the Surrogates, which is the extension

* *http://unicode.org/Public/PROGRAMS/CVTUTF/*

mechanism for the UTF-16 encoding form. The High and Low Surrogates should never be handled individually when converting to and from the UTF-8 and UTF-32 encoding forms, and instead should be used together to map directly code points in Planes 1 through 16. Table 9-6 illustrates correct and incorrect interpretation of UTF-16 High and Low Surrogates, using the first characters of Planes 1 and 2, along with the last code point of Plane 16 (which is classified as a noncharacter) as examples.

Table 9-6. UTF-16 Surrogate conversion examples

Scalar value	UTF-16BE	UTF-8	UTF-32BE	UTF-8 (Incorrect)	UTF-32BE (Incorrect)
U+10000	D8 00 DC 00	F0 90 80 80	00 01 00 00	ED A0 80 ED B0 80	00 00 D8 00 00 00 DC 00
U+20000	D8 40 DC 00	F0 A0 80 80	00 02 00 00	ED A1 80 ED B0 80	00 00 D8 40 00 00 DC 00
U+10FFFF	DB FF DF FF	F4 8F BF BF	00 10 FF FF	ED AF BF ED BF BF	00 00 DB FF 00 00 DF FF

Note how the incorrect handling of the UTF-16 High and Low Surrogates is directly related to the handling of each code unit as a separate character rather than as a single unit. If your software generates UTF-8 and UTF-32 sequences that look like the UTF-8 (Incorrect) or UTF-32BE (Incorrect) columns in Table 9-6, then something is clearly wrong. The fact that the result becomes the wrong number of code units is a good indicator. In the case of UTF-8, incorrect interpretation results in six code units, instead of the expected four. And for UTF-32, it results in two code units, instead of the expected single code unit.

Conversion Between ISO-2022 and EUC

EUC encoding is what I often refer to as escape sequence– or shift-less ISO-2022 encoding with the eighth bit set. Some email transport systems (and news readers) strip the eighth bits from email messages—if one sends an EUC-encoded file through such mailers, the file becomes damaged (and unreadable) because it is transformed into escape sequence– or shift-less ISO-2022. This should indicate to you that conversion between ISO-2022 and EUC is a simple matter of subtracting or adding 128 (0x80), applied to both bytes—this has the effect of toggling the eighth bit.* Although the conversion of bytes is a simple process, one must properly detect and insert designator sequences, escape sequences, or shift characters for ISO-2022–encoded text.

First, we assume two variables, one for holding each of the two bytes to be converted:

```
int p1,p2;
```

I am not showing how you go about assigning the initial values to these variables—I assume that they already contain appropriate values.

* Or, if you prefer to treat both bytes as a single unit, you subtract or add 32,896 (<80 80>).

Converting ISO-2022 to EUC is a simple matter of using the following two assignment statements in C:

```
p1 += 128;
p2 += 128;
```

These assignment statements have an effect of adding 128 (0x80) to the current values of the variables p1 and p2. These statements could also have been written as follows:

```
p1 = p1 + 128;
p2 = p2 + 128;
```

Both styles perform the same task. C (and other programming languages) has a shorthand method for doing these sort of variable assignments. There are even shorthand methods for turning the eighth bit on or off.

Next, converting EUC to ISO-2022 requires the following two statements (or their equivalent):

```
p1 -= 128;
p2 -= 128;
```

These assignment statements have an effect of subtracting 128 (0x80) from the current value of the variables p1 and p2. That's really all there is to do.

One difficult issue to contend with is how to handle half-width katakana, which can be represented in EUC-JP encoding using code set 2, but have no official representation in ISO-2022-JP encoding. I suggest that they be converted into their full-width counterparts (see the section later in this chapter entitled "Half- to Full-Width Katakana Conversion—in Java").

Conversion Between ISO-2022 and Row-Cell

Conversion from ISO-2022 to Row-Cell is a matter of subtracting 32 (0x20) from each of the ISO-2022 bytes.* Similarly, conversion from Row-Cell to ISO-2022 is a matter of adding 32 (0x20) to each of the Row-Cell bytes (or, more properly, adding 32 (0x20) to the Row and 32 (0x20) to the Cell). This may not be very useful for converting Japanese text since Row-Cell is not typically used internally to represent characters on computer systems—there are exceptions, of course. It may often be useful to determine the Row-Cell value for CJKV characters, such as for indexing into a dictionary whose entries are listed by Row-Cell code.

To convert from ISO-2022 to Row-Cell, use the following assignment statements:

```
p1 -= 32;
p2 -= 32;
```

* Or, if you prefer to treat both bytes as a single unit, use 8,224 (<20 20>) instead.

The reverse conversion (Row-Cell to ISO-2022) uses the following assignment statements:

```
p1 += 32;
p2 += 32;
```

ISO-2022 and Row-Cell are related more closely than you would think. They are different only in the fact that ISO-2022 is the encoded value, which does not happen to begin at value 1, and that Row-Cell represents an encoding-independent way of indexing characters within the 94×94 character matrix. The only software system I know of that processes CJKV characters by Row-Cell values is the Japanese version of TEX, a typesetting language. For other systems it is simply not very efficient or practical to process Row-Cell codes internally.

Conversion Between ISO-2022-JP and Shift-JIS

The ability to convert between ISO-2022-JP and Shift-JIS encodings is fundamental for most software that is designed to support Japanese. Half-width katakana, which can be represented in Shift-JIS encoding, have no official representation in ISO-2022-JP encoding. As with conversion from EUC-JP, I suggest that these characters be converted into their full-width counterparts.

ISO-2022-JP to Shift-JIS conversion

Conversion from ISO-2022-JP to Shift-JIS requires the use of the following conversion algorithm, given in C code, or its equivalent. A call to this function must pass variables for both bytes to be converted, and pointers are used to return the values back to the calling statement. Following is the algorithm, represented as working C code:

```
   void jis2sjis(int *p1, int *p2)
   {
1    unsigned char c1 = *p1;
2    unsigned char c2 = *p2;
3    int rowOffset = c1 < 95 ? 112 : 176;
4    int cellOffset = c1 % 2 ? (c2 > 95 ? 32 : 31) : 126;
5    *p1 = ((c1 + 1) >> 1) + rowOffset;
6    *p2 += cellOffset;
   }
```

Assuming that variables have been defined already, a typical call to this function may take the following form:

```
jis2sjis(&p1,&p2);
```

Table 9-7 provides a step-by-step listing of the conversion process used in the preceding function. The target character is 漢 (*kan*; the "kan" from the Japanese word 漢字 *kanji*). Its ISO-2022-JP code is 52-65, and the Shift-JIS code is 138-191. Changes are highlighted.

Table 9-7. ISO-2022-JP to Shift-JIS conversion example

Variable	Line 1	Line 2	Line 3	Line 4	Line 5	Line 6
`c1`	52	52	52	52	52	52
`c2`	…	65	65	65	65	65
`rowOffset`	…	…	112	112	112	112
`cellOffset`	…	…	…	126	126	126
`*p1`	52	52	52	52	138	138
`*p2`	65	65	65	65	65	191

Now for some explanation by line number:

1. The variable `c1` is assigned the value of the object to which `*p1` points. In this case, it is the value of the first byte, specifically 52.
2. The variable `c2` is assigned the value of the object to which `*p2` points. In this case, it is the value of the second byte, specifically 65.
3. The variable `rowOffset` is initialized by testing a condition. This condition is whether the value of the variable `c1` is less than 95. If its value is less than 95, `rowOffset` is initialized to 112. Otherwise, it is initialized to 176. Because `c1` is less than 95 in the example, `rowOffset` is initialized to 112.
4. The variable `cellOffset` is initialized by testing one or more conditions. The first condition is whether the variable `c1` is odd. If this first condition is not met, `cellOffset` is initialized to 126. If this first condition is met, another condition is tested. If the variable `c2` is greater than 95, `cellOffset` will be initialized to 32, and to 31 otherwise. Because `c1` is not odd in the example, `cellOffset` is initialized to 126.
5. The object to which `*p1` points is assigned the value of adding 1 to `c1` (52 + 1 = 53), performing a right-shift, which is the same as dividing a number by two and throwing away the remainder (53 ÷ 2 = 26), and then finally adding `rowOffset` (26 + 112 = 138).
6. The object to which `*p2` points is assigned the value of adding `cellOffset` to itself (126 + 65 = 191).

Besides simple code conversion, it is also very important to be able to detect the escape sequences used in ISO-2022-JP encoding. Escape sequences signal the software when to change modes. Good software should also keep track of the current *n*-byte-per-character mode so that redundant escape sequences can be ignored (and absorbed). Remember that Shift-JIS encoding does not use escape sequences, so you will have to make sure that they are not written to the resulting output file.

Shift-JIS to ISO-2022-JP conversion

Conversion from Shift-JIS to ISO-2022-JP is not as simple as just reversing the preceding algorithm, but requires the use of the following dedicated conversion algorithm, given again in C code, or its equivalent. A call to this function must pass variables for both bytes to be converted, and pointers are used to return the values back to the calling statement. The following is the C code:

```
  void sjis2jis(int *p1, int *p2)
  {
1   unsigned char c1 = *p1;
2   unsigned char c2 = *p2;
3   int adjust = c2 < 159;
4   int rowOffset = c1 < 160 ? 112 : 176;
5   int cellOffset = adjust ? (c2 > 127 ? 32 : 31) : 126;
6   *p1 = ((c1 - rowOffset) << 1) - adjust;
7   *p2 -= cellOffset;
  }
```

Assuming that variables have been defined already, a typical call to this function may take the following form:

```
sjis2jis(&p1,&p2);
```

Table 9-8 provides a step-by-step table of the conversion process used in the preceding function. The target character is 漢 again. Its Shift-JIS code is 138-191, and its ISO-2022-JP code is 52-65. Changes are highlighted.

Table 9-8. Shift-JIS to ISO-2022-JP conversion example

Variable	Line 1	Line 2	Line 3	Line 4	Line 5	Line 6	Line 7
c1	138	138	138	138	138	138	138
c2	…	191	191	191	191	191	191
adjust	…	…	0	0	0	0	0
rowOffset	…	…	…	112	112	112	112
cellOffset	…	…	…	…	126	126	126
*p1	138	138	138	138	138	52	52
*p2	191	191	191	191	191	191	65

Now for some explanation by line number:

1. The variable c1 is assigned the value of the object to which *p1 points. In this case, it is the value of the first byte, specifically 138.
2. The variable c2 is assigned the value of the object to which *p2 points. In this case, it is the value of the second byte, specifically 191.
3. The variable adjust is assigned the value 0 or 1, depending on the result of a test. This test checks whether the value of the variable c2 is less than 159. If the result of this

test results in true, then the variable `adjust` is assigned the value 1; otherwise, it is assigned the value `0`. In this example, the variable `c2` is 191, which is not less than 159, so the variable `adjust` is assigned the value `0`.

4. The variable `rowOffset` is initialized by testing a condition. This condition is whether the value of the variable `c1` is less than `160`. If its value is less than `160`, `rowOffset` is initialized to 112. Otherwise, it is initialized to 176. Because `c1` is less than `160` in the example, `rowOffset` is initialized to 112.
5. The variable `cellOffset` is initialized by testing one or more conditions. The first condition is whether the variable `adjust` is equal to 1. If this first condition is not met, the variable `cellOffset` is initialized to 126. If this first condition is met, another condition is tested. If the variable `c2` is greater than 127, `cellOffset` will be initialized to 32, and to 31 otherwise. Because `c1` is not equal to 1 in the example, `cellOffset` is initialized to 126.
6. The object to which `*p1` points is assigned the value of subtracting `rowOffset` from *c1* (138 − 112 = 26), performing a left-shift, which is equivalent to multiplying a number by two (26 × 2 = 52), then finally subtracting `adjust` (52 − `0` = 52).
7. The object to which `*p2` points is assigned the value of subtracting `cellOffset` from itself (191 − 126 = 65).

In addition, it is also very important to be able to properly insert escape sequences into ISO-2022-JP–encoded text streams. Be sure that redundant escape sequences are not written.

Conversion Between EUC-JP and Shift-JIS

There is no need to elaborately explain how one goes about converting Shift-JIS to EUC-JP here. What you have already learned is sufficient. You simply need to use ISO-2022-JP encoding as the middle ground for JIS X 0208:1997 characters.* The only exceptional handling that is required is for those pesky half-width katakana, which require a one-byte representation in Shift-JIS, but a two-byte representation in EUC-JP. The relationship between them is useful to know—the second byte of EUC-JP–encoded half-width katakana is the same as the Shift-JIS equivalent. Converting Shift-JIS half-width katakana to EUC-JP encoding is a matter of prefixing a byte with the value of 142 (`0x8E`, also known as SS2) to each half-width katakana byte. Likewise, converting EUC-JP–encoded half-width katakana to Shift-JIS is a simple matter of removing the first byte, specifically 142 (`0x8E`). Note that escape-sequence handling is not required for either encoding.

* ISO-2022-JP encoding cannot be used as the middle ground for half-width katakana, because the official definition of ISO-2022-JP encoding explicitly excludes half-width katakana.

Other Code Conversion Types

What you have learned already is enough to guide you through additional code conversion types, so we haven't covered every type of code conversion. Table 9-9 details how to implement other conversions, and the integer values that are provided are in decimal notation.

Table 9-9. Code conversion matrix

		ISO-2022	Shift-JIS	EUC	Row-Cell
From	**ISO-2022**	n/a	*jis2sjis*	+128	−32
	Shift-JIS	*sjis2jis*	n/a	*sjis2jis* then +128	*sjis2jis* then −32
	EUC	−128	−128 then *jis2sjis*	n/a	−160
	Row-Cell	+32	+32 then *jis2sjis*	+160	n/a

The string *jis2sjis* refers to the ISO-2022-JP to Shift-JIS conversion algorithm; likewise, *sjis2jis* refers to the Shift-JIS to ISO-2022-JP conversion algorithm—both were described in detail earlier in this chapter. The numbers prefixed with either + (plus) or – (minus) mean that you must add or subtract those amounts, in decimal, from both bytes. Also note in the table how ISO-2022 is used as the middle ground for code conversion when Shift-JIS encoding is involved—this does not mean such implementation is absolutely necessary, but I find it efficient to do so.

Java Programming Examples

The following sections illustrate, with example code, how trivial CJKV code conversion can be when using the Java programming language. Specifically, code conversion and text stream handling techniques are provided.

Java Code Conversion

The Java programming language, beginning with version 1.1, provides extremely useful built-in code conversion facilities that allow the programmer to easily convert between legacy (that is, non-Unicode) encodings and Unicode. This was a significant development because the proper handling of multiple encodings has always been a tricky issue for programmers developing software that manipulates CJKV text or data.

Starting with version 1.4, Java now provides NIO (*New Input/Output*) facilities in the *java.nio* package, specifically in the *java.nio.charset* package, which further enhances and simplifies the handling of code conversion and related tasks.* The *CharsetDecoder*

* *http://java.sun.com/javase/6/docs/api/*

(*java.nio.charset.CharsetDecoder*) and *CharsetEncoder* (*java.nio.charset.CharsetEncoder*) classes perform this type of code conversion.

The Java code examples in this section demonstrate how to take advantage of Java's built-in code conversion facilities for handling small chunks of data, such as single characters or short strings of characters. You will notice that these examples are short and concise, which clearly illustrates how Java trivializes code conversion.

Non-Unicode to Unicode conversion—CharsetDecoder

One of the ways in which non-Unicode data can be converted to Unicode using Java is to simply convert byte arrays or buffers into character buffers using the *CharsetDecoder* class. The following is sample code that first creates a byte array (containing the two-byte value <B0 EC> for the EUC-JP–encoded kanji 一 *ichi*, meaning "one") then converts it into Unicode:

```
Charset eucjpCharset = Charset.forName("EUC-JP");
CharsetDecoder decoder = eucjpCharset.newDecoder();
byte eucjpData[] = {176, 236};
ByteBuffer eucjpBytes = ByteBuffer.wrap(eucjpData);
CharBuffer unicodeChar = null;
unicodeChar = decoder.decode(eucjpBytes);
```

Simply specifying a "charset" value, such as *EUC-JP* for EUC-JP encoding, as the argument to the *Charset* object creation method is sufficient to invoke code conversion on the *ByteBuffer* object *eucjpBytes*. It is really that simple.

The *CharBuffer* object *unicodeChar* ends up containing the Unicode scalar value U+4E00 (expressed as \u4E00 in Java notation).

Unicode to non-Unicode conversion—CharsetEncoder

Because the Java programming language processes Unicode internally, it may become necessary to convert an internal Unicode representation into a legacy encoding when dealing with non-Unicode environment. The *CharsetEncoder* class is used for this purpose. Following is an example that performs the reverse of what the previous example did, specifically that it converts the *CharBuffer* object *unicodeChar*, which contains the character U+4E00, into its EUC-JP–encoded equivalent, resulting in a new *ByteBuffer* object *neweucjpBytes* that contains the values 0xB0 and 0xEC:

```
Charset eucjpCharset = Charset.forName("EUC-JP");
CharsetEncoder encoder = eucjpCharset.newEncoder();
ByteBuffer neweucjpBytes = null;
neweucjpBytes = encoder.encode(unicodeChar);
```

It is really amazing to realize how trivial code conversion can become when using Java's built-in facilities, as the preceding two lines of Java code demonstrate.

Java Text Stream Handling

This section provides example code for handling text streams in Java through the underlying use of the standard (but private) *ByteToCharConverter* and *CharToByteConverter* classes found in the *java.io* package. These algorithms fall into two basic types:

- Non-Unicode to Unicode (considered a text "import")
- Unicode to non-Unicode (considered a text "export")

None of these text stream conversion types require any special handling, such as the proper handling of designator sequences, escape sequences, or shifting characters as used in ISO-2022 encoding.

Before Java, keeping track of the current *n*-byte-per-character mode and current character set was very important when dealing with ISO-2022–encoded data. Java performs the following tasks for you:

- Recognize and remove redundant escape sequences
- Ensure that lines terminate in one-byte mode
- Ensure that the file terminates in one-byte mode

This list may not seem very important to you now, but as you begin to encounter ISO-2022–encoded files with redundant or missing escape sequences, you will soon appreciate it.

Non-Unicode to Unicode conversion—import

Converting a text stream from a non-Unicode encoding to Unicode is greatly simplified in Java through its text stream classes. Non-Unicode encodings are treated as raw data by Java. The following three lines of Java open a file called *input* and proceed to convert its contents to Unicode as it is being read:

```
File i = new File("input");
FileInputStream tmpin = new FileInputStream(i);
BufferedReader in = new BufferedReader(new InputStreamReader(tmpin,"Shift_JIS"));
```

Once the *BufferedReader* object *in* is established, as accomplished here, data can be read using the *readLine()* method. The following is an example use of this method:

```
inputStr = in.readLine();
```

Note the use of *Shift_JIS* as the second argument to the *InputStreamReader()* method. This parameter invokes the built-in conversion to Unicode assuming Shift-JIS encoding as input.

As you can see, Java takes away all of the pain associated with importing non-Unicode data. This is actually a big win for programmers because what has traditionally been a formidable task in developing CJKV-capable software is now trivialized.

Unicode to non-Unicode conversion—export

Here is the Java code for converting a Unicode-encoded text stream into Shift-JIS encoding. Notice that this function does not return any information to the calling statement—it merely reads in a text stream and outputs to another text stream.

```
File o = new File("output.sjs");
FileOutputStream tmpout = new FileOutputStream(o);
BufferedWriter out = new BufferedWriter(new OutputStreamWriter(tmpout,"Shift_JIS"));
```

After the *BufferedWriter* object *out* is established, Unicode data that is subsequently output is automatically converted to Shift-JIS encoding.

```
out.println("\u6CB3\u8C5A");
out.close();
```

The two Unicode characters U+6CB3 (河) and U+8C5A (豚) that are fed to the *println()* method become Shift-JIS <89 CD> and <93 D8> in the output file called *output.sjs*. Creating a UTF-8–encoded output file is accomplished in the same way, but the *UTF-8* charset designator should be used instead of *Shift_JIS*.

It is also possible to output directly in Unicode, demonstrated as follows:

```
PrintWriter out = new PrintWriter (
  new BufferedWriter (
    new OutputStreamWriter (
      new FileOutputStream("output.ucs"), "UTF-16BE"
    )
  )
);
```

We can then output the same Unicode characters, without any code conversion applied, as follows:

```
out.println("\u6CB3\u8C5A");
out.close();
```

This time, the output is exactly U+6CB3 (河) and U+8C5A (豚).

Java Charset Designators

In order to take advantage of Java's built-in code conversion facilities, you need to be aware of the valid charset designators in order to properly invoke them. While Java includes the concept of "preferred" charset designators for each meaningful character set and encoding combination, it also supports an aliasing mechanism to support alternate charset names.

Table 9-10 provides a partial listing of Java's charset designators, along with the encodings and character sets that they support. What is provided are the Java NIO charset designators, specifically for the *java.nio* package. The *java.io* and *java.lang* packages may use slightly different charset designators. Alternate charset designators have been explicitly excluded from this table, as my way of to discouraging their use. These charset designators can be used to invoke the built-in code conversion routines. Also, those that are required to be supported on every implementation of the Java platform have been highlighted.

Table 9-10. Java NIO charset designators—examples

Charset designator	Encoding	Character sets
US-ASCII	ASCII	ASCII
ISO-8859-1	ISO 8859-1:1998	ISO 8859-1:1998
UTF-8	UTF-8	Unicode
UTF-16	UTF-16	Unicode
UTF-16BE	UTF-16BE	Unicode
UTF-16LE	UTF-16LE	Unicode
UTF-32	UTF-32	Unicode
UTF-32BE	UTF-32BE	Unicode
UTF-32LE	UTF-32LE	Unicode
ISO-2022-CN	ISO-2022-CN-EXT	ASCII, CNS 11643-1992
ISO-2022-JP	ISO-2022-JP	ASCII/JIS-Roman, half-width katakana, JIS X 0208:1997, JIS X 0212-1990
ISO-2022-KR	ISO-2022-KR	ASCII/KS-Roman, KS X 1001:2004
GB2312	EUC-CN	ASCII/GB-Roman, GB 2312-80
EUC-JP	EUC-JP	ASCII/JIS-Roman, half-width katakana, JIS X 0208:1997, JIS X 0212-1990
EUC-KR	EUC-KR	ASCII/KS-Roman, KS X 1001:2004
x-EUC-TW	EUC-TW	ASCII, CNS 11643-1992
GBK	GBK	ASCII, GBK
GB18030	GB 18030	ASCII, GB 18030
Big5	Big Five	ASCII, Big Five
Big5-HKSCS	Big Five	Hong Kong SCS
Shift_JIS	Shift-JIS	ASCII/JIS-Roman, half-width katakana, JIS X 0208:1997

A much more complete and up-to-date listing of Java charset designators is available online—be sure to consult the latest Java programming language specification to ensure that you are using the correct charset designators. This will best guarantee that your program will function in all possible environments.*

Miscellaneous Algorithms

This section covers three miscellaneous algorithms that are useful, but are not directly associated with either code conversion or text stream handling, as covered in the two previous sections.

* *http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html*

The first algorithm is for the automatic detection of the input file's encoding. Some software requires that you specify the encoding method used by the input file: many people who use Japanese code conversion utilities may not be familiar with the various Japanese encoding methods. If you do not know, all you can do is guess. The Japanese code detection algorithm examines the input file in order to determine the encoding method. This usually makes it unnecessary to specify the input file's encoding method. However, there are times when the input file's encoding may be ambiguous—the Shift-JIS and EUC-JP encoding ranges overlap considerably, for example.

The second algorithm converts half-width katakana into their full-width counterparts. Some environments do not provide half-width katakana support, so this algorithm converts these characters into their full-width versions, which are more commonly supported. This algorithm is also quite useful as a filter for outgoing email transmissions in order to ensure that information interchange is maintained on the receiving end.

The third and final algorithm repairs damaged ISO-2022-JP–encoded files; that is, files that had their escape sequences stripped out by unfriendly email or news reading software. I occasionally received email in this damaged format, and spent a lot of time reinserting those lost escape characters. This algorithm is simply a way to automate this repair process.

The C functions described in this section use the following C #define statements:

```
#define NEW         1
#define OLD         2
#define NEC         3
#define EUC         4
#define SJIS        5
#define EUCORSJIS   6
#define ASCII       7
#define SS2         142
#define ESC         27
```

Japanese Code Detection

This algorithm is useful for automatically detecting the Japanese encoding used in a Japanese text file. This is helpful when you receive Japanese text files with various encodings: it is not always obvious what encoding a given text file uses, so it is easier to let the software decide for you.

This C function requires only an input stream as a parameter, but returns a value to the calling statement indicating what Japanese code, if any, was found. This value can specify either the Japanese encoding detected or that none was detected (or was ambiguous). If Japanese encoding is found, possible values include JIS C 6226-1978 (also called Old-JIS), JIS X 0208-1983 (also called New-JIS), NEC Kanji (also called NEC-JIS), EUC-JP (packed format), and Shift-JIS. It also returns special values if no Japanese encoding is detected, or if the Japanese encoding is ambiguous (Shift-JIS and EUC-JP overlap considerably, and it is possible to encounter text streams that may be ambiguous). I use this algorithm in two

of the tools described at the end of this chapter, specifically *JConv* and *JCode*. A typical call to this function takes the following form:

```
DetectCodeType(in);
```

Following is a C function for detecting the Japanese encoding of an input stream. Most of the statements check encoded value ranges. The results of these checks are then used to determine whether a particular encoding has been detected in the stream. ISO-2022 encodings are easily found by the occurrence of escape characters, along with other characters that constitute a valid two-byte character escape sequence.

```
int DetectCodeType(FILE *in)
{
  int c = 0;
  int whatcode = ASCII; /* The detected code, set to ASCII. */
  while ((whatcode == EUCORSJIS || whatcode == ASCII) && c != EOF) {
    if ((c = fgetc(in)) != EOF) { /* Read one byte until EOF. */
      if (c == ESC) { /* Maybe ISO-2022-JP encoding. */
        c = fgetc(in);
        if (c == '$') { /* Maybe two-byte escape sequence. */
          c = fgetc(in);
          if (c == 'B')
            whatcode = NEW; /* JIS X 0208-1983 detected. */
          else if (c == '@')
            whatcode = OLD; /* JIS C 6226-1978 detected. */
        }
        else if (c == 'K')
          whatcode = NEC; /* NEC Japanese detected. */
      }
      else if ((c >= 129 && c <= 141) || (c >= 143 && c <= 159))
        whatcode = SJIS;
      else if (c == SS2) { /* Maybe EUC-JP half-width katakana. */
        c = fgetc(in);
        if ((c >= 64 && c <= 126)||(c >= 128 && c <= 160)||(c >= 224 && c <= 252))
          whatcode = SJIS; /* Shift-JIS detected. */
        else if (c >= 161 && c <= 223)
          whatcode = EUCORSJIS; /* Ambiguous (Shift-JIS or EUC-JP). */
      }
      else if (c >= 161 && c <= 223) {
        c = fgetc(in);
        if (c >= 240 && c <= 254)
          whatcode = EUC; /* EUC-JP detected. */
        else if (c >= 161 && c <= 223)
          whatcode = EUCORSJIS; /* Ambiguous (Shift-JIS or EUC-JP). */
        else if (c >= 224 && c <= 239) {
          whatcode = EUCORSJIS; /* Ambiguous (Shift-JIS or EUC-JP). */
          while (c >= 64 && c != EOF && whatcode == EUCORSJIS) {
            if (c >= 129) {
              if (c <= 141 || (c >= 143 && c <= 159))
                whatcode = SJIS; /* Shift-JIS detected. */
              else if (c >= 253 && c <= 254)
                whatcode = EUC; /* EUC-JP detected. */
            }
            c = fgetc(in);
```

```
          }
        }
        else if (c <= 159)
          whatcode = SJIS; /* Shift-JIS detected. */
      }
      else if (c >= 240 && c <= 254)
        whatcode = EUC; /* EUC-JP detected. */
      else if (c >= 224 && c <= 239) {
        c = fgetc(in); /* Read next byte to c. */
        if ((c >= 64 && c <= 126) || (c >= 128 && c <= 160))
          whatcode = SJIS; /* Shift-JIS detected. */
        else if (c >= 253 && c <= 254)
          whatcode = EUC; /* EUC-JP detected. */
        else if (c >= 161 && c <= 252)
          whatcode = EUCORSJIS; /* Ambiguous (Shift-JIS or EUC-JP). */
      }
    }
  }
  return whatcode; /* Return the detected code. */
}
```

Appendix C provides Perl code for a much more flexible way to automatically detect the encoding of CJKV text files, not only those for Japanese. That Perl code shows how powerful regular expressions can be when used in specific contexts.

Half- to Full-Width Katakana Conversion—in Java

It sometimes is necessary to convert half-width katakana to their full-width counterparts. This is most useful as a filter to ensure that no half-width katakana characters are included within email messages. It is also useful when you need to move files from one platform to another and the new platform does not support half-width katakana characters. Example usage of this Java method is as follows:

```
String half = "\uFF76\uFF9E";
String full = KatakanaFilter.halfToFullWidthKatakana(half);
```

There is no simple conversion algorithm that you can use to accomplish this task. In fact, such conversion requires a mapping table between half- and full-width katakana (table-driven conversion), as well as special handling to accommodate *dakuten* and *handakuten*, the marks that modify kana characters. You see, these marks are encoded as separate characters in the half-width katakana character set, but in the case of full-width katakana, they are integrated with katakana characters within the same encoded character.

The following Java class defines a method called *halfToFullWidthKatakana()* that represents the algorithm for converting half-width katakana to their full-width counterparts, and includes proper handling for dakuten and handakuten marks.

```
public class KatakanaFilter {

  // Zero-base table for mapping half-width katakana to full-width
  private final static char FWKatakana[] = {
    '\u3002','\u300C','\u300D','\u3001','\u30FB', // U+FF61 - U+FF65
```

```
  '\u30F2','\u30A1','\u30A3','\u30A5','\u30A7', // U+FF66 - U+FF6A
  '\u30A9','\u30E3','\u30E5','\u30E7','\u30C3', // U+FF6B - U+FF6F
  '\u30FC','\u30A2','\u30A4','\u30A6','\u30A8', // U+FF70 - U+FF74
  '\u30AA','\u30AB','\u30AD','\u30AF','\u30B1', // U+FF75 - U+FF79
  '\u30B3','\u30B5','\u30B7','\u30B9','\u30BB', // U+FF7A - U+FF7E
  '\u30BD','\u30BF','\u30C1','\u30C4','\u30C6', // U+FF7F - U+FF83
  '\u30C8','\u30CA','\u30CB','\u30CC','\u30CD', // U+FF84 - U+FF88
  '\u30CE','\u30CF','\u30D2','\u30D5','\u30D8', // U+FF89 - U+FF8D
  '\u30DB','\u30DE','\u30DF','\u30E0','\u30E1', // U+FF8E - U+FF92
  '\u30E2','\u30E4','\u30E6','\u30E8','\u30E9', // U+FF93 - U+FF97
  '\u30EA','\u30EB','\u30EC','\u30ED','\u30EF', // U+FF98 - U+FF9C
  '\u30F3','\u309B','\u309C'                    // U+FF9D - U+FF9F
};

// Class method for converting half-width katakana to full-width
public static String halfToFullWidthKatakana (String string_input) {
  int ixIn = 0;
  int ixOut = 0;
  int bufferLength = string_input.length();
  char[] input = string_input.toCharArray();
  char[] output = new char[bufferLength + 1];

  while (ixIn < bufferLength) {
    if (input[ixIn] >= '\uFF61' && input[ixIn] <= '\uFF9F') {
      if (ixIn + 1 >= bufferLength) {
        output[ixOut++] = FWKatakana[input[ixIn++] - '\uFF61'];
      } else {
        if (input[ixIn + 1] == '\uFF9E' || input[ixIn + 1] == '\u3099'
        || input[ixIn + 1] == '\u309B') {
          if (input[ixIn] == '\uFF73') {
            output[ixOut++] = '\u30F4';
            ixIn += 2;
          } else if (input[ixIn] >= '\uFF76' && input[ixIn] <= '\uFF84'
            || input[ixIn] >= '\uFF8A' && input[ixIn] <= '\uFF8E') {
            output[ixOut] = FWKatakana[input[ixIn] - '\uFF61'];
            output[ixOut++]++;
            ixIn += 2;
          } else {
            output[ixOut++] = FWKatakana[input[ixIn++] - '\uFF61'];
          }
        } else if (input[ixIn + 1] == '\uFF9F'
          || input[ixIn + 1] == '\u309A' || input[ixIn + 1] == '\u309C') {
          if (input[ixIn] >= '\uFF8A' && input[ixIn] <= '\uFF8E') {
            output[ixOut] = FWKatakana[input[ixIn] - '\uFF61'];
            output[ixOut++] += 2;
            ixIn += 2;
          } else {
            output[ixOut++] = FWKatakana[input[ixIn++] - '\uFF61'];
          }
        } else {
          output[ixOut++] = FWKatakana[input[ixIn++] - '\uFF61'];
        }
      }
    } else {
```

```
            output[ixOut++] = input[ixIn++];
          }
        }
        String output_string = new String(output);
        return output_string.substring(0,ixOut);
      }
    }
```

Appendix C provides a half- to full-width katakana conversion program written in Perl. It is different in that it is not based on Unicode, but rather supports EUC-JP and Shift-JIS encodings.

Encoding Repair

ISO-2022-JP–encoded files often become damaged or corrupt from software that strips out escape characters. Some programs have a tendency to filter out control characters from files; the escape character (0x1B or U+001B), which is an essential part of ISO-2022-JP encoding, is a control character. Luckily, there are ways to repair corrupted ISO-2022-JP–encoded files.

You can make a few assumptions before you proceed to repair damaged ISO-2022-JP–encoded files. The first assumption is that the text stream begins, and also ends, in one-byte mode. In addition, each line begins and ends in one-byte mode. The next assumption is that the other characters that make up a complete escape sequence are still intact. These may include such strings as $@, $B, (J, and (B. Depending on the *n*-byte-per-character mode, you need to scan for different strings.

While in one-byte mode, you need only to scan for the string $B or $@, which should signify the beginning of two-byte mode. The chances of encountering such strings of characters while in one-byte mode are quite low (but can happen!). You need to repair such string occurrences by inserting an escape character immediately before the string that determined the context for it (in this case, either $B or $@). The current mode is then shifted to two-byte.

While in two-byte mode, you need to scan for the strings (J and (B. Also, since you are in two-byte mode, you must scan two characters, and then compare them to the search strings. The two bytes that represent the strings (J and (B are within the ISO-2022-JP encoding space, but have no characters assigned to them. This means that you should never run into those strings other than when they are part of a damaged escape sequence. Like before, you need to insert an escape character right before the string that was found (in this case, either (J or (B). The current mode is then shifted to one-byte.

Other processing may be necessary if you reach the end of a line, but are still in two-byte mode. You must then insert an entire escape sequence.

The following C function represents an algorithm for automatically inserting escape sequences into a damaged ISO-2022-JP–encoded file. Note that undamaged escape sequences are also recognized by this C function. A modified version of this function is

used in one of the Japanese code-processing tools, specifically *JConv*, described at the end of this chapter.

```
void repairjis(FILE *in,FILE *out)
{
  int p1; /* First byte. */
  int p2; /* Second byte. */
  int p3; /* Third byte. */
  int shifted_in = FALSE; /* The initial one-byte mode. */
  while ((p1 = getc(in)) != EOF) {
    if (shifted_in) { /* If in two-byte mode. */
      if (p1 == ESC) {
        p2 = getc(in);
        if (p2 == '(') {
          p3 = getc(in);
          switch (p3) {
            case 'J' : /* JIS-Roman. */
            case 'B' : /* ASCII. */
            case 'H' : /* False JIS-Roman. */
              shifted_in = FALSE; /* Change to one-byte mode. */
              break;
            default :
              break;
          }
          fprintf(out,"%c%c%c",p1,p2,p3); /* Print the escape sequence. */
        }
      }
      else if (p1 == '(') { /* If p1 is (. */
        p2 = getc(in);
        switch (p2) {
          case 'J' : /* JIS-Roman. */
          case 'B' : /* ASCII. */
          case 'H' : /* False JIS-Roman. */
            shifted_in = FALSE; /* Change to one-byte mode. */
            fprintf(out,"%c%c%c",ESC,p1,p2); /* Print the escape sequence. */
            break;
          default :
            fprintf(out,"%c%c",p1,p2); /* Print p1 and p2. */
            break;
        }
      }
      else {
        p2 = getc(in);
        fprintf(out,"%c%c",p1,p2); /* Print p1 and p2. */
      }
    }
    else { /* If in one-byte mode. */
      if (p1 == ESC) {
        p2 = getc(in);
        if (p2 == '$') {
          p3 = getc(in);
          switch (p3) {
            case 'B' : /* JIS X 0208-1983. */
            case '@' : /* JIS C 6226-1978. */
```

```
            shifted_in = TRUE; /* Change to two-byte mode. */
            break;
          default :
            break;
        }
        fprintf(out,"%c%c%c",p1,p2,p3); /* Print the escape sequence. */
      }
    }
    else if (p1 == '$') {
      p2 = getc(in);
      switch (p2) {
        case 'B' : /* JIS X 0208-1983. */
        case '@' : /* JIS C 6226-1978. */
          shifted_in = TRUE; /* Change to two-byte mode. */
          fprintf(out,"%c%c%c",ESC,p1,p2); /* Print the escape sequence. */
          break;
        default :
          fprintf(out,"%c%c",p1,p2); /* Print p1 and p2. */
          break;
      }
    }
    else
      fprintf(out,"%c",p1); /* Print p1. */
  }
 }
}
```

Yep, you guessed it, Appendix C provides a similar program, but written in Perl. I encourage you to compare and contrast the C and Java examples in this chapter with Perl versions in Appendix C.

Byte Versus Character Handling

Most Western encoding methods have the luxury of assuming that one byte equals one character, so inserting, deleting, and searching text becomes a simple matter of comparing one byte with another. However, this is not the case with encodings that require more than one byte to represent a single character, such as those used for representing CJKV text. Life gets much more complex! A multiple-byte character is still a character. Consider it an *atomic* unit. After all, you would gawk at Western-style software that split characters into four-bit units for some strange design reason. What I discuss next falls into what I would call text-processing behavior, because it is what one would expect from programs such as text editors, word processors, and the like.

What you should learn from this section is that a multiple-byte character should never be broken down into its component bytes. This whole discussion points out the best reason for adopting fixed 16- or 32-bit representations inside your software— you can much more safely deal with atomic units. The topics that follow are examples of areas in which many text-processing programs fail to handle multiple-byte characters properly. There are several examples of programs that fall into this category—unless a program was specifically designed to handle multiple-byte characters, it is unlikely that multiple-byte

characters are handled properly. For example, the standard (that is, unlocalized) version of Microsoft Word (for Mac OS) is one of the most popular word-processing applications ever, but at one time failed to handle two-byte characters properly.

Character Deletion

It is quite likely that you will encounter text-processing software that deletes only one byte of a two-byte character. Those that have been properly adapted to CJKV locales are able to detect whether the character in front of the insertion point is represented by two bytes, and subsequently deletes both bytes. This problem can be avoided if you remember to press the delete key twice when dealing with two-byte characters. If you are not careful, loss or corruption of data may result.

Let's take a closer look at this problem. Table 9-11 provides a sample Shift-JIS–encoded Japanese text string. The first process that will be applied is the deletion of the last character. The first example deletes the last character (consisting of two bytes), and the second deletes the last byte (more precisely, the last byte of the last character). Finally, we add another character, 典, at the insertion point. Note how the undeleted first byte left over from the second example affects the interpretation of the added character (the encoded value of this added character is highlighted).

Table 9-11. Character deletion example—Shift-JIS

		Text representation	Shift-JIS representation				
Original string		漢字辞書	漢	字	辞	書	
			8A BF	8E 9A	8E AB	8F 91	
Correct	Delete	漢字辞	漢	字	辞		
			8A BF	8E 9A	8E AB		
	Add character	漢字辞典	漢	字	辞	典	
			8A BF	8E 9A	8E AB	**93 54**	
Incorrect	Delete	漢字辞	漢	字	辞		
			8A BF	8E 9A	8E AB	8F	
	Add character	漢字辞諸T	漢	字	辞	諸	T
			8A BF	8E 9A	8E AB	8F **93**	**54**

A lack of sync occurs when the first byte of a two-byte character is left behind. Any two-byte characters that follow will be interpreted incorrectly—their first byte will be interpreted as the second byte for the previous character, and their second byte will be interpreted as a first byte.

Table 9-12 illustrates what happens with the same character string, but when EUC-JP–encoded.

Table 9-12. Character deletion example—EUC-JP

		Text representation	EUC-JP representation				
Original string		漢字辞書	漢	字	辞	書	
			B4 C1	BB FA	BC AD	BD F1	
Correct	Delete	漢字辞	漢	字	辞		
			B4 C1	BB FA	BC AD		
	Add character	漢字辞典	漢	字	辞	典	
			B4 C1	BB FA	BC AD	C5 B5	
Incorrect	Delete	漢字辞	漢	字	辞		
			B4 C1	BB FA	BC AD	BD	
	Add character	漢字辞重	漢	字	辞	重	
			B4 C1	BB FA	BC AD	BD **C5**	**B5**

This problem is fixed by keeping track of the characters at the insertion point—whether they are represented by one or more bytes. If a byte happens to be the second byte of a two-byte character, both bytes must be deleted with a single keystroke. In the case of three-byte characters (for example, characters from EUC-JP code set 3—JIS X 0212-1990 characters), three bytes must be deleted. Extreme examples include EUC-TW and GB 18030 encodings, both of which have a four-byte representation. That's a lot of bytes, meaning that a lot can go wrong if you're not careful. Also, anything outside of the BMP, when dealing with the UTF-8 encoding form, is four bytes.

Character Insertion

Inserting characters is problematic only when the insertion point—that is, the cursor—is between the two bytes that represent a two-byte character. This then splits the two-byte character and results in data loss. This section, as you may have expected, relates to cursor movement.

Let's now look at some examples of inserting characters between the two bytes of a two-byte character. The example string is 仮名漢字, and the character と is mistakenly inserted between the two bytes of the character 名—in an ideal world, it should be added between the two characters 名 and 漢. Table 9-13 provides an example that is Shift-JIS–encoded, and the byte values for the inserted character are highlighted.

Table 9-13. Character insertion example—Shift-JIS

	Text representation	Shift-JIS representation					
Original string	仮名漢字	仮	名	漢	字		
		89 BC	96 BC	8A BF	8E 9A		
Correct	仮名と漢字	仮	名	と	漢	字	
		89 BC	96 BC	82 C6	8A BF	8E 9A	
Incorrect	仮魔ニシ漢字	仮	魔	ニ	シ	漢	字
		89 BC	96 **82**	**C6**	BC	8A BF	8E 9A

Notice how the two-byte character 名 is split right down the middle, and that unexpected characters have resulted, two of which are interpreted as half-width katakana. Now you can see why incorrect character insertion must never be allowed to happen—it leads to corruption and data loss. Integrity is retained only with proper handling of two-byte characters.

Table 9-14 provides this same example, but this time EUC-JP–encoded. Notice how different characters result from incorrect insertion—the expected 名と string becomes the unexpected 未半.

Table 9-14. Character insertion example—EUC-JP

	Text representation	EUC-JP representation				
Original string	仮名漢字	仮	名	漢	字	
		B2 BE	CC BE	B4 C1	BB FA	
Correct	仮名と漢字	仮	名	と	漢	字
		B2 BE	CC BE	A4 C8	B4 C1	BB FA
Incorrect	仮未半漢字	仮	未	半	漢	字
		B2 BE	CC **A4**	**C8** BE	B4 C1	BB FA

The solution to this problem is simply to have the cursor move one or more bytes—the number of bytes to move corresponds to the number of bytes used to represent the current character.

Character Searching

The various instances of the *grep* program represent the most commonly used utilities on Unix and some other platforms—grep is short for *Global Regular Expression Print*.*

* Perhaps jokingly, one source suggests that *grep* represents the first letters of its authors' last names: *Gregior*, *Ritchie*, *Ebersole*, and *Pike*. I dunno. Maybe it's not a joke.

The grep program performs a search based on regular expressions, which is a topic that is covered in greater detail later in this chapter. The standard Unix version of grep, unfortunately, does not treat two or more bytes—that constitute a single character—as a single unit. Some versions of Unix, such as IBM's AIX, include versions of grep that recognize multiple-byte characters.

So, you may ask, what problem does this cause? Well, take, for instance, a case when you are searching for the kanji 剣 in a large Japanese file. Assuming Shift-JIS encoding, you may end up with matches in quite unexpected places. In fact, some lines for which a match is reported may not even contain the kanji 剣 because the comparison that is being performed during the searching is done on a per-byte, not per-character, basis. This means that one byte is compared to another without regard to multiple-byte characters. In the case of a search pattern that contains multiple-byte characters, the following conditions must be met:

- One or more bytes of the search string must be compared with one or more bytes in the document being searched.
- The current index into the text being searched must advance either one or more bytes depending on whether the character at that index is represented by one or more bytes. This simply means that the index must advance one character, which is not always represented by one byte.
- Because CJKV character sets are supported by multiple encodings, even in the context of Unicode, the search engine must be completely aware of the encoding used in the search string and in the document to be searched. If the encodings are different, they must be made compatible (the easiest approach would be to convert the search string into the same encoding used in the document to be searched).

If these conditions are not met, matches may sometimes be made with the second or subsequent bytes of one multiple-byte character plus the first or subsequent bytes of the next one. This, of course, produces completely undesirable results. Table 9-15 provides an example using the kanji 剣 as the search string in a Shift-JIS–encoded file. The character codes of successful matches, whether they are correct or not, have been highlighted for convenience.

Table 9-15. Character searching example—Shift-JIS

	Characters	Shift-JIS representation		
Search string	剣	剣		
		8C 95		
Correct	剣道	剣	道	
		8C 95	93 B9	
Incorrect	白血病	白	血	病
		94 92	8C **8C**	**95** 61

Note how the example of an incorrect match spans two characters, specifically the second byte of one character and the first byte of the next one. The incorrect match was made by treating every byte as a single character. Clearly, the *one-byte-equals-one-character* barrier or mind set must be overcome in order to handle CJKV text properly. This is a crucial issue for those who are writing multiple-byte–capable search engines, a topic covered later in this chapter in the section entitled "Search Engines."

Line Breaking

Many text-processing programs allow users to break long lines into shorter ones, usually by specifying a maximum number of columns per line. As you can expect, breaking a line between the bytes of a two-byte character can result in a loss of information and end up corrupting surrounding characters.

Let's look at what may happen when ISO-2022-JP, Shift-JIS, and EUC-JP strings are broken into two lines. In the example string given in Tables 9-16 through 9-18, a line break is inserted between the two bytes that represent the katakana character サ (*sa*). Note how that character is apparently lost, and how some characters after the line break become scrambled. Some Japanese telecommunications programs, such as the older ASLTelnet that I once used on Mac OS, inserted their own one-byte character escape sequences at the end of each line to ensure that no errors took place. This means when a line is broken, the software automatically inserts a one-byte character escape sequence. However, when the line is broken in this fashion, the lack of an additional two-byte character escape sequence causes the following line to be interpreted as though it were in one-byte mode.

Table 9-16 provides a sample string, along with two examples of how the sample string can be broken when it is represented as ISO-2022-JP–encoded text. The first example, specified as version 1 in the table, is when the software automatically inserts a one-byte character escape sequence at the end of the broken line, which means that the subsequent two-byte characters are interpreted as though they were one-byte characters. The second example, specified as version 2 in the table, demonstrates what may happen if the software—in this case, NinjaTerm, which ran on Mac OS—does not automatically insert one-byte character escape sequences at the ends of lines. The second byte of the split

character, specifically サ (*sa*), is now treated as the first byte for the following two-byte character, which causes a lack of sync. Also note how the following line does not start on the left margin.

Table 9-16. Examples of broken ISO-2022-JP strings

Type	String
Original	カキクケコサシスセソタチツテト
Broken version 1	カキクケコ 5%7%9%;%=%?%A%D%F%H
Broken version 2	カキクケコ 汽轡好札愁織船張謄

Table 9-17 provides a comparable Shift-JIS example. This time there is no lack of sync, because there are no escape sequences with which to contend, and because of the liberal use of seven-bit bytes in Shift-JIS encoding. The only character that is effectively lost or destroyed is the one that was split. Its first byte, because it is in the eight-bit range, becomes invisible and hence not displayed, and because the second byte—at least for this particular example, サ (*sa*)—falls into the seven-bit range, it is not treated as the first byte of a two-byte character, and displays as is, as its ASCII equivalent.

Table 9-17. Example of broken Shift-JIS string

Type	String
Original	カキクケコサシスセソタチツテト
Broken	カキクケコ Tシスセソタチツテト

Finally, Table 9-18 provides a comparable EUC-JP example. You should immediately notice the lack of sync problem here again, like we did in the ISO-2022-JP–encoded example in Table 9-16. The first byte of the split character is effectively invisible, meaning not displayed, due to the fact that it is a lone eight-bit byte, and its second byte is now treated as the first byte of a two-byte character.

Table 9-18. Example of broken EUC-JP string

Type	String
Original	カキクケコサシスセソタチツテト
Broken	カキクケコ 汽轡好札愁織船張謄

As you can see from these examples, this problem varies in intensity depending on the encoding method, and even on the software you are using—compare the two types of output you get for ISO-2022 encoding, using different applications. Some encodings require slightly more overhead than simplistically treating multiple-byte characters as an inseparable unit. For example, when dealing with ISO-2022 encoding, you must also remember to insert and perhaps even delete escape sequences.

Character Attribute Detection Using C Macros

A useful function often supported in CJKV text-processing programs—or for that matter in most text-processing systems—is the ability to determine the attributes of characters within a file. For example, it is often convenient to obtain a listing of the numbers of Chinese characters, kana, and other characters in a file. One can even break those categories down further, such as kana into katakana and hiragana, ideographs into separate levels, and so on.

The C programming language has a useful macro facility that allows programmers to specify simple commands that can be used often within a program. Macros are similar in concept to functions, but require less work, although perhaps more thought.

As an example, several C macro definitions for detecting the attributes of Japanese (JIS X 0208:1997) characters are provided. They all assume Row-Cell values as input. How you implement this depends on the purpose of the program you are writing. You simply need to convert the Japanese code of each character to Row-Cell values right before executing each of these macros. The macros are as follows:

```
#define ISLEVEL1(A)    (A >= 16 && A <= 47)
#define ISLEVEL2(A)    (A >= 48 && A <= 84)
#define ISKANJI(A)     (ISLEVEL1(A) || ISLEVEL2(A))
#define ISHIRAGANA(A)  (A == 4)
#define ISKATAKANA(A)  (A == 5)
#define ISKANA(A)      (ISKATAKANA(A) || ISHIRAGANA(A))
#define ISKANAKANJI(A) (ISKANA(A) || ISKANJI(A))
```

Seasoned C programmers should be able to recognize what each of these macro definitions does.

The first two macros:

```
#define ISLEVEL1(A)    (A >= 16 && A <= 47)
#define ISLEVEL2(A)    (A >= 48 && A <= 84)
```

use the first byte (row) value to determine if a character is in JIS Level 1 or 2 kanji. You may recall that in JIS X 0208:1997, the kanji are contained in two ranges—rows 16 through 47, and rows 48 through 84—which is exactly what the macro checks for.

The next macro:

```
#define ISKANJI(A)     (ISLEVEL1(A) || ISLEVEL2(A))
```

combines the first two macros. Quite often you won't care whether a kanji is in JIS Level 1 or 2 kanji, but rather if it is a kanji at all. Again, it is sufficient to use only the first byte as input to this macro.

The next three macros:

```
#define ISHIRAGANA(A)  (A == 4)
#define ISKATAKANA(A)  (A == 5)
#define ISKANA(A)      (ISHIRAGANA(A) || ISKATAKANA(A))
```

do the same as the first three macros, but with kana. The first two detect whether a character is a hiragana or katakana, and the last one combines them. Like before, only the first byte is used for this.

The last macro:

```
#define ISKANAKANJI(A) (ISKANA(A) || ISKANJI(A))
```

checks for a larger set of characters, kana and kanji.

Similar macros can be written to accommodate other languages, such as Korean (KS X 1001:2004):

```
#define ISJAMO(A)        (A == 4)
#define ISHANGUL(A)      (A >= 16 && A <= 40)
#define ISHANJA(A)       (A >= 42 && A <= 93)
#define ISHANGULHANJA(A) (ISHANGUL(A) || ISHANJA(A))
```

and Chinese (GB 2312-80):

```
#define ISLEVEL1(A) (A >= 16 && A <= 55)
#define ISLEVEL2(A) (A >= 56 && A <= 87)
#define ISHANZI(A)  (ISLEVEL1(A) || ISLEVEL2(A))
```

Writing such macro definitions can be carried to almost any extreme, and represents a very useful tool in the hands of a C or C++ programmer. Of course, these macros could have been implemented as C functions or written in yet other programming languages.

Character Sorting

You can sort English text in a multitude of ways—low to high, high to low, dictionary, numeric; the possibilities are seemingly endless. CJKV locales have even more possibilities for sorting text. In English, despite all the possible variations, there are really only two basic ways to sort text. The first is case-insensitive, meaning that upper- and lowercase Latin characters are sorted as though they were the same. The other is an ASCII sort, which sorts by increasing value of the byte that represents each character. This has the effect of separating upper- and lowercase Latin characters whereby uppercase is sorted first. This is sometimes referred to as sorting *ASCIIbetically*.

Japanese, for example, has the equivalent of an ASCII sort, that is, characters are ordered by the values of the bytes used to represent them. This is often called a JIS sort. In Chapter 3, you learned that JIS X 0208:1997 Level 1 kanji are arranged by reading, and that JIS X 0208:1997 Level 2 kanji are arranged by radical, and then the total number of strokes of

the nonradical part. Consequently, a JIS sort produces a list of characters sorted in that way. And although they represent the same set of sounds, hiragana and katakana are separated when performing a JIS sort—the hiragana come first.

The *iroha* order is another collation sequence in addition to the 50 Sounds order. The name of this ordering comes from its first three sounds, specifically *i*, *ro*, and *ha*, the Japanese analogy to *a*, *b*, and *c*. The iroha collation sequence is not commonly used and is based on the Buddhist poem listed in Table 9-19.

Table 9-19. The iroha order as a poem

Japanese	Transliterated
いろはにほへと	*i ro ha ni ho he to*
ちりぬるを	*chi ri nu ru (w)o*
わかよたれそ	*wa ka yo ta re so*
つねならむ	*tsu ne na ra mu*
うゐのおくやま	*u (w)i no o ku ya ma*
けふこえて	*ke fu ko e te*
あさきゆめみし	*a sa ki yu me mi shi*
ゑひもせす	*(w)e hi mo se su*

Other types of sorts include by radical, by total number of strokes, and by pronunciation. Yes, these were listed previously in the JIS sort, but I am referring to the coverage of all kanji. For example, a sort by radical should include JIS Level 1 kanji, too. The implementation of these various sorting methods is limited to the database of information you have. Hiragana and katakana can also be sorted together like a case-insensitive sort of Latin characters, but some dictionaries sort hiragana separately from katakana.

Due to the unique nature of kana, the examples provided in Table 9-20 consist of four words written solely with kana, and will serve to illustrate some issues that arise when sorting kana.

Table 9-20. Sorting kana

Unordered	Byte-value order	Desired order
バンドフ	はんとう	ハンド
はんとう	ハントン	はんとう
ハントン	ハンド	バンドフ
ハンド	バンドフ	ハントン

All four words share several characteristics with regard to their first three characters. All begin with the kana whose basic (unannotated) form is read *ha* (バ, は, or ハ), followed by a syllabic *n* (ン or ん), then finally the kana whose basic form is read *to* (ド, と, or ト). Note how the byte-value order—also known as the JIS order because it is also the order in which these characters appear in JIS X 0208:1997—produces a significantly different result from the desired order. There are two things going on here to cause these radically different results:

- The distinction between hiragana and katakana is ignored—all hiragana come before any katakana in JIS X 0208:1997; this can be considered a form of case-folding.
- The distinction between unannotated and annotated kana is ignored—unannotated forms come before the annotated ones in JIS X 0208:1997.

Kana sorting can also be affected by small kana (which always come before the standard forms).

JSA published the standard designated JIS X 4061-1996, *Collation of Japanese Character String* (日本語文字列照合順番 *nihongo mojiretsu shōgō junban*), which provides detailed information and suggestions for sorting characters found in the Japanese character set standards. I suggest that you use this standard for guidance.

Chinese text, written using hanzi, is usually sorted by Pinyin reading. Pinyin transliteration, because it uses Latin characters, is thus sorted according to the English alphabet.

An example that illustrates different sorting requirements for the same writing system is Korean hangul. North and South Korea (DPRK and ROK, respectively), although they use the same set of jamo for constructing hangul, sort them differently. Table 9-21 illustrates the sequence in which jamo are sorted in the two Korean locales, subcategorized by the position in which they appear in hangul: initial (consonants), medial (vowels), and final (consonants).

Table 9-21. Korean jamo sorting sequences

Position	Locale	Sorting sequence
Initial	DPRK	ㄱㄴㄷㄹㅁㅂㅅㅈㅊㅋㅌㅍㅎㄲㄸㅃㅆㅉㅇ
	ROK	ㄱㄲㄴㄷㄸㄹㅁㅂㅃㅅㅆㅇㅈㅉㅊㅋㅌㅍㅎ
Medial	DPRK	ㅏㅑㅓㅕㅗㅛㅜㅠㅡㅣㅐㅒㅔㅖㅚㅟㅢㅘㅝㅙㅞ
	ROK	ㅏㅐㅑㅒㅓㅔㅕㅖㅗㅘㅙㅚㅛㅜㅝㅞㅟㅠㅡㅢㅣ
Final	DPRK	ㄱㄲㄴㄵㄶㄷㄹㄺㄻㄼㄽㄾㄿㅀㅁㅂㅄㅅㅇㅈㅊㅋㅌㅍㅎㄲㅆ
	ROK	ㄱㄲㄳㄴㄵㄶㄷㄹㄺㄻㄼㄽㄾㄿㅀㅁㅂㅄㅅㅆㅇㅈㅊㅋㅌㅍㅎ

In general, North Korea sorts double consonants after all other consonants. The vowels, in medial position, are also sorted quite differently.

Natural Language Processing

Attempting to derive meaning from text requires natural language processing. The most fundamental level of natural language processing is the ability to parse the text into words. The words themselves can be further broken down through morphological analysis. Applications for this technology include spelling and grammar checkers, along with tagging and indexing. The following sections will explore natural language-processing techniques, along with some of its applications.

Word Parsing and Morphological Analysis

Parsing most Western-language sentences into their constituent words is a somewhat trivial operation due to the intervening—and necessary—spaces between words. Other than the occasional punctuation and dealing with case issues, there is very little difficulty. Most CJKV sentences, however, offer significant challenges in this area of information processing. The ultimate goal of parsing sentences into their component words is for common purposes such as determining the key words of a document, which is called *tagging*. Tagging is useful for categorizing or indexing documents based on their content, and it is a key function for search engines.

Most CJKV sentences, such as those for Chinese and Japanese, include no intervening spaces between words. Korean, on the other hand, does use spaces to separate words. Typical Chinese sentences include only hanzi, along with a select few punctuation marks and symbols. So, how does one decide how to break up the words when there are only hanzi with which to deal? Typical Japanese sentences include mostly kana, to the tune of about 70%, plus some kanji. And, to make matters more interesting or challenging, some Japanese words are combinations of kana and kanji.

In order to successfully parse any CJKV sentence, a suitable dictionary is required. The importance of a dictionary cannot be understated. Of course, rules drive the process, but without a dictionary, the rules mean nothing. This dictionary can be very simple, listing only words of the language. Any parsing that is performed must result in chunks that match entries in this dictionary. More complex parsing dictionaries may be necessary to handle language-specific phenomena such as inflectional endings and other grammatical components. It is safe to state that a full-blown morphological analyzer is required, and many have been developed and are being used in real products.

Because Japanese text seems to be laced with the most difficulty, let's examine a sample Japanese sentence that includes some typical words that need to be identified: 漢字が含まれているテキストは読み易い.* Table 9-22 lists the constituent words and parts-of-speech that make up this Japanese sentence.

* This sentence can be translated into English as "Texts that include kanji are easy to read."

Table 9-22. A parsed sentence—Japanese

Word	Reading	Meaning
漢字	*kanji*	kanji
が	*ga*	Subject marker
含まれている	*fukumarete iru*	included
テキスト	*tekisuto*	text
は	*wa*	Topic marker
読み	*yomi*	read
易い	*yasui*	easy

Three of the constituent words that resulted from the parsing of this sentence can be further analyzed into other forms, such as their so-called base form. 含まれている (*fukumarete iru*), for example, is an inflected form of the verb 含む (*fukumu*). Likewise, 読み (*yomi*) is an inflected form of the verb 読む (*yomu*). Finally, 易い (*yasui*) is the base form of an adjective, but could have easily appeared as an inflected form, such as the form 易かった (*yasukatta*) or entirely in hiragana as やすい (*yasui*). One of my favorite inflected Japanese words, back from my college days during which I studied Japanese, is 片付けさせられませんでした (*katazuke sase raremasen deshita*). The base form of the verb is 片付ける (*katazukeru*), and the remaining hiragana characters, させられませんでした, represent the inflectional endings. These are examples of what operations a morphological analyzer would perform.

Some complex or compound words, such as 日本語情報処理 (*nihongo jōhō shori*, meaning "Japanese information processing"), can cause additional potential problems or issues, such as the need to determine how to break up the word into its component words, some of which may be single ideographs. Of course, such words can also be treated as a single unit, but doing so suggests that the dictionary includes an entry for it. In any case, there are several ways in which this compound word can be broken up, which are best described as levels of a hierarchy. Table 9-23 illustrates the various levels to which this particular compound Japanese word can be parsed.

Table 9-23. Parsing compound words—Japanese

Compound word	First level parsing	Second level parsing
日本語情報処理	日本語　Japanese	日本　Japan 語　language
	情報処理　information processing	情報　information 処理　processing

Chinese and Japanese sentences are typically laced with compound words, composed of hanzi and kanji, respectively. Korean sentences are, too, to some extent, but they use

hangul instead of hanja, and the intervening spaces help in the effort to parse them into the constituent parts.

Fujitsu Laboratories in Japan had developed a Japanese morphological analyzer called *Breakfast* that had the ability to parse Japanese text into morphemes, and had a customizable POS* (*part-of-speech*) system. This customizable feature enabled Breakfast to use the dictionaries of *JUMAN* (寿満 *juman*)† and *ChaSen* (茶筌 *chasen*),‡ both of which still seem to be available in some form, though Breakfast seems to have gone away. Another called *Sumomo* (すもも *sumomo*), developed by NTT, also seems to have gone away. Chances are, Breakfast and Sumomo were either sold and renamed, or simply renamed.

In addition to JUMAN and ChaSen, other more current Japanese morphological analyzers include *MeCab* (和布蕪 *mekabu*)§ and *KAKASI* (*Kanji Kana Simple Inverter*).¶

Without a doubt, Basis Technology's *Rosette Base Linguistics for Asian Languages*, which provides a morphological analyzer that supports Chinese, Japanese, and Korean, is one of the top-performing libraries of its kind.** The fact that Google and Amazon use it states something about its effectiveness. Its abilities include segmentation, tokenization, noun decompounding, part-of-speech tagging, sentence boundary detection, and other analyzing functions. They also provide related software, such as the *Rosette Japanese Orthographic Analyzer* (JOA) that detects alternate representations for Japanese words, and the *Rosette Chinese Script Converter* that converts between Simplified and Traditional Chinese. Later in this chapter you will learn that this level of conversion—or rather, translation—must take place at many levels, not only at the individual character level.

Spelling and Grammar Checking

Spelling- and grammar-checking software is extremely common for English, especially spellchecking software. In fact, most word processors and page composition systems include spellcheckers. However, finding such software for languages spoken in CJKV locales can be a daunting task—at least it was in the past.

SpellViser (日本語文書校正支援ライブラリ *nihongo bunsho kōsei shien raiburari* in Japanese), originally developed by Sumitomo Metal Industries (SMI), is an example of the Japanese equivalent of a spelling and grammar checker that was bundled with some Mac OS and Windows applications. Although it seems to have lost its name, the current version of EDICOLOR, which was also originally developed by SMI and is available for Mac OS X and Windows, includes SpellViser's functionality as part of its 文書校正支援機能

* Needless to say, at least for native English speakers, this is a very unfortunate abbreviation, because it suggests something about the quality of the thing being discussed or debated.

† *http://www-nagao.kuee.kyoto-u.ac.jp/nl-resource/juman.html*

‡ *http://chasen-legacy.sourceforge.jp/*

§ *http://mecab.sourceforge.net/*

¶ *http://kakasi.namazu.org/*

** *http://www.basistech.com/base-linguistics/asian/*

(*bunsho kōsei shien kinō*) feature. EDICOLOR, and thus SpellViser, are now owned and developed by Canon.* Apparently, SpellViser was licensed to Microsoft for MS Word-J, to Corel for its Japanese versions of Word Perfect and DRAW, to ERGOSOFT for EGWord, and to Kuni Research for Eudora Pro-J.

JustSystems also developed a similar Japanese grammar-checking technology called 修太 (*shūta*). It is included in their Japanese word processor called 一太郎 (*ichitarō*), and it is made accessible through its 文書校正 (*bunsho kōsei*) feature.†

Chinese-Chinese Conversion

Chinese is discussed throughout this book in the context of Simplified versus Traditional, whereby the former is used in China and in Singapore, and the latter is used in Taiwan, Hong Kong, and other regions where Chinese is spoken. While it is easy to fall into the trap to think that they differ only to the degree to which Simplified Chinese uses simplified ideographs, the truth is far more complex. Converting between these forms of Chinese is more of a translation than a conversion, because it takes place on many levels: *orthographic*, *lexemic*, and *contextual*.

In the past, prior to the broad adoption of Unicode, there was a fourth level to consider: *encoding*. Simplified Chinese was encoded according to GB 2312-80 or a related character set standard, and Traditional Chinese was encoded according to Big Five, and in some cases, CNS 11643-2007. Now, through the use of a uniform character set and encoding, specifically Unicode, at least one aspect of this issue is no longer problematic.

If we consider individual ideographs, there are hundreds of prototypical simplified/traditional pairs for which a one-to-one conversion works without problems. Consider 门 (U+95E8) and 門 (U+9580), and 国 (U+56FD) and 國 (U+570B), as two such pairs.

The issue starts to become more complex when we consider simplified ideographs that correspond to more than one traditional form—in other words, multiple traditional forms collapsed into a single simplified form. In some cases, one of the traditional forms serves as the simplified form. Consider 台 (U+53F0), which has four possible traditional forms: itself, 檯 (U+6AAF), 臺 (U+81FA), and 颱 (U+98B1).The following simplified/traditional *word* pairs exemplify correct uses of these four traditional forms: 台球/台球 (*táiqiú*), 球台/球檯 (*qiútái*), 印台/印臺 (*yìntái*), and 台风/颱風 (*táifēng*).

Another issue that needs to be addressed in such conversions are words that effectively transcend the simplified/traditional paradigm, and simply have different representations in these forms of Chinese. An excellent example is the Chinese word that means "computer." In Simplified Chinese it is written 计算机 (*jìsuànjī*), but in Traditional Chinese it is 電脑 (*diànnǎo*). One might naïvely think that 計算機 is the proper Traditional Chinese rendering of 计算机, or that 电脑 is the proper Simplified Chinese rendering of 電脑. At

* *http://ps.canon-its.jp/ec/*

† *http://www.ichitaro.com/*

the orthographic level, and in some limited contexts, these conversions may be appropriate, but they do not represent current usage. In other words, 计算机 is a Simplified Chinese word that must map to its lexemic equivalent in Traditional Chinese, which is 電腦.

For further reading about this topic, I suggest an enlightening article by Jack Halpern and Jouni Kerman entitled *The Pitfalls and Complexities of Chinese to Chinese Conversion*.* Much of this article led to the development of Basis Technology's *Rosette Chinese Script Converter*.

Special Transliteration Considerations

Although the transliteration and Romanization systems that were covered in Chapter 2 were primarily unidirectional in that Latin characters are used to represent CJKV text, there are other transliteration issues to consider.

It is common practice to transliterate Western names in Chinese using hanzi. For example, the name "Bush" is commonly transliterated into Chinese as 布希 (*bùxī*), 布什 (*bùshí*), or 布殊 (*bùshū*). In Japanese and Korean, Western names are transliterated using katakana and hangul syllables, respectively. Our example is thus expressed as ブッシュ (*busshu*) in Japanese, and as 부시 (*busi*) in Korean.

Some transliterations are hybrids, meaning part translation and part transliteration. The name "Starbucks" is a good example in that its Chinese transliteration is 星巴克 (*xīngbākè*). The first hanzi, 星 (*xīng*), means "star" in English and conveys meaning, and is thus the translated portion. The last two hanzi, 巴克 (*bākè*), are simply transliterations, meaning that they convey phonemic information.

Ideographs as used in Japan and Korea can be transliterated into kana and hangul syllables, respectively. In the case of Japanese, such transliteration is useful for indexing purposes, or perhaps for automatically generating ruby. In Japanese, many kanji have multiple readings, and while dictionaries can usually assist in determining which readings are appropriate or correct, there are some cases that require more sophsticated techniques. An excellent example is the common Japanese phrase 今日は, which can correspond to こんにちは (*kon'nichi-wa*) or きょうは (*kyō-wa*), depending on how it is used in a sentence. The former is a greeting that means "hello" or "good day," and the latter is a phrase that means "today." In Korean, many hanja also have multiple readings, so care must be taken to use the correct reading when performing such transliterations, especially when dealing with names. For example, the name 李明博 is transliterated as 이명박 (*i myeongbak*). Although the hanja 李 has two readings in Korean, 리 (*ri*) and 이 (*i*), the latter is appropriate for this name.

ICU, mentioned earlier in this chapter, is a powerful internationalization library that includes a *Transforms* package that is designed to manipulate Unicode text, and that

* *http://www.cjk.org/cjk/c2c/c2cbasis.htm*

performs script-to-script transliteration.* CLDR, also mentioned earlier in this chapter, is useful in that it provides transliterations and translations of country, language, and script names.

Regular Expressions

Regular expressions (*regexes* is a short way to express this; 正規表現 *seiki hyōgen* in Japanese) provide a very powerful mechanism to search for, replace, shred, or otherwise manipulate text or data. The most common regex engines, as found in popular Unix tools such as *awk*, *GNU Emacs*, *grep*, *Perl*, *Ruby*, *sed*, *Tcl*, and so on, have no inherent CJKV-specific capabilities. However, several CJKV-specific regex implementations have been developed over the years. The most noteworthy of these include JPerl (Japanese Perl) and GNU Emacs (version 20 or greater).

Adding CJKV or multiple-byte support to regex engines is a matter of being able to use multiple-byte characters in places where one-byte characters are expected. This may sound simple enough at first glance, but there is much complexity to consider. The character-class feature of regexes, for example, is an immediate candidate for this sort of extension. The following is a typical regex character class definition in Perl:

```
/[0-9A-Fa-f]/
```

This character class includes any upper- or lowercase hexadecimal digits; the entire regex (that is, what appears between the slashes) matches exactly one character in this character class. However, when we deal with CJKV text, the following character class would be very useful:

```
/[あ-んア-ケ]/
```

This character class is intended to include all hiragana and katakana characters. JPerl allows you to specify such character classes. Without explicit Japanese (or, multiple-byte) support, the previous regex would be meaningless (or, at least, result in an incorrect match).† There are, however, clever ways to fake such character classes, covered later in this section. You may think that the following is equivalent to the previous kana character class:

```
/[\xA4-\xA5][\xA1-\xFE]/
```

This regex should match any character in the range <A4 A1> through <A5 FE>, which means the hiragana and katakana rows in EUC-JP encoding, right? Nope. Because standard (that is, those that are not CJKV-capable) regex engines match on a per-byte basis, a match may be made with the second byte of a two-byte character followed by the first byte of the next two-byte character. See the "Character Searching" section, earlier in this chapter, for more details on why this is important to avoid at all costs.

* *http://www.icu-project.org/userguide/Transform.html*

† This regex, if EUC-JP–encoded, would be interpreted as `/[\xA4\xA1-\xA4\xF3\xA5\xA1-\xA5\xF6]/`, and would match a single byte whose values are in the following ranges: 0xA1–0xA5, 0xF3, and 0xF6. Not what you would expect, eh?

The only reliable way to handle multiple-byte encodings using standard regex engines is to use techniques that effectively trap all characters. You may not need to perform any transformations on most of the trapped characters—they can be easily output as is. This means that there are two types of operations to consider:

- Process all characters or all characters of a particular class—some sort of converter or filter (a side-effect of applying a regex to all characters in a file allows an encoding-integrity check to be performed with little additional effort).
- Selectively process characters—search or search/replace.

Other aspects of regexes that need to be extended for supporting multiple-byte characters include the definitions of many metacharacters, such as . (*dot*). This metacharacter matches any (single) byte, and to make it useful in the context, CJKV information processing requires that it match any (single) character.

JPerl is unique in that its regex engine provides a way to directly use Japanese characters. Other regex engines, such as those found in GNU Emacs, predefine Japanese-specific character classes as metacharacters. Table 9-24 provides a listing of these predefined character classes, according to the latest version of GNU Emacs, along with a definition plus the equivalent JPerl regex.

Table 9-24. Japanese-specific regular expression implementations

GNU Emacs	GNU Emacs definition	JPerl equivalent
\cA	Alphanumeric (row 3)	[０-９Ａ-Ｚａ-ｚ]
\cH	Hiragana (01-11, 01-12, 01-21, 01-22, row 4)	[゛゜ヽヾぁ-ん]
\cK	Katakana (01-11, 01-12, 01-19, 01-20, 01-28, row 5)	[゛゜ヽヾーア-ケ]
\cG	Greek (row 6)	[Α-Ω α-ω]
\cY	Cyrillic (row 7)	[А-Я а-я]
\cC	Kanji (01-24 through 01-27, 05-86, rows 16–84)	[仝-〇ケ亜-腕弋-熙]

When running GNU Emacs in other language modes, such as Chinese or Korean, additional multiple-byte metacharacters spring into existence. In particular, *ch* (matches any KS X 1001:2004 character) and *cc* (matches any GB 2312-80, CNS 11643-2007, or Big Five character). The *cA*, *cG*, *cH*, *cK*, and *cY* multiple-byte metacharacters also function for Chinese and Korean, but match characters in the appropriate rows.

In the context of Unicode, regex engine developers are strongly encouraged to read UTS (*Unicode Technical Standard*) #18, entitled *Unicode Regular Expressions*, to get guidance for how regex engines are to handle Unicode text.*

* *http://unicode.org/reports/tr18/*

I encourage you to study the Perl code examples found in Appendix C to learn more about how regexes can be used to manipulate CJKV text.

For more information on regular expressions, I highly suggest Jeffrey Friedl's *Mastering Regular Expressions*, Third Edition (O'Reilly Media, 2006).* *X/Open Guide: Internationalisation Guide* (X/Open Company Limited, 1993) also includes a chapter on internationalized regular expressions.

Search Engines

One important function of the Web is the ability to conduct searches for particular items. While most of the popular search engines accept only ASCII characters (or regexes that reflect ASCII text) for this task, there are now a number of CJKV-capable search engines available.

The toughest issues faced by CJKV-capable search-engine developers include the following:

- The proper handling of multiple-byte characters that appear in the search string, including multiple-byte support for regexes.
- The proper handling of multiple encodings for both the search string and searched text (for example, a user-entered Korean EUC-KR search string must be able to match in documents encoded according to EUC-KR and ISO-2022-KR encodings), which effectively means that the encodings for the search string and searched text must be regularized (because the searched text may be large, it is much easier to regularize the search string to match the encoding of the searched text).

Japanese-enabled search engines include the obvious Google,† but also goo‡ and Yahoo!§ Thankfully, the methodology for handling CJKV text is now well-established, and everything is stored and indexed according to Unicode. And, commercial-grade morphological analyzers are being used as the basis for indexing.

Code-Processing Tools

Following you will find brief descriptions of and the printed help pages for three Japanese code-processing tools that I have written and maintained until 1993: *JConv*, *JChar*, and *JCode*. Also included is a description of JChar's replacement: the *CJKV Character Set Server*, available online as a web service. This section also includes some contexts in which these tools may be useful to your work. The latest source code for these tools is available

* *http://www.regex.info/*

† *http://www.google.co.jp/*

‡ *http://www.goo.ne.jp/*

§ *http://www.yahoo.co.jp/*

online.* I find it intriguing that source code that I haven't touched in 15 years is still useful today.

Three of these tools were written in ANSI C, and are portable on compilers that conform to this standard. This means that their source code, without any modifications, should compile on multiple platforms, which has been confirmed by their many users. Each of these three tools displays its help page by using the `-h` option on the command line. These same help pages are listed in the following sections, and are provided in order to illustrate the full potential of the tools' functionality.

JConv—Code Conversion Tool

The most basic Japanese text-processing requirement is a tool that converts Japanese text from one encoding to another. This is most important when moving Japanese text from one platform to another, and when receiving email messages or news articles. All in all, this is a general workhorse tool. I use it often.

This tool, called JConv, implements the routines for converting from one Japanese encoding to another. The main features of JConv are as follows:

- Supports the JIS X 0208:1997 character set
- Handles ISO-2022-JP, Shift-JIS, and EUC-JP encodings
- Lists code specifications for Japanese encoding methods
- Filters half-width katakana by converting them to their full-width counterparts
- Repairs damaged ISO-2022-JP–encoded files
- Can forcibly damage ISO-2022-JP–encoded files (so they can be restored later with the repair option)
- Lets one check files for their encoding without actually performing any code conversion
- Includes a verbose mode option that displays more information about what the tool is doing

Many of the algorithms and routines listed and explained earlier in this chapter are used in this code-processing tool.

Here is this tool's help page:

```
** jconv v3.0 (July 1, 1993) **
Written by Ken R. Lunde, Adobe Systems Incorporated
lunde@adobe.com
Usage: jconv [-options] [infile] [outfile]
Tool description: This tool is a utility for converting the Japanese code of
textfiles, and supports Shift-JIS, EUC, New-JIS, Old-JIS, and NEC-JIS for
```

* *http://examples.oreilly.com/9780596514471/src/*

```
both input and output. It can also display a file s input code, repair
damaged Old- or New-JIS files, and display the specifications for any of the
handled codes.
Options include:
  -c       Displays the detected input code, then exits -- the types
           reported include EUC, Shift-JIS, New-JIS, Old-JIS, NEC-JIS, ASCII
           (no Japanese), ambiguous (Shift-JIS or EUC), and unknown (note
           that this option overrides "-iCODE")
  -f       Converts half-width katakana to their full-width equivalents (this
           option is forced when output code is New-, Old-, or NEC-JIS)
  -h       Displays this help page, then exits
  -iCODE   Forces input code to be recognized as CODE
  -o[CODE] Output code set to CODE (default is Shift-JIS if this option is
           not specified, or if the specified CODE is invalid)
  -r[CODE] Repairs damaged New- and Old-JIS encoded files by restoring lost
           escape characters, then converts it to the CODE specified (the
           default is to convert the file to New-JIS if CODE is not
           specified -- cannot be used in conjunction with "-s")
  -s[f]    Removes escape characters from valid escape sequences of New- and
           Old-JIS encoded files -- "f" will force all escape characters
           to be removed (default extension is .rem -- cannot be used in
           conjunction with "-r")
  -t[CODE] Prints a table listing the specifications for the specified CODE,
           then exits (all code tables will be displayed if CODE is not
           specified, or if CODE is invalid)
  -v       Verbose mode -- displays information such as automatically
           generated file names, detected input code, number of escape
           characters restored/removed, etc.
NOTE: CODE has five possible values (and default outfile extensions):
      "e" = EUC (.euc); "s" = Shift-JIS (.sjs); "j" = New-JIS (.new);
      "o" = Old-JIS (.old); and "n" = NEC-JIS (.nec)
```

JConv has been ported to Mac OS, MS-DOS, and MS Windows. One of the Mac OS ports was done by Natsu Sakimura, and it is called JCONV-DD (DD stands for "Drag and Drop").

If you are interested in cross-locale code conversion, consider using *CJKVConv.pl*, *tcs*, or *Uniconv*, all of which were described in Chapter 4's section entitled "Code Conversion Across CJKV Locales."

JChar—Character Set Generation Tool

Another general Japanese code-processing need is the ability to generate a listing of Japanese character sets. Generating a file that contains a complete electronically encoded Japanese character set can be done most effectively with the use of loops found in most programming languages. After all, who wants to manually input several thousand characters? Generating the coded character sets is trivial, as loops do all the work for you. The problem occurs when you want to generate a list containing only the characters in a noncoded character set, such as Jōyō Kanji. There is no algorithm you can use, since the necessary kanji are scattered throughout JIS X 0208:1997 Level 1 kanji. The only way to handle such a task is to key the kanji in manually, and then to be sure to save your work!

I have written a tool, called *JChar*, that generates these problematic character sets (and nonproblematic ones, too!). Listings of these noncoded Japanese character set standards, as generated by JChar, are in Appendix J.

The JChar tool has many features and options that you will find useful at some time or another—the main ones are as follows:

- Supports the JIS X 0208:1997, ASCII/JIS-Roman, half-width katakana, Jōyō Kanji, Gakushū Kanji, and Jinmei-yō Kanji character sets
- Outputs data in ISO-2022-JP, Shift-JIS, or EUC-JP encoding
- Wraps output lines at *n* columns
- Can suppress header information

Algorithms used in this tool are primarily loops (for the coded character sets) and data structures (for the noncoded character sets). Encoding range bounds are used, though, to generate the whole character encoding space, and not just the code positions that contain characters. For example, when choosing to generate the JIS X 0208:1997 list, it does not generate 6,879 code positions, but does 8,836 code positions, which is what you get from a complete 94×94 matrix.

Here is this tool's help page:

```
** jchar v3.0 (July 1, 1993) **
Written by Ken R. Lunde, Adobe Systems Incorporated
lunde@adobe.com
Usage: jchar [-options] [outfile]
Tool description: This tool is a utility for generating various Japanese
character sets in any code. This includes all the characters specified in
JIS X 0208-1990, half-width katakana (EUC and Shift-JIS output only), the
94 printable ASCII/JIS-Roman characters, the 1945 Joyo Kanji, the 284
Jinmei-yo Kanji, and the 1006 Gakushu Kanji.
Options include:
  -a       Builds an ASCII/JIS-Roman list (printable characters only)
  -g       Builds the Gakushu Kanji list
  -h       Displays this help page, then exits
  -j       Builds the Joyo Kanji list
  -k       Builds the JIS X 0208-1990 list
  -o[CODE] Builds lists in CODE format (default is Shift-JIS if this option
           is not specified, if CODE is not specified, or if CODE is invalid)
  -p       Builds the Jinmei-yo Kanji list
  -s       Suppresses headers and row number information
  -w[NUM]  Wraps output lines at NUM columns (if NUM is not specified, 78
           is used as the default value)
NOTE: CODE has five possible values: "e" = EUC; "s" = Shift-JIS;
      "j" = New-JIS; "o" = Old-JIS; and "n" = NEC-JIS
```

CJKV Character Set Server

\I decided to upgrade and enhance JChar—sometime in 1996—to support CJKV character sets and encodings. Now, instead of being available as C source code that must be

compiled and then run on specific OSes, it is a web service with a CGI program under the hood.* The underlying CGI program is written in Perl.

Like JChar, the *CJKV Character Set Server* supports both coded and noncoded character sets. And, you can decide whether you'd like a file emailed to you—automatically uuencoded for safety if there are any eight-bit characters used in the selected encoding—or else display the character set directly in your web browser, which can be easily copied and pasted.

JCode—Text File Examination Tool

Every programmer—or even nonprogrammer types with enough interest—may occasionally like to take a closer peek at Japanese codes and how they relate to each other. The non-Japanese analogy is a hex dump of a file. However, since most Japanese characters consist of two bytes, a normal hex dump may not be very useful. Such a tool designed for use with Japanese text should treat two-byte characters as single entities. It should also make use of all the routines for converting between the various encoding methods, but instead of converting characters, it lists each character, along with its associated value in a variety of encodings.

A tool I wrote, called *JCode*, fills this gap, and offers two basic functions, indicated as follows:

- Accepts actual encoded Japanese characters in a variety of encodings, and then performs the equivalent of a hex dump
- Accepts four- and five-digit codes, one per line, that represent the encoded value of a character—for instance, "3021" or "k1601" for the JIS X 0208:1997 kanji 亜—then performs the equivalent of a hex dump

Using a hex dump as the non-Japanese analogy to this tool is not entirely correct. The tool JCode also allows you to perform an octal or decimal dump, depending on what notation you want (the default is to use hexadecimal notation).

Now it's time to see some sample output of JCode. First, you will see how this tool can handle actual Japanese text. Note that the file cannot be of a mixed encoding (that is, Shift-JIS plus EUC-JP, and so on). The following four characters serve as the example input to JCode:

かな漢字

They characters are pronounced *ka na kan ji* (meaning "kana [and] kanji"). The resulting output is shown in Table 9-25.

* *http://lundestudio.com/cjkv-char.html*

Table 9-25. JCode output—first example

Character	Shift-JIS	EUC	JIS	ASCII	KUTEN
か	82-A9	A4-AB	24-2B	$+	04-11
な	82-C8	A4-CA	24-4A	$J	04-42
漢	8A-BF	B4-C1	34-41	4A	20-33
字	8E-9A	BB-FA	3B-7A	;z	27-90

Next, you will see how this tool can handle four- and five-digit codes. To automatically detect all the main encodings, you must add a prefix before EUC-JP and Row-Cell (called KUTEN in its output) values, and require hexadecimal notation for ISO-2022-JP, Shift-JIS, and EUC-JP encodings. Here is the input I used:

```
82A9
xa4cA
3441
k2790
```

The first line is a hexadecimal Shift-JIS code, the second line is a hexadecimal EUC-JP code (note the "x" prefix), the third line is a hexadecimal ISO-2022-JP code, and the last line is a Row-Cell (KUTEN) code (note the "k" prefix). Table 9-26 provides the resulting output.

Table 9-26. JCode output—second example

Character	Shift-JIS	EUC	JIS	ASCII	KUTEN
か	82-A9	A4-AB	24-2B	$+	04-11
な	82-C8	A4-CA	24-4A	$J	04-42
漢	8A-BF	B4-C1	34-41	4A	20-33
字	8E-9A	BB-FA	3B-7A	;z	27-90

As you can see, a completely different set of input data produced the exact same output. Also note how the handling of the four- and five-digit codes is not case-sensitive, and how each line can specify a different encoding.

This tool has other options, most of which allow you to better format the output, as follows:

- Supports octal, decimal, and hexadecimal notations for output (the default is hexadecimal)
- Pads columns with spaces or a tab (the default is padding with spaces)
- Shows control characters
- Includes a verbose mode that provides more information, such as which Japanese encoding was detected

This tool, as you might expect, uses many of the code conversion algorithms and routines described earlier in this chapter. The remainder is simply fancy formatting of the output.

Now for this tool's help page:

```
** jcode v3.0 (July 1, 1993) **
Written by Ken R. Lunde, Adobe Systems Incorporated
lunde@adobe.com
Usage: jcode [-options] [infile] [outfile]
Tool description: This tool is a utility for displaying the electronic values
of Japanese characters within textfiles, and supports Shift-JIS, EUC, New-JIS,
Old-JIS, and NEC-JIS for both input and output.
Options include:
  -c[DATA]     Reads codes, one per line, rather than characters as input --
               if DATA is specified, only that code is treated, then exits
               (KUTEN codes must be prefixed with "k," and EUC codes with
               "x" -- EUC, JIS, and Shift-JIS codes must be hexadecimal)
  -h           Displays this help page, then exits
  -iCODE       Forces input code to be recognized as CODE
  -n[NOTATION] Output notation set to NOTATION (default is hexadecimal if this
               option is not specified, if NOTATION is not specified, or if
               the specified NOTATION is invalid)
  -o[CODE]     Output code set to CODE (default is Shift-JIS if this option
               is not specified, if CODE is not specified, or if the
               specified CODE is invalid)
  -p[CHOICE]   Pads the columns with CHOICE whereby CHOICE can be either t
               for tabs or "s" for spaces (default is spaces if this option
               is not specified, if CHOICE is not specified, or if the
               specified CHOICE is invalid)
  -s           Shows control characters (except escape sequences)
  -v           Verbose mode -- displays information such as automatically
               generated file names, detected input code, etc.
NOTE: CODE has five possible values:
      "e" = EUC; "s" = Shift-JIS; "j" = New-JIS; "o" = Old-JIS;
      and "n" = NEC-JIS
NOTE: NOTATION has three possible values:
      "o" = octal; "d" = decimal; and "h" = hexadecimal
```

Other Useful Tools and Resources

There are many more tools that perform very useful and time-saving tasks that are, in one way or another, CJKV-related. The following is a brief listing of some more well-known sources for such utilities or data:

- Basis Technology's *Rosette Linguistics Platform*[*]
- Erik Peterson's Chinese tools[†]
- *International Components for Unicode* (ICU)[‡]

* *http://www.basistech.com/products/*

† *http://www.mandarintools.com/*

‡ *http://www.icu-project.org/*

- Jack Halpern's *The CJK Dictionary Institute*[*]
- Koichi Yasuoka's *Kanji Bukuro* (漢字袋 *kanji bukuro*)[†]
- Koichi Yasuoka's various CJK mapping tables[‡]

I encourage you to explore these URLs, along with the many books that are listed in this book's Bibliography, to find suitable tools or data for your needs. It goes without saying that it is best not to reinvent the wheel.

* *http://www.cjk.org/*

† *http://kanji.zinbun.kyoto-u.ac.jp/~yasuoka/kanjibukuro/*

‡ *http://kanji.zinbun.kyoto-u.ac.jp/~yasuoka/CJK.html*

CHAPTER 10

OSes, Text Editors, and Word Processors

Here we cover basic or minimal functionality, specifically the use of OSes, text editors, and word processors. These environments and applications completely satisfy the basic requirements for some users, even without regard to the CJKV implications that are discussed throughout this book. One could even argue that today's mobile devices, including some cell phones, satisfy the needs for some users, because they provide all the features and functionality that they desire, such as the ability to send and receive email, to browse the Web, to keep contact information, and for recording text notes.

In this chapter, we discuss software that companies and individuals have developed, which, when properly integrated and configured, provides the ability to create, format, display, print, send, or receive CJKV text by electronic means. Figure 10-1 illustrates how these various text-processing tools interact with each other.

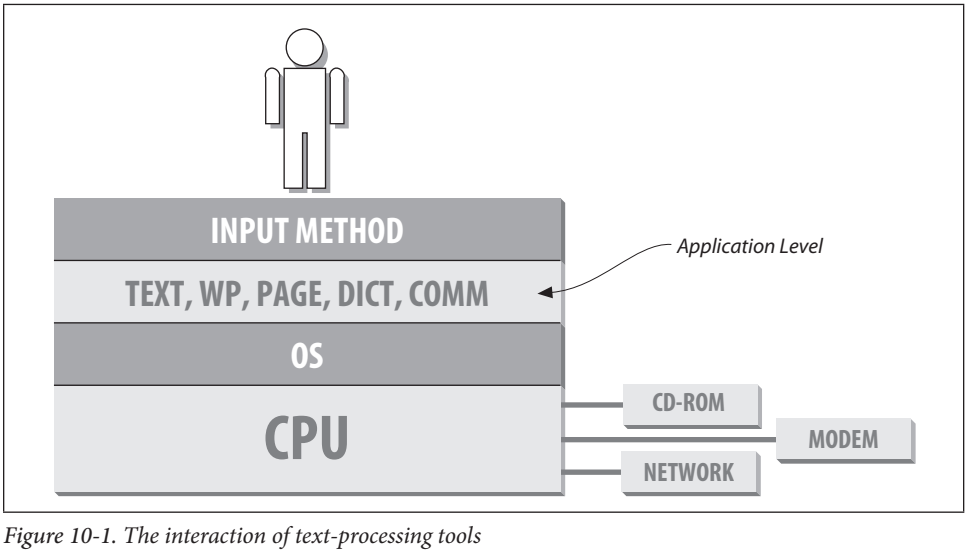


Figure 10-1. The interaction of text-processing tools

The Web, of course, is changing the way we think about software, and applications no longer need to be tied to a specific OS or platform. The introduction of web browsers started this trend, because they effectively act as portals for online applications. At the end of this chapter I cover online word processors, which continue this trend. As this technology improves, online applications are likely to become more sophisticated, even rivaling conventional applications.

One of the goals of this chapter is to give readers, whether they are developers or users, a basic understanding or overview of the types of CJKV text-processing utilities that are available, and what capabilities to expect from each. This, of course, includes the OSes on which they run. Note that certain classes of these utilities and applications have already been discussed in earlier and more appropriate chapters: code conversion tools at the end of Chapter 4; input methods in Chapter 5; and page composition and graphics software in Chapter 7. Some of the code-processing tools described at the end of Chapter 9 are relatively specialized programmer utilities that I developed, or were developed by other creative individuals and companies. Others are discussed in later and again more appropriate chapters: dictionary software and learning aids in Chapter 11; and web-related software in Chapter 12.

Mind you, the capabilities of these types of applications range from the most basic to very complex. With all of their differences, they share one common feature: they are all multiple-byte–aware or multilingual. Many support Unicode. In fact, I would claim that those that do not support Unicode are uninteresting, and thus not compelling. Not all software has been localized for all CJKV locales, but some provide an adequate or basic level of CJKV support, such as multiple-byte awareness. This was especially true of Mac OS applications that were considered to be WorldScript II–compatible. Such applications became multiple-byte–aware when the appropriate script (language) resources were installed into the OS, either due to the use of a fully localized OS or else the installation of one of Apple's Language Kits.

I make an effort to describe at least one example for each of the categories listed on the first page of this chapter, and for the following platforms: Mac OS X, Unix (with a focus on Linux), and Windows Vista. And, the fact that I list particular software in this chapter does not mean that I necessarily prefer it over software that I have not listed. Bear in mind that it is this chapter that is likely to become quickly outdated.

To an unprecedented extent, Unicode has changed the landscape for the better for OSes, text editors, word processors, and other applications. Put simply, Unicode has enabled multilingual support, and to a significant extent, has trivialized this effort. Languages and scripts that at one time competed and conflicted with one another can now peacefully coexist, and even flourish.

Included in this chapter are terms such as *freeware*,* *shareware*,† and *commercial software*.‡ *Open source* is another term that will pop up from time to time. These terms relate to the licensing terms of the software, and to what extent the source code is made available. This chapter will provide you with a basic working knowledge of what functions to expect, but stops short of providing all the information about such software. To do so is clearly beyond the scope of this book. If you are interested in a particular OS or application, I encourage you to obtain more detailed information on it from its creator, manufacturer, or distributor. In some cases, the manual may be available as a downloadable PDF file, or there may be a trial version available.

Viewing CJKV Text Using Non-CJKV OSes

For a significant number of users, simply being able to display CJKV text is sufficient to suit their needs. The ability to input or otherwise manipulate CJKV text may not be so important to this class of user. If you do not fall into this category of user, I strongly encourage you to consider obtaining an OS or other environment that provides a full range of CJKV support, not simply the ability to display CJKV text. As long as you are using Mac OS X or Windows Vista as your OS, your needs will be met.

In the past, Microsoft provided, as part of their OS CD-ROMs and on their website, what were referred to as *Language Packs* that enabled users to add CJKV functionality to their earlier OSes. If you installed the appropriate Language Pack, it would enable the display—but not input—of the corresponding CJKV text in many applications, such as Internet Explorer. Windows 2000 changed this and effectively eliminated the need for Language Packs. Windows Vista took this one step further through a variety of refinements and enhancements.

Apple provided comparable functionality through the distribution of *Language Kits*, which enabled not only the display of CJKV text, but also their input. The introduction of Mac OS X eliminated the need to use Language Kits, and instead rich multilingual support was built into the OS core.

* Freeware software is free, meaning that there is no fee to use it. But it is not public domain, which means that the developer still maintains the copyright.

† Shareware software is usually freely available, but there is a minimal fee that you must pay the developer in order to continue using it. This effectively allows you to try-before-you-buy. If you end up regularly using shareware programs, I strongly encourage you to pay the license fee in fairness to the software author who spent countless hours developing it.

‡ Some commercial applications are available as demo or trial versions at no charge, which means that you are able to try-before-you-buy, but unlike shareware, there are may be restrictions, such as the inability to print or save files. Some trial versions are fully functional, such as those offered by Adobe Systems, and are so for 30 days. If you're interested in a particular commercial application, I strongly encourage you to seek out a demo or trial version, either at the company's website, by making a telephone call, or sending an email.

AsianSuite X2—Microsoft Windows

UnionWay provides CJKV functionality on English versions of Windows, including input methods and TrueType fonts, through their *AsianSuite X2* product.* Although legacy encodings are supported, Unicode is also.

NJStar CJK Viewer—Microsoft Windows

Also known as NJWIN, NJStar CJK Viewer enables the display and printing of CJK text in non-CJK versions of Windows.† For those users who also need the ability to enter CJK text, *NJStar Communicator*, which additionally provides CJK input methods and other utilities, should be explored.‡

TwinBridge Language Partner—Microsoft Windows

TwinBridge provides CJKV functionality on English versions of Windows, to include input methods and TrueType fonts, through their TwinBridge Language Partner products. Although these products are specifically designed for use with English versions of Windows, they will work with any version to add CJKV functionality. Of course, Unicode is supported by these products.

Although they have separate Chinese, Japanese, and Korean versions of TwinBridge Language Partner, they also have developed a *TwinBridge CJK Partner*, which presumably includes all of their functionality in a single product.

Operating Systems

Many of the text-processing tools that are described in the sections that follow, or that were described in previous chapters, may require that you are using a CJKV-capable OS or that you add CJKV support to the OS that you are using. This may involve replacing your OS entirely with a fully localized version, or else adding extensions to your current OS. The latest versions of popular OSes include an adequate level of multilingual support, to the extent that they can be used as is. I am speaking, of course, of Mac OS X and Windows Vista.

CJKV-capable—meaning multilingual—OSes typically provide two main benefits, described as follows:

- They provide multilingual text-handling capability at the OS level, which usually—but not always—gives non-CJKV (or unlocalized) applications the ability to handle multilingual text.

* *http://www.unionway.com/*

† *http://www.njstar.com/cms/njstar-cjk-viewer/*

‡ *http://www.njstar.com/cms/njstar-communicator/*

- They provide a UI (user interface) that is tailored for a target locale (in Korea, for example, in the form of a Korean OS)—this is a major convenience for users who happen to be native speakers of the locale targeted by the OS developer.

To a great extent, the material in this section lays the foundation for the discussions that follow in the later sections of this chapter.

There are far too many OSes and OS extensions to describe in this book. What is provided in the following sections are brief descriptions of CJKV-capable OSes and OS extensions—in essence, how to add CJKV support to computers. What is not described are the computers that are manufactured specifically for CJKV locales. Older computers required special ROM to support CJKV locales—the fonts to support the locale, for example, were included in the ROM. Fortunately, today's trend is to use standard hardware.

At the end of Chapter 6 was a brief description of OS-bundled fonts, along with a strong recommendation that software should not reference them directly. This is because they are subject to change, either by being removed entirely in a new version of the OS, or by being renamed. In other words, it is important that developers and users do not become dependent on OS-bundled fonts nor the specific characteristics that they exhibit, because both are subject to change with little or no warning.

FreeBSD

FreeBSD is yet another freeware version of Unix, similar to Linux in that it is designed to run on PC compatibles.* As its name suggests, FreeBSD is based on BSD Unix. While new versions of the Linux kernel are released frequently, FreeBSD's releases are more infrequent. Unix is covered later in this chapter.

Linux

Linux (りぬくす *rinukusu* in Japanese;† 리눅스 *rinukseu* in Korean) is arguably the most widely used OS today, simply because it has grown into the preferred OS for servers. It is broadly used for personal use, embedded systems, mobile devices, and even games.‡ It is Unix-like and Unix-compatible, and for many purposes, has effectively replaced Unix. Linux is best described as a fully matured Unix-compatible OS that runs on ordinary Intel-processor–powered PCs. Linux also runs on a variety of other machines and architectures, including embedded systems.

Part of the success of Linux is due to its outstanding support for Unicode. Like other Unix implementations, the UTF-8 encoding form is the preferred way in which to

* *http://www.freebsd.org/*

† Wikipedia claims that there are a myriad of ways in which to express Linux in Japanese, such as リーヌークス (*rīnūkusu*), リナックス (*rinakkusu*), リヌックス (*rinukkusu*), リヌクス (*rinukusu*), ライナックス (*rainakkusu*), and リーナクス (*rīnakusu*).

‡ *http://www.linux.org/*

represent Unicode on Linux. As described in Chapter 4, it is compatible with the UTF-16 and UTF-32 encoding forms and is considered one of the three fundamental ways in which Unicode text is represented.

The number of Linux distributions that are available today is staggering. Yet, they all share one common attribute, specifically that they all use the Linux kernel. Linus Torvalds first developed the Linux kernel in 1991, and remains the primary force behind Linux in that he is still responsible for the kernel. But, there is a cast of tens of thousands performing actual development on Linux as a whole.

Many large companies provide Linux distributions, such as IBM, SGI, and Sun Microsystems. Some companies, such as Red Hat, specialize in providing Linux distributions. Linux itself is open source, meaning free. Companies charge for Linux, either for the installation conveniences that they provide, or for the support or additional applications and features that they provide.

Popular commercial Linux distributions include *Red Hat Linux* and Novell's *SUSE Linux*. Novell also provides a community version of their SUSE Linux called *OpenSUSE*, and Red Hat does the same with *Fedora*. *Ubuntu Linux* and *Debian Linux* are other popular Linux distributions. This list could go on and could fill a book, but we won't do that here.

Like Unix, Linux is an OS that lacks what I would consider to be adequate built-in font support, comparable to what Mac OS X and Windows 2000/XP/Vista provide. This functionality gap has effectively widened when Adobe Systems licensed its ATM font-rendering engine to Apple and Microsoft, which made PostScript-based fonts first-class citizens on those OSes. Of course, there are libraries available that can be used by applications that run on Linux, such as FreeType 2, which is included with virtually every Linux distribution.* But, until OpenType fonts are supported by Linux at the OS level, I consider its font support to be inadequate.

An excellent introduction to Linux is *Running Linux*, Fifth Edition (O'Reilly Media, 2005),† and *Linux in a Nutshell*, Fifth Edition (O'Reilly Media, 2005) provides incredibly useful reference material.‡

Mac OS X

In my opinion, and perhaps due to my computing background that includes Unix and Mac OS, Apple revolutionized the world of OSes by effectively merging Mac OS and Unix into a single OS called *Mac OS X*.§ For those who are not aware, the "X" of Mac OS X is the Roman numeral for 10. For those who needed to run applications that were not built to run on Mac OS X, Apple provided *Classic Mode*, which allowed Mac OS to run

* *http://freetype.sourceforge.net/index2.html*

† *http://oreilly.com/catalog/9780596007607/*

‡ *http://oreilly.com/catalog/9780596009304/*

§ *http://www.apple.com/macosx/*

simultaneously with Mac OS X. For a while, it was a "best of both worlds" situation. As soon as Apple started to use Intel processors in lieu of PowerPC processors, their machines could no longer run Classic Mode, and users were forced to make the complete switch over to Mac OS X. Some applications, such as Adobe FrameMaker, were never developed to run on Mac OS X.*

Apple has always been committed to making Macintosh a multilingual platform. Development of a localized Japanese version of its OS, originally called *KanjiTalk* (漢字 Talk *kanji tōku*), began very early on. KanjiTalk then changed its name to Mac OS-J, and both versions processed Shift-JIS encoding internally. Genuine Unicode enablement was introduced with Mac OS X, and with each subsequent release of Mac OS X, the extent to which Apple's OS supports Unicode improves.

Another development or achievement of Mac OS X, which many users (and developers) take for granted, is that Adobe Systems licensed to Apple its ATM (*Adobe Type Manager*) rasterizer, which effectively added support for PostScript-based fonts at the OS level. This means that Type 1 fonts, CID-keyed fonts, sfnt-CID fonts, and OpenType fonts are fully supported in Mac OS X, though OpenType is clearly the most important of these PostScript-based font formats. This effectively means that ATM is no longer required and will not run on Mac OS X.

For more information about Mac OS X, I recommend David Pogue's witty and informative *The Missing Manual* book, the latest of which is entitled *Mac OS X Leopard: The Missing Manual* (O'Reilly Media, 2007).†

Mac OS X fonts

Beginning with the very early versions of Mac OS X, Apple began bundling a somewhat large set of OpenType Japanese fonts, all of which were developed by Dainippon Screen. The designs themselves came from a Japanese type foundry called Jiyu-Kobo, and are part of the now-famous *Hiragino* (ヒラギノ *hiragino*) typeface family.‡ A total of six Hiragino fonts were bundled.

Beginning with Mac OS X version 10.5 (aka *Leopard*), Apple effectively doubled the number of bundled OpenType Japanese by virtue of providing JIS2004-*savvy* versions of the fonts, in addition to continuing to bundle the non–JIS2004-*savvy* versions. As stated in Chapter 6, the fonts required for Mac OS X are in the */System/Library/Fonts/* directory, and those that are considered optional are in the */Library/Fonts/* directory. The required fonts are necessary for UI or other purposes and cannot be deleted without Administrator

* *http://www.fm4osx.org/*

† *http://oreilly.com/catalog/9780596529529/*

‡ *http://www.screen.co.jp/ga_product/sento/*

or "root" privileges.* Tables 10-1 and 10-2 list the required and optional Chinese, Japanese, and Korean fonts that are bundled with Mac OS X, current as of version 10.5.

Table 10-1. Mac OS X version 10.5 fonts—required

Language	In /System/Library/Fonts/	Type foundry	Glyphs
Simplified Chinese	华文细黑.*ttf*	SinoType	37,256
	华文黑体.*ttf*		32,493
Traditional Chinese	儷黑 *Pro.ttf*	DynaComware	22,581
Japanese[a]	ヒラギノ角ゴ *ProN W3.otf*	Dainippon Screen	20,325
	ヒラギノ角ゴ *ProN W6.otf*		20,325
	ヒラギノ明朝 *ProN W3.otf*		20,325
	ヒラギノ明朝 *ProN W6.otf*		20,325
Korean	*AppleGothic.ttf*	Apple	19,810

a. These OpenType Japanese fonts are based on the Adobe-Japan1-5 character collection, which includes 20,317 glyphs, along with 8 additional glyphs from the Adobe-Japan1-6 character collection so that they are JIS2004-*savvy*.

Table 10-2. Mac OS X version 10.5 fonts—optional

Language	In /Library/Fonts/	Type foundry	Glyphs
Simplified Chinese	华文宋体.*ttf*	SinoType	32,493
	华文楷体.*ttf*		32,493
	华文仿宋.*ttf*		32,493
Traditional Chinese	儷宋 *Pro.ttf*	DynaComware	22,581
Japanese	ヒラギノ角ゴ *Pro W3.otf*	Dainippon Screen	20,317
	ヒラギノ角ゴ *Pro W6.otf*		20,317
	ヒラギノ明朝 *Pro W3.otf*		20,317
	ヒラギノ明朝 *Pro W6.otf*		20,317
	ヒラギノ角ゴ *StdN W8.otf*		9,498
	ヒラギノ角ゴ *Std W8.otf*		9,354
	ヒラギノ丸ゴ *ProN W4.otf*		20,325
	ヒラギノ丸ゴ *Pro W4.otf*		20,317
Korean	*AppleMyungjo.ttf*	Apple	19,745

As stated earlier in this chapter, it is strongly advised that developers and users not become dependent on the fonts provided by Mac OS X, or for any other OS for that matter.

* The only time circumstance during which it may be necessary to remove a required font is to check to what extent your application depends on the presence of any of those fonts.

The fact that the bundled Chinese and Korean fonts are TrueType is a good indicator that OpenType fonts may one day replace them.

Mac OS X versus Mac OS

In the past, meaning prior to the introduction of Mac OS X, multilingual versions of Mac OS came in two basic flavors or configurations, as follows:

- Fully localized OS
- English OS plus a Language Kit

Beginning with Mac OS version 7.1, which was called *System 7.1*, Apple introduced OS extensions or libraries called *WorldScript I* and *WorldScript II*. Prior to that, comparable functionality came from an OS component called *ScriptManager*. WorldScript I supported non-Latin, one-byte–encoded scripts, such as Arabic, Cyrillic, Hebrew, Thai, and so on. WorldScript II supported non-Latin, two-byte–encoded scripts, such as Chinese, Japanese, and Korean.

In terms of CJKV support through fully localized OSes, Apple provided Mac OS-S (originally named *HanziTalk*) for Simplified Chinese, Mac OS-T (originally named *ChineseTalk*) for Traditional Chinese, Mac OS-J (originally named *KanjiTalk*) for Japanese, and Mac OS-KH (originally named *HangulTalk*) for Korean.* These fully localized OSes provided a localized UI, a basic set of TrueType fonts appropriate for each language and script, and a basic set of input methods. Early versions of KanjiTalk included an input method called 2.0 変換 (*2.0 henkan*), which later became 2.1 変換 (*2.1 henkan*). This was later replaced by ことえり (*kotoeri*), which remains the name of the Japanese input method used in Mac OS X. In terms of encoding, Unicode was not yet supported, so the corresponding legacy encodings were used, meaning Big Five for Traditional Chinese, EUC-CN for Simplified Chinese, Shift-JIS for Japanese, and EUC-KR for Korean.

Apple also introduced a series of Language Kits that effectively added Chinese, Japanese, or Korean support to the standard English version of Mac OS. The UI remained in English, but the ability to input, display, and print CJKV text was enabled through the use of appropriate input methods and fonts. JLK (*Japanese Language Kit*) was the first to be released. Simplified and Traditional Chinese were combined into a single Language Kit called CLK (*Chinese Language Kit*). The last one to be released was KLK (*Korean Language Kit*). These Language Kits were extremely popular with Apple's customers and sold quite well.

More details about the character sets and encodings used by these fully localized versions of Mac OS, including the Language Kits, can be found in Appendixes E and F of this book.

* Interestingly, Mac OS-KH was not developed by Apple, and was instead developed by a company called Elex Computer.

Microsoft Windows Vista

Windows Vista represents the latest and greatest version of Microsoft's Windows OS.* The previous version of their OS, called Windows XP, was very popular, and from a multilingual point of view, was quite robust, with strong Unicode support. Windows Vista ups the ante by providing a fully multilingual OS, without the need to download or install language- or script-specific components. Everything is included.

The very early versions of Windows were simply windowing environments for MS-DOS, thus its name. Starting with Windows 95, it was a genuine OS, and did not require that MS-DOS be running under the hood. Microsoft developed several localized versions of Windows 95 and 98, including five for CJKV locales: Simplified Chinese, Traditional Chinese, Japanese, Korean, and Vietnamese. They did the same for Windows NT, which was the first version of Windows that supported Unicode. Prior to Windows NT, only some components, such as the bundled fonts, supported Unicode. Looking as far back as Windows 95J and 98J, their bundled Japanese fonts had absolutely no trace of Shift-JIS encoding in their 'cmap' tables. They had only Unicode encoding in them. So, how did Shift-JIS–based applications work with such Unicode-encoded fonts? Windows converted between legacy encodings, such as Shift-JIS, and Unicode on-the-fly whenever necessary. This was remarkably seamless, and users really had no idea that conversion between Unicode and legacy encodings was even taking place. And, they didn't need to know or care. Microsoft had effectively insulated users from such issues.

Windows 2000 (followed by Windows XP then Windows Vista) established a new milestone by providing support for PostScript-based fonts at the OS level, eliminating the need for ATM. Adobe Systems licensed to Microsoft the ATM-rendering engine, which gives the OS the ability to handle such fonts, to include the most important PostScript-based font format, specifically OpenType. Because Windows now supports OpenType fonts as first-class citizens, many users and developers alike take this for granted.

David Pogue's *Windows Vista: The Missing Manual* (O'Reilly Media, 2006) is a very informative book for those who are unfamiliar with Windows Vista.† And, Preston Gralla's *Windows Vista in a Nutshell* (O'Reilly Media, 2006) serves as an excellent reference.‡ Developers of internationalized Windows Vista software should consult Microsoft's website for appropriate information.§

Windows Vista versus Windows XP

If Unicode support is included in Windows Vista and Windows XP, how do they differ? Reports indicate that Windows XP is faster and consumes less resources than Windows Vista.

* *http://www.microsoft.com/windows/products/windowsvista/*

† *http://oreilly.com/catalog/9780596528270/*

‡ *http://oreilly.com/catalog/9780596527075/*

§ *http://www.microsoft.com/globaldev/vista/vistahome.mspx*

Given the multilingual features of Windows Vista, it is no surprise that it consumes more resources, because the OS itself includes more resources. For users who do not require multilingual functionality, Windows XP remains a compelling choice.

Still, at some point in the future, Windows XP will go away, and before that, support for Windows XP will be dropped. It is inevitable. Likewise, a new version of Windows is likely to someday replace Windows Vista.

Windows Vista fonts

Because Windows Vista is multilingual in its standard configuration, it means that a large number of CJKV fonts are available without the need to download or install Language Packs. And, because Microsoft needs to address legacy issues, some of the CJKV fonts that were bundled with earlier versions of its OS, such as Windows XP, continue to be bundled with Windows Vista. The fonts that are considered new for Windows Vista have been highlighted in Tables 10-3 through 10-6.

Table 10-3 lists the TrueType and TrueType Collection fonts for Simplified Chinese that are included with Windows Vista, all of which support GB 18030. The last one, *simsunb.ttf*, supports CJK Unified Ideographs Extension B. The first two were developed by Founder, and use Microsoft's *ClearType* technology for improved onscreen display.* The rest were developed for Microsoft by ZhongYi Electronics. English and Chinese menu names are provided, and the fonts introduced in Windows Vista are highlighted.

Table 10-3. TrueType Chinese fonts in Windows Vista—simplified

TrueType font file	Font instances	Glyphs
msyh.ttf	Microsoft YaHei/微软雅黑 (*wéiruǎn yǎhēi*)	29,126
msyhbd.ttf	Microsoft YaHei Bold/微软雅黑 Bold (*wéiruǎn yǎhēi Bold*)	29,126
simfang.ttf	FangSong/仿宋 (*fǎngsòng*)	28,562
simhei.ttf	SimHei/黑体 (*hēitǐ*)	28,562
simkai.ttf	KaiTi/楷体 (*kǎitǐ*)	28,562
simsun.ttc	SimSun/宋体 (*sòngtǐ*) NSimSun/新宋体 (*xīnsòngtǐ*)	28,762
simsunb.ttf	SimSun-ExtB	42,809

For those who are interested in GB 18030-2005 support issues, Microsoft's Simplified Chinese fonts have some interesting history that I'd like to record in these pages; otherwise, it may be forgotten or lost. Prior to Windows Vista, Microsoft provided a GB18030 Support Package for Windows XP users. In addition to providing OS resources that were necessary to support GB 18030, a special version of *simsun.ttc*, named *simsun18030.ttc*, was

* *http://www.microsoft.com/typography/ClearTypeInfo.mspx*

included. It differed from the *simsun.ttc* that was included with the Simplified Chinese version of Windows XP in two primary ways, as follows:

- The menu names in *simsun18030.ttc* included a "-18030" suffix, specifically "SimSun-18030" and "NSimSun-18030" in English, and 宋体-18030 and 新宋体-18030 in Chinese.
- Whereas *simsun.ttc* (on Windows XP) supported only GBK with 22,141 glyphs, *simsun-18030.ttc* supported GB 18030 by including all of the glyphs for CJK Unified Ideographs Extension A, along with those for four of its regional scripts, specifically Mongolian, Tibetan, Uyghur, and Yi, for a total of 30,533 glyphs.

One immediately wonders why the *simsun.ttc* font on Windows Vista, which presumably supports GB 18030, includes 1,771 fewer glyphs. Quite simply, the glyphs for the four GB 18030 regional scripts were removed. In Windows Vista, those glyphs are available in separate fonts. Support for Mongolian and Tibetan, for example, are available in the fonts *monbaiti.ttf* (*Mongolian Baiti*) with 1,824 glyphs and *himalaya.ttf* (*Microsoft Himalaya*) with 1,482 glyphs.

In addition, some of these Windows Vista fonts, such as *simkai.ttf*, included far fewer glyphs in Windows XP, specifically 7,580 glyphs to cover the GB 2312-80 character set. And, they were built by a different company for Microsoft. In the case of this example, the Windows XP version was built by GreatWall Computer. The versions that are included in Windows Vista now support GB 18030, as their glyph complements suggest.

Table 10-4 lists the TrueType and TrueType Collection fonts for Traditional Chinese that are included with Windows Vista. The first three were made for Microsoft by DynaComware, and the last two were made for Microsoft by Monotype Imaging. English and Chinese menu names are provided, and the fonts introduced in Windows Vista are highlighted.

Table 10-4. TrueType Chinese fonts in Windows Vista—traditional

TrueType font file	Font instances	Glyphs
kaiu.ttf	DFKai-SB/標楷體 (*biāo kǎiti*)	22,134
mingliu.ttc	MingLiU/細明體 (*xìmíngtǐ*) PMingLiU/新細明體 (*xīn xìmíngtǐ*) MingLiU_HKSCS/細明體_HKSCS (*xìmíngtǐ_HKSCS*)	33,987
mingliub.ttc	MingLiU-ExtB/細明體-ExtB (*xìmíngtǐ-ExtB*) PMingLiU-ExtB/新細明體-ExtB (*xīn xìmíngtǐ-ExtB*) MingLiU_HKSCS-ExtB/細明體_HKSCS-ExtB (*xìmíngtǐ_HKSCS-ExtB*)	44,857
msjh.ttf	Microsoft JhengHei/微軟正黑體 (*wéiruǎn zhèng hēiti*)	28,969
msjhbd.ttf	Microsoft JhengHei Bold/微軟正黑體 Bold (*wéiruǎn zhèng hēitǐ Bold*)	28,961

Table 10-5 lists the TrueType Collection fonts for Japanese that are bundled with Windows Vista. C&G, Eiichi Kono, and Matthew Carter were involved with the design of the two Meiryo fonts, and Ricoh made the rest. English and Japanese menu names are provided, and the fonts introduced in Windows Vista are highlighted.

Table 10-5. TrueType Japanese fonts in Windows Vista

TrueType font file	Font instances	Glyphs
meiryo.ttc	Meiryo/メイリオ (*meirio*) Meiryo Italic/メイリオ イタリック (*meirio itarikku*)	20,684
meiryob.ttc	Meiryo Bold/メイリオ ボールド (*meirio bōrudo*) Meiryo Bold Italic/メイリオ ボールド イタリック (*meirio bōrudo itarikku*)	20,684
msgothic.ttc	MS Gothic/MS ゴシック (*MS goshikku*) MS PGothic/MS P ゴシック (*MS P goshikku*) MS UI Gothic	22,213
msmincho.ttc	MS Mincho/MS 明朝 (*MS minchō*) MS PMincho/MS P 明朝 (*MS P minchō*)	19,321

As discussed in Chapter 6, the difference between the font instances in *msmincho.ttc* and *msgothic.ttc* is that the ones that include a "P" in their name use proportional glyphs, specifically for Latin characters, kana, punctuation, and some symbols. Even the half-width katakana are proportional. These same TrueType Collection fonts, including their "P" instances, were also bundled as far back as Windows 95J.

Finally, the *name.ID=10* (*Description*) string of the *meiryo.ttc* and *meiryob.ttc* fonts states the following:

> Meiryo is a very versatile modern sans serif type designed to give an exceptionally clean appearance on screen, as well as in print. It is optimized for on-screen reading. The letterforms are generously open and well-proportioned; legible and clear at smaller sizes, and dynamic at larger display sizes. The beauty of this face is that it sets text lines in Japanese with Roman seamlessly and harmoniously. The balanced inter-letter spacing enhances horizontal alignment, facilitating smooth reading flow. Meiryo has a very large character set with Japanese and Roman combined, fully scalable outline technology, making it extremely functional for all aspects of communication and publishing. It is a robust legible typeface yet compact enough to enable tight inter-line spacing which is good for space economy.

In other words, a lot of effort went into the design of the Meiryo fonts in order to make sure that they work well at a variety of point sizes and resolutions.

Table 10-6 lists the TrueType Korean fonts that are bundled with Windows Vista, and because two of them are TrueType Collections, they include multiple font instances—four, to be exact. These TrueType Collections were developed by HanYang. The two new TrueType fonts were developed for Microsoft by Sandoll. Ignoring the presence of "Che" (체 *che*) for the moment, note how each TrueType Collection includes two basic font instances, both of which have different hangul designs, but share the same set of hanja

glyphs. English and Korean menu names are provided, and the fonts introduced in Windows Vista are highlighted.

Table 10-6. TrueType Korean fonts in Windows Vista

TrueType font file	Font instances	Glyphs
batang.ttc	Batang/바탕 (*batang*) BatangChe/바탕체 (*batangche*) Gungsuh/궁서 (*gungseo*) GungsuhChe/궁서체 (*gungseoche*)	39,680
gulim.ttc	Gulim/굴림 (*gullim*) GulimChe/굴림체 (*gullimche*) Dotum/돋움 (*dotum*) DotumChe/돋움체 (*dotumche*)	40,194
malgun.ttf	Malgun Gothic/맑은 고딕 (*malgeun godik*)	12,747
malgunbd.ttf	Malgun Gothic Bold/맑은 고딕 Bold (*malgeun godik Bold*)	12,740

Interestingly, these same TrueType Collections were included with Windows as far back as Windows 95K, though obviously changes to these fonts have taken place between versions of Windows OS, such as to support additional hangul.

There is some interesting history behind these TrueType Korean font instance names, specifically the presence or absence of the apparent suffix "Che." The fact that these font instances exist can be somewhat confusing, so this paragraph may help to understand why they are named so. For example, we can see that there are pairs of related font instances, such as *Batang* and *BatangChe*. In this case, the Batang instance uses proportional Latin glyphs as the default, but the BatangChe instance uses half-width ones instead. The difference in their name, specifically the presence or absence of "Che" (체 *che*), must apparently mean "proportional" or "half-width." Not true. When expressing 체 using hanja, it becomes 體, and we can clearly see that it is simply a common suffix that means "typeface." Versions of these fonts that included half-width Latin glyphs were originally bundled with Windows 3.1K, and included the "Che" suffix in their names. When instances of these fonts that use proportional Latin glyphs by default were created, whoever was in charge of font-naming at Microsoft decided to simply drop the final "Che" (체).

MS-DOS

MS-DOS (*Microsoft Disk Operating System*) is covered in this chapter primarily because it demonstrates how far the software industry has advanced since the glory days of this OS. MS-DOS represents the classic OS for personal computers. Long ago, in the early years of this OS, it was necessary for the computer itself to contain special ROM that contained CJKV fonts and CJKV text-handling routines. That requirement came to an abrupt end with IBM's introduction of DOS J/V, which allowed anyone to install a Japanese-capable

version of MS-DOS onto a non-Japanese IBM PC or compatible computer. IBM DOS J/V processed Shift-JIS encoding internally.

The hardware requirements for IBM DOS J/V, as shown in the following list, are considered amazingly minimal by today's standards, and for many, some aspects of these requirements are unknown, and thankfully shall remain that way:

- PS/55, PC/AT compatible, or PS/2 computer
- VGA, XGA, or PS/55 display adaptor
- 80286 CPU or greater
- A whopping 1 MB of RAM, though 4 MB was recommended

Today's software runs under Windows Vista or other modern OSes, so naturally the number of users of MS-DOS has decreased significantly. In my opinion, the greatest achievement of Microsoft Windows, which was covered earlier in this chapter, is its ability to effectively bridge incompatible hardware. For example, there were countless versions of MS-DOS available, each of which was designed to run under a different underlying architecture. When you purchased software that ran under MS-DOS, you had to be sure that it was designed for the particular version of MS-DOS running on your computer. When you buy Windows software, it will run on Windows regardless of the underlying architecture.

Plan 9

Plan 9—originally designed in the late 1980s by Ken Thompson, Rob Pike, Dave Presotto, and Phil Winter—is an experimental multilingual Unix-based OS under seemingly constant development at Bell Laboratories. It is currently in its fourth edition.* Its four editions were released in 1992, 1995, 2000, and 2002, respectively.

Plan 9's multilingual support is not based on the locale model, but instead simply uses Unicode as its standard character set and the UTF-8 encoding form. No switching between locales is necessary, because all of the characters necessary for many languages, scripts, or regions are supported by Unicode.

Solaris and OpenSolaris

The Unix-based OS developed by Sun Microsystems is called *Solaris*, and there are fully localized versions available, including those for Simplified Chinese, Traditional Chinese, Japanese, and Korean.† There is also an open source version of Solaris called *OpenSolaris*.‡ The latest versions of Solaris and OpenSolaris, of course, support Unicode. Because Solaris and OpenSolaris are based on Unix, their preferred Unicode encoding form is, of

* *http://plan9.bell-labs.com/plan9/*

† *http://www.sun.com/software/solaris/*

‡ *http://www.opensolaris.com/*

course, UTF-8. And, its CJKV locale designators include UTF-8, such as *zh_CN.UTF-8* for Simplified Chinese, *zh_TW.UTF-8* for Traditional Chinese as used in Taiwan, *zh_HK.UTF-8* for Traditional Chinese as used in Hong Kong, *ja_JP.UTF-8* for Japanese, and *ko_KR.UTF-8* for Korean.

Perhaps of historical interest, there are even locales established to support GBK (*zh_CN.GBK*), GB 18030 (*zh_CN.GB18030*), and Hong Kong SCS (*zh_HK.BIG5HK*). Of course, Solaris includes additional locales to support the more common legacy character sets and encodings.

TRON and Chokanji

TRON, an acronym for "The Real-time Operating system Nucleus," was originally architected by Ken Sakamura (坂村健 *sakamura ken*) in Japan in 1984 as a set of interfaces and design guidelines for creating a kernel.* This allows the specification to remain public, yet implementations based upon it can be proprietary. There are many instantiations of TRON, such as those listed in Table 10-7.

Table 10-7. TRON instantiations

Designation	Full Name	Primary Purpose
BTRON	Business TRON	Computers and mobile devices
CTRON	Central and Communications TRON	Mainframe computers and telecommunications
eTRON	Entity and Economy TRON	Security and authentication
ITRON	Industrial TRON	Embedded systems
JTRON	Java TRON	A Java version of ITRON
MTRON	Macro TRON	Network-based control over other architectures
μITRON[a]	Micro Industrial TRON	Small embedded systems

a. Also called MITRON.

Personal Media Corporation in Japan had developed the BTRON OS, which was an instance of TRON. BTRON is now called *Chokanji* (超漢字 *chō kanji*, which is best translated into English as "Super Kanji").† Chokanji is designed to run directly on the same machines that run Windows OS, and it can also run on top of Windows through the use of *VMware Player*.‡

TRON was one of the first OSes to support the JIS X 0212-1990 character set in its entirety. Naturally, TRON additionally supports the JIS X 0213:2004 character set, and much

* *http://www.tron.org/index-e.html*

† *http://www.chokanji.com/*

‡ *http://www.vmware.com/products/player/*

more. Interestingly, TRON has become a very prolific OS, and its ITRON instantiation is widely used in consumer electronics and automobiles.

The most up-to-date information about TRON can be found online* and in *TRONWARE* magazine, published bimonthly in Japanese by Personal Media Corporation.†

Unix

One of the oldest OSes still in use today is Unix. While there are still two primary Unix implementations available, specifically BSD (*Berkeley Standard Distribution*) and System V, countless derivatives have been developed over the years.

Each Unix workstation manufacturer offers its own proprietary Unix variant with manufacturer-specific enhancements. In addition, there are various "free" and "copylefted" versions of Unix around. Naturally, some are more robust or more user-friendly than others.

One aspect of Unix to be careful of—in the context of this book—is the fact that the behavior of some of its utilities changes depending on what "language" is set by the user. This is called the *LANG environment variable*. The most basic LANG setting is "C." Using the C shell, one can set the LANG environment variable as follows:

```
% setenv LANG C
```

When you use common Unix tools when LANG is set to "C," such as *wc*, it predictably calculates the number of words and characters based on the number of bytes in the file.‡ But, when the LANG environment variable is set to a value such as "Japanese," this command instead calculates the number of words and characters based on the number of characters—which may be composed of more than one byte—in the file.

I find the book entitled *Unix Power Tools*, Third Edition (O'Reilly Media, 2002) to be an overall useful Unix reference, with a focus on time-saving tips.§ Arnold Robbins' *Unix in a Nutshell*, Fourth Edition (O'Reilly Media, 2005) is also a must-have reference for those who use Unix.¶

Hybrid Environments

The environments described up until now are considered to be fully functional OSes, running on their own and on what are considered to be their native machines. In other words, they are not add-ons to existing OSes, but rather they wholly replace your current OS with

* *http://tronweb.super-nova.co.jp/homepage.html*

† *http://www.personal-media.co.jp/book/genre/tw.html*

‡ Note that *wc* stands for "word count," not "water closet."

§ *http://oreilly.com/catalog/9780596003302/*

¶ *http://oreilly.com/catalog/9780596100292/*

something entirely new. The software described in this section, however, are considered to be hybrid OSes.

If you are a serious user or developer, I strongly suggest that you use a fully localized OS. It is especially prudent for developers to use a fully localized OS, because that is what the vast majority of your customers are using. If you do not feel that such an investment is right for you, these hybrid OSes may provide an acceptable level of CJKV functionality for your needs. There are also those who need the ability to process CJKV data, but whose language skills are not sufficiently developed to understand a fully localized interface (in other words, its menus or dialogs). For this class of user, these hybrid environments become more enticing.

Boot Camp—Run Windows on Apple Hardware

When Apple decided to abandon the PowerPC processor in favor of an Intel processor, it opened the door to being able to run Windows, as an OS, on their machines, given that Windows uses the same type of processor. Boot Camp, which is now a standard part of Mac OS X, performs this through the use of the *Boot Camp Assistant* application.*

One potential issue is drivers, specifically Windows drivers for the Apple-specific hardware, such as keyboard, trackpad, and so on. Thankfully, Apple provides the necessary drivers. Once Windows is installed, you can simply choose at startup time which OS you wish to use. The only problem is that both OSes cannot run simultaneously—but for many users this is not a disadvantage.

CrossOver Mac—Run Windows Applications on Mac OS X

As long as you are using Mac OS X with an Intel processor, you can use CrossOver Mac to install and run Windows applications.† CodeWeavers, the developers of CrossOver Mac, provides a list of supported Windows applications. Given that there is a trial version available, it is worth exploring if its functionality appeals to you.

GNOME—Linux and Unix

GNOME (*GNU Network Object Model Environment*), developed by the GNOME Foundation, is a desktop environment for Linux and Unix that provides strong multilingual and Unicode support.‡ *GnomeOffice* is the application suite that is available for GNOME, which includes a word-processing application called *AbiWord*.§ GNOME's origins stem from KDE, which is covered next.

* *http://www.apple.com/macosx/features/bootcamp.html*

† *http://www.codeweavers.com/products/cxmac/*

‡ *http://www.gnome.org/*

§ *http://live.gnome.org/GnomeOffice/*

KDE—Linux and Unix

KDE (*K Desktop Environment*) is another desktop environment for Linux and Unix that is worth mentioning.* In addition to its strong multilingual and Unicode support, its application suite, *KOffice*, and the word-processing application that is included, *KWord*, have become very popular.†

VMware Fusion—Run Windows on Mac OS X

Put simply, VMware Fusion allows Windows, as an OS, to run simultaneously with Mac OS X.‡ VMware Fusion creates a Windows virtual machine that is optimized for the Mac OS X machine on which it is installed. Interestingly, the ability to create an OS virtual machine allows VMware Fusion to do the same for other OSes, such as Linux. It is not limited to Windows.

Wine—Run Windows on Unix, Linux, and Other OSes

Wine (an acronym that stands for *Windows Emulator*) is an open source implementation of the Windows API.§ Because applications that run on Windows necessarily use standard Windows APIs to interact with the OS and other applications, Wine acts as a suitable substitute, but without the need to be running a Windows OS.¶

X Window System—Unix

The X Window System (also known as *X11R7*, meaning "X Window System, version 11, Release 7," or simply as *X11*), originally developed by the now disbanded MIT X Consortium, is a GUI windowing system and general multilingual environment that runs on top of Unix that has recently become available for Mac OS X and other platforms and OSes. The X.Org Foundation now develops the X Window System, and the latest instantiation is *X11R7*, meaning version 11, Release 7.** *XFree86*, which is developed by the XFree86 Project, is another popular derivative of the X Windows System that is commonly used with Linux.†† Mac OS X even includes an *X11* application.

Each flavor of Unix had taken its own path towards localization and internationalization, though there is now convergence, thanks in part to Unicode by providing the encoding

* *http://www.kde.org/*

† *http://koffice.org/*

‡ *http://www.vmware.com/products/fusion/*

§ *http://www.winehq.org/*

¶ Speaking of wine, approximately 250 bottles, primarily reds, were consumed during the writing and production of this book.

** *http://www.x.org/*

†† *http://www.xfree86.org/*

architecture, and also to some related standards developed by The Unicode Consortium, specifically the CLDR.*

For more information on X11R7 or on the X Window System in general, I suggest exploring the X.Org Foundation website.

Text Editors

The most basic text-processing utility is clearly the text editor. Text editors allow you to input, manipulate, and save text. The functionality that is provided by a text editor may seem to be very basic, but, believe it or not, there are times when one would choose a text editor over a word processor. One such circumstance is if you are composing CJKV text for transmission by email. Special formatting, such as you would expect from a word processor, does not travel well over email. Still, plain text continues to be the preferred way in which to send email and other text messages. In other words, the characters and formatting to which one is limited when using text editors are precisely what usually travels well through email.

The feature set provided by text editors is limited, yet useful. Most of them come with search and replace functions, and some even allow the user to write complex macros or to use regular expressions. Limitations typically include lack of word wrap, inadequate line breaking, font limitations, font-size limitations, and font-style limitations.

Of the text editors that are currently available, they can be categorized into two basic types:

- Those that require an underlying CJKV-capable OS to correctly handle CJKV data
- Those that provide their own CJKV support independently of the underlying OS

My overall favorite text editor is *GNU Emacs* because there is no dependency on a mouse for any editing command, and its built-in editing commands are extremely powerful and robust and include support for regular expressions. Additionally, a fair number of Mac OS X applications support many of the GNU Emacs keyboard commands, which makes it even more compelling for me. I had been using its multilingual version, called *Mule*, on Unix for years. Mule's multilingual functionality was integrated back into GNU Emacs (as of version 20). I also used a Japanese-capable version of Emacs designed for Mac OS called *Nitemacs*. Now I use GNU Emacs on Mac OS X through the use of its *Terminal* application. If you haven't yet explored Emacs and its multilingual variants, I encourage you to do so. And, if you are a fan of *vi* as a text editor, there are also CJKV-capable versions available.

I would also like to point out that many contemporary text editors effectively eliminate the need for dedicated code conversion tools in some important contexts, because they

* *http://www.unicode.org/cldr/*

provide import and export facilities that support a broad set of character sets and encodings, the most important of which are those for Unicode.

In Chapter 5, specifically in the section entitled "Mobile Keyboard Arrays," I stated that several books that were published in Japan were written through the use of a cell phone. Obviously, a cell phone was not used to typeset the books, but instead served as a handheld or extremely portable text editor. The fact that cell phone and other mobile devices are treated as "always with you" devices means that text can be entered or edited on a whim.* In Japan, it is common practice for people to have their cell phones with them when they sleep.

Mac OS X Text Editors

In the past, before the days of Mac OS X, nearly all CJKV-capable text editors for Mac OS required that one use either a fully localized version of Mac OS or else an appropriate Language Kit, such as JLK. And, most Japanese text editors supported ISO-2022-JP and EUC-JP encodings only for import or export purposes, and processed Shift-JIS encoding internally because that is what Mac OS-J (and Mac OS plus JLK) processed internally. This has changed, obviously for the better, with Mac OS X. Unicode is more broadly supported, but legacy encodings are still supported for import and export purposes.

For those who spend significant time in the *Terminal* application, like me, standard Unix text editors are available, such as *emacs* and *vi*, and their level of multilingual support has increased with each subsequent version of Mac OS X. Though, instead of *vi*, I suggest using *Vim*.

TextEdit

TextEdit is the standard text editor for Mac OS X. It evolved from *SimpleText*, which was used on Mac OS. Unlike other text editors, TextEdit allows the user to toggle between plain and rich text (aka RTF, meaning *Rich Text Format*) modes, and it even provides a keyboard shortcut for this purpose: *Shift-Command-T*. There is even some limited OpenType feature support, such as the ability to use old forms of ideographs through the use of the 'trad' GSUB feature. Although this works in rich text mode, it is obviously not preserved in the plain text representation.

In addition, I discovered early on that many of the keyboard commands used by Emacs function the same in TextEdit, so document navigation can be done as in Emacs, instead of using the cursor keys, mouse, or trackpad.

Interestingly, TextEdit can open Microsoft Word documents, in rich text mode, of course. This is mighty convenient, especially if someone sends to you a Microsoft Word document, but you don't have that application installed.

* But hopefully not while driving a motor vehicle for which there are safety issues and concerns.

BBEdit

Programmers and developers tend to prefer BBEdit due to its powerful features, of which there are far too many to list here.* It sufficient to state that the latest version of BBEdit supports Unicode, meaning that its multilingual support, at least from the CJKV perspective, is adequate. In terms of Unicode encoding forms, both UTF-8 and UTF-16 are fully supported.

Jedit X

Jedit X is the Mac OS X version of a popular Japanese text editor that was originally called *Jedit*.† Jedit X provides useful editing facilities, and the ability to toggle between plain text and rich text modes, along with import and export facilities for legacy encodings, such as ISO-2022-JP, EUC-JP, and Shift-JIS. Of course, Jedit X supports Unicode. Its name suggests that it is tailored for Japanese use.

Windows Text Editors

There is an incredible number of CJKV-capable text editors available for Windows, and unlike those for Mac OS X, many do not require underlying CJKV support in the OS, though with the availability of Language Packs for Windows XP, and the broad multilingual support in Windows Vista, this is less of a concern these days.

Popular Japanese-capable Windows text editors include *Hidemaru Editor* (秀丸エディタ *hidemaru edita*),‡ *MIFES* (マイフェス *maifesu*; a Linux version is also available),§ and *WZ Editor* (a mobile version is also available).¶ Some of these text editors offer extremely powerful text-manipulation facilities, often through the use of regular expressions or rectangular selection.

Notepad

Notepad is the bare-bones text editor that is include with Windows OS that supports only plain text. Its functionality is very basic, but there are obviously times when plain text is preferred over stylized text, and the use of Notepad helps to enforce this paradigm.

Naturally, Unicode is used by Notepad, meaning it can open and save text files that are encoded according to Unicode.

* *http://www.barebones.com/products/bbedit/*

† *http://www.artman21.com/jp/jedit_x/*

‡ *http://hide.maruo.co.jp/software/hidemaru.html*

§ *http://www.megasoft.co.jp/mifes/*

¶ *http://www.villagecenter.co.jp/soft/wz50/*

WordPad

To some extent, WordPad can be treated as a word processor, because it supports rich text, meaning that font attributes can be set at the character level, as opposed to at the document level, and there is more control over the layout of the text. Some people treat it as a simplified version of Microsoft Word.

BabelPad

BabelPad, developed by Andrew West, deserves special mention here, not because it is free and not because it is a text editor that supports Unicode in general and CJKV text, but that it additionally supports a large number of complex scripts and includes a large number of input methods.* For readers of this book, I would like to point out that BabelPad supports the regional scripts set forth in GB 18030-2005, specifically Korean, Mongolian, Tai Le, Tibetan, Uyghur, and Yi. I should also point out that supporting complex scripts such as Mongolian and Tibetan is no small feat and is done by calling *Uniscribe*, which is Windows' complex script text layout engine. There is a reason why they are referred to as complex scripts.

BabelPad supports a large number of encodings, including the various encoding forms of Unicode, whether it is for opening or saving text files.

In addition to its text-editing features, BabelPad includes many useful tools and utilities, such as character maps, character lookup facilities, font analysis facilities, and so on.

Vietnamese Text Editing

In the past, before Unicode was widely supported, Vietnamese text editing—and word processing—required the use of specialized fonts, along with applications that knew how to support them. I am speaking, of course, about the Latin-based Vietnamese writing system called Quốc ngữ. In many ways, Unicode has trivialized the extent to which Quốc ngữ can be used. Many fonts, such as Minion Pro and Myriad Pro that are used for this book, now include a complete set of glyphs that support Quốc ngữ. As long as the application supports Unicode, and as long as the selected font includes the appropriate glyphs, Vietnamese support is facilitated. Of course, Vietnamese input methods then become an issue, but it is comforting to know that Mac OS X and Windows Vista provide appropriate input methods for Vietnamese.

VietPad—Cross-platform Vietnamese Unicode text editor

Although many of the text editors already described support Vietnamese, by virtue of Unicode and the extent to which Vietnamese is supported by the OSes, VietPad is unique in that it is truly cross-platform, because it is written in Java.† Multiple Vietnamese input

* *http://www.babelstone.co.uk/Software/BabelPad.html*

† *http://vietpad.sourceforge.net/*

methods are provided by VietPad, and it naturally uses Unicode to represent Vietnamese text.

Emacs and GNU Emacs

Described in this section are several freely available variants of Emacs and GNU Emacs. Emacs is the name of a text editor that has been ported to many OSes, and GNU Emacs is the most widely used version. If you have more than one working environment, you can use the software described in this section to have similar text-editing features and functionality across them.

All of these Emacs and GNU Emacs variants depend on a CJKV-capable environment for displaying CJKV characters on the screen—they handle only the internal manipulation of CJKV character codes, whether they are based on a Unicode encoding form or a legacy encoding method.

Most variants of Emacs or GNU Emacs can be extensively customized by the user. In fact, Emacs or GNU Emacs can also constitute a complete working environment, at least on most Unix systems—email can be sent and received, source code can be compiled, and so on. Customizing is usually done by adding entries to its configuration file called *.emacs* ("dot" emacs) or by writing Emacs LISP programs. Extensive tutorials are also included in the complete GNU Emacs distribution.

There are a large number of Emacs and GNU Emacs variants available for a number of OSes. While I describe only the most widely used versions, there are also Emacs variants, such as *Demacs*, *Han Emacs*, *Mg*, *Ng*, *Nitemacs*, and *SaLLY*. At one time I was particularly fond of *Nitemacs*, because it ran on Mac OS. Now, I use GNU Emacs on Mac OS X through the use of its *Terminal* application.

For more information on GNU Emacs in general, I suggest *Learning GNU Emacs*, Third Edition (O'Reilly Media, 2004),* by Debra Cameron et al., and Richard Stallman's *GNU Emacs Manual*, Sixteenth Edition (Free Software Foundation, 2007).† The latter is likely to be most useful because it documents the multilingual features and functionality that have been integrated into GNU Emacs since version 20. More about this in the next section.

GNU Emacs, NEmacs, and Mule

GNU Emacs and its variants are developed by the Free Software Foundation‡ and are copylefted software distributed under the terms of the *GNU General Public License*.§ The terms of the GNU General Public License protect software from being exploited for commercial use.

* *http://oreilly.com/catalog/9780596006488/*

† *http://www.gnu.org/software/emacs/manual/emacs.html*

‡ *http://www.gnu.org/*

§ *http://www.gnu.org/copyleft/gpl.html*

There have been two major CJKV-capable GNU Emacs developments over the years: NEmacs and Mule, each of which is installed as a series of patches to the GNU Emacs source code. NEmacs stands for *Nihongo Emacs* (*Nihongo*, written 日本語, is the Japanese word that means "Japanese [language]"). Mule stands for *MULtilingual enhancement to GNU Emacs*. While it may be obvious by their names, NEmacs was a Japanese-only version of GNU Emacs, whereas Mule provided support for a large number of languages, including Chinese, Japanese, and Korean.

Both NEmacs and Mule were developed by a core team made up of Ken'ichi Handa (半田剣一 *handa ken'ichi*), Satoru Tomura (戸村哲 *tomura satoru*), and Mikiko Nishikimi (錦見美貴子 *nishikimi mikiko*). Mule was also one of the very first programs to support the characters and encoding methods for the JIS X 0212-1990 character set standard. Both ISO-2022-JP-2 and EUC-JP encodings were supported for the encoding of this character set in Mule. GNU Emacs version 20 and greater incorporates all functionality of Mule. In other words, the Mule extensions were integrated back into the standard GNU Emacs distribution as of version 20.

With GNU Emacs, it is possible to use any encoding within any given language environment. Changing the language environment affects only the following behaviors:

- Which encoding is used as the default
- The priority of encodings during automatic encoding detection

Another advantage of GNU Emacs, particularly for those who use Mac OS X, is that many of its keyboard commands (such as for document navigation) function the same in other applications. TextEdit, the standard text editor for Mac OS X, is but one example.

vi and Vim

The vi text editor is another popular editing environment, and several CJKV-capable versions have been developed over the years. Like GNU Emacs, it has Unix heritage, but unlike GNU Emacs, it is modal in operation. Ignoring the various CJKV-capable derivatives, there are several vi clones, such as *Calvin*, *Elvis*, *Nvi*, *Stevie*, *VILE*, *Vim*, *WinVi*, and *xvi*. Note that all of them have "vi" somewhere in their names. Anyway, unlike GNU Emacs, which had its multilingual extensions integrated back into its own source code, the original vi editor still exhibits weak multilingual support and is effectively an ASCII-based text editor. Vim (*Vi iMproved*) deserves special mention here, because it includes support for Unicode, meaning that it provides a much better multilingual editing environment.* When compared to vi, Vim is also available for more OSes, making it more compelling than vi. For example, it is included with Mac OS X and accessible through its *Terminal* application.

* *http://www.vim.org/*

Prior to Vim, several CJKV-capable versions of vi were developed, most of which were based upon vi clones. For example, Jstevie and jelvis were Japanese-enabled versions of Stevie and Elvis, respectively. There was even a Korean-enabled version of Elvis called Hangul Elvis. Also, *nvi-m17n* was a multilingual version of Nvi.

For those who wish to learn more about vi, and especially about Vim, and how to use either of them, I suggest reading *Learning the vi and Vim Editors*, Seventh Edition (O'Reilly Media, 2008), by Arnold Robbins et al.*

Word Processors

Word processors or word-processing applications (ワードプロセッサ *wādopurosessa* or ワープロ *wāpuro* in Japanese) represent the next step up from text editors in terms of features and functionality. Supported features typically include character- or paragraph-based font selection, multiple font styles, character-based point-size adjustment, line-breaking, various types or levels of justification, CJKV-specific line-breaking capability, somewhat complex formatting capabilities, the ability to use tabs, and sometimes a basic set of graphics-building tools. Some word-processing applications even rival the features of some page-layout applications, especially when it comes to the ability to manipulate text. Some even include functionality that allows them to perform many of the tasks expected to be done by page-layout applications.

Of the large number of word-processing applications on the market that provide CJKV or multilingual functionality, there are those that depend on an underlying CJKV-capable or Unicode-enabled OS, and there are those that establish their own environment, making them usable on English-only OSes. Given the current state of multilingual and Unicode support in today's OSes, the former are clearly of more importance.

Because there are far more word-processing applications available today than can conceivably be described in a book such as this, this section thus provides a mere sampling of what is currently available on the market.

In addition, I should point out that many of these word-processing applications are included as part of software suites. *Microsoft Word*, for example, is included in *Microsoft Office*. While the sections that follow mention specific software suites, some software suites deserve to be mentioned here, such as *OpenOffice*,† which is an open source version of Microsoft Office that is available for many OSes, and *NeoOffice*‡ and *OpenOSX Office*,§ both of which are based on OpenOffice and designed for Mac OS X.

* *http://oreilly.com/catalog/9780596529833/*

† *http://www.openoffice.org/*

‡ *http://www.neooffice.org/*

§ *http://www.openosx.com/*

AbiWord

AbiWord, designed to be an alternative to Microsoft Word, is an open source multilingual word processing application that is available for Windows, and is included with some Linux distributions.* *GnomeOffice* is a software suite that includes AbiWord as one of its many applications. One of the goals of AbiWord is for it to become a cross-platform word processing application.

Haansoft Hangul—Microsoft Windows

The Korean word-processing application called Haansoft Hangul (formerly HWP, which stood for *Hangul Word Processor*), developed by a company called Haansoft (formerly Hangul & Computer), provides full Korean functionality, but does not require an underlying Korean version of the Windows OS.†

Ichitaro—Microsoft Windows

JustSystems' Ichitaro (一太郎 *ichitarō*) is one of the most widely used Japanese word-processing applications and is currently available only for Windows.‡ It also one of the first Japanese word processors that is now completely Unicode-based internally. Their engineers have effectively insulated the users from ever knowing this.§

One significant benefit of Ichitaro is that it is bundled with *ATOK*, which is JustSystems' powerful input method. Ichitaro is also included as part of *JUST Suite*, which is an application suite comparable in functionality to Microsoft Office.

KWord

KWord is the frame-based word-processing application that is included with *KOffice*, the application suite for KDE.¶ Because KWord is frame-based, it is suitable for some tasks that are normally suited for more complex page-layout applications.

Microsoft Word—Microsoft Windows and Mac OS X

One of the longest-selling word-processing applications is clearly Microsoft Word, which is also referred to as *MS Word* or simply *Word*.** It is one of the applications included with *Microsoft Office*,†† though it is also sold individually as a point product. The advantage

* *http://www.abisource.com/*

† *http://www.haansoft.com/hnc/haansoft_en/product/Haansoft_Hangul2007.jsp*

‡ *http://www.ichitaro.com/*

§ Well, at least until Ichitaro users read this page.

¶ *http://koffice.org/kword/*

** *http://office.microsoft.com/word/*

†† *http://office.microsoft.com/*

of purchasing Microsoft Office, as opposed to simply Microsoft Word, beyond the mere savings and especially for Windows XP users, is that Microsoft makes Language Packs available to those users who are running Microsoft Office. This means, for example, that one can purchase the English version of Microsoft Office, then download the appropriate Language Pack to enable the language or script of your choice. For Windows Vista users, there is only the financial benefit of Microsoft Office.

At least for the Mac OS X version of Microsoft Word, there are still differences between the English and localized versions of the application. The Japanese version, for example, includes the ability to perform vertical layout. I have yet to discover this functionality in the English version of Microsoft Word, at least for the Mac OS X version.

Microsoft Word 97 (and later), Microsoft Excel* 97 (and later), and Microsoft PowerPoint† 97 (and later) are Unicode-enabled. The Language Packs that Microsoft makes available for Microsoft Office customers who are using Windows XP can effectively CJKV-enable these and other Windows applications.

Nisus Writer—Mac OS X

Nisus Writer, currently available in Express and Pro flavors, is one of the oldest multilingual word-processing applications.‡ The latest versions of Nisus Writer embraces Unicode and runs on Mac OS X. A trial version of Nisus Writer is available, and I encourage you to try it out.

Looking back to its earlier days, when it ran on Mac OS, Nisus Writer was one of the very few word-processing applications that could adequately handle CJKV text, although it was not designed for CJKV handling *per se*. An effort began to localize and tailor Nisus Writer to the Japanese market, and to add extra features and functionality specific to Japanese text handling. The resulting application was called *SoloWriter*, but wasn't very successful, perhaps for sheer marketing reasons. In any case, Nisus Software continued to develop Nisus Writer, and it is still considered one of the top-rated multilingual word-processing applications.

Many of Nisus Writer's most noteworthy features are not necessarily specific to handling CJKV text, such as the ability to open a wide variety of word processor formats, regular expression support, and so on. It has also been one of the easiest multilingual word-processing applications to obtain outside of the CJKV locales.

Perhaps of interest is the fact that Nisus Writer, running on Mac OS, was used to prepare the manuscript for *Understanding Japanese Information Processing* (O'Reilly Media, 1993), which was the predecessor of the first edition of this book.

* *http://office.microsoft.com/excel/*

† *http://office.microsoft.com/powerpoint/*

‡ *http://www.nisus.com/*

NJStar Chinese/Japanese WP—Microsoft Windows

NJStar (南极星/南極星 *nánjíxīng*), which began as applications written by Hongbo Ni (倪鸿波 *ní hóngbō*), who also went on to found NJStar Software Corporation, was originally designed as a Chinese word-processing application, but was modified to handle Japanese in a dedicated Japanese version.* This means that there are now Chinese and Japanese versions of NJStar WP available, called *NJStar Chinese WP* and *NJStar Japanese WP*, respectively. Its interface includes pull-down menus and mouse support, and the ability to use an English UI. It also supports a rich set of editing functions, to include multiple-file editing, undo, two-direction fast search, flexible search/replace, and extensive block manipulations. NJStar WP does not require a localized OS because it establishes its own Chinese or Japanese environment, as appropriate. The latest versions are compatible with Windows Vista.

The lowest-price version of NJStar WP, which is the Basic, or shareware, version, includes only bitmapped fonts, but the Pro and Pro Plus versions include TrueType fonts that can be used for better-quality printing. NJStar Japanese WP includes two TrueType Japanese fonts in the Pro version, and four in the Pro Plus version. NJStar Chinese WP supports both Simplified and Traditional Chinese, and is available in a Pro version with four TrueType Chinese fonts (two for Simplified Chinese, and two for Traditional Chinese), along with several flavors or levels of Pro Plus that differ in the number of additional TrueType Chinese fonts that are included.

NJStar Chinese WP supports EUC-CN, Big Five, HZ, and Unicode encodings for import and export purposes, and provides the user with nearly 20 Chinese input methods, some of which were described in Chapter 5 of this book. In fact, the NJStar Chinese WP manual itself is useful for understanding these Chinese input methods. Also provided is instant access to a Chinese-English dictionary.

NJStar Japanese WP supports ISO-2022-JP, EUC-JP, Shift-JIS, and Unicode encodings for import and export purposes, includes approximately 10 Japanese input methods, and provides users with instant access to a Japanese-English dictionary, specifically Jim Breen's EDICT, ENAMDICT, and KANJIDIC dictionary files. These and other dictionary files are covered in Chapter 11.

For those who wish to run NJStar WP on other OSes, Wine makes it possible for Linux users, and CrossOver Mac makes it possible for Mac OS X user. These were covered early in this chapter.

* *http://www.njstar.com/*

Pages—Mac OS X

Pages is the name of the powerful word-processing application that is included with Apple's *iWork* application suite.* For those who use Mac OS X, it deserves exploration. Given the broad multilingual support in Mac OS X in general, the same can be said for Pages.

Online Word Processors

There has been a recent trend to move applications from the desktop, as applications that are tied to a specific OS or architecture, to those that are accessed online, through the use of web browsers or similar clients. While the portal through which such applications are accessed may be tied to a specific platform, out of obvious necessity, the applications themselves are not. Such applications effectively remove the notions of "platform" and "OS" from the equation.

One of the touted benefits of online word processors is the ability to collaborate. Another obvious benefit is the ability to work from anywhere, on any machine that has web access. For developers and users alike, another compelling benefit of online word processors—which is really a benefit of any online application—is the ability to have much shorter release cycles. In other words, bug fixes, updates, and new versions get into customers' hands much more quickly. This is a very good thing.

Adobe Buzzword

Adobe Systems has developed an online word processor called Adobe Buzzword.† It is unique among other online word processors in that it performs genuine WYSIWYG text layout, meaning that users have at their fingertips a precise layout and line-breaking paradigm without being tied to a specific platform or OS. The current version of Adobe Buzzword supports high-quality English-language layout, and future versions are expected to support many more languages and scripts, including Japanese, all with the same consistent, true-to-print layout facilities. Of course, these multilingual enhancements will be done in the context of Unicode.

Given the ability to rapidly release updates, the multilingual functionality of Adobe Buzzword is likely to advance very quickly.

Google Docs

Google has invested heavily into its internationalization efforts, and it really shows when you explore their website and offerings such as Google Docs, which is their online word-processing solution.‡ In terms of Unicode support, the UTF-8 encoding form is used.

* *http://www.apple.com/iwork/pages/*

† *http://www.adobe.com/acom/buzzword/*

‡ *http://docs.google.com/*

Google Docs is more than a word processor. It provides other functionality as well, such as the ability to make spreadsheets.

Advice to Developers

First and foremost, and in case this message hasn't been clear throughout this book, embrace Unicode. Much of the progress that has been made in the area of software internationalization is directly due to Unicode and the extent to which it is supported in today's OSes and applications.

When developing OSes and applications, in addition to supporting Unicode, bear in mind locale-specific practices and conventions, and to the extent possible, build them into your software. Each modern OS has its own rich support for locale conventions. CLDR now provides a rich and consistent framework for providing and specifying locale information, which makes it possible to trigger locale-specific behaviors in a genuinely cross-platform manner. Deciding between using OS resources and CLDR is not necessarily a simple task, because each one has its pros and cons. Such a discussion is clearly outside the scope of this book.

And, as mentioned in Chapter 6 and earlier in this chapter, bear in mind that the fonts that are bundled or otherwise included in an OS are subject to change over time. In other words, their names, their glyph complements, and even the fonts themselves, may change with little or no warning. Instead of referencing specific fonts by name, today's OSes provide convenient APIs for referencing fonts in more generic ways. Mac OS X, for example, provides APIs that effectively mean "get system font" and "get application font" that serve this purpose, and are highly recommended for application developers to use. Explore the OS APIs and bear in mind that new functionality is often provided through the use of new APIs, as opposed to modifying existing APIs.

CHAPTER 11

Dictionaries and Dictionary Software

Everyone, or nearly everyone, reading this book may be wondering why there is even mention of—let alone a dedicated chapter for—dictionaries in a book about information processing. The usefulness of dictionaries should become painfully obvious once you give it some thought: dictionaries, in a variety of forms, provide users and developers the ability to locate information about characters and words. For most CJKV locales, this usually entails meaningful sequences of two of more ideographs in word dictionaries, and sometimes for individual ideographs in character dictionaries. Armed with the ability to rapidly and conveniently access information about words and the ideographs that are used to compose them—or to be able to access this type of information at all—makes using and developing software a much more pleasant experience.

Dictionaries play an especially important role during text input. As covered in Chapter 5, the ability to enter ideographs is governed by the use of appropriate dictionaries, which provide input methods with the ability to convert transliterated or other native scripts into sequences of one or more ideographs.

Word and character dictionaries come in many forms. Traditionally, they are printed on paper and bound into a book-like format. Some contain so much information that they are published as several volumes. Given that we're discussing computers in the context of this book, machine-readable, electronic, or "digital" dictionaries are of particular interest to its audience. These traditional dictionaries still play an important role today, such as for researching the history and development of ideographs and the words that use them.

Applications, such as those that perform machine-aided translation or help in the study of a language, are also related to dictionaries, because they make use of them. Some very basic information about these related topics is covered near the end of this chapter.

Reading this chapter is clearly not enough. The number of dictionaries that are available today is genuinely mind-boggling. I provide names, descriptions, and URLs for some of those that I consider to be key or useful. Some dictionaries change on a daily basis.

Dive into this chapter with the understanding that there is more to learn elsewhere. Explore the URLs that are provided and take advantage of today's search engines.

Ideograph Dictionary Indexes

In order to begin taking advantage of the wealth of information contained within ideograph dictionaries, you first need to know how to use them. The typical ideograph dictionary contains two main parts:

Main entries
: Provide all the useful information about the ideograph in one convenient location

One or more indexes
: Provide more convenient ways with which to find the target main entry

Needless to say, the more indexes a dictionary contains, the greater chance that you can find the target main entry—many ideographs are difficult to find in dictionaries for a variety of reasons, such as due to rare readings, difficult-to-determine indexing radicals, ambiguous stroke counts, and so on.

The main entries in ideograph dictionaries are typically ordered in a standard way, such as by radical plus the number of remaining strokes, total number of strokes, or reading. The most common of these is by indexing radical plus the number of remaining strokes. In fact, there is usually a simple radical index right inside the front (or back) cover of most ideograph dictionaries to aid in quick lookup.

The following sections describe these indexing methods in greater detail and provide tips for faster lookup.

Reading Index

One of the most effective ways to locate an ideograph in a dictionary is by its reading. Once you locate the appropriate reading in the reading index, you will then usually find the ideographs ordered by radical, number of strokes, or other well-established criteria.

However, there are some characteristics to bear in mind when making use of reading indexes detailed as follows:

- Many ideographs have multiple readings, which means that some reading indexes may include multiple entries for such ideographs.
- Not all ideographs have well-known or well-understood readings, which means that some reading indexes may not contain a single entry for such ideographs.

The fact that ideographs often have multiple readings actually can work to your advantage. In the case of Japanese, for example, kanji with multiple readings often have a reading that is fairly frequent—that is, many other kanji share the same reading, and are thus considered to be homophones—and one that is rather infrequent. In short, searching for a ideograph using an infrequent reading is typically faster because there are less homophones that are presented as candidates. Table 11-1 lists several kanji, along with their frequent and infrequent readings.

Table 11-1. Kanji readings of different frequency

Kanji	Frequent reading	Infrequent reading
剣	ケン *ken*	つるぎ *tsurugi*
中	チュウ *chū*	なか *naka*
生	セイ *sei*, ショウ *shō*	なま *nama*
犬	ケン *ken*	いぬ *inu*

Many rare ideographs do not have very well-known readings. But, even if you happen to know a reading for a particular ideograph, that doesn't necessarily mean that all reading indexes will include an entry for it. This is when other indexes come into play, such as those that denote structure.

Some character set standards order ideographs according to reading. Level 1 of GB 2312-80 and JIS X 0208:1997 are good examples, as is KS X 1001:2004. Interestingly, and as pointed out in Chapter 3, KS X 1001:2004 includes multiple instances of over 200 ideographs because they have multiple readings.

Radical Index

Ideographs are most frequently categorized by radical. As discussed in Chapters 2 and 5, indexing radicals and radical-like components are the basic building blocks of ideographs;* they thus serve as a way to organize and categorize ideographs. Some character set standards order ideographs according to their indexing radical, such as Level 2 of GB 2312-80 and JIS X 0208:1997, JIS X 0212-1990, JIS X 0213:2004, and each of Unicode's CJK Unified Ideograph blocks.

The most commonly used set of indexing radicals today is the classic set of 214. This set of indexing radicals, along with its classification system, was established in the classic ideograph dictionary entitled 康熙字典 (*kāngxī zìdiǎn*), which is nearly 300 years old. The radical-indexing scheme that was set forth by this dictionary is widely used in Japan, Korea, and Taiwan. China, because of its simplified hanzi, uses a reduced set of 186 radicals. Both of these common radical systems are listed in appendixes of this book. The classic set of 214 are listed in Appendix J, and the reduced set of 186 are in Appendix G.

Radical indexes require that the user first identify what is considered to be the indexing radical. In most cases, this is simple. For some ideographs, determining the indexing radical can be a challenge. Consider the two ideographs 龘 (U+56B2) and 鬱 (U+61FF). The indexing radical of the former is 口 (30), though 亠 (8) and 子 (39) seem as plausible or possible. Likewise, the indexing radical of the latter is 心 (61), though 士 (33) and 豆 (151)

* Actually, the individual strokes that are used to compose these components are the true building blocks, but that's a different topic.

seem to be as reasonable. In any case, once the indexing radical has been determined, the next task is to count the remaining strokes in the ideograph. Table 11-2 lists several hanzi, along with their indexing radical and remaining strokes, using the two common indexing radical systems: the classic 214 system and the simplified 186 system.

Table 11-2. Identifying hanzi using different radical systems

Hanzi	214 radicals	186 radicals	Remaining elements and residual strokes
剑	刀 (18)	刂 (11)[a]	佥, 7 strokes
汉	水 (85)	氵 (55)[b]	又, 2 strokes
林	木 (75)	木 (72)	木, 4 strokes
边	辵 (162)	辶 (57)[c]	力, 2 strokes
语	言 (149)	讠 (22)[d]	吾, 7 strokes

a. A variant form of 刀; used as a right-side component.

b. A variant form of 水; used as a left-side component.

c. A variant form of 辵.

d. The simplified form of 言.

Appendix G provides a radical index for Level 2 of GB 2312-80, and Appendix J does the same for Level 2 of JIS X 0208:1997, along with JIS X 0212-1990. While the two JIS standards conform to the set of 214 classical radicals, Level 2 of GB 2312-80 uses the reduced set of 186 based on the principles of simplified hanzi.

Some ideograph dictionaries that use a reduced set of radicals provide a table that serves as a cross-reference to equivalent radicals in the set of 214 classical radicals. Mark Spahn and Wolfgang Hadamitzky's *The Kanji Dictionary* (Charles E. Tuttle Company, 1996) uses a significantly reduced set of only 79 indexing radicals, and does so quite effectively. Furthermore, some dictionaries provide special indexes that list ideographs whose indexing radical is not very obvious, or provide cross-references for such cases.

Stroke Count Index

I consider this type of index to be the least effective of the three principal ways to locate an ideograph. The first two methods discussed in this chapter, specifically access by reading and radical indexes, are almost always more effective in quickly locating ideographs.

A stroke count index first separates ideographs into groups whose common characteristic is their total number of strokes. The ideographs within each stroke-count group then need to be ordered—otherwise, you'd sometimes need to scan several hundred (or thousand)

characters to find the one you're looking for. The most common ordering within each stroke-count group is by indexing radical.*

Many character dictionaries published in China use a very useful and effective method for ordering hanzi within each stroke-count group. The characters are ordered according to the shape of their first strokes. This system employs a total of five such strokes, indicated in Table 11-3, along with their names.

Table 11-3. Five basic strokes

Stroke	Stroke names
一	横 *héng*
丨	竖/竪 *shù*
丿	撇 *piě*
丶	点/點 *diǎn*
乙	折 *zhé* or 横折 *héngzhé*

The ideograph 中 can thus be described using the following four strokes, listed in the order in which they are used to draw the character: 丨, 乙, 一, and 丨. The convention in the stroke-count indexes of these dictionaries is to use a single stroke to order ideographs whose stroke counts are either small or great (because there are not that many characters under such stroke counts), but to use two strokes for those stroke counts that include a large number of candidate ideographs. This is obviously done for the sake of searching efficiency.

Note that some ideographs are known to have ambiguous stroke counts, typically because there are multiple ways to write their indexing radical or radical-like components. Table 11-4 lists some indexing radicals that have ambiguous stroke counts, either due to variations of the indexing radical itself or different ways in which to write the indexing radical.

Table 11-4. Indexing radicals with ambiguous stroke counts

Indexing radical	Stroke counts	Reason for discrepancy
臣	6 or 7	Different ways to write indexing radical
食	9 or 10	Different ways to write indexing radical
舛	6 or 7	Different ways to write indexing radical

* This is sort of the reverse of radical lookup whereby you first find the radical, then you count the number of strokes or the number of remaining strokes, depending on how the dictionary is designed.

Table 11-4. Indexing radicals with ambiguous stroke counts

Indexing radical	Stroke counts	Reason for discrepancy
阝	2 or 3	Different ways to write indexing radical
辶, 辶	3 or 4	Different number of strokes
礻, 示	4 or 5	Variations

Good dictionaries include ideographs with ambiguous stroke counts in all applicable stroke-count groups or, at the very least, provide adequate cross-references.

Some character set standards, specifically Big Five and the first seven planes of CNS 11643-2007, order ideographs by total number of strokes, then by indexing radical.

Other Indexes

While reading, radical, and stroke count indexes are the most commonly found in ideograph dictionaries, they are not necessarily the most efficient in terms of lookup speed. Other useful or proven indexes include Jack Halpern's SKIP, the Four Corner method, and the character codes themselves.

SKIP

One particularly efficient ideograph index method is known as SKIP (*System of Kanji Indexing by Patterns*), developed and patented by Jack Halpern (春遍雀來 *harupen jakku*), implemented in his own dictionaries for the ordering of the main entries, and included (by permission) in Jim Breen's KANJIDIC file (described later in this chapter). In fact, KANJIDIC includes SKIP codes for all JIS X 0208:1997 kanji, not only those that appear in Jack Halpern's *New Japanese-English Character Dictionary* (Kenkyusha, 1990; NTC, 1993). Jim Breen's kanji dictionary files also include SKIP codes. Jack Halpern maintains a useful web page that provides details about SKIP.*

A SKIP code consists of three numbers separated by a hyphen. The first part represents the basic pattern. SKIP identifies four basic patterns, and numbers them accordingly:

1. Left-right
2. Top-bottom
3. Enclosure
4. Solid

For the first three SKIP patterns, the second number in the SKIP code represents the number of strokes in the pattern (Pattern 1: left; Pattern 2: top; Pattern 3: the enclosure).

* *http://www.kanji.org/*

The third and final number in the SKIP code represents the remaining number of strokes. Table 11-5 provides three examples.

Table 11-5. Example SKIP codes

Kanji	SKIP code	Description
剣	1-8-2	First pattern, eight strokes in pattern, two remaining strokes
書	2-6-4	Second pattern, six strokes in pattern, four remaining strokes
国	3-3-5	Third pattern, three strokes in pattern, five remaining strokes

The fourth pattern, Solid, has four subpatterns, as follows:

1. Top line
2. Bottom line
3. Through line
4. Others

The second SKIP code for the fourth pattern is the total number of strokes. The third SKIP part is the subpattern. An example is the kanji 下, whose SKIP code is 4-3-1 (fourth pattern, three total strokes, first subpattern).

Although SKIP has thus far been used only for Japanese, it is applicable to other CJKV locales as well. For those who wish to use SKIP in their own products, please read its terms of use.*

Four Corner Code

The Four Corner Code has been used for many years in China and Japan as a way to identify ideographs using exactly four digits. This system is apparently losing popularity in China due to the now widespread use of Pinyin. There are a number of rules that effectively describe how this system works and its underlying principles:

- The Four Corner Code elements are divided into 10 shapes, numbered 0 through 9. These are described in Table 11-6.
- The four digits that comprise a Four Corner Code are derived from the four corners in a Z-shaped sequence—upper-left, upper-right, lower-left, then lower-right.
- A shape is only used once. If it fills several corners, it is counted as 0 (zero) in subsequent corners.

* *http://www.kanji.org/kanji/dictionaries/skip_permission.htm*

- When the upper or lower half of an ideograph is comprised of only a single shape, it is, regardless of its position, counted as a left-corner code—the right corner is counted as 0 (zero).
- When there is no additional element to the four sides of the ideographs 口, 門, 鬥 (and the split form of 行), whatever is inside these ideographs is taken as the lower two corners.

Table 11-6 lists the 10 elements that can be used to build Four Corner Codes, along with some notes about their use.

Table 11-6. The Four Corner Code elements

Code	Name	Example shapes	Description
0	头/頭 *tóu*	亠	Lid
1	横 *héng*	一	Horizontal bar
2	垂 *chuí*	丨丿	Vertical bar
3	点/點 *diǎn*	丶	Dot
4	叉 *chā*	十	Cross
5	插 *chā*	扌	Skewer
6	方 *fāng*	口	Box
7	角 *jiǎo*	乙	Angle
8	八 *bā*	八人入	Eight
9	小 *xiǎo*	小个忄	Small

Japan's 13-volume 大漢和辭典 (*dai kanwa jiten*) dictionary and China's 辭源 (*cíyuán*) dictionary are examples of dictionaries that include a Four Corner Code index.

Multiple-component

Given that an ideograph can be broken down to not only its indexing radical and strokes, but also to all of the radical-like components that make up its structure, it becomes possible to enter ideographs by identifying two or more of its components. This is an especially useful lookup technique for ideographs whose indexing radical is not obvious.

Jim Breen developed two files, named *KRADFILE* and *RADKFILE*, that provide such functionality for the kanji in JIS X 0208:1997, and that also list software that uses these files or technique.* Jim Rose developed comparable files for the kanji in JIS X 02120-1990, called *KRADFILE2* and *RADKFILE2*.

* *http://www.csse.monash.edu.au/~jwb/kradinf.html*

The multiple-component indexing method is useful beyond cases when the indexing radical is not obvious or cannot otherwise be easily determined. Some of the components used in ideographs are not considered radicals, such as 𠂉, which looks very much like katakana マ (*ma*). In addition, it becomes possible to look up ideographs with similar components, beyond having a common indexing radical. This is a truly powerful ideograph indexing method.

Character code

When dealing with ideographs across multiple locales, there are often cases when finding an ideograph in one locale is easier than in another. For example, if you happen to be well-versed in Japanese, it may be relatively easy to input most kanji. But, trying to input that same ideograph in Chinese or Korean may prove to be a difficult or impossible task.* The character code itself can serve as a cross-reference to the Chinese or Korean equivalents.

Consider the contents of Table 11-7, which provide various character codes for the ideograph 中, meaning "middle" or "center."

Table 11-7. CJKV character code indexes

Character set	Row-Cell	ISO-2022	EUC	Other
JIS X 0208:1997	35-70	4366	C3E6	9286—Shift-JIS
GB 2312-80	54-48	5650	D6D0	
CNS 11643-2007	1-36-67	1-4463	C4E3	A4A4—Big Five
KS X 1001:2004	81-73	7169	F1E9	F3E9—Johab
KPS 9566-97	74-27	6A3B	EABB	
TCVN 6056:1995	42-21	4A35	CAB5	

In case it is not painfully obvious, all of the various character codes listed in Table 11-7 map to the same Unicode code points, specifically U+4E2D. This also demonstrates why Unicode is so compelling, and why it has been embraced as the *de facto* standard for encoding characters in today's OSes and applications. A dictionary that provides cross-references or indexes for character codes, as you can see, can be quite useful at times.

But, in order for a character code indexing system to be useful, it must also account for character form differences across locales. Character form differences, such as the simplifications used in China, can often render simplistic cross-references useless. Consider the case as illustrated in Table 11-8. This table provides the cross-reference information as provided by Unicode in the Traditional column, but what is considered to be the more correct cross-reference is in the Simplified column.

* Especially if you're trying to write a book entitled *CJKV Information Processing* on a Japanese-only OS with only Japanese input methods and Shift-JIS–encoded Chinese, Korean, and Vietnamese fonts. While this statement was true when the first edition was written, it is not entirely true for this edition.

Table 11-8. Ideograph cross-referencing issues

Locale	Traditional—U+6F22	Simplified—U+6C49
China	漢 GB/T 12345-90 26-26[a]	汉 GB 2312-80 26-26[b]
Taiwan	漢 CNS 11643-2007 1-73-39	汉 CNS 11643-2007 15-01-70
Japan	漢 JIS X 0213:2004 1-87-05[c]	漢 JIS X 0208:1997 20-33[d]
South Korea	漢 KS X 1001:2004 89-51	n/a
North Korea	漢 KPS 9566-97 82-51	n/a
Vietnam	漢 TCVN 6056:1995 61-30	n/a

a. This GB/T 12345-90 code point corresponds to GB 18030-2005 <9D 68>.

b. This GB 2312-80 code point corresponds to GB 18030-2005 <BA BA>.

c. This JIS X 0213:2004 code point corresponds to U+FA47, which is a CJK Compatibility Ideograph.

d. This JIS X 0208:1997 code point corresponds to U+6F22.

Note how the character code for the example from China is the same for both traditional and simplified, but the character set designation differs. This is by design. As you learned in Chapter 3, GB/T 12345-90 is the traditional analog of GB 2312-80.

Ideograph Dictionaries

Dictionaries represent one of the most useful resources for inputting or otherwise accessing CJKV characters. Yep, these are usually good ol' fashioned books![*] In fact, many of this book's appendixes, to some extent, can fulfill this purpose. What you will find in this book's appendixes are complete listings of characters enumerated in the most common CJKV character sets, followed by reading and radical indexes. The dictionaries discussed in this section go one step further in helping you to more easily access characters from those CJKV character set standards.

Ideograph dictionaries that are CJKV-specific typically fall into one of two categories, described as follows:

Conventional dictionaries
: Provide information such as readings, compounds, or words in which the ideograph occurs, and perhaps even offer a short definition—these dictionaries were not necessarily designed with computer users in mind.

Specialized dictionaries
: Give you information, such as readings, one or more encoded values, and perhaps other information, such as compounds or words in which the ideograph occurs.

* The nice thing about books is that you can take them virtually anywhere—often to places where computers or other electronic devices dare not venture.

Needless to say, conventional dictionaries are most useful to the student or scholar of a language, but specialized dictionaries are a valuable resource for the computer user and software developer, and typically contain entries for all the ideographs in one or more CJKV character set standards, such as GB 2312-80 or GB/T 12345-90 for Chinese. One of the largest conventional ideograph dictionaries was published in Japan. It is entitled 大漢和辭典 (*dai kanwa jiten*), contains 50,294 kanji, and is published as a 13-volume set.

Dictionaries that include the encoded value of ideographs prove their usefulness when you are trying to input a particular ideograph, and the input method that you are using simply doesn't seem to know it exists. Mind you, there are ideographs that are not included in any CJKV character set standard, but those ideographs are not frequently used. There are many ideographs, such as those enumerated in JIS X 0208:1997 Level 2 kanji and GB 2312-80 Level 2 hanzi, that typically cannot be input by reading. The user must finally resort to unintuitive means, such as *input by encoded value*, which is also known as *code input*.

Most input methods provide a mechanism for code input, and come with character tables—printed or available in software—arranged by encoded value. The usefulness of these character tables is very limited. Remember that GB 2312-80 Level 1 hanzi are arranged by reading, so forget about trying to locate a hanzi there by its indexing radical. GB 2312-80 Level 2 hanzi, on the other hand, are arranged by radical, so this makes locating hanzi a bit easier. However, there are always circumstances when you may wish to use other means to locate ideographs, such as by reading (or multiple readings), number of strokes, indexing radical, or radical-like components—for both GB 2312-80 Level 1 and 2 hanzi. This is when a specialized ideograph dictionary becomes invaluable. These dictionaries go far beyond what CJKV character set tables offer: they provide two or more methods for locating all ideographs in CJKV character set standards. I highly recommend purchasing at least one of these types of dictionaries. I have purchased several such dictionaries over the years, and they still make their way into my hands from time to time.

Character Set Standards As Ideograph Dictionaries

The CJKV character set standards themselves—in the form of a printed manual—can function as a limited-use ideograph dictionary strictly for the purpose of locating characters more effectively than the inherent ordering of the characters themselves.

Table 11-9 lists the major CJKV character set standards, along with information about what type of indexes they provide.

Table 11-9. Ideograph indexes provided by CJKV standards

Character set standard	Indexes
Unicode	Cross-references to character codes in CJKV national standards[a]
GB 2312-80[b]	Radical, stroke-count (limited), reading (ordered by Pinyin), and simplified versus traditional

Table 11-9. Ideograph indexes provided by CJKV standards

Character set standard	Indexes
CNS 11643-1992	None[c]
CNS 11643-2007	None[d]
JIS X 0208:1997	Radical and reading, plus cross-references to variants, 大漢和辭典 and 新字源
JIS X 0212-1990	Radical plus cross-references to variants
JIS X 0213:2000	Radical and reading (including JIS X 0208:1997), Unicode, plus cross-references to JIS X 0212-1990, variants, 大漢和辭典, and 新字源
JIS X 0221-1995	Cross-references to 大漢和辭典 and 康熙字典
KS X 1001:2004	None[e]
KS X 1002:2001	None[e]
TCVN 5773:1993	Reading
TCVN 6056:1995	Cross-references to *Tự Diển Chữ Nôm* and *Bảng Tra Chữ Nôm*

a. *The Unicode Standard, Version 5.0* (Addison-Wesley, 2006) provides a radical index.

b. Level 1 hanzi entries are annotated with Pinyin readings, and Level 2 hanzi entries are annotated with radicals (hanzi that represent the first using a particular radical are printed in Hei style; all others are in Song style).

c. Each hanzi entry includes fields that indicate its total number of strokes, its indexing radical, and the number of strokes that make up the indexing radical.

d. Part two of the standard provides a listing of components that make up the hanzi in Planes 1 and 2.

e. The hanja themselves are annotated with hangul to indicate reading.

I have personally found the indexes provided in the GB 2312-80 manual to be very useful and complete. These indexes have been repeated verbatim in the appendixes of the ideograph dictionary entitled 常用汉字编码字典 (described in the very next section). Kohji Shibano's JIS 漢字字典 includes the same indexes as in the JIS X 0208:1997 standard, which makes obtaining the actual JIS X 0208:1997 standard, for the purpose of obtaining its indexes, somewhat pointless.*

Locale-Specific Ideograph Dictionaries

Every CJKV locale has published a number of ideograph dictionaries. While most of these dictionaries are not directly applicable to computer users and software developers, a handful have been designed with computers in mind. A handful of these computer-oriented dictionaries are briefly described in the following list.

The specialized character dictionaries that I have used over the years provide access to different sets of characters, offer different methods for locating characters, and provide different information once you locate the target character:

* Besides, computer-oriented ideograph dictionaries are typically cheaper and easier to find than the character set standards that they cover.

- The hanzi dictionary entitled 常用汉字编码字典 (*chángyòng hànzì biānmǎ zìdiǎn*, meaning "Character Dictionary of Codes for Frequently Used Chinese Characters") contains the same hanzi indexes found in the GB 2312-80 character set standard, along with nearly two dozen assorted input codes for all 6,763 GB 2312-80 hanzi.
- The hanzi dictionary entitled 汉字属性字典 (*hànzì shǔxìng zìdiǎn*, meaning "Character Dictionary of Chinese Character Properties") is essentially a giant database of information for all 6,763 hanzi in the GB 2312-80 character set standard. It provides cross-references to other CJKV character set standards, such as Japan's JIS X 0208 series and Taiwan's CCCII.
- A pocket-size Chinese dictionary entitled 汉字输入速查手册 (*hànzì shūrù sùchá shǒucè*, meaning "Handbook of Chinese Character Input and Searching") allows lookup only through Pinyin readings (with no tones), and lists codes for two input methods (Renzhi and Wubi), along with Row-Cell values.
- The hanzi dictionary entitled 標準中文輸入碼大字典 (*biāozhǔn zhōngwén shūrùmǎ dà zìdiǎn*, meaning "Big Character Dictionary of Standard Chinese Input Codes") provides stroke-count, reading, and radical indexes for the Big Five character set, and also provides cross-references to the GB 2312-80 character set standard (using EUC-CN character codes). Also included are approximately 3,000 Hong Kong hanzi that are not part of the standard Big Five definition, but are in Hong Kong GCCS (it is published in Hong Kong).
- The kanji dictionary entitled パソコンワープロ漢字辞典 (*pasokon wāpuro kanji jiten*, meaning "Personal Computer and Word Processor Kanji Dictionary") allows users to locate all of the characters found in JIS X 0208-1983 (not the 1990 or 1997 vintages) by reading, indexing radical, and total number of strokes. It then provides hexadecimal ISO-2022-JP, hexadecimal Shift-JIS, Row-Cell, and the two printable ASCII characters that correspond to the two hexadecimal ISO-2022-JP bytes.
- The kanji dictionary entitled 最新JIS漢字辞典 (*saishin JIS kanji jiten*, meaning "New JIS Kanji Dictionary") allows users to locate all the characters in JIS X 0208-1990 and JIS X 0212-1990 by radical and reading. It then provides hexadecimal ISO-2022-JP and Row-Cell values for JIS X 0208-1990 characters, and only the Row-Cell value for JIS X 0212-1990 characters.
- Although the kanji dictionary entitled 大漢語林 (*dai kango rin*, meaning "Large Chinese Word Forest") is on the expensive side due to its professional-level quality, it is extremely authoritative, being authored by 2 students of the author of the 13-volume 大漢和辭典 (*dai kanwa jiten*). It includes entries for all the kanji in JIS X 0208-1990 and JIS X 0212-1990, and provides Row-Cell and hexadecimal ISO-2022-JP codes (Row-Cell only for JIS X 0212-1990).
- JSA itself published a kanji dictionary entitled JIS 漢字字典 (*JIS kanji jiten*, meaning "JIS Kanji Dictionary") in 1997, which provided extensive information and cross-references based on the JIS X 0208:1997 character set standard. This dictionary

offered Row-Cell, GL (ISO-2022-JP), GR (EUC-JP), Shift-JIS, and CJK (Unicode) code points for each kanji, and included a staggering amount of readings. Also included for every entry were cross-references to 大漢和辭典 (*dai kanwa jiten*) and 新字源 (*shinjigen*). I considered this dictionary a model for others based on national standards to follow. It was extensively revised in 2002 to include JIS X 0213:2000, and is now appropriately entitled 増補改訂 JIS 漢字字典 (*zōho kaitei JIS kanji jiten*, meaning "Expanded and Revised JIS Kanji Dictionary"). Interestingly, the 1997 edition of this dictionary ordered kanji according to the JIS X 0208:1997 standard, but its 2002 revision changed the order to the more traditional convention of using indexing radical and number of residual strokes.

There are a number of other computer-oriented character dictionaries available, at least another two dozen such titles in Japan alone, most of which support some flavor of the JIS X 0208 series (a select few cover JIS X 0212-1990 to varying degrees, and newer ones obviously provide coverage for JIS X 0213:2004). I have found the ones listed here to be exceptionally useful in my work. Strangely, I have not yet been able to discover a computer-oriented hangul or hanja dictionary from Korea. Appendix K represents my attempt to provide such a reference for hangul.

The dictionary entitled JIS 漢字字典, along with its newer version entitled 増補改訂 JIS 漢字字典, deserve special mention. The former covered JIS X 0208:1997 in its entirety, and the latter expanded coverage to include all of JIS X 0213:2000. There was a virtual explosion of computer-oriented kanji dictionaries in the late 1980s and 1990s, and my guess is that the broad and rich nature of these two kanji dictionaries effectively stopped the publishing of competing dictionaries. In other words, these dictionaries were of such broad coverage and high quality that there apparently was no point in publishing competing dictionaries.

Interestingly, the dictionary entitled 大漢語林 (*dai kango rin*) and its smaller sibling 漢語林 (*kango rin*) had set a precedent in Japan for conventional kanji dictionaries to include character codes, such as Row-Cell. They were apparently the very first conventional kanji dictionaries to do so. 新漢語林 (*shin kango rin*), which was written by the same authors as 漢語林, has taken this a step further by covering the latest JIS standards, including JIS X 0213:2004.*

Another kanji dictionary worth mentioning is entitled 新潮日本語漢字辞典 (*shinchō nihongo kanji jiten*). It is noteworthy because, like 新漢語林 described previously, it covers all of the kanji in multiple JIS standards, specifically JIS X 0208:1997 andJIS X 0213:2004.†

John Haig's revision of Andrew Nelson's *The New Nelson Japanese-English Character Dictionary* (Charles E. Tuttle Company, 1997), a favorite among English-speaking learners of Japanese, has joined this trend. As some dictionaries demonstrate, an ideograph

* *http://www.taishukan.co.jp/item/sinkangorin/*

† *http://www.shinchosha.co.jp/book/730215/*

dictionary does not need to include all the characters in a character set standard (such as JIS X 0208:1997) in order for the inclusion of character codes (such as Row-Cell) to be useful to its readership. In fact, I have categorized conventional Japanese kanji dictionaries in the five ways listed in Table 11-10.

Table 11-10. Categories of conventional Japanese kanji dictionaries

Coverage of JIS standards	Dictionaries
None	新字源, 大漢和辭典
Partial JIS X 0208:1997	*The Kodansha Kanji Learner's Dictionary*
JIS X 0208:1997	*The New Nelson Japanese-English Character Dictionary*, 漢語林
Partial JIS X 0208:1997, JIS X 0212-1990	角川必携漢和辞典, 福武漢和辞典, 旺文社漢和辞典
JIS X 0208:1997, JIS X 0212-1990	大漢語林, 五十音引き講談社漢和辞典
JIS X 0208:1997, JIS X 0213:2004	新潮日本語漢字辞典
JIS X 0208:1997, JIS X 0212-1990, JIS X 0213:2004	新漢語林

Of the dictionaries listed in Table 11-10, one deserves to be singled out because it is a hybrid dictionary, and to this day is still unique for this reason. Dictionaries in Japan typically come in two flavors. One is obviously a kanji dictionary, and it includes a main entry for each kanji that it covers. The other is a word dictionary, and it lists words, consisting of one or more characters, and ordered according to reading, and provides definitions. *Kodansha Kanwa Jiten* (五十音引き講談社漢和辞典 *gojūon biki kōdansha kanwa jiten*), published in 1997, mixes both types of dictionaries, and orders both kanji main entries and words by reading. These two types of dictionary entries thus appear side by side. Of course, this dictionary provides the expected stroke count and radical indexes so that kanji main entries can be located by means other than reading.

Ever since I began working with Chinese, Korean, and Vietnamese character sets, I have been searching for character dictionaries that span more than a single CJKV locale. There are times when I need to quickly determine the Chinese or Korean character code for a kanji in a Japanese character set standard. The limited indexes provided in *The Unicode Standard* and ISO 10646 run out of steam far too quickly, although once you find the ideograph, there are useful cross-references to various CJKV character set standards.

Vendor Ideograph Dictionaries and Ideograph Tables

A number of vendor character set standards are described in Appendix E. Where do they fit into this discussion of character dictionaries? Most companies that develop their own CJKV character set standards, such as operating system and font manufacturers, also publish a character dictionary, or at least a set of code tables, along with some indexes, which allow users to locate all of its special characters. I have a large collection of such references.

It goes without saying that some of these references are more useful than others. For example, a Japanese code table from Hewlett-Packard is identical to a JIS X 0208-1983 code table, except that it lists the Shift-JIS, ISO-2022-JP, and Row-Cell values for each character—this doesn't help you much in finding obscure characters. The set of Japanese character dictionaries published by NEC, on the other hand, are superb, and allow users to search by radical, reading, total number of strokes, and encoding. The point of the paragraph is that you should examine such references before you buy, or else get a recommendation from someone whose opinion you trust.

CJKV Ideograph Dictionaries

What can be considered the world's very first—and still only—CJKV character dictionary, designed for computer users and software developers, was published by Sanseido in 2000, and is entitled *Sanseido's Unicode Kanji Information Dictionary* (ユニコード漢字情報辞典 *yunikōdo kanji jōhō jiten*). Due to the year of its publication, and the work that was necessary to lead up to it, its coverage is the URO, meaning the first block 20,902 ideographs, the main entries of which provide a variety of character codes, readings, cross-references to related characters, classification symbols, indexing radical, stroke count, and so on. CJK Unified Ideographs Extension A is covered, but as an appendix, not as main entries.

Other Useful Dictionaries

Believe it or not, there are many dictionaries, some of which are nondigital, that are in some way useful in the context of this book. In the following sections I categorize several such dictionaries in the hope that you will also find them useful for your own work or research.

Conventional Dictionaries

In my attempts to understand Korean text, I have found a particularly useful dictionary: *Dong-A's Prime Korean-English Dictionary*, Second Edition (Doosan Dong-A Company, 1996). This dictionary is helpful in that it illustrates the hanja that correspond to the hangul that are conventionally used to express Korean words. Because of my experience in dealing with ideographs, once I can see the hanja that correspond to hangul, my comprehension of Korean text increases significantly. Those of you with limited Korean experience but with extensive Chinese or Japanese knowledge may find this and similar dictionaries useful.

I'll provide you with an example of what I mean. Suppose I come across the Korean phrase 정보 교환 (*jeongbo gyohwan*). When I look up this word in the dictionary, as two separate words 정보 (*jeongbo*) and 교환 (*gyohwan*), I am immediately greeted with the hanja 情報 and 交換. Because I am familiar with the equivalent words in Japanese, specifically 情報 (*jōhō*) and 交換 (*kōkan*), I gain an instant understanding of the Korean text. These two words together mean "information interchange."

Variant Ideograph Dictionaries

When working with ideographs, there is always the issue of simplified or variant characters to wrestle with. Some dictionaries, such as 汉字写法规范字典 (*hànzì xiěfǎ guīfàn zìdiǎn*), illustrate the traditional forms of simplified hanzi. Still other dictionaries, such as 漢字簡繁體字對照字典 (*hànzì jiǎnfántǐzì duìzhào zìdiǎn*) and 简化字 繁体字 异体字辨析手册 (*jiǎnhuàzì fántǐzì yìtǐzì biànxī shǒucè*), are specialized in the sense that they serve only to provide information on the relationship between traditional, simplified, and variant hanzi.

For Japanese, Jack Halpern's *New Japanese-English Character Dictionary* (NTC, 1993; Kenkyusha, 1990) is a virtual gold mine of kanji variant information and cross-references. 漢字異体字典 (*kanji itai jiten*; Nichigai Associates, 1994) is a specialized kanji variant dictionary. Also of interest is 誤字俗字・正字一覧表 (*goji zokuji seiji ichiranhyō*; Teihan, 1995), which lists kanji variants that were registered for use in Japanese names. In my experience, however, the most useful Japanese dictionary that contains kanji variants is entitled 角川大字源 (*kadokawa daijigen*; Kadokawa, 1991). This dictionary catalogs a large number of kanji variants.

A CD-ROM–based product called 今昔文字鏡 (*konjaku moji kyō*) provides over 150,000 ideographs and their variants. It runs on Windows.*

The relationships among traditional, simplified, and variant ideographs are somewhat locale-specific, especially when dealing with characters that were either created or simplified within a single locale. This is why dedicated reference materials are indispensable if you are serious about fully understanding the relationship among ideographs and their variants.

Dictionary Hardware

Electronic dictionaries can come in the form of dedicated hardware. Obviously, these devices still use software—to be more precise, dedicated software—for searching and other purposes.

Canon, Seiko, and Sharp, among other companies, manufacture hand-held electronic character dictionaries that support the languages used in CJKV locales. For over 20 years, Canon's *WordTank* series has been exceptionally popular with students studying Japanese. I myself once owned a Canon WordTank ID 7100 in the late 1980s, and thinking of it brings back good memories from when I was first learning the Japanese language. The electronic dictionaries that are available today are relatively inexpensive, and include scores of large dictionaries.

WizCom Technologies and Japan21 more recently teamed up to develop the *Quicktionary 2 Kanji Reader*, which is a hand-held scanning device that performs OCR operations

* *http://www.mojikyo.org/*

on Japanese text, set horizontally and vertically. It translates words and phrases between English and Japanese.* Horizontal and vertical scanning is specified by a setting. This pen dictionary includes three of Sanseido's *Daily Concise* dictionaries, which allow English text to be translated into Japanese, Japanese text to be translated into English, and definitions of Japanese text to be displayed on its LCD screen.

The primary disadvantage of dedicated dictionary hardware is that, due to its hardware nature, it cannot be easily upgraded, though some of them allow users to install their own CD-ROM– or DVD-based dictionaries, effectively bypassing this limitation. While many of these devices have optional dictionaries, usually specialized, there is no way to add entries to the main dictionary. WYBIWYG.† The nonupgradable nature of dedicated dictionary hardware may be true of those offered in the past, but given the introduction of mobile devices that can perform computing operations, including many cell phones, I doubt that it will be true in the future.

Dictionary Software

Character dictionaries that reside in software are a more recent phenomenon. Some are conventional, some are specialized, and some are even a combination of both. These electronic dictionaries take the form of dictionary software for your computer, acting as a frontend to one or more dictionary files, and may or may not require that you use a CJKV-capable OS. However, given the multilingual capabilities of today's OSes, there is generally an adequate level of CJKV support already built in.

Dictionary CD-ROMs

Electronic dictionary software was a very useful resource for computer users. Such software allows you to look up individual characters, words, and even phrases. This software usually comes in two parts: the actual dictionary, which is machine-readable text in a database-like format, and software that acts as a frontend, and thus accesses the information and displays it to the screen. Software packages almost always include both parts.

A past market trend was to store dictionaries on a single CD-ROM. CD-ROMs can store several hundred megabytes of data and make an excellent media for large distributions of software (and music!). Now, DVDs have taken over, given that they are able to store nearly 10 times the amount of data of a CD-ROM.

Most of these disc-based dictionaries were originally designed to be interfaced using an electronic book player. These are commonly referred to as electronic books (電子ブック *denshi bukku* in Japanese). Creative people and companies have written software that allows you to read these CD-ROMs on platforms such as Linux, Mac OS X, and Windows using standard CD-ROM or DVD drives, and many of these dictionaries now come

* *http://wizcom.jp21.jp/*

† What You Buy Is What You Get.

bundled with such software. The following is a short list of Japanese-related electronic books that were current when the first edition of this book was published, and many of them are still available today:

広辞苑 by 岩波書店

大辞林 by 三省堂

新英和・和英中辞典 by 研究社

漢字源 by 学研

辞書パック 10 by 三省堂

新漢英字典電子ブック版 by 日外アソシエーツ

科学技術用語大辞典 by 日外アソシエーツ

25 万語医学用語大辞典 by 日外アソシエーツ

コンピュータ用語辞典 by 日外アソシエーツ

These and other electronic books can usually be ordered through Kinokuniya and other Japanese bookstores in the U.S. and Japan, but perhaps *amazon.co.jp* is the most convenient place through which to order them.* They may even have certain titles in stock now that they are becoming more popular. Prices usually begin at about $50 (U.S.). Note that these are almost always 8cm CD-ROMs—most CD-ROM drives accept 12cm CD-ROMs. Most music stores sell adapters that let you play 8cm CD-ROMs in 12cm CD-ROM drives. For DVD-based electronic books, only the 12cm size is offered.

Electronic Publishing WING (EPWING) has gained acceptance, at least in Japan, as the leading format for disk-based or other electronic dictionaries.† In fact, EPWING is now reflected in a JIS standard, specifically JIS X 4081:2002 (日本語電子出版検索データ構造 *nihongo denshi shuppan kensaku dēta kōzō*).

Frontend Software for Dictionary CD-ROMs

There are a number of freeware and commercial software packages that serve as frontends for dictionary CD-ROMs, which are obviously designed to eliminate the need for a dedicated electronic book player.

Table 11-11 lists several popular frontends for EPWING-based electronic dictionaries, for a variety of platforms. More, beyond what is listed in Table 11-11, are available.‡

* *http://www.amazon.co.jp/*

† *http://www.epwing.or.jp/*

‡ *http://hp.vector.co.jp/authors/VA037273/viewers.htm*

Table 11-11. Frontend software for EPWING electronic dictionaries

Software	Platforms	Availability
CDROM2	MS-DOS and Unix	*http://www.nerimadors.or.jp/~jiro/cdrom2/*
CeDar	Mac OS and Mac OS X	*http://hp.vector.co.jp/authors/VA004654/cedar/*
DDwin	Windows	*http://homepage2.nifty.com/ddwin/*
EBPocket/EBWin	Windows	*http://www31.ocn.ne.jp/~h_ishida/EBPocket.html*
EBView	Linux and Windows	*http://ebview.sourceforge.net/*
Jamming	Mac OS X and Windows	*http://dicwizard.jp/jamming.html*
かものす (*kamonosu*)	Mac OS X	*http://homepage2.nifty.com/andom/kamonos.html*
コトノコ (*kotonoko*)	Mac OS X	*http://www.afternooncafe.jp/kotonoko/*

One of the best is called *Jamming*, also available as *Jamming Light*, both of which serve as a frontend not only to disk-based dictionaries, but also for dictionaries of other formats. Both are available for Windows and Mac OS X.

Dictionary Files

Jim Breen deserves extra special mention here, because he has managed and coordinated the development of several very useful machine-readable dictionary files for Japanese: *EDICT*, *ENAMDICT*, *KANJIDIC*, *KANJD212*, and so on. More recently, he has combined and XML-ized some of them into newer and improved dictionary files: *JMdict*, *JMnedict*, and *KANJIDIC2*. In 1999, the EDICT data was placed into a database format that was capable of holding richer and more complex content. From this database, both EDICT and the XML-ized JMdict are generated daily. Likewise, the kanji-related dictionary files are also in a database format, and the text-based and XML-ized versions are generated from it on a daily basis. These dictionary files are described in the next section, followed by a listing of applications that act as frontends to one or more of them. Paul Denisowski had initiated a similar effort for Chinese. His first (and only) work was *CEDICT*, which is the Chinese analog of Jim Breen's EDICT. Jim Breen's work is copyrighted under the Electronic Dictionary Research and Development Group, Monash University.*

A recent trend, as demonstrated by Jim Breen's JMdict, JMnedict, and KANJIDIC2, is to move from a text-based format to XML. While this leads to some amount of increase in overall file size, it does provide a much better encapsulation of the data, and also increases its usability. Given the inherent structure of an XML file and the fact that structure is an important part of a dictionary, this is a natural and logical development. Any ambiguity that may be present in a text-based format is effectively removed by virtue of XML.

* *http://www.edrdg.org/*

In addition, another recent trend is to provide web-based interfaces for these dictionaries, which effectively solves any platform-dependent issues. These web-based frontends additionally make it easier to hide the actual dictionary files, yet does not hinder access. In other words, web-based frontends help to protect the intellectual-property aspect of dictionaries, but at the same time do not prevent their use. In fact, one can argue that they encourage their use.

It goes without saying that these dictionary projects represent ongoing efforts, involving several dedicated individuals and organization. Contributions in whatever form are always welcome.

Unihan Database

Perhaps the single most useful dictionary file is the Unihan Database, which provides a large number of cross-references, readings, and meanings for all of the ideographs in Unicode.* It is maintained by The Unicode Consortium, and covers not only all CJK Unified Ideographs, but CJK Compatibility Ideographs as well. It is kept up to date, with new versions being released in sync with new versions of Unicode, but also when entirely new fields are added.

CEDICT—Chinese-English dictionary

Started in the fall of 1997, CEDICT is work that was inspired by Jim Breen's suite of Japanese dictionary files.† CEDICT is a Chinese-English dictionary file whose development was originally managed by Paul Denisowski. As of this writing, CEDICT has well over 40,000 entries, and is still expected to grow significantly over time. The following are some example entries from CEDICT:

筆畫 笔画 [bi3 hua4] /strokes of a Chinese character/

漢字 汉字 [han4 zi4] /Chinese character, kanji/

人權 人权 [ren2 quan2] /human rights/

中華人民共和國 中华人民共和国 [zhong1 hua2 ren2 min2 gong4 he2 guo2] /The People's Republic of China/

中日韓越 中日韩越 [zhong1 ri4 han2 yue4] /China, Japan, Korea, and Vietnam/

字集 字集 [zi4 ji2] /character set/

自由 自由 [zi4 you2] /freedom/free/liberty/

Note how CEDICT now includes two forms for the Chinese entry, specifically the traditional form followed by the simplified form. In some cases they are the same, such as for

* *http://www.unicode.org/charts/unihan.html*

† *http://www.mandarintools.com/cedict.html*

the example entries 字集 and 自由 shown previously. The CEDICT file itself was originally encoded according to EUC-CN encoding, which accommodated only a limited set of Simplified ideographs, but its current form is encoded according to UTF-8 encoding, which accommodates any ideograph in Unicode. Its format consists of four basic fields, detailed as follows:

- One or more traditional hanzi that represent a word
- The simplified form of the word
- Reading in Pinyin
- One or more English glosses

The objective of the CEDICT project is to create an online, downloadable (as opposed to searchable-only) public-domain Chinese-English dictionary. For the most part, the project was modelled after Jim Breen's hugely successful EDICT (Japanese-English dictionary) project, and it is intended to be a collaborative effort, with users providing entries and corrections to the main file. For specific limitations regarding its use, please read the CEDICT documentation.* Also of potential interest is that CEDICT has been adapted for NJStar use.

CEDICT forked several years ago, and the most active version is called *CC-CEDICT* (CC stands for *Creative Commons*, and refers to its usage license), which includes nearly 75,000 entries.†

HanDeDict—Chinese-German dictionary

For those who are speakers of German with an interest in Chinese, HanDeDict is a web-based dictionary that allows lookup of words in either language.‡ This dictionary serves as an excellent reminder that English is limiting for many people and is not as universal as some think.

EDICT/JMdict—Japanese-English dictionaries

EDICT is a freeware Japanese-English dictionary in machine-readable form. It was initially intended for use with MOKE (*Mark's Own Kanji Editor*) and related applications, such as JDIC. However, this file has the potential to be used in a large number of situations. The copyright on EDICT and its documentation is held by Jim Breen; however, it is freely available for noncommercial use. EDICT is in the EDICT format as originally specified by MOKE, and uses EUC-JP encoding for the Japanese portions. It is a text file with one entry per line.

* *http://www.mandarintools.com/download/cedict_readme.txt*

† *http://cc-cedict.org/wiki/*

‡ *http://chdw.de/*

The following are a few example entries as found in the latest version of the EDICT dictionary file:

教科書 [きょうかしょ] /(n) textbook/text book/(P)/

字体 [じたい] /(n) type/font/lettering/(P)/

字典 [じてん] /(n) character dictionary/

辞書 [じしょ] /(n) (1) dictionary/lexicon/(2) (arch) letter of resignation/(P)/

辞典 [じてん] /(n) dictionary/(P)/

電子計算機 [でんしけいさんき] /(n) (comp) computer/

日本 [にっぽん] /(n) Japan/

日本 [にほん] /(n) Japan/(P)/

日本語 [にっぽんご] /(n) Japanese (language)/

日本語 [にほんご] /(n) Japanese (language)/(P)/

和英辞典 [わえいじてん] /(n) Japanese-English dictionary/

辭典 [じてん] /(oK) (n) dictionary/

EDICT now has over 150,000 entries and is similar in size to a high-quality commercial dictionary. The EDICT documentation that is distributed with EDICT describes its history, its lexicographical principles, and its usage license statement, and it lists its many contributors.*

There are two main EDICT formats. One is the legacy EDICT format. The other is the EDICT2 format that is richer, and is more like MJdict. While the legacy EDICT format allows single readings and no cross-references, the EDICT2 format allows both. Consider the following two EDICT entries:

行き方 [いきがた] /(n) (one's) whereabouts/

行き方 [ゆきがた] /(n) (one's) whereabouts/

The following single EDICT2 entry consolidates these two entries and provides a useful cross-reference:

行き方 [ゆきがた;いきがた] /(n) (See 行方) (one's) whereabouts/

* *http://www.csse.monash.edu.au/~jwb/edict_doc.html*

JMdict, in addition to including German, French, Russian, and Dutch glosses, is also an XML file, as opposed to plain text.* To provide you a better sense of the XML structure of JMdict, when compared to EDICT, consider the following single entry, which corresponds to the first sample EDICT entry shown previously:

```
<entry>
<ent_seq>1237020</ent_seq>
<k_ele>
<keb>教科書</keb>
<ke_pri>ichi1</ke_pri>
<ke_pri>news1</ke_pri>
<ke_pri>nf04</ke_pri>
</k_ele>
<r_ele>
<reb>きょうかしょ</reb>
<re_pri>ichi1</re_pri>
<re_pri>news1</re_pri>
<re_pri>nf04</re_pri>
</r_ele>
<sense>
<pos>&n;</pos>
<gloss>textbook</gloss>
<gloss>text book</gloss>
<gloss xml:lang="fre">livre de classe</gloss>
<gloss xml:lang="fre">manuel scolaire</gloss>
<gloss xml:lang="rus">учебник</gloss>
<gloss xml:lang="ger">Lehrbuch</gloss>
<gloss xml:lang="ger">Schulbuch</gloss>
</sense>
</entry>
```

Given the extent to which XML is being used by today's OSes and applications, JMdict is clearly the next step for EDICT's evolution.

WaDokuJT—Japanese-German dictionary

Ulrich Apel developed an online Japanese-Germany dictionary called WaDokuJT (和独辞典 *wadoku jiten*).† This dictionary deserves mention, not only because it is another example of a useful non-English resource, but also because it serves as the source of the German-language material in Jim Breen's JMdict.

ENAMDICT/JMnedict—Japanese proper name dictionaries

ENAMDICT, which now contains over a half-million Japanese proper name entries, was once part of EDICT, well over 10 years ago. Because this portion of EDICT caused it to bloat considerably, Jim decided to split it into two separate entities: EDICT, which now has the personal and place names removed; and ENAMDICT, which contains the personal and place names formerly part of EDICT.

* *http://www.csse.monash.edu.au/~jwb/j_jmdict.html*

† *http://www.wadoku.de/*

Table 11-12 provides a list of the tags used by ENAMDICT, along with their meaning, and the number of entries that include the tag. As you can see, the number of entries is both staggering and impressive.

Table 11-12. Explanations of ENAMDICT tags

Tag	Explanation	Number of entries
s	Surname	138,500
p	Place name	99,500
u	Person name, either given or surname, as yet unclassified	139,000
g	Given name, as yet not classified by sex	64,600
f	Female given name	106,300
m	Male given name	14,500
h	Full—family plus given—name of a particular person	30,500
pr	Product name	55
co	Company name	34

The following are sample entries from the ENAMDICT file, some of which include multiple tags:

剣 [けん] /Ken (g)/

小塚 [こづか] /Kodzuka (p,s)/

工藤 [くどう] /Kudou (p,s)/

太郎 [たろう] /Tarou (p,s,m)/

山本 [やまもと] /Yamamoto (p,s)/

瑠美 [るび] /Rubi (f)/

ENAMDICT, like EDICT, is encoded according to EUC-JP encoding. The *JMnedict* (*Japanese Multilingual Named Entity Dictionary*) dictionary file is an XML form of ENAMDICT that is encoded according to UTF-8 encoding.

More detailed information about ENAMDICT can be found by reading its documentation.*

* *http://www.csse.monash.edu.au/~jwb/enamdict_doc.html*

KANJIDIC—JIS X 0208:1997 kanji

KANJIDIC is simply a kanji database file that is generated on a daily basis from a database that also generates the XML-ized KANJIDIC2, which is covered later in this chapter.* There are 6,355 entries, one per line, and one for each of the kanji in JIS X 0208:1997. The first two fields of each entry are always the kanji itself (EUC-JP–encoded) followed by its hexadecimal ISO-2022-JP code. The remaining fields correspond to additional information. Table 11-13 lists the prefixes and the information that their values represent. Clearly, some of these fields may not mean much unless you have a particular dictionary or reference handy.

Table 11-13. Explanations of KANJIDIC fields

Field	Meaning
B	Radical number assigned by Nelson's kanji dictionary (from a set of 214)
C	Classical radical number (from the standard set of 214) when assigned differently from Nelson's kanji dictionary
D	Dictionary-based codes, to be progressively used to support additional dictionaries and similar references
E	Index number from Kenneth Henshall's *A Guide to Remembering Japanese Characters*
F	Frequency-of-use ranking, if present (applies only to 2,501 kanji, based on Alexandre Girardi's analysis of word frequencies in the *Mainichi Shimbun* spanning four years)
G	Jōyō Kanji, Gakushū Kanji, and Jinmei-yō Kanji field (a value of 1 to 6 indicates the grade level for Gakushū Kanji; a value of 8 indicates Jōyō Kanji; and a value of 9 indicates Jinmei-yō Kanji)
H	Index number from Jack Halpern's *New Japanese-English Character Dictionary*
I	Index number from Mark Spahn and Wolfgang Hadamitzky's *The Kanji Dictionary* and *Kana & Kanji*
J	The *Japanese Language Proficiency Test* (JLPT) level in which the kanji occurs (1 through 4 are possible values)
K	Gakken dictionary index number
L	Index number from James Heisig's *Remembering the Kanji* series
M	Index number from Morohashi's 大漢和辭典
N	Index number from Andrew Nelson's *The Modern Reader's Japanese-English Character Dictionary*
O	Index number from P.G. O'Neill's *Japanese Names*
P	SKIP pattern code
Q	Four Corner Code
S	Total number of strokes (more than one such field is acceptable in the case of kanji with varying stroke counts)
U	Unicode scalar value
V	Index number from Andrew Nelson and John Haig's *The New Nelson Japanese-English Character Dictionary*
W	Korean reading

* *http://www.csse.monash.edu.au/~jwb/kanjidic.html*

Table 11-13. Explanations of KANJIDIC fields

Field	Meaning
X	Cross-reference code
Y	Pinyin reading
Z	Mis-classification code

KANJIDIC, like EDICT and ENAMDICT, has a copyright. However, Jim has made it freely available on the same basis as EDICT. The copyright on some fields is held by others. For example, the SKIP field was included with the permission of Jack Halpern and his publishers.

The final fields are one or more pronunciations written in kana and English meanings. English meanings are enclosed in curly braces. The KANJIDIC documentation explains these fields in greater detail.*

The following are three complete sample entries taken from KANJIDIC. Compare the information to the listing given in Table 11-13.

漢 3441 U6f22 B85 G3 S13 F1487 J3 N2662 V3281 H657 DK471 L1578 K1394 O1860 DO401 MN18068P MP7.0189 E442 IN556 DS572 DF212 DH265 DT418 DJ527 DB2.19 DG1230 P1-3-10 I3a10.17 Q3413.4 DR363 Yhan4 Whan カン T1 はん {Sino-} {China}

剣 3775 U5263 B18 G8 S10 XJ05178 XJ05179 XJ0517A XJ0517B XJ06E5F F1305 J1 N696 V498 H1672 DK1096 L1671 K1248 O1151 DO1436 MN2076 MP2.0295 E1214 IN879 DJ791 DG198 P1-8-2 I2f8.5 Q8250.0 DR2843 Yjian4 Wgeom ケン つるぎ {sabre} {sword} {blade} {clock hand}

和 4F42 U548c B115 C30 G3 S8 XJ16D61 F124 J2 N3268 V770 H1130 DK769 L897 K151 O638 DO277 MN3490 MP2.0969 E416 IN124 DS338 DF412 DH440 DT318 DC85 DJ352 DB3.5 DG326 P1-5-3 I5d3.1 Q2690.0 DR2277 Yhe2 Yhe4 Yhuo2 Yhuo4 Yhuo5 Yhai1 Yhe5 Whwa ワ オ カ やわ.らぐ やわ.らげる なご.む なご.やか T1 あい いず かず かつ かつり かづ たけ ち とも な にぎ まさ やす よし より わだこ わっ {harmony} {Japanese style} {peace} {soften} {Japan}

KANJD212—JIS X 0212-1990 kanji

The KANJD212 dictionary file is best described as the JIS X 0212-1990 analog of the KANJIDIC file, and includes 5,801 entries, one for each of the kanji enumerated in the JIS

* *http://www.csse.monash.edu.au/~jwb/kanjidic_doc.html*

X 0212-1990 character set standard.* Its format is different from that of KANJIDIC in the following respects only:

- EUC-JP encoding, code set 3, for the main entries (KANJIDIC, by comparison, uses EUC-JP encoding, code set 1, for its main entries)
- Includes only the following KANJIDIC fields: U, B, S, M, W, and Y (see Table 11-13 for a description of these fields)

Japanese readings are also included. The following are some sample entries from the KANJD212 file:

圃 3729 U5715 B31 S13 MN4829P P3-3-10 ショ としょかん {library}

辷 6133 U8fb5 B162 S7 XJ16134 H1945 MN38700 P2-1-6 Ychuo4 チャク T2 しんにょう しんにゅう {walk} {walking} {road radical (no. 162)}

辶 6134 U8fb6 B162 S3 XJ16133 H1932 MN38702 P2-1-2 Ychuo4 チャク T2 しんにょう しんにゅう {walk} {walking} {road radical variant (no. 162)}

Whether this dictionary file can be used in your environment depends on whether your OS supports the JIS X 0212-1990 character set. Read the KANJD212 documentation for more details.†

KANJIDIC2—XML form of KANJIDIC, KANJD212, and more

Jim Breen has taken the necessary steps to develop an XML form of KANJIDIC, which provided an opportunity to not only merge KANJIDIC and KANJD212 into a single file, but also to add the additional kanji necessary to cover JIS X 0213:2004. This new dictionary is called *KANJIDIC2*.‡ The following is an example KANJIDIC2 dictionary entry, which is the same as the first example entry for the KANJD212 dictionary file:

```
<!-- Entry for Kanji: 圃 -->
<character>
<literal>圃</literal>
<codepoint>
<cp_value cp_type="ucs">5715</cp_value>
<cp_value cp_type="jis212">23-9</cp_value>
<cp_value cp_type="jis213">2-04-58</cp_value>
</codepoint>
<radical>
<rad_value rad_type="classical">31</rad_value>
</radical>
<misc>
<stroke_count>13</stroke_count>
</misc>
<dic_number>
```

* *http://www.csse.monash.edu.au/~jwb/kanjidic.html*

† *http://www.csse.monash.edu.au/~jwb/kanjd212_doc.html*

‡ *http://www.csse.monash.edu.au/~jwb/kanjidic2/*

```
<dic_ref dr_type="moro">4829P</dic_ref>
</dic_number>
<query_code>
<q_code qc_type="skip">3-3-10</q_code>
</query_code>
<reading_meaning>
<rmgroup>
<reading r_type="ja_on">ショ</reading>
<reading r_type="ja_kun">としょかん</reading>
<meaning>library</meaning>
</rmgroup>
</reading_meaning>
</character>
```

Note how the same information is provided, but in a highly structured XML format. In addition, the JIS X 0213:2004 code point is provided.

LSD—Japanese-English life sciences dictionary

Although its development is not coordinated by Jim Breen, the LSD dictionary, coordinated by the Life Science Dictionary (LSD) project, and led by Shuji Kaneko (金子周司 *kaneko shūji*), is nonetheless another useful dictionary file that has been adapted to the same format as the previously described dictionary files. It provides a web-based interface to its contents for easy and platform-independent access.*

This LSD dictionary file contains over 40,000 entries for terms used in the life sciences—physiology, pharmacology, biophysics, biochemistry, organic chemistry, biology, and so on—along with their corresponding English terms. The following are some example entries—some not very suitable for discussion at the dinner table—from the LSD dictionary. These entries are formatted according to the same conventions as used by EDICT, from when the LSD dictionary was made available as a static file:

甲状腺機能低下症 [こうじょうせんきのうていかしょう] /hypothyroidism/

ジアスターゼ [じあすたーぜ] /diastase/†

手根管症候群 [しゅこんかんしょうこうぐん] /carpal tunnel syndrome/

虫垂炎 [ちゅうすいえん] /appendicitis/

腸 [ちょう] /intestine/intestinal/bowel/gut/entero/

Needless to say, LSD is useful for those who deal with the life sciences and Japanese.

* *http://lsd.pharm.kyoto-u.ac.jp/*

† $2.50

HANJADIC—Korean hanja dictionary

For those who work with ideographs in a Korean context, Dan Bravender developed HANJADIC, which is the Korean analog of KANJIDIC.* Instead of being a static dictionary file, HANJADIC is a program, written in Ruby, that provides four ways with which to search: in English, by reading, by character compound, and by the character itself.

HANJADIC is also available through a web interface, which arguably allows it to reach a wider audience.†

Frontend Software for Dictionary Files

There are a variety of programs, freeware, and commercial software, that serve as excellent frontends to Jim Breen's various dictionary files: EDICT, ENAMDICT, KANJIDIC, and KANJD212. These frontends also work with the EDICLSD3 file. There is also one, CEL (*Chinese-English Lookup*), that serves as a frontend to CEDICT. Table 11-14 lists these programs and includes information about what platform they run on, who authored them, and so on.

Table 11-14. Dictionary frontend software

Software	Platforms	Availability
CEL	Windows	Freeware by Richard Warmington[a]
CJEDictionary	Windows	Freeware by Jeremy Thorpe[b]
Gjiten	Linux	Freeware by Botond Botyanszki[c]
JavaDict	Java	Freeware by Todd Rudick[d]
JEDict	Mac OS X	Commercial software by Sergey Kurkin[e]
JquickTrans	Windows	Shareware by Alex Schonfeld[f]
JREADER	MS-DOS	Freeware by Jim Breen; does not require Japanese-capable OS[g]
Rikai	Online	Freeware by Todd Rudick[h]
Rikaichan	Firefox plug-in	Freeware by Jon Zarate[i]
StarDict	Linux and Windows	Freeware[j]
UniDict	Mac OS	Commercial software by Enfour Media[k]

* *http://www.msu.edu/~bravend2/hanjadic/*

† *http://hanjadic.bravender.us/*

Table 11-14. Dictionary frontend software

Software	Platforms	Availability
WWWJDIC	Firefox plug-in	Freeware by Jim Breen[l]
XJDIC	Unix	Freeware by Jim Breen[m]

a. *http://home.iprimus.com.au/richwarm/cel/cel.htm*
b. *http://www.cjedictionary.com/*
c. *http://gjiten.sourceforge.net/*
d. *http://www.cs.arizona.edu/projects/japan/JavaDict/*
e. *http://www.jedict.com/*
f. *http://www.coolest.com/jquicktrans/*
g. *http://www.csse.monash.edu.au/~jwb/jreader.html*
h. *http://www.rikai.com/*
i. *http://www.polarcloud.com/rikaichan/*
j. *http://stardict.sourceforge.net/*
k. *http://www.enfour.co.jp/unidict/*
l. *http://www.csse.monash.edu.au/~jwb/wwwjdicfirefox.html*
m. *http://www.csse.monash.edu.au/~jwb/xjdic/*

Some word processors, such as NJStar, also provide support for these dictionary files as a way for users to input text or look up unknown words.

There are other frontend software packages for dictionary files, but the dictionary files themselves are in what I consider to be proprietary formats.

Web-Based Dictionaries

There are now many dictionaries and dictionary-like resources that are accessible through web browsers. Some of these simply act as a frontend to dictionaries that are easily accessible, and some are accessible only through the Web. The following are some samples of those that are currently available—more and more are coming online every day:

- Jim Breen's *WWWJDIC*, which is a web frontend to his various dictionary files—definitely a must-visit site.*
- Kim Ahlström's *Denshi Jisho*, which is a powerful derivative of WWWJDIC.†
- Jeffrey Friedl's frontend to Jim Breen's Japanese dictionary files—such as EDICT, KANJIDIC, and so on—long has been a mainstay on the Web.‡

* *http://www.csse.monash.edu.au/~jwb/cgi-bin/wwwjdic.cgi*
† *http://jisho.org/*
‡ *http://dict.regex.info/cgi-bin/j-e/dict*

- Rick Harbaugh's *Chinese Character Genealogy* (based on his software called Zhongwen Zipu)* and *Chinese Characters Dictionary Web* (really cool because it links together about a dozen web dictionaries).†
- Charles Muller's *CJKV-English Dictionary.*‡
- Erik Peterson's *Chinese-English Dictionary* and *Chinese Character Dictionary.*§
- *KanjiBase* by Christian Wittern and Urs App.¶
- Hongjie Xin's (忻宏杰 *xīn hóngjié*) *Online English-Chinese and Chinese-English Dictionary.***
- *StarDict*, which is also available through a web interface.††
- Richard Cook's frontend to The Unicode Consortium's *Unihan Database.*‡‡

Some of these web dictionaries—in particular, Charles Muller's, Jim Breen's, and Rick Harbaugh's—have been cross-linked at the character level. This represents an exciting development in web-based lexicography.

Expect to find more and more dictionaries accessible through the Web as time goes on. In many ways, web-based dictionaries provide the best of both worlds. They effectively eliminate the platform issue, due to the inherent cross-platform nature of the Web. All you need is a suitable web-browsing application that is running on a CJKV-enabled OS, and you have a very convenient frontend to a large and growing number of useful dictionaries.

Machine Translation Applications

Machine translation (MT) applications are a bit different from electronic dictionaries in that they handle not only single characters, words, and phrases, but entire sentences. Machine translation applications are best used to perform the first pass translation, thus reducing the translation burden for humans. The state of machine translation technology is not yet to the point where human intervention is not required.§§ These applications are best referred to as machine-*aided* translation software—they merely assist you in translating text faster.

* *http://zhongwen.com/*

† *http://zhongwen.com/zi.htm*

‡ *http://www.buddhism-dict.net/dealt/*

§ *http://www.mandarintools.com/*

¶ *http://www.chibs.edu.tw/~chris/gwdg/kanjibas/cefintro.htm*

** *http://www.tigernt.com/*

†† *http://www.stardict.org/*

‡‡ *http://linguistics.berkeley.edu/~rscook/cgi/zunihan.html*

§§ We can all dream about the time when there will be a "universal translator" as used in *Star Trek* and other science fiction series.

Because these applications cannot not provide a perfect translation, pre- and post-editing functions are usually available. *Pre-editing* is a form of massaging input text so that the translation application does a better job. *Post-editing* is the process of correcting errors and inconsistencies in translation that are made by the application.

One of the keys to adequate translation quality is the availability of specialized dictionaries that contain terms specific to fields of study. Such fields include computer science, medicine, science, and so on. Very few machine translation applications come bundled with specialized dictionaries—they are available for additional cost.

The highest-rated and most widely-used machine translation applications are offered by SYSTRAN, and have been used by Yahoo! and Google.* Also consider those offered by LEC (*Language Engineering Corporation*), specifically their *Translate* and *Power Translator* series, all of which are available for Windows.† They even provide a web-based demo service.‡ Babylon offers machine translation software for Windows and Mac OS X users.§ Another to consider is *ATLAS* by Fujitsu, which supports English to Japanese and Japanese to English (for which demo versions are available), and is bundled with Fujitsu laptop computers.¶ Finally, Amikai's *Amikai Enterprise* is designed for business users and is web-based.**

Machine Translation Services

With the explosive growth of the Web came the need to translate information on the same medium. This market has clearly matured since the first edition of this book was written 10 years ago. As you will discover, and if you were not yet aware, there are now free machine translation services, along with commercial ones. While the "you get what you pay for" expression certainly applies here, to what extent the free machine translation services suit your needs depends on a variety of factors.

Free Machine Translation Services

Given the proliferation of web use, and the number of companies that provide useful services within that context, it is of no surprise that the "usual suspects" provide very useful—and free—machine translation services.

Google, as a primary player on the Web, offers a series of language tools, some of which provide translation services.†† Blocks of text can be translated—or entire web pages if

* *http://www.systransoft.com/*

† *http://www.lec.com/translate-software.asp*

‡ *http://www.lec.com/translate-demos.asp*

§ *http://www.babylon.com/*

¶ *http://www.fujitsu.com/global/services/software/translation/atlas/*

** *http://www.amikai.com/products/enterprise/*

†† *http://www.google.com/language_tools*

the URL is provided. It is of no surprise that Yahoo! offers comparable functionality to Google's language tools with *Babel Fish*.*

While specific to Japanese, the *goo* search engine offers very useful dictionaries and translation tools.† Infoseek's Japanese website offers a Japanese-English and English-Japanese translation service, along with the very unique ability to translate standard Japanese text into Kansai dialect (関西弁 *kansai ben*).‡

SDL offers their FreeTranslation website,§ and Babylon offers comparable translation services.¶

Each of these machine translation services deserves to be explored, if for no other reason than to determine which best suits your needs. For those who find their capabilities too limiting, consider some of the commercial options, which are covered next.

Commercial Machine Translation Services

In the spirit of the "you get what you pay for" expression, some machine translation services are commercial, meaning that you payment is via subscription or based on usage.

LEC offers machine translation subscriptions, such as *Transate DotNet* and *Translate2Go*.** Likewise, LogoVista offers their *e-Trans* service.†† And SYSTRAN offers various translation services.‡‡

Language-Learning Aids

This section provides information on some of the Japanese learning aids that are available—this does not mean that similar software for Chinese, Korean, or Vietnamese does not exist.

The learning of a foreign language can often be supplemented or reinforced through the use of software.§§ There are many software-based learning aids available, far too many to exhaustively list here. If you plan to study—or have already studied to some extent—a language such as Chinese, Japanese, Korean, or Vietnamese, I encourage you to seek out software-based learning aids.

* *http://babelfish.yahoo.com/*

† *http://dictionary.goo.ne.jp/*

‡ *http://translation.infoseek.co.jp/*

§ *http://www.freetranslation.com/*

¶ *http://translation.babylon.com/*

** *http://www.lec.com/translation-subscriptions.asp*

†† *http://e-trans.logovista.co.jp/*

‡‡ *http://www.systransoft.com/translation/translation-products/online-services/*

§§ But make no mistake, although software-based learning aids are useful in the language-learning process, they are by no means a substitute for genuine human interaction, especially during the early stages of learning.

Table 11-15 lists some of the language-learning aids of which I am aware, but make no mistake, there are many, many more available. Some of these applications are much more than simply language learning aids—some also function as electronic dictionaries or full-featured word processors, among which Wenlin is a prime example.

Table 11-15. Language-learning aids

Software	Platforms	Availability
Chinese Character Tutor	Windows	Commercial software by Flashware International[a]
Kanji-Flash/BTJ	MS-DOS	Commercial software by Kanji-Flash Softworks[b]
ReadWrite, FlashCards, and Dictionary	Windows	Commercial software by Declan Software[c]
Rosetta Stone	Mac OS X, Windows, and Web	Commercial software by Rosetta Stone[d]
Wenlin	Mac OS X and Windows	Commercial software by Wenlin Institute[e]

a. *http://www.bridgetochina.com/*
b. *http://ourworld.compuserve.com/homepages/KanjiFlash/*
c. *http://www.declan-software.com/*
d. *http://www.rosettastone.com/*
e. *http://www.wenlin.com/*

I suggest that you also explore World Language Resources as a source for language-learning products.*

Many readers may also require classroom study of the Japanese language. If local classes are not available, or if there is a group within a company that wants to learn Japanese, there are still options open. Some schools, such as The University of Wisconsin–Madison (which I attended), offer Technical Japanese programs that cater to the working professional through distance education.†

* *http://www.worldlanguage.com/*

† *http://metj.engr.wisc.edu/*

CHAPTER 12
Web and Print Publishing

This chapter wraps up the book by first delving into the subject of email and its CJKV implications, to include coverage of popular email clients. The remainder of the chapter focuses on the topics of publishing in the contexts of the Web and in print, which means discussions surrounding markup languages for web publishing, and Adobe Acrobat and PDF for print publishing. All of these publishing technologies provide adequate CJKV support, as long as they're configured and used correctly. And, as you will discover, Acrobat and PDF play an important role today in the realm of print publishing. As you are no doubt aware, there has been explosive growth of the Web in recent years. Virtually every company has established a website for letting customers know of their products and services, and for some companies, their web page has become their primary venue for providing information to their customers. Television, radio, and billboard advertisements these days often boldly include URLs. But, how do you go about displaying or creating web pages that include CJKV text? These are good questions with reasonable and understandable answers.

The handling of CJKV text within the context of email clients clearly falls into the realm of *information interchange* (信息交换 *xìnxī jiāohuàn* or 資訊交換 *zīxùn jiāohuàn* in Chinese, 情報交換 *jōhō kōkan* in Japanese, and 정보교환/情報交換 *jeongbo gyohwan* in Korean), whereby text data is sent from one computer or mobile device to another without, one hopes, any loss of data. How CJKV text is handled internally by a single computer system is not necessarily the same as the external handling of the same data—these days, it's almost guaranteed that it is different. Here we can make a distinction between an internal and external code. An *internal code* is one which is most efficiently processed directly on a computer system. Examples of internal codes include the various locale-specific instances of EUC encoding, such as EUC-CN, EUC-JP, EUC-KR, and EUC-TW, and any of the encoding forms for Unicode, specifically UTF-8, UTF-16, and UTF-32. An *external code*, however, is used somewhat as a machine-independent encoding that allows for the interoperability of data from one encoding to another—an external code is also called an *information interchange code*. Examples of external codes include the various instances of ISO-2022 encoding. Such encodings are designed to be transmitted quite reliably through most email networks.

First, it is useful to consider an example of information interchange in action. Figure 12-1 illustrates how Japanese text data can be sent or transmitted from an OS that processes Shift-JIS encoding to one that processes EUC-JP encoding, using ISO-2022-JP encoding as the common encoding for electronic transmission purposes. ISO-2022-JP encoding is thus used as the information interchange encoding. The ISO-2022-JP encoding step may be bypassed if the Japanese data are being moved by other means, such as by a direct connection, wireless transfer, or by removable media (such as a memory stick, CD, or DVD).

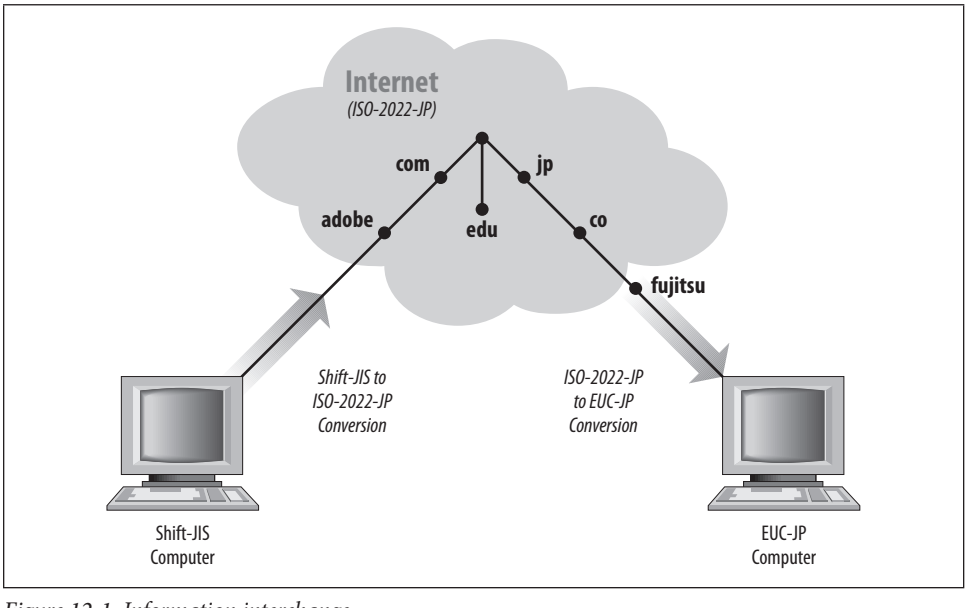


Figure 12-1. Information interchange

Nevertheless, true information interchange is achieved only when data is moved from one platform to another—and perhaps even from one encoding to another—with absolutely no loss of information. Of course, loss of information is considered to be a bad thing. Naturally, preserving information is a very good thing. Information interchange is usually a simple affair when using the ASCII character set, but in the case of CJKV text, there are more problems and complications. Examples are different encoding methods for the same locale—such as ISO-2022-KR, EUC-KR, Johab, and Unicode for Korean—and multiple character sets for a single locale, though Unicode has effectively put a damper on that issue. As you should have learned in Chapter 4, not all encodings for a specific locale support all of its character sets. An excellent example is the lack of support for the JIS X 0212-1990 character set in Shift-JIS encoding. All of these factors must be taken into consideration when deciding how to best implement information interchange in your working environment.

And, times and technologies change. Some of the Internet applications and services that were covered in the first edition of this book, such as News clients, have become outdated. Some, such as FTP, are still useful from time to time, though much of FTP's functionality has been taken over by web browsers. To a great extent, web forums have effectively replaced News as the preferred medium through which to exchange information or ideas within an interest group.

Line-Termination Concerns

No matter how complex the Web becomes, files still need to be transferred. In fact, one can argue that file transfer is the very nature of the Web. Furthermore, each platform still maintains its own notion of what constitutes line termination. As the Web becomes more complex, more and more platforms interact with one another. In the past, services such as FTP (*File Transfer Protocol*), *rlogin*, and *telnet* were common methods for connecting to remote servers, for the purpose of transferring files or to perform tasks. To some extent, these services are still in use today, but sftp (*secure FTP*) and ssh (*secure shell*) are preferred by many for their added security features. I know, because I still use them as part of my daily work.

While it is not the purpose of this book to show you how to establish an FTP connection, nor how to transfer files using FTP, there are some valid concerns that relate to the transfer of CJKV data. In case you were not aware, there are effectively two ways to use FTP:

- Using the *ftp* program itself, which is available on Mac OS X, Unix, and Windows.
- Using a web browser, and specifying a URL that includes *ftp* as the Web protocol, instead of the more common *http*.

The first thing to remember is that nearly all legacy CJKV encoding methods, such as Big Five, EUC, ISO-2022, Shift-JIS, and so on, do not constitute binary data. The same is true of UTF-8 encoding. Of course, some of these encoding methods use generous amounts of eight-bit data, and some even control characters. But, text data that is encoded according to these encoding methods is still considered to be text data. In most cases, though, transferring CJKV text through a binary FTP connection is not harmful, but will result in the preservation of the original line termination. For those who are not aware, different OSes use different characters or different sequences of characters to terminate lines of text. Table 12-1 lists the three most common line-termination methods, along with the OSes that make use of them.

Table 12-1. Line-termination conventions

Line termination	Encoding	OSes
Carriage return[a]	13 (0x0D or U+000D)	Mac OS and Mac OS X
Carriage return + Line feed[b]	13 (0x0D or U+000D) + 10 (0x0A or U+000A)	MS-DOS and Windows

Table 12-1. Line-termination conventions

Line termination	Encoding	OSes
Line feed[c]	10 (0x0A or U+000A)	Unix and Mac OS X[d]

a. Also expressed as CR.

b. Also expressed as CR + LF or CR + NL (*New Line*).

c. Also expressed as LF or NL.

d. Some aspects of Mac OS X, such as working in the *Terminal* application, prefer this line-termination convention.

A nonbinary file transfer session will convert the line-termination characters appropriately. For example, if you are using Windows and connect to a Linux or Unix server to transfer a file from Linux or Unix to Windows, a nonbinary FTP connection will change line termination appropriately, from LF to CR+LF. For those who use Unix, or for Mac OS X using the *Terminal* application, the following *tr* command changes line termination, from carriage returns (CR) to line feeds (LF):

```
tr '\015' '\012'
```

And, when transferring files that were authored on Windows to Unix or Linux, the following *tr* command performs a similar function by simply removing the carriage returns (CR):

```
tr -d '\015'
```

Files that contain real binary data must be transferred in binary transfer mode; otherwise, their data will most certainly become corrupt. Compressed files are the most commonly encountered binary files. Binary data, for example, may contain byte values that correspond to carriage-return or line-feed characters, but they are not to be interpreted as those characters. So, when these characters are converted, the entire file becomes damaged, and repair is almost always impossible.

Some encodings must be treated as binary data, such as the following two Unicode encodings: UTF-16 and UTF-32. UTF-8 encoding can be treated in nonbinary mode with no worries. My recommendation is to effect FTP transfers in binary mode when there is any doubt.

Email

One of the most commonly used Internet services is, without a doubt, email, and it still represents the most basic form of today's electronic communication. Email, of course, is the abbreviated form of electronic mail. Emails can range from short text messages sent from a mobile device, such as a cell phone, to elaborate texts that include numerous attachments. People often wonder how to send and receive email in languages other than English. That is what this section is all about. As a recurring theme of this book, Unicode has been very effective in trivializing the ability to send and receive email using characters beyond those in ASCII. It is no different for email.

Email is still very much a plain-text paradigm, and is likely to be so for many years. Of course, it is possible to use stylized text in an email message, but as the message becomes thread-like, with multiple replies or forwards, much of the "style" is stripped away. And, for some email clients, such as mobile devices, the plain-text paradigm adds a layer of simplicity and robustness to an otherwise complex electronic device.

Sending Email

Today, sending email has become trivial and is now as simple as launching your favorite email client, typing in one or more email addresses, composing the message, and then sending it. In the past, email clients tended to be much more text-based. The user was once exposed to some of the nitty-gritty details, such as encoding conversion and similar operations. The current generation of email clients have successfully insulated users from these details, and they can instead focus on using email as a tool.

Looking back only a decade, one was forced to make a few preparations before a CJKV text data could be reliably transmitted through email. I originally suggested that the following guidelines be adhered to as closely as possible:

- Break long lines to less than 80 columns (75 columns or fewer was preferred).
- Compose the document using a monospaced font, because most email clients used monospaced fonts for display purposes.
- Do not include any half-width katakana characters since they are not fully supported in all environments (this applies only to Japanese).
- Convert the text file to an ISO-2022 encoding, according to the appropriate RFC.

Speaking of ISO-2022 encoding and RFCs, Jun Murai, Erik van der Poel, and Mark Crispin wrote RFC (*Request For Comments*) 1468, *Japanese Character Encoding for Internet Messages*. This RFC described how Japanese was to be encoded for use in email, and paved the way for comparable RFCs that specified similar treatment for Chinese and Korean.* Table 12-2 lists these RFCs, along with the encodings they defined and the languages they supported. Those that are shaded were widely used, and in some circles are still used today. The others were defined too late, and never became widely used, if used at all.

Table 12-2. RFCs that define ISO-2022 encodings

RFC	Encoding	Character sets
1922	ISO-2022-CN, ISO-2022-CN-EXT	ASCII, GB 2312-80, CNS 11643-1992
1468	ISO-2022-JP	ASCII, JIS-Roman, JIS C 6226-1978, JIS X 0208-1983[a]
2237	ISO-2022-JP-1	ISO-2022-JP plus JIS X 0212-1990

* *http://www.ietf.org/rfc.html*

Table 12-2. RFCs that define ISO-2022 encodings

RFC	Encoding	Character sets
1554	ISO-2022-JP-2	ISO-2022-JP plus JIS X 0212-1990[b]
1557	ISO-2022-KR	ASCII, KS X 1001:1992

a. Support for JIS X 0208-1990 and JIS X 0208:1997 is also implied.

b. As you've read in Chapter 4, ISO-2022-JP-2 also supports GB 2312-80, KS X 1001:1992, and two parts of ISO 8859, but the GB 2312-80, KS X 1001:1992 character sets are better handled through RFCs 1922 and 1557, respectively.

Interestingly, although Unicode has become broadly adopted in today's OSes and applications, ISO-2022 encoding is still used for email messages to a great extent. But, the conversion between Unicode and the appropriate ISO-2022 encoding is handled by email clients.

Receiving Email

Receiving CJKV text is considerably easier than sending it, especially using today's email clients. Displaying text is obviously easier than composing it. Still to this day, whether or not CJKV text is displayed properly in your email client depends heavily on the extent to which your OS has the ability to display CJKV text.

Some email clients, such as those that ran on VMS systems, did not allow control characters to function—that is, if they didn't simply strip them out! This had the effect of rendering CJKV text unreadable within the email client even if you were using CJKV-capable terminal-emulation software. For example, you may sometimes see some text that looks like this:

```
$B$3$l$OOBJ8$NJ8>O$NNc$G!"$=$l$O(JEnglish$B$NJ8>O$NNc$G$9!#(J
```

There sure are a lot of $ (dollar) characters, which happen to represent the first byte of hiragana characters. This should tell you that something has gone wrong.

It was possible to trick or coerce email clients into permitting control characters to perform their proper functions. For example, on VMS systems, you could make use of the *extract* command followed by `tt:` (two t's followed by a single colon) to accomplish this. Below is an example command line within the VMS email client:

```
Mail> extract tt:
```

If this worked, and if the CJKV text had not been damaged, the previous text should have displayed properly as a mixture of CJKV and English text, such as the following sample Japanese text:

これは和文の文章の例で、それはEnglishの文章の例です。

If there was no method with which you could coerce your email client into displaying CJKV text, you were then forced to save the message to a file, exit the email client, and then attempt to view the message through other means. The most successful method almost always involved saving the email message as a file outside your email client, and

then, if necessary, downloading the file to your computer. Unix newsreaders often had a similar problem, and users had to resort to saving the article as a text file, and then view it through other means.

We are fortunate that today's email clients do an excellent job in sheltering and insulating us from resorting to cumbersome methods to perform seemingly simple and trivial tasks, such as displaying email messages.

Email Troubles and Tricks

Thankfully, the likelihood of encountering issues when sending and receiving email with CJKV content has diminished. With the current generation of email clients being dedicated applications or web browsers, the self-contained nature of such environments has helped to a great extent. Still, most of the problems that you are likely to encounter when sending CJKV text through email relate to encodings becoming damaged.

When using a text-based email client, while using a terminal, there are times when an erratic encoding issue makes it seem as though your terminal went nuts. And, there are times when the escape sequences that are commonly used for Japanese text become scrambled or corrupted, for one reason or another. This might be due to a poorly written terminal application, or simply a problem with the integrity of the Japanese text itself, such as it terminating in two-byte mode. This can leave you stuck in two-byte mode, meaning that text, such as your system prompt, will be interpreted as two bytes per character and will make no sense whatsoever. A solution to this problem is to create a short shell script that outputs a one-byte–character escape sequence when invoked. The following two lines can be added to your *.cshrc* file (if you are using the C shell on a Unix system):

```
set e = "`echo x | /bin/tr x \\033`"
alias ko 'echo "${e}(J"; echo "*** FORCED KANJI-OUT ***"'
```

The first line sets the value of the variable *e* to be the same as the escape character. This variable is then used in the second line to complete the one-byte–character escape sequence. It is not wise to directly use an escape character in this sort of settings file—it may be detected as a redundant escape sequence by certain CJKV-capable text editors, and subsequently—and quite appropriately, I might add—deleted. When invoked, this newly-established alias "ko" outputs two lines to the terminal: the first is a valid one-byte–character escape sequence, and the second is simply a line that informs you that you have successfully returned to one-byte mode.

Email Clients

The availability of email clients always seems to be on the rise, perhaps because email is the most widely used—and thus most frequently abused—Internet service. In case you are not aware, an email client is also known as an *Mail User Agent* (MUA). Fortunately, obtaining an CJKV-capable email client has become an easier task than compared to several years ago. Much of this change is due to Unicode, and the fact that supporting Unicode

helped to trivialize, or at least ease, the effort that is necessary to provide multilingual support to customers.

Using a web browser as an email client

The advantages of using a web browser as an email client is that you can access your email from almost any computer. And, given the high level of multilingual support in today's OSes and web browsers, as long as the email service you use has adequate multilingual support, you will experience little or no problems. Google and Yahoo! are examples of companies that offer email accounts that are easily accessed through the use of any web browser, and they provide an excellent level of multilingual support. This is precisely why their email services are popular: they work.

Using a mobile device as an email client

Mobile devices, including cell phones, can serve as convenient email clients. The primary issue that you are likely to encounter is the extent to which they can—or cannot—handle CJKV text.

Unfortunately, the current state is rather poor, which is likely due to font and input method limitations. The size of these devices necessitates that they be carefully configured and tailored for specific markets. Those sold in Japan, for example, support Japanese well. Those sold in the U.S. have very poor or nonexistent support for CJKV text.

Becky!—Windows

RimArts has developed a Windows-based email client called Becky!, and its standard version includes CJKV support.* Although this email client was developed many years ago, it is still being maintained and developed.

Eudora

Eudora, developed by QUALCOMM, remains one of the most popular email clients for Mac OS X and Windows users.† Various versions have been available, from commercial to free or inexpensive Light versions. An open source version, currently being called *Penelope*, is in development, and is being coordinated with Thunderbird (which is covered later in this section).‡

Although localized versions have been developed, Eudora's multilingual support has been relatively weak, especially when compared to other contemporary email clients.

* *http://www.rimarts.co.jp/becky.htm*

† *http://www.eudora.com/*

‡ *http://wiki.mozilla.org/Penelope*

Mac OS X Mail

The very first versions of Mac OS X included a multilingual email client that is simply called Mail.* This happens to be the primary email client that I use, and I find its multilingual features to be quite robust. It has other useful features, such as the ability to view single-page PDFs inline, as part of email, instead of needing to open them using Acrobat or Adobe Reader. Naturally, Mail has a home turf advantage, because it is provided as part of Mac OS X and is thus supported by Apple.

Microsoft Entourage—Mac OS X

Microsoft provides an application called Entourage as the email client that is included with Microsoft Office for Mac OS X.† Like other components of Microsoft Office, Entourage provides multilingual support. And, unlike other email clients, Entourage is more than merely an email client, and it also serves as a calendaring application for managing schedules.

Microsoft Outlook and Outlook Express—Windows

The Windows version of Microsoft Office includes an application called Outlook that serves as an email client, and also as a calendaring application.‡ Interestingly, Outlook Express, the freely available relative of Outlook, has historically provided users with better overall multilingual support.

Mutt—Linux and Unix

Mutt is a text-based email client for Linux and Unix OSes that provides multilingual capabilities.§ This email client can even be used on Mac OS X, thanks to Unix being under the hood. Mutt's slogan is: *All mail clients suck. This one just sucks less*.

PowerMail—Mac OS X

CTM Development has developed a Mac OS X email client called PowerMail.¶ Its multilingual support is due to the fact that it uses Mac OS X APIs to accomplish this feat, which is the way it should be done.

Thunderbird

The same organization that developed Firefox, specifically the Mozilla Foundation, developed an equally robust email client called Thunderbird.** Like Firefox, the multilingual

* *http://www.apple.com/macosx/features/mail.html*
† *http://www.microsoft.com/mac/products/entourage2008/*
‡ *http://office.microsoft.com/outlook/*
§ *http://www.mutt.org/*
¶ *http://www.ctmdev.com/powermail/*
** *http://www.mozilla.com/thunderbird/*

support in Thunderbird is broad. Furthermore, it is available for FreeBSD, Linux, Mac OS X, and Windows.

mh-e—GNU Emacs

I have always been a fan of the GNU Emacs, and at one time used its mh-e package as my primary email client. The mh-e package is based on, and depends on the availability of, the MH email system. When used with GNU Emacs version 20 or later, the mh-e package inherits its CJKV support.

Network Domains

The Web, as it is used today, began as a defense-funded network known as ARPANET. The identity of each node on the network was first categorized into different domains, which are now referred to as *Top-Level Domains* (TLDs). This tells users, for example, that *adobe.com* represents a commercial node, and that *nasa.gov* is a government node. The so-called "classic" gTLDs (*generic Top-Level Domains*) are listed in Table 12-3.

Table 12-3. Classic Top-Level Domains

Domain	Description
com	Corporations
edu	Educational institutions
gov	Government agencies
mil	Military
net	Network providers
org	Nonprofit organizations

In 1998 and beyond, however, additional gTLDs were established to provide a more meaningful set of Top-Level Domains for today's commercial web. Table 12-4 lists these newly established web domains.

Table 12-4. New Top-Level Domains

Domain	Description
aero	Aviation-related businesses
asia	Asia-Pacific region
biz	Businesses
cat	Catalan language and culture
coop	Cooperatives
info	Informative websites

Table 12-4. New Top-Level Domains

Domain	Description
jobs	Employment-related websites
mobi	Mobile devices
museum	Museums
name	Individuals
post	Postal services
pro	Professionals
tel	Web-communication services
travel	Travel industry

Besides these gTLDs, there are also ccTLDs (*country code Top-Level Domains*), which cover specific countries, and, as its name suggests, by country code. The ccTLD for the United States is *us*. The United States is also divided into regional subdomains, all of which correspond to a state. The State of California, for example, is registered as *ca.us*. The use of the classic gTLDs has traditionally been restricted to sites in the United States, due to the initial U.S.-centricity of the Web, but that trend has changed. Now, it is almost impossible to determine the physical location of websites that use the domains specified in Tables 12-3 or 12-4.

Keep in mind that the networking environment throughout the world is ever changing and improving. URLs are included for each *Network Information Center* (NIC) so that more up-to-date information on each domain can be obtained.

Internationalized Domain Names

The widespread adoption of Unicode as the standard character set and encoding enabled internationalization of not only OSes and applications, but of the Web's namespace as well. Prior to Unicode, the multitude of character sets and encodings prevented this from happening, but Unicode leveled the proverbial playing field, and every character in Unicode was given the same treatment, whether it has its roots in ASCII or is deep within CJK Unified Ideographs Extension B in Plane 2. As long as a single character set is used, IDNs (*Internationalized Domain Names*) are now possible.

ICANN (*Internet Corporation for Assigned Names and Numbers*) has published the guidelines for IDNs.* Note that some countries have implemented IDNs to an extent that is not yet recognized by ICANN, though this has opened up recently. IANA maintains a collection of IDN tables, organized by TLD and language, which indicate specific Unicode code points that are permitted on a per-registry basis.†

* *http://www.icann.org/topics/idn/*

† *http://www.iana.org/domains/idn-tables/*

The CN Domain

The CN domain, which is a ccTLD that covers China, is managed by CNNIC (*China Internet Network Information Center*).* This domain includes six subdomains, as indicated in Table 12-5.

Table 12-5. CN subdomains

Subdomain	Description
ac	Academic or scientific research institutions
com	Corporations
edu	Educational institutions
gov	Government agencies
net	Network providers
org	Organizations

There are also regional subdomains that indicate the province, such as *bj* for Beijing. Also of interest is that CNNIC has proposed Chinese names for these subdomains, such as 公司 for *com*, 网络 for *net*, and even 中国 for *cn* as a ccTLD.

The HK Domain

The HK domain, which is a ccTLD, covers Hong Kong, which is now a Special Administrative Region (SAR) of China. HKDNR (*Hong Kong Domain Name Registration Company*) manages this domain.† Table 12-6 lists the six HK subdomains, along with their official names in Chinese.

Table 12-6. HK subdomains

Subdomain	In Chinese	Description
com	公司	Corporations
edu	教育	Educational institutions
gov	政府	Government agencies
idv	個人	Individuals
net	網絡	Network providers
org	組織	Organizations

* *http://www.cnnic.net.cn/*

† *http://www.hkdnr.hk/*

As Table 12-6 indicates, it is possible to both register HK subdomains and represent them in Chinese. But, bear in mind that these subdomain names, when expressed in Chinese, are not yet officially recognized by ICANN.

The JP Domain

The ccTLD that specifies websites in Japan uses the two-letter country code JP. JPNIC (*Japan Network Information Center*) manages the JP domain.* Table 12-7 lists the JP subdomains.

Table 12-7. JP subdomains

Subdomain	Description
ac	Academic institutions
ad	JPNIC members
co	Corporations
ed	Educational institutions for minors
go	Government agencies
gr	Individual entities
lg	Local government agencies
ne	Network providers
or	Organizations

Some entities, such as NTT, originally belonged to the JP domain itself, as *ntt.jp*—it did not belong to a subdomain, but rather formed its own subdomain of sorts. NTT is now properly *ntt.co.jp*, under the expected *co* subdomain.

The KR Domain

The KR domain, which is a ccTLD that is managed by NIDA (*National Internet Development Agency of Korea*), is primarily intended for websites that are located in South Korea.† Table 12-8 lists the KR subdomains.

Table 12-8. KR subdomains

Subdomain	Description
ac	Academic institutions
co	Corporations

* *http://www.nic.ad.jp/*

† *http://www.nida.or.kr/*

Table 12-8. KR subdomains

Subdomain	Description
es	Elementary schools
go	Government agencies
hs	High schools
kg	Kindergarten
mil	Military
ms	Middle schools
ne	Network providers
or	Organizations
pe	Personal
re	Research institutes
sc	Other schools

Note that geographical domains, such as *seoul.kr* and *busan.kr*, also exist, along with IDNs, such as 한글*.kr*.

The KP domain also exists, which corresponds to North Korea, but as of this writing, there are a mere two websites registered under the KP domain.

The TW Domain

The TW domain is managed by TWNIC (*Taiwan Network Information Center*).* Table 12-9 lists the TW subdomains.

Table 12-9. TW subdomains

Subdomain	Description
club	Clubs
com	Corporations
ebiz	E-businesses
edu	Educational institutions
game	Gaming websites
gov	Government agencies
idv	Individuals
mil	Military

* *http://www.twnic.net.tw/*

Table 12-9. TW subdomains

Subdomain	Description
net	Network providers
org	Organizations

Like China and Hong Kong, Taiwan also allows some of the subdomains to be represented in Chinese, specifically 商業 for *com*, 網路 for *net*, and 組織 for *org*. Of course, ICANN does not yet recognize these subdomains when expressed in Chinese.

The VN Domain

The VN domain is managed by VNNIC (*Vietnam Internet Network Information Center*).* Table 12-10 lists the VN subdomains.

Table 12-10. VN subdomains

Subdomain	Description
ac	Academic institutions
biz	Businesses
com	Corporations
edu	Educational institutions
gov	Government agencies
health	Medical agencies
info	Informative websites
int	International organizations
name	Individuals
net	Network providers
org	Organizations
pro	Professionals

Content Versus Presentation

Now we turn our attention away from email and toward the area of publishing, whether it is for the Web, for printing, or for both mediums. Everything about this book culminates toward publishing, and these two forms represent the vast majority of today's documents.

* *http://www.vnnic.vn/*

HyperText Markup Language (HTML), an application of *Standard Generalized Markup Language* (SGML, described in ISO 8879:1986), provides the author with full control over the content of web documents, but it is up to the individual browser to handle presentation. HTML is effectively the publishing language of the Web. When including graphics in your web documents, for example, you must keep in mind that text-based browsers, such as Lynx, exist, and therefore cannot display the graphics directly.

Printed materials, to include scans, preserve the presentation of documents, but because they are not machine-readable, they have no content *per se*. Content can be derived through the use of *Optical Character Recognition* (OCR) or by manually entering the data.

Portable Document Format (PDF), on the other hand, provides the author with full control over both the content and presentation of web documents. Both aspects are *preserved*, which effectively means that the "look and feel" of the original document is fully maintained. Adobe Acrobat version 3.0, based on PDF version 1.2, was the first release that provided minimal CJKV support. It could also plug into Netscape Communicator so that PDF files display directly in the browser window (as opposed to running Adobe Acrobat as a separate process or application). Adobe Acrobat version 4.0, based on PDF version 1.3, provided significantly more enhanced CJKV support, such as the ability to embed CJKV fonts.

Cascading Style Sheets (CSS) represents a language for controlling the presentation of web pages, to include color, fonts, layout, and other aspects of how web pages display.* CSS is used in conjunction with HTML.

Extensible Markup Language (XML), which was once thought of as a possible replacement for HTML, provides much better control over content and also has better overall CJKV support.

HTML, CSS, XML, and other advanced web developments are coordinated by the World Wide Web Consortium (W3C).† I encourage you to explore the W3C website for the latest information on these and related standards, because this represents an ever-changing area.

In order to determine which publishing method is best for your needs, you first must determine or decide whether it is important to preserve content, presentation, or both. You also need to consider the delivery method and the potential readership. Also bear in mind that the majority of web browsers can display PDF files within the application, as though the file were a typical web page, so to some extent you can treat PDF files as though they were normal web documents.

* *http://www.w3.org/Style/CSS/*

† *http://www.w3.org/*

Web Publishing

Web publishing, quite simply, is the development of web pages. The vast majority of web pages are written using HTML, and various related technologies have been developed to simplify the development and management of web pages, such as *Active Server Pages* (ASP), *JavaServer Pages* (JSP), and *Personal Home Page* (PHP). Some technologies, such as Adobe Flash* and Adobe AIR,† were specifically designed to deliver enhanced capabilities to the Web.

Web Browsers

The most popular and utilitarian web application is the now infamous web browser. It is hard to imagine accomplishing much of the work that I do without the conveniences and features of a browser. Web browsers act as portals. To some extent, they act as platforms into and of themselves. In fact, some technologies effectively use browsers as their platform. Take, for instance, Adobe Buzzword and Google Docs. These online word processors use web browsers as their platform. The web browser can thus be thought of as a portal to the Web.

Years ago, there were two major web browsers that competed for top position: Internet Explorer and Netscape Communicator. While Internet Explorer is still very popular, other web browsers have been developed. In other words, there are choices. The strong competition between Internet Explorer and Netscape Communicator resulted in better web-browsing applications, and ultimately became a "win" situation for all users—especially for those who needed to display CJKV web pages.

One common feature of a browser that is multilingual in nature is the ability to automatically detect the encoding of a web page, in case one is not declared, and the ability to manualy override the encoding in case automatic encoding detection fails for whatever reason. The underlying OSes on which these web browsers run also include improved multilingual support.

Chrome

Chrome is the name of Google's own web-browsing application.‡ Although the initial release of Chrome was for Windows, clearly in an effort to compete with Microsoft's Internet Explorer, Linux and Mac OS X versions are in the works. Given Google's strong multilingual capabilities, I expect no less from Chrome.

* *http://www.adobe.com/products/flash/*

† *http://www.adobe.com/products/air/*

‡ *http://www.google.com/chrome/*

Firefox

Firefox, developed by the Mozilla Foundation, is the most popular web browser that is not developed by an OS vendor.* Clearly, Internet Explorer and Safari have home-turf advantages on their respective OSes, yet Firefox can still compete. Like Internet Explorer and Safari, the multilingual functionality of Firefox is impressive. I use Firefox as my preferred web browser, and the only functionality that I miss is the ability to display PDF files directly in the browser.

Internet Explorer

Of the web browsers in use today, Microsoft's Internet Explorer, often abbreviated as simply IE, is one of the oldest, if not oldest.† And, because it is the standard web browser for Windows, it is ranked as the most popular. Internet Explorer is included with Microsoft Windows, and is also available at no charge from Microsoft's website. At one point, Microsoft developed a Mac OS X version of Internet Explorer, but development has ceased, most likely due to Apple's release of their own web browser called Safari. Like Safari, Internet Explorer has the ability to display PDF files directly in the browser.

Lynx

Believe it or not, the text-based web browser called Lynx is still in development.‡ Given the extent to which graphics and other nontext elements are used in today's web pages, one would think that development of text-based web browsers would surely have ceased. Lynx was initially popular for users with disabilities, such as those who are blind. But, A11Y§ has improved significantly in GUI-based browsers, so Lynx's use within this community has naturally decreased.

Of course, the Mac OS X version is run through the use of the *Terminal* application.¶ The continued development and use of Lynx makes it clear that "plain text" still has meaning, power, and purpose.

Opera

Opera Software has developed a very popular web browser called Opera.** Unlike other web browsers, which typically come in only one form, there are several versions of Opera available, from the traditional web browser for desktop or laptop computers to versions that are tailored for use on mobile devices.

* *http://www.mozilla.com/firefox/*

† *http://www.microsoft.com/windows/products/winfamily/ie/*

‡ *http://lynx.browser.org/*

§ An abbreviated form of *accessibility*.

¶ *http://www.apple.com/downloads/macosx/unix_open_source/lynxtextwebbrowser.html*

** *http://www.opera.com/*

Safari

Not long after Apple released Mac OS X, they also released their own web browser called Safari.* It is based on Apple's open source *WebKit*.† Like other components of Mac OS X, Safari has excellent multilingual support. One incredibly useful feature of Safari is its ability to display PDF files directly in the browser, instead of using Adobe Reader, the commercial viewer, or Apple's own *Preview* application.

Displaying Web Pages

Today's OSes and web browsers have, to a great extent, trivialized the displaying of web pages. Unicode has obviously played a role in this. The Fallback Font mechanisms that the OSes provide, which applications such as web browsers can take advantage of, also play a role in the ability to display arbitrary text in web pages.

In the past, however, the process was more complex and required a greater level of user intervention. The following list chronicles four common methods that were once necessary to correctly display CJKV text in previous-generation web browsers:

- Obtain a web browser that includes CJKV support, such as Netscape Communicator or Internet Explorer. An underlying CJKV-capable OS may be required.
- Patch your web browser to support CJKV text. Again, an underlying CJKV-capable OS may be required.
- Use a gateway that transforms CJKV text into graphic images, such as *Shodouka*—this technique was not terribly useful for text-based web browsers, such as Lynx.
- Use an application that forcibly displays CJKV text, such as NJStar Software's NJStar CJK Viewer. This technique was also quite useful outside the context of web-browsing applications and could be used to view CJKV text in most non-CJKV applications.

Note that the gateway method required that you use a graphics-based browser, and the results could be painfully slow. Text-based browsers, such as Lynx, do not (and, obviously, cannot) directly support graphics or images of any kind. It is possible, however, for Lynx to use a "helper application," such as *xv*, for displaying graphics. Worse yet, web pages that made use of Java, such as Java Applets or JavaScript, are not usable in today's text-based browsers. Too bad.

HTML—HyperText Markup Language

HTML, first documented in RFC 1866, *Hypertext Markup Language—2.0*,‡ is a language used to describe the content and structure of web documents. It originally specified ISO

* *http://www.apple.com/safari/*

† *http://www.webkit.org/*

‡ *http://www.ietf.org/rfc/rfc1866.txt*

8859-1:1998 (also known as ISO Latin 1 or ISO-8859-1) as its default character set and encoding. RFC 2070, *Internationalization of the Hypertext Markup Language*, effectively changed the default character set for HTML to ISO 10646-1:1993 (Unicode), and also extended HTML to be more suitable for multilingual purposes.* The latest HTML specification is version 5.0.† W3C (*World Wide Web Consortium*) now manages the internationalization aspects of HTML, and it provides documentation, tips, and test cases.‡

In addition to HTML, there is also *Extensible HyperText Markup Language* (XHTML), which is HTML reformulated using XML. Perhaps as a best of both worlds, XHTML thus combines the strength of HTML and the flexibility of XML. The latest version of HTML, specifically version 5.0, brings HTML and XHTML back together.

CSS provides the web page author with more control over how web pages are presented, through the specification of color, fonts, some aspects of layout, and other presentation-related attributes. Given the multilingual nature of many fonts, there are clearly CJKV-related implications when such fonts are specified through CSS.

Some companies, such as Microsoft and Netscape Communications, have defined their own extensions to HTML in the past, in the form of additional tags that were above and beyond the HTML specification proper, at least at that time. These tags were not part of the official HTML specification and were guaranteed to work only in specific web browsers. The world has fortunately become a much better place, because today's web browsers are much more compliant than those developed several years ago. In other words, web-browser development has matured.

With the large number of HTML-related books on the market, it can be difficult for the beginner to choose an appropriate title. My favorite is Chuck Musciano and Bill Kennedy's *HTML & XHTML: The Definitive Guide*, Sixth Edition (O'Reilly Media, 2006); as its title suggests, it covers both HTML and XHTML.§ For those who merely need a convenient HTML reference, Jennifer Niederst Robbins' *HTML & XHTML Pocket Reference*, Third Edition (O'Reilly Media, 2006) is highly recommended.¶ For learning more about CSS, I suggest reading David McFarland's *CSS: The Missing Manual* (O'Reilly Media, 2006).**

Authoring HTML Documents

The most widely used legacy CJKV encodings—EUC, ISO-2022, Big Five, Shift-JIS, and so on—all support a mixed one- and two-byte code space.†† All of the HTML tags can be

* *http://www.ietf.org/rfc/rfc2070.txt*

† *http://www.w3.org/TR/html5/*

‡ *http://www.w3.org/International/*

§ *http://oreilly.com/catalog/9780596527327/*

¶ *http://oreilly.com/catalog/9780596527273/*

** *http://oreilly.com/catalog/9780596526870/*

†† If you have read Chapter 4 very carefully, you'd know I'm sort of lying. EUC-JP defines a mixed one-, two-, and three-byte code space; EUC-TW defines a mixed one-, two-, and four-byte code space; and UTF-8 defines a mixed one- through six-byte code space.

represented by the characters in the ASCII character set, which is fully supported by virtually all CJKV encoding methods. And, for this reason, UTF-8 is the preferred encoding for HTML documents that include Unicode content. In fact, UTF-8 is most commonly used encoding for web pages today, and I expect its use to only increase.

The following brief example represents a minimal HTML document with no multilingual content:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=us-ascii">
<TITLE>Ken Lunde's Home Page</TITLE>
</HEAD>
<BODY>
<H1>Ken Lunde's Home Page</H1>
</BODY>
</HTML>
```

(I feel that HTML tags and attributes stand out better if they are in uppercase, but they do not need to be—this is more important if you write your own HTML and intend it to be human-readable; the use of uppercase becomes a way to more readily distinguish tags from text.) The following is the same HTML document as just shown, but this time with Japanese content, declaring EUC-JP as the encoding:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=euc-jp">
<TITLE>小林劍のホームページ</TITLE>
</HEAD>
<BODY>
<H1>小林劍のホームページ</H1>
</BODY>
</HTML>
```

As you can see, creating multilingual HTML documents is actually quite simple. The real question is how to go about creating the raw HTML. Do you prefer to use a dedicated application, or your favorite word processor, or do you want to craft your own HTML using a text editor? Depending on the complexity of the HTML document, which method you use can vary.

The current HTML specification defines several attributes and tags that allow the document author to define multilingual aspects of a document. More information about these and other attributes and tags can be found in the HTML specification.

Character Entity References

Besides the obvious method of using characters themselves in their binary forms in a known encoding, it is also possible to use character references in HTML, XML, and SGML. Character Entity References (CERs), known quite well to most HTML content creators, include examples such as `&` for an ampersand. The use of CERs began with

SGML, and was carried over to HTML and XML. The W3C website maintains a list of the CERs that are commonly supported.*

These character references, whether they are CERs or NCRs (to be covered next), are delimited by an ampersand (&, U+0026) and a semicolon (;, U+003B). CERs are most commonly used to represent characters that could otherwise be confused with characters that are used for syntactic purposes. The use of some CERs are more required than others. Some are simply used as a measure of good practice. The CER for an ampersand is required, because the ampersand is used as the initial character for CERs and NCRs.

Numeric Character References

In the context of this book, Numeric Character References (NCRs) are more important than CERs, and have CJKV implications and uses. In case you recall, NCRs were a topic of discussion in Chapter 4. I should point out that every CER can be represented by an equivalent NCR. The `&` CER can thus be alternatively represented by `&#x0026;`, `&#x26;`, or `&#38;`.

When using character codes for character references, traditional SGML allowed only decimal notation. The same was true for HTML up until version 3.2. Hexadecimal is now the preferred notation for NCRs in HTML, SGML (after a correction), and XML. Today, the use of hexadecimal notation is widespread. Hexadecimal notation, of course, is preferred for Unicode-encoded characters, mainly because it is the same notation as used for Unicode scalar values, which are conveniently encoding-independent.

The character encoded at hexadecimal 0xFF (decimal 255) can take on the following two forms:

`&#xFF;` (hexadecimal)
`&#255;` (decimal)

I should also point out that regardless of the encoding of the HTML document, declared or otherwise, when an NCR is used, it is interpreted as a Unicode character, specifically a Unicode scalar value. The `&#xFF;` example just shown is thus treated as though it were U+00FF. Likewise, a typical Unicode scalar value, such as U+4E00, should only be represented as `&#x4E00;`. Likewise, the first character of Plane 2, which is also the first ideograph in CJK Unified Ideographs Extension B, is represented as `&#x20000;`. Even if UTF-16 or UTF-8 were to be used as the encoding of the HTML file itself, neither `&#xD840;&#xDC00;` nor `&#xF0; &#x80;&#x80;` should be used to represent U+20000.

NCRs are recommended when the binary form of the character may introduce some level of confusion or ambiguity. Thus, the use of an NCR. While it may detract from the readability aspects of more standard characters, it provides clarity to some situations. The use of IVSes (*Ideographic Variation Sequences*) comes to mind. Consider the following IVS: <U+528D, U+E0101>. The Base Character, specifically U+528D, can be conveniently used in

* *http://www.w3.org/TR/REC-html40/sgml/entities.html*

its binary form, specifically 劍, and was used in a previous HTML example as the third ideograph:

```
<TITLE>小林劍のホームページ</TITLE>
```

Because the Variation Selector component of an IVS has no standard glyph *per se*, it would be considered good practice to specify them using NCRs, as follows:

```
<TITLE>小林劍&#xE0101;のホームページ</TITLE>
```

All current web browsers recognize NCRs. In fact, the entire Japanese string 小林劍のホームページ can be represented using only NCRs, as follows:

```
&#x5C0F;&#x6797;&#x528D;&#x306E;&#x30DB;&#x30FC;&#x30E0;&#x30DA;&#x30FC;&#x30B8;
```

Of course, readability of the string just went downhill, but to a great extent, content became well-preserved through the use of this notation.

The LANG attribute

HTML's `LANG` attribute provides the ability to tag smaller portions of text with language-specific attributes. The entire HTML file can also be tagged with the `LANG` attribute, by including it as part of the `<HTML>` tag, as follows:

```
<HTML LANG="ja">
```

When the entire HTML document is tagged for a specific language as just shown, it means that any text is interpreted as the language specified (unless specified otherwise through the use of the `LANG` attribute within other tags). If I were to include a short Korean string in the same HTML file, I would then include the `LANG` attribute with the appropriate tag, such as illustrated in the following example:

```
<P LANG="ko">켄 런디의 홈페이지</P>
```

Although it is typically necessary to indicate the character set and encoding of the entire document, which is a topic that is covered next, this special tag allows a similar type of specification on a much smaller scale. This is especially critical when using a character set and encoding that is ambiguous as to what language is being used. Unicode is the best example.

Examples of this tag in use can be illustrated when dealing with Unicode data. A Unicode code point by itself does not carry with it sufficient information to properly render most CJKV texts, particularly ideographs. While such a document may be flagged as being Unicode, the `LANG` attribute allows the HTML author to indicate the language of the text, using whatever granularity is desired, thus allowing the characters to be correctly displayed. W3C has more information about the `LANG` attribute and its uses.*

* *http://www.w3.org/International/articles/language-tags/*

The <META> tag

Did you notice the <META> tag in the previous HTML sample? It is considered good practice to explicitly specify character set and encoding information between the balancing <HEAD> and </HEAD> tags. This sequence uses the <META> tag extension, as described in the specification for internationalizing HTML, and looks like the following, which indicates EUC-KR encoding:

```
<HEAD>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=euc-kr">
</HEAD>
```

I must stress that the <META> tag must be the very first tag after the <HEAD> tag in an HTML file.

IANA (*Internet Assigned Numbers Authority*) is responsible for maintaining the complete list of recognized CHARSET values that today's web browsers recognize.*

Specifying the CHARSET tag is not only useful for HTML documents that include CJKV text, but also for non-CJKV ones. In case an HTML document includes accented Latin characters, such as those found in ISO 8859-1:1998, specifying *iso-8859-1* as the CHARSET value will ensure that the accented Latin characters are not accidentally interpreted as though they were encoded according to a CJKV encoding. Obviously, using Unicode makes this all go away, but it is still good practice to explicitly specify the encoding, which makes life a much more pleasant experience for those staring at a web browser.

The CHARSET value is subsequently passed from the server to the client as part of the HTTP negotiation. There are many other uses for the <META> tag beyond simply providing character set and encoding information to browsers. I suggest that you consult HTML documentation to explore these other uses.

Providing charset information—where and how?

There is a big debate about which method for indicating character set and encoding information is best. There are two methods available:

- Embed the language, character set, and encoding information using the LANG attribute or the <META> tag, as appropriate, and as described in the previous sections.
- Require that the server send the character set and encoding information separately from the document itself to the client in the HTTP header—the LANG attribute may still be necessary, depending on the character set and encoding.

Instead of arguing which method is best, why not simply support both methods whenever possible? Consider a situation where you receive a CD or DVD that is jam-packed full of HTML files. There is no server present that can send the character set and encoding information to the client, because the client must open the documents as local files. You must then rely on embedded information.

* *http://www.iana.org/assignments/character-sets*

There are three ways to specify charset information in CSS, specifically the in-file method (similar to the use of the <META> tag in HTML), via the HTTP header, and from the referring document. Interestingly, the HTML standard states that the HTTP header has higher precedence than the <META> tag declarations, and contemporary web browsing applications adhere to this.

Automatic encoding detection issues

Most contemporary browsers provide a feature that automatically detects what encoding is being used, and the algorithms that perform this have improved significantly. EUC-JP and Shift-JIS encodings used for Japanese text can often be difficult to differentiate. There is a useful trick that can help browsers more reliably detect the encoding by including an HTML comment near the top of the HTML file. This comment includes a single character whose encoding is unambiguously either EUC-JP or Shift-JIS encoding. I suggest the following two meaningful characters for this purpose: 東京 (*tōkyō*, meaning "Tokyo"). The character codes for these two characters are <C5 EC> and <B5 FE> for EUC-JP, and <93 8C> and <8B 9E> for Shift-JIS. Note how the second kanji is unambiguously either EUC-JP or Shift-JIS. The following is an example HTML comment that includes these two kanji:

```
<!--- 東京 used for correct automatic encoding detection --->
```

This technique provides a reliable backup for the technique of specifying the character set and encoding using the <META> tag.

HTTP language negotiation

Many websites can serve different versions of an HTML document, such as *index.html* and *index-ja.html*, where one provides English-language content, and the other provides Japanese-language content. The same information can be used to redirect the user to an alternate URL. The bottom line is that it is possible, through HTTP negotiation, to serve a specific HTML file, or to redirect to a specific URL, depending on how the client or server is configured.

HTTP negotiation involves a conversation or dialog between a client, such as your browser, and a server, such as the machine specified by the URL. It is during this negotiation when each party informs the other of its capabilities and preferences. The client, for example, can inform the server that it prefers Korean-language content. If the server has a file named something such as *index-ko.html*, that document can be provided to the client instead of the default *index.html* file, or it can be automatically redirected to the Korean-language website.

Apache, a popular web server, automatically performs correct language negotiation of filenames containing ISO 639 language codes, such as *ja* for Japanese and *ko* for Korean.

For more information about HTTP negotiation between clients and servers, I suggest *HTTP: The Definitive Guide*, First Edition (O'Reilly Media, 2002), by David Gourley and Brian Totty.*

Web-Authoring Tools

Some of the many HTML authoring tools available today that provide a WYSIWYG paradigm include Adobe Dreamweaver,† Apple's iWeb,‡ Google Page Creator,§ and Microsoft Expression.¶ Many word processors and page-layout applications, such as Microsoft Word, Nisus Writer, and Adobe FrameMaker, include features for saving documents in web formats. You compose to your heart's delight, and then save as an HTML or XML file.

Because I maintain a single website—consisting of my home page, along with a small number of other pages—I prefer to craft my own HTML documents and write my own CGI programs, using GNU Emacs and Perl, respectively. This is not a WYSIWYG environment. Learning the simple HTML syntax is really a no-brainer, and such knowledge is downright useful when special HTML tools are not available when something goes wrong. If I were to maintain many websites, for example, as a full-time job, I would immediately start looking for tools to ease or trivialize the effort.

Embedding CJKV Text As Graphics

If you expect that your target audience is not equipped with the ability to display CJKV text, which is rather doubtful these days, given the current states of today's OSes and web browsers, you can resort to embedding CJKV text as graphics. The most common format to use for this purpose is *Graphics Interchange Format* (GIF), developed by the folks that brought us CompuServe, along with JPEG and PNG. Most graphics applications, such as Adobe Photoshop, can generate these graphics formats.**

XML—Extensible Markup Language

XML, an abbreviation for *Extensible Markup Language*, put simply, makes the power and flexibility of SGML available for web use. Unlike HTML, it does not restrict the author to a fixed set of tags. On the contrary, it allows the author to freely define tags. And, its default character set and encoding is Unicode. This in itself allows XML to wield extraordinary power, given the broad implications that Unicode support entails. Due to the highly

* *http://oreilly.com/catalog/9781565925090/*

† *http://www.adobe.com/products/dreamweaver/*

‡ *http://www.apple.com/ilife/iweb/*

§ *http://pages.google.com/*

¶ *http://www.microsoft.com/Expression/*

** To quote the character Johner from the movie entitled *Alien Resurrection*, played by Ron Perlman, "What a waste of ammo!" Using Adobe Photoshop for the purpose of creating CJKV text as graphic images is a lot like driving in a thumb tack through the use of a sledge hammer. But what the heck, it does the trick.

structured and customizable nature of XML, it has become the preferred way in which a wide variety of data is exchanged through the Web and related mediums. In other words, XML's use and functionality extends well beyond the Web.

The XML specification is available online, and I encourage you to explore it.* An excellent printed reference for XML is *XML in a Nutshell*, Third Edition (O'Reilly Media, 2004), by Elliotte Rusty Harold and W. Scott Means.† I have also found that Robert Eckstein's *XML Pocket Reference*, Third Edition (O'Reilly Media, 2005) is one of those books to have close at hand.‡

Authoring XML Documents

Authoring XML document is very much like authoring HTML documents, but there are well-defined places in XML to encapsulate "character set" and "encoding" information that tells the browser or other client software how to interpret the text data of the document.

The following is a brief XML file that is a lot like the HTML example provided earlier, but includes two additional lines at the very beginning that are for XML:

```
<?xml version="1.0" encoding="euc-kr"?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" "http://www.w3.org/TR/
REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>켄 런디의 홈페이지</TITLE>
</HEAD>
<BODY>
<H1>켄 런디의 홈페이지</H1>
</BODY>
</HTML>
```

The first line identifies the document as XML, and also provides encoding information. If no encoding information is provided, UTF-8 or UTF-16 encoding is assumed. The presence of a *Byte Order Mark* (BOM) at the beginning of the XML file is used to identify UTF-16 encoding. Because UTF-8 encoding includes seven-bit ASCII as part of its specification, it serves as an incredibly useful default encoding for compatibility with existing web pages. The second line indicates the *Document Type Definition* (DTD), which tells the client how to treat the document as far as information structure and tags are concerned.

* *http://www.w3.org/XML/*

† *http://oreilly.com/catalog/9780596007645/*

‡ *http://oreilly.com/catalog/9780596100506/*

Including these first two lines of an XML file, with meaningful values, provides the following three benefits:

- The character set and encoding are declared, and thus unambiguous. This is mandatory in XML.
- The document is unambiguously declared as an XML file. This is mandatory in XML.
- The DTD that the document uses is unambiguous. The DOCTYPE declaration is not mandatory in XML, but W3C recommends its use, even for HTML files.

As pointed out earlier, XML is useful well beyond web pages and is used in dictionaries and OS-level resources.

The xml:lang attribute

Similar to HMTL's `LANG` attribute, XML provides the `xml:lang` attribute that performs a near-identical function, and allows tags to be assigned a language attribute.* When an XML tag is associated with data that has a clear language attribute, the `xml:lang` attribute should be used and given an appropriate value.

CGI Programming Examples

There is much more to the Web than simply static HTML and PDF documents. One can dynamically provide content through what is known as *Common Gateway Interface* (CGI) programming. The most common programming language that drives CGI programs is Perl. But, virtually any programming language will do.

CGI programming involves interaction between HTML forms and a program that does something (hopefully) intelligent with data from the forms—a prime example of a client-server relationship. Users fill out HTML forms much like they would fill out conventional forms. Once the form is complete, the user submits the form to the server to be processed. This is when the CGI program takes over.

There are two ways through which the data in HTML forms are passed to a CGI program: the GET and POST methods. The GET method provides the form data to the CGI program on the server in a single step by simply appending the data to the URL. The POST method uses two steps: the server is first contacted, and then the form data is supplied in a separate transmission.

* *http://www.w3.org/International/articles/language-tags/*

Once you have handled some simple HTML-related parsing tasks, such as managing the HTML form data, the remainder is simply standard programming. The following are two functions (written in Perl) that I use for parsing HTML form data:

```
sub parse_form_data {
  local(*DATA) = @_;
  local($method,$query_string,@key_value_pairs,$key_value,$key,$value);

  $method = $ENV{REQUEST_METHOD};
  if ($method eq "GET") {
    $query_string = $ENV{QUERY_STRING};
  } elsif ($method eq "POST") {
    read(STDIN,$query_string,$ENV{CONTENT_LENGTH});
  } else {
    &return_error(500,"Server Error","Server uses unsupported method");
  }
  $query_string =~ tr/+/ /;
  @key_value_pairs = split(/&/,$query_string);

  foreach $key_value (@key_value_pairs) {
    ($key,$value) = split(/=/,$key_value);
    $value =~ s/%([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/eg;
    if (defined($DATA{$key})) {
      $DATA{$key} = join("\0",$DATA{$key},$value);
    } else {
      $DATA{$key} = $value;
    }
  }
}

sub return_error {
  local($status,$keyword,$message) = @_;

  print "Content-type: text/html\n";
  print "Status: $status $keyword\n\n";
  print "<TITLE>CGI Program - Unexpected Error</TITLE>\n";
  print "<H1>$keyword</H1>\n<HR>$message</HR>\n";
  print "Please contact lunde\@oreilly.com for more information.\n";
  exit(1);
}
```

Now for some explanations. It is not until the following line that we start manipulating the data provided by the HTML form:

```
@key_value_pairs = split(/&/,$query_string);
```

Each key-value pair in the query string returned by the HTML form is separated by an ampersand (&, U+0026) character, so we need to split each pair using an ampersand as the separator. Each pair is stuffed into its own element of an array.

Next we iterate through the array that contains each key-value pair:

```
foreach $key_value (@key_value_pairs) {
  …
}
```

With each iteration of the loop, we copy the next key-value pair into the temporary variable *$key_value*. The body of this loop is as follows:

```
($key,$value) = split(/=/,$key_value);
$value =~ tr/+/ /;
$value =~ s/%([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/eg;
if (defined($DATA{$key})) {
  $DATA{$key} = join("\0",$DATA{$key},$value);
} else {
  $DATA{$key} = $value;
}
```

While key-value pairs were separated by an ampersand, the keys and values themselves are separated by an equals sign (=, U+003D). Thus, we must split each key-value pair using an equals sign as the separator. Once we have split a key-value pair into its key and value, we must decode the result. Because most nonalphanumeric characters are not permitted to be sent as form data through HTTP, they have been converted or otherwise encoded to mask their true identity. Most everything that is nonalphanumeric is converted into a three-character string: a percent sign (%, U+0025) followed by two hexadecimal digits (known as *URL transformation*). For example, a hyphen (-, U+002D) is represented as %2D according to this method. Bytes with the MSB set that are so commonly used for CJKV encodings, such as 0xA1, can also be passed using this method: %A1. The following single line of Perl decodes this type of encoded data:

```
$value =~ s/%([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/eg;
```

Spaces have also been converted, either to a plus sign (+, U+002B) or according to the encoding described earlier, so we must convert them back:

```
$value =~ tr/+/ /;
```

This line of code must be executed before the general decoding takes place, as follows:

```
$value =~ tr/+/ /;
$value =~ s/%([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/eg;
```

The resulting keys and values are then stored in a hash (associative array) called *%DATA* for subsequent lookup.

The only other aspect of CGI programming to be aware of is that sending results back to the user requires a valid HTTP header. This effectively means that the very first data sent back must be the text *Content-type: text/html* followed by not one but two newline characters. The following single line of Perl code is all you need:

```
print STDOUT "Content-type: text/html\n\n";
```

Of course, if you'd like to avoid dealing with all of these CGI headaches and instead concentrate and focus on writing some useful CGI programs, I strongly suggest that you consider using Lincoln Stein's excellent *CGI.pm* module, which has been included in the

standard Perl distribution since version 5.004. This Perl module gracefully handles all tricky aspects of CGI programming—so that you don't have to.

There is much more to CGI programming than I cover in these few pages. The book entitled *CGI Programming with Perl*, Second Edition (O'Reilly Media, 2000), by Scott Guelich et al., is an excellent guide for writing CGI programs,* as is Lincoln Stein's own book entitled *Official Guide to Programming with CGI.pm: The Standard for Building Web Scripts* (John Wiley & Sons, 1998).

There are many URLs that provide content through the magic of CGI programming. Jeffrey Friedl's *Japanese-English Dictionary Server*† and my own *CJKV Character Set Server*‡ are examples that serve multilingual content. Check them out! In both cases, the Perl programming language is being used to handle multilingual text.

Print Publishing

The last part of this chapter covers the topic of print publishing, with an obvious focus on PDF and its advantages when used as the basis for a publishing workflow. If you're not embracing PDF, you are strongly encouraged to do so. It solves many of problems that have plagued publishing workflows.

Looking back before the days of PDF, PostScript was the first technology to revolutionize the publishing industry, which gave birth to the desktop publishing industry. Much of the conveniences that we enjoy today, and for which we often take for granted, are the result of PostScript. Adobe Acrobat and PDF have done so again, and have taken PostScript to the next level, which can be thought of as removing the dependency on a printer.

PDF—Portable Document Format

PDF, short for *Portable Document Format*, is a PostScript-derived language for describing and structuring documents in a platform-independent manner.§ But, some of you may have heard that PostScript itself was designed to be platform-independent. PostScript, as a programming language, is typically generated by other software, such as high-end publishing applications and PostScript printer drivers. The quality and integrity of the resulting PostScript files varied greatly. There was also the issue of font availability—if the fonts that are referenced in a PostScript file were not available to the PostScript interpreter, the document would not print.¶ These were all motivations for developing a truly portable and cross-platform format for interchanging and managing documents. In the beginning, PDF was a write-only format, meaning that once you created a PDF file, you could only

* *http://oreilly.com/catalog/9781565924192/*

† *http://linear.mv.com/cgi-bin/j-e/euc/tty/dict*

‡ *http://lundestudio.com/cjkv-char.html*

§ *http://www.adobe.com/devnet/pdf/pdf_reference.html*

¶ Or it may print, but use Courier as the substitution font, which results in typographic chaos.

view, print, and search its contents. Today, many of Adobe Systems' applications, such as Adobe Illustrator and Adobe InDesign, can open and edit PDF files or can save them directly in PDF format through the use of built-in PDF-generating libraries.

The primary way in which PDF files are viewed and printed is through the use of the free Adobe Reader application.* The commercial version, called Adobe Acrobat, provides more functionality, such as the ability to edit PDF files in various ways.† Some applications, such as Adobe InDesign, include PDF-generating libraries so that PDF files can be made without any additional help. Some OSes, such as Mac OS X, include their own applications and libraries for generating, displaying, and printing PDF files. The Mac OS X *Preview* application, for example, can open, display, and print PDF files. After all, PDF is based on a specification that is published for anyone to use. Planet PDF is an excellent example of this.‡

The early versions of Adobe Acrobat, and the PDF specifications on which they were based, did not include any CJKV support whatsoever. While some people had discovered that there were ways to embed CJKV glyphs as graphics, this method had three serious drawbacks:

- It was not possible to search for text. The text that was converted to graphics was, well, simply graphics. Thus, the "content" portion was gone, and only the "presentation" remains.
- Embedding graphics required that one specify a resolution. Such documents may display fine at screen resolutions, but may print horribly, or vice versa.
- Embedding higher-resolution graphics takes up a lot of space. The resulting files can be huge.

PDF version 1.2, which had been incorporated into Adobe Acrobat version 3.0, provided a minimal level of CJKV support. PDF version 1.3 provided a much more enhanced level of CJKV support, to include the ability to embed glyphs from CJKV fonts, and it was incorporated in Adobe Acrobat version 4.0. In other words, what I consider to be true CJKV support began with version 4.0.

PDF was published as an open standard on July 1, 2008, as the standard designated ISO 32000-1:2008, which is entitled *Document management—Portable document format—Part 1: PDF 1.7*. PDF version 1.7 corresponds to the functionality set forth in Acrobat version 8.0.

I need to state that at some point the preferred printing workflow will be purely PDF-based. In other words, the font burden will be on PDF files, not the printer. This effectively means that PDF files will be expected to have the necessary fonts, subsetted and embedded, as part of their content. No longer will printer-resident fonts be required *or allowed*.

* *http://www.adobe.com/products/acrobat/readstep.html*

† *http://www.adobe.com/products/acrobat/*

‡ *http://www.planetpdf.com/*

While printer-resident fonts have historically been in a variety of formats, such as OCF and CID, a PDF-based workflow necessarily entails the use of OpenType fonts in order to better guarantee cross-platform use by authoring applications. In case it is not painfully obvious, one of the greatest advantages of a PDF-based workflow is that it better guarantees that the fonts used to author documents are the same as what are used to print them, because they are the same fonts. The Japanese market is well ahead of the curve in this transition, with approximately 50% of printing jobs being submitted as PDF files. This is due to the broad extent to which OpenType fonts are used in that market. Other markets that have traditionally depended on printer-resident fonts, such as China, Taiwan, and Korea, are expected to follow this trend.

Authoring PDF Documents

PDF files can be created in a number of ways, and several options with regard to CJKV font embedding are made available in Adobe Acrobat version 4.0 and later. Adobe Systems' Technical Note #5641, *Enabling PDF Font Embedding for CID-Keyed Fonts*, illustrated to developers how they may enable their CID-keyed fonts, to include derivative formats such as sfnt-CID fonts, for embedding in PDF.*

PDF files typically begin their life as documents created by using standard word-processing or page-composition applications. My favorite happens to be Adobe InDesign due to its high-end typographic controls. There are four basic methods for converting application documents into PDF, listed in order of preference:

- Simply save or export the document as PDF. Some applications, such as Adobe Illustrator, can include PDF as part of its native file format. Other applications, such as Adobe InDesign, include PDF-generating libraries that enable the direct creation of PDFs.
- When Acrobat is installed, to include the free Adobe Reader, a PDF-generating printer driver is installed at the same time. If an application has the ability to print, this special printer driver can be used. Simply select "Adobe PDF *x*.0" instead of a specific printer when printing a document. The result is a PDF file that is saved to your hard disk, instead of being output to a printer.
- Use TEX- or LATEX-based tools to generate PDF files, such as *pdfTeX* and *PDFLaTeX*.† *DVIPDFMx* is an extended version of *dvipdfm* that provides CJKV support.‡ CSS-styled HTML and XML files can be converted to PDF using *Prince*, which is available for a variety of OSes.§

* *http://www.adobe.com/devnet/font/pdfs/5641.CID_Embed.pdf*

† *http://www.tug.org/applications/pdftex/*

‡ *http://project.ktug.or.kr/dvipdfmx/*

§ *http://www.princexml.com/*

- Print the document to a PostScript file, and then use the Acrobat Distiller application to convert the PostScript file to PDF. In the past, this was considered the most reliable and robust way in which to create a PDF file. Times change. If you generate your own PostScript files with the intent to turn them into PDFs, like I do, Acrobat Distiller will remain useful.

When using the last method, specifically the use of PostScript files, don't be overly alarmed or concerned by the size of the PostScript files that you plan to push through Acrobat Distiller. The resulting PDF file is usually a fraction of the size of the original PostScript file. Of course, YMMV.*

One of the more difficult decisions that must be made when creating PDF files is whether or not to embed fonts.† Actually, today's default behavior is to embed fonts, so the decision is really whether to choose not to embed fonts. In any case, there are advantages and disadvantages of each approach, as shown in Table 12-11.

Table 12-11. To embed or not to embed, that is the question

Font embedding	Advantages	Disadvantages
No	Reduced file size	Adobe Reader and other viewers require fonts
Yes	Display or print anywhere	Increased file size

So, how does one decide whether or not to embed fonts in PDF files? In the latest versions of Adobe Acrobat and the corresponding PDF-generating libraries, font embedding is turned on by default. One thus needs to explicitly choose not to embed fonts.

Some class of fonts can be embedded in their entirety, regardless to what extent its glyphs are referenced in the document. Other classes of fonts, such as those with large glyph sets, are automatically subsetted prior to embedding. This is done for reasons of practicality, considering the size of such fonts. Thus, PDF files that embed glyphs from large fonts never include the entire font, unless, of course, every glyph in the font is referenced in the document. This scenario is rather doubtful.‡ All the glyphs that are referenced in the documented are subsetted prior to embedding. In Japanese, for example, 70% of running text is composed of kana, which represents approximately 200 unique glyphs in a typical font; the rest are kanji and other symbols, which may or may not take up much space.

The *fsType* field of the 'OS/2' table specifies the embedding permissions for OpenType fonts, and thus controls embedding. For CIDFont resources, the corresponding field is called *FSType* and is an entry in the *CIDSystemInfo* dictionary. Table 12-12 lists the 'OS/2'

* Your Mileage May Vary

† As a purely historical note, the embedding of CJKV fonts in Adobe Acrobat version 3.0 was not possible. version 4.0 and greater provide this capability.

‡ Appendix G of first edition of this book, which is now part of Appendix H of this edition, served as an exception to this general principle. The font used for that appendix included 55,880 glyphs, and all of them were referenced in that appendix.

table *fsType* field settings that are possible, and how they control the extent to which fonts are embedded.*

Table 12-12. CIDFont and OpenType embedding permissions

fsType/FSType	Meaning
0	Installable embedding
2	Restricted license embedding
4	Print and preview embedding
8	Editable embedding

In other words, an *fsType* value of 0 is the most liberal setting, allowing the font to be installed, and a value of 2 is the most restrictive, denying the embedding of the font. From a practical point of view, when dealing with TrueType fonts with large glyph sets or CID-keyed fonts, to include OpenType fonts with CID-keyed CFFs, there is no functional difference between the values 4 and 8. Only smaller name-keyed fonts have the ability to embed to PDF in their entirety, regardless to what extent their glyphs are referenced. The value of 8 is useful when you want to edit the PDF and use a character that is not yet referenced in the document, but for which there is a corresponding glyph in the embedded font. Given that larger fonts are always subsetted prior to embedding, it is clear that there is no functional difference between the values 4 and 8 for this class of font.

Unless you can guarantee that all people who display or print your PDF file have the exact same fonts used to create it, which also means these people will have the same versions of the fonts, embedding the fonts is the prudent thing to do. As already stated, the current default is to embed the fonts unless the embedding permissions dictate otherwise. Embedding usually leads to slightly larger files, but a small and undisplayable document is obviously useless.

PDF Eases Publishing Pains

Most individuals—and many companies—do not own their own typesetting hardware, such as a photo imagesetter, and for a very good reason: it is a huge expense and requires frequent maintenance and attention. This is precisely when and where service bureaus live up to the first word in their title: *service*.

There are many service bureaus that provide CJKV support, meaning that they accept printing jobs that include CJKV text. Years ago, service bureaus would typically accept "jobs" in one of only two formats:

- Application files, limited to specific applications
- PostScript files, limited to specific fonts

* *http://www.microsoft.com/typography/otspec/os2.htm#fst*

Both of these methods of submitting printing jobs to service bureaus had major disadvantages, such as the following:

- Submitting application files required that the service bureau have the same application—and perhaps the same version of the application—installed onto at least one of their computers, along with the same fonts. This is why some service bureaus supported only specific applications. In Japan, it was common for service bureaus to additionally support only specific versions of applications.
- Submitting PostScript files did not require the presence of the application or fonts from which they were generated, but they generally cannot be previewed.* More importantly, it also required that the service bureau have the same fonts installed onto their photo imagesetter, unless they were embedded in the PostScript files. Given the relatively high cost of using photo imagesetters, it was common to print a proof using a lower-resolution—and cheaper—device.
- Furthermore, both methods of submitting printing jobs to service bureaus suffered from font-version discrepancy, between the version that the customer used to author the document and the version that was installed onto the service bureau's photo imagesetter. Minor glyph or glyph metrics changes between font versions was to blame for any printing inconsistency.

Today, submitting documents to service bureaus in the form of PDF files suffers from none of these problems. In short, PDF does not require that the original application and fonts be available, and PDF files can be previewed prior to printing using freely available viewers, such as Adobe Reader.

Some fonts, due to their licensing restrictions, are not allowed to be embedded within a PDF file. For OpenType fonts, the *fsType* field in the 'OS/2' table controls this parameter, and Table 12-12 detailed the possible values and their meanings. The font's license or documentation should indicate any such restrictions. Read or inquire before you buy. I claim that an OpenType font is useless if PDF embedding is not allowed. Fonts are part of a document workflow, and if they cannot be embedded into PDF, the workflow is obviously broken.

In summary, PDF functions as a reliable "digital master" if the fonts referenced in the document are embedded, which is possible in the context of Adobe Acrobat version 4.0 and greater (PDF version 1.3), and as long as the fonts' embedding permissions are set appropriately. This provides service bureaus with the ability to print directly from PDF

* Some environments and applications can be used to preview PostScript files, such as Ghostscript and the Mac OS X *Preview* application.

to plate.* Those in the professional publishing industry will immediately recognize this benefit.

Where to Go Next?

Wow. You finished all 12 chapters. Although there are some appendixes, some of which are available as downloadable and printable PDFs, keep in mind that this book also serves as a reference. Be sure to use its index to find the pages that cover specific topics.

I also encourage you to explore the many URLs that have been provided throughout this book, most of which are in footnotes. If you purchased the PDF version of this book, you can simply click on a URL in order to open the web page using your favorite web browser.

There are two types of web users: those who author documents, and those who view—that is, "surf"—them. Many, like me, are both. One of the most addicting aspects of the Web is the ability to follow links to virtually no end. This can be good and bad. It is good in that you may come across some incredibly useful information, which means you feel compelled to bookmark the URL for future reference. But, it can be bad in that you probably have other—perhaps more meaningful—work you should be doing instead, such as what you're probably being paid to do.

An excellent way to find your way around the Web is to use a search engine, such as Google or Yahoo! Another useful reference to explore, which is often overlooked, is Wikipedia.† In addition to using Wikipedia as a resource, consider contributing to its continued development.

Anyway, now that you're done reading this book, or at least its chapters, I encourage you to explore my home page, which will be updated from time to time, to provide late-breaking information about this book.‡ In addition, my home page can serve as one way to start exploring the numerous CJKV information resources that are available.

Cheers!

* The first edition of this book was printed in late 1998 by submitting to the publisher photographic paper output that was produced using a high-resolution photo imagesetter. The second printing of the first edition of this book was submitted to the publisher in the latter half of 2002 as a PDF file. Guess which one was easier? It should be quite obvious how this edition was published.

† *http://www.wikipedia.org/*

‡ *http://lundestudio.com/*

APPENDIX A
Code Conversion Tables

The first table in this appendix, which spans the following two pages, is useful when dealing with material indexed by the various native CJKV encoding methods. The second table provides the correspondences between Row-Cell rows 95 through 120 and the Shift-JIS user-defined encoding range.

All of these columns are fairly self-explanatory, except perhaps for the two Shift-JIS columns. Confusion may occur when you try to convert the first Shift-JIS byte to another code, or when you try to convert the second byte of another code to Shift-JIS. The following two sets of rules should help:

Row-Cell/ISO-2022/EUC to Shift-JIS

1. The first byte converts using both conversion tables—find the first byte value in Row-Cell/ISO-2022/EUC code, and then simply slide over to the Shift-JIS First Byte column.
2. The second byte is a bit tricky. If the first byte (yes, the first byte!) of the Row-Cell/ISO-2022/EUC code is odd (the hexadecimal digits B, D, and F are odd), select the lefthand value in the Shift-JIS Second Byte column. Otherwise, select the right-hand value.

Shift-JIS to Row-Cell/ISO-2022/EUC

1. If the second Shift-JIS byte is the lefthand entry in the Shift-JIS Second Byte column, use the first occurrence of the first Shift-JIS byte to determine the first byte value of another code. Otherwise, use the second occurrence of the first Shift-JIS byte.
2. The second Shift-JIS byte converts unambiguously using the code conversion table.

Row-Cell	ISO-2022	EUC	Shift-JIS First Byte	Shift-JIS Second Byte
01	21	A1	81	40 9F
02	22	A2	81	41 A0
03	23	A3	82	42 A1
04	24	A4	82	43 A2
05	25	A5	83	44 A3
06	26	A6	83	45 A4
07	27	A7	84	46 A5
08	28	A8	84	47 A6
09	29	A9	85	48 A7
10	2A	AA	85	49 A8
11	2B	AB	86	4A A9
12	2C	AC	86	4B AA
13	2D	AD	87	4C AB
14	2E	AE	87	4D AC
15	2F	AF	88	4E AD
16	30	B0	88	4F AE
17	31	B1	89	50 AF
18	32	B2	89	51 B0
19	33	B3	8A	52 B1
20	34	B4	8A	53 B2
21	35	B5	8B	54 B3
22	36	B6	8B	55 B4
23	37	B7	8C	56 B5
24	38	B8	8C	57 B6
25	39	B9	8D	58 B7
26	3A	BA	8D	59 B8
27	3B	BB	8E	5A B9
28	3C	BC	8E	5B BA
29	3D	BD	8F	5C BB
30	3E	BE	8F	5D BC
31	3F	BF	90	5E BD
32	40	C0	90	5F BE
33	41	C1	91	60 BF
34	42	C2	91	61 C0
35	43	C3	92	62 C1
36	44	C4	92	63 C2
37	45	C5	93	64 C3
38	46	C6	93	65 C4
39	47	C7	94	66 C5
40	48	C8	94	67 C6
41	49	C9	95	68 C7
42	4A	CA	95	69 C8
43	4B	CB	96	6A C9
44	4C	CC	96	6B CA
45	4D	CD	97	6C CB
46	4E	CE	97	6D CC
47	4F	CF	98	6E CD

Row-Cell	ISO-2022	EUC	Shift-JIS First Byte	Shift-JIS Second Byte
48	50	D0	98	6F CE
49	51	D1	99	70 CF
50	52	D2	99	71 D0
51	53	D3	9A	72 D1
52	54	D4	9A	73 D2
53	55	D5	9B	74 D3
54	56	D6	9B	75 D4
55	57	D7	9C	76 D5
56	58	D8	9C	77 D6
57	59	D9	9D	78 D7
58	5A	DA	9D	79 D8
59	5B	DB	9E	7A D9
60	5C	DC	9E	7B DA
61	5D	DD	9F	7C DB
62	5E	DE	9F	7D DC
63	5F	DF	E0	7E DD
64	60	E0	E0	80 DE
65	61	E1	E1	81 DF
66	62	E2	E1	82 E0
67	63	E3	E2	83 E1
68	64	E4	E2	84 E2
69	65	E5	E3	85 E3
70	66	E6	E3	86 E4
71	67	E7	E4	87 E5
72	68	E8	E4	88 E6
73	69	E9	E5	89 E7
74	6A	EA	E5	8A E8
75	6B	EB	E6	8B E9
76	6C	EC	E6	8C EA
77	6D	ED	E7	8D EB
78	6E	EE	E7	8E EC
79	6F	EF	E8	8F ED
80	70	F0	E8	90 EE
81	71	F1	E9	91 EF
82	72	F2	E9	92 F0
83	73	F3	EA	93 F1
84	74	F4	EA	94 F2
85	75	F5	EB	95 F3
86	76	F6	EB	96 F4
87	77	F7	EC	97 F5
88	78	F8	EC	98 F6
89	79	F9	ED	99 F7
90	7A	FA	ED	9A F8
91	7B	FB	EE	9B F9
92	7C	FC	EE	9C FA
93	7D	FD	EF	9D FB
94	7E	FE	EF	9E FC

The following table provides the correspondences between Row-Cell and Shift-JIS encodings for the Shift-JIS user-defined region (0xF0 through 0xFC). The principles for the previous code conversion table apply to this table.

Row-Cell	Shift-JIS First Byte
95	F0
96	F0
97	F1
98	F1
99	F2
100	F2
101	F3
102	F3
103	F4
104	F4
105	F5
106	F5
107	F6
108	F6
109	F7
110	F7
111	F8
112	F8
113	F9
114	F9
115	FA
116	FA
117	FB
118	FB
119	FC
120	FC

APPENDIX B
Notation Conversion Table

The following two-page table lists all 256 8-bit byte values in binary (base 2), octal (base 8), decimal (base 10), and hexadecimal (base 16) notations. While I commonly use decimal and hexadecimal notations throughout this book, there are some readers who are more familiar with other notations, such as octal. It is unlikely that many readers will be converting between binary and other notations, but you never know.

Use this table as your guide for converting data throughout this book between notations. The 94 printable ASCII characters are included, when appropriate, for reference purposes.

Base 2	Base 8	Base 10	Base 16	ASCII
00000000	000	0	00	<NUL>
00000001	001	1	01	<SOH>
00000010	002	2	02	<STX>
00000011	003	3	03	<ETX>
00000100	004	4	04	<EOT>
00000101	005	5	05	<ENQ>
00000110	006	6	06	<ACK>
00000111	007	7	07	<BEL>
00001000	010	8	08	<BS>
00001001	011	9	09	<HT>
00001010	012	10	0A	<LF>
00001011	013	11	0B	<VT>
00001100	014	12	0C	<FF>
00001101	015	13	0D	<CR>
00001110	016	14	0E	<SO>
00001111	017	15	0F	<SI>
00010000	020	16	10	<DLE>
00010001	021	17	11	<DC1>
00010010	022	18	12	<DC2>
00010011	023	19	13	<DC3>
00010100	024	20	14	<DC4>
00010101	025	21	15	<NAK>
00010110	026	22	16	<SYN>
00010111	027	23	17	<ETB>
00011000	030	24	18	<CAN>
00011001	031	25	19	<EM>
00011010	032	26	1A	<SUB>
00011011	033	27	1B	<ESC>
00011100	034	28	1C	<FS>
00011101	035	29	1D	<GS>
00011110	036	30	1E	<RS>
00011111	037	31	1F	<US>
00100000	040	32	20	<SP>
00100001	041	33	21	!
00100010	042	34	22	"
00100011	043	35	23	#
00100100	044	36	24	$
00100101	045	37	25	%
00100110	046	38	26	&
00100111	047	39	27	'
00101000	050	40	28	(
00101001	051	41	29	)
00101010	052	42	2A	*
00101011	053	43	2B	+
00101100	054	44	2C	,
00101101	055	45	2D	-
00101110	056	46	2E	.
00101111	057	47	2F	/
00110000	060	48	30	0
00110001	061	49	31	1
00110010	062	50	32	2
00110011	063	51	33	3
00110100	064	52	34	4
00110101	065	53	35	5
00110110	066	54	36	6
00110111	067	55	37	7
00111000	070	56	38	8
00111001	071	57	39	9
00111010	072	58	3A	:
00111011	073	59	3B	;
00111100	074	60	3C	<
00111101	075	61	3D	=
00111110	076	62	3E	>
00111111	077	63	3F	?

Base 2	Base 8	Base 10	Base 16	ASCII
01000000	100	64	40	@
01000001	101	65	41	A
01000010	102	66	42	B
01000011	103	67	43	C
01000100	104	68	44	D
01000101	105	69	45	E
01000110	106	70	46	F
01000111	107	71	47	G
01001000	110	72	48	H
01001001	111	73	49	I
01001010	112	74	4A	J
01001011	113	75	4B	K
01001100	114	76	4C	L
01001101	115	77	4D	M
01001110	116	78	4E	N
01001111	117	79	4F	O
01010000	120	80	50	P
01010001	121	81	51	Q
01010010	122	82	52	R
01010011	123	83	53	S
01010100	124	84	54	T
01010101	125	85	55	U
01010110	126	86	56	V
01010111	127	87	57	W
01011000	130	88	58	X
01011001	131	89	59	Y
01011010	132	90	5A	Z
01011011	133	91	5B	[
01011100	134	92	5C	\
01011101	135	93	5D	]
01011110	136	94	5E	^
01011111	137	95	5F	_
01100000	140	96	60	`
01100001	141	97	61	a
01100010	142	98	62	b
01100011	143	99	63	c
01100100	144	100	64	d
01100101	145	101	65	e
01100110	146	102	66	f
01100111	147	103	67	g
01101000	150	104	68	h
01101001	151	105	69	i
01101010	152	106	6A	j
01101011	153	107	6B	k
01101100	154	108	6C	l
01101101	155	109	6D	m
01101110	156	110	6E	n
01101111	157	111	6F	o
01110000	160	112	70	p
01110001	161	113	71	q
01110010	162	114	72	r
01110011	163	115	73	s
01110100	164	116	74	t
01110101	165	117	75	u
01110110	166	118	76	v
01110111	167	119	77	w
01111000	170	120	78	x
01111001	171	121	79	y
01111010	172	122	7A	z
01111011	173	123	7B	{
01111100	174	124	7C	|
01111101	175	125	7D	}
01111110	176	126	7E	~
01111111	177	127	7F	<DEL>

Base 2	Base 8	Base 10	Base 16	ASCII
10000000	200	128	80	
10000001	201	129	81	
10000010	202	130	82	
10000011	203	131	83	
10000100	204	132	84	
10000101	205	133	85	
10000110	206	134	86	
10000111	207	135	87	
10001000	210	136	88	
10001001	211	137	89	
10001010	212	138	8A	
10001011	213	139	8B	
10001100	214	140	8C	
10001101	215	141	8D	
10001110	216	142	8E	
10001111	217	143	8F	
10010000	220	144	90	
10010001	221	145	91	
10010010	222	146	92	
10010011	223	147	93	
10010100	224	148	94	
10010101	225	149	95	
10010110	226	150	96	
10010111	227	151	97	
10011000	230	152	98	
10011001	231	153	99	
10011010	232	154	9A	
10011011	233	155	9B	
10011100	234	156	9C	
10011101	235	157	9D	
10011110	236	158	9E	
10011111	237	159	9F	
10100000	240	160	A0	
10100001	241	161	A1	
10100010	242	162	A2	
10100011	243	163	A3	
10100100	244	164	A4	
10100101	245	165	A5	
10100110	246	166	A6	
10100111	247	167	A7	
10101000	250	168	A8	
10101001	251	169	A9	
10101010	252	170	AA	
10101011	253	171	AB	
10101100	254	172	AC	
10101101	255	173	AD	
10101110	256	174	AE	
10101111	257	175	AF	
10110000	260	176	B0	
10110001	261	177	B1	
10110010	262	178	B2	
10110011	263	179	B3	
10110100	264	180	B4	
10110101	265	181	B5	
10110110	266	182	B6	
10110111	267	183	B7	
10111000	270	184	B8	
10111001	271	185	B9	
10111010	272	186	BA	
10111011	273	187	BB	
10111100	274	188	BC	
10111101	275	189	BD	
10111110	276	190	BE	
10111111	277	191	BF	

Base 2	Base 8	Base 10	Base 16	ASCII
11000000	300	192	C0	
11000001	301	193	C1	
11000010	302	194	C2	
11000011	303	195	C3	
11000100	304	196	C4	
11000101	305	197	C5	
11000110	306	198	C6	
11000111	307	199	C7	
11001000	310	200	C8	
11001001	311	201	C9	
11001010	312	202	CA	
11001011	313	203	CB	
11001100	314	204	CC	
11001101	315	205	CD	
11001110	316	206	CE	
11001111	317	207	CF	
11010000	320	208	D0	
11010001	321	209	D1	
11010010	322	210	D2	
11010011	323	211	D3	
11010100	324	212	D4	
11010101	325	213	D5	
11010110	326	214	D6	
11010111	327	215	D7	
11011000	330	216	D8	
11011001	331	217	D9	
11011010	332	218	DA	
11011011	333	219	DB	
11011100	334	220	DC	
11011101	335	221	DD	
11011110	336	222	DE	
11011111	337	223	DF	
11100000	340	224	E0	
11100001	341	225	E1	
11100010	342	226	E2	
11100011	343	227	E3	
11100100	344	228	E4	
11100101	345	229	E5	
11100110	346	230	E6	
11100111	347	231	E7	
11101000	350	232	E8	
11101001	351	233	E9	
11101010	352	234	EA	
11101011	353	235	EB	
11101100	354	236	EC	
11101101	355	237	ED	
11101110	356	238	EE	
11101111	357	239	EF	
11110000	360	240	F0	
11110001	361	241	F1	
11110010	362	242	F2	
11110011	363	243	F3	
11110100	364	244	F4	
11110101	365	245	F5	
11110110	366	246	F6	
11110111	367	247	F7	
11111000	370	248	F8	
11111001	371	249	F9	
11111010	372	250	FA	
11111011	373	251	FB	
11111100	374	252	FC	
11111101	375	253	FD	
11111110	376	254	FE	
11111111	377	255	FF	

APPENDIX C
Perl Code Examples

This appendix provides Perl equivalents of some algorithms presented in Chapter 9 as C or Java code. While the C and Java code examples in Chapter 9 are useful for developing your own commercial-grade software, these Perl equivalents are helpful for internal-use tools.

If you do not use Perl, feel free to skip this appendix. But, in the same vein, I encourage you to explore the Perl programming language to see whether it can offer you something. I use nothing but Perl for virtually all of my programming needs.*

A few of these Perl code examples began their life as code courtesy of regex wizard Jeffrey Friedl, author of a most excellent book entitled *Mastering Regular Expressions*, Third Edition (O'Reilly Media, 2006). Tom Christiansen and Nathan Torkington's *Perl Cookbook*, Second Edition (O'Reilly Media, 2003) should also be used, because it provides a wealth of incredibly useful Perl examples.

Japanese Code Conversion

The programs presented in the following sections perform Japanese code conversion. Note that all of them support ASCII (JIS-Roman), JIS X 0208:1997, and half-width katakana (even for ISO-2022-JP encoding, which, technically, does not support half-width katakana).

* In retrospect, I should have instead learned the Ruby programming language, if for no other reason than my daughter's name is Ruby.

ISO-2022-JP to EUC-JP Conversion

The following Perl program performs ISO-2022-JP to EUC-JP conversion, and fully supports half-width katakana:

```
#!/usr/local/bin/perl -w

while (defined($line = <STDIN>)) {
  $line =~ s{                              # JIS X 0208:1997
    \e\$[\@B]                              # ESC $ plus @ or B
    ((?:[\x21-\x7E][\x21-\x7E])+)          # Two-byte characters
  }{($x = $1) =~ tr/\x21-\x7E/\xA1-\xFE/,  # From 7- to 8-bit
    $x
  }egx;
  $line =~ s{                              # JIS X 0201-1997 half-width katakana
    \e\(I                                  # ESC ( I
    ([\x21-\x7E]+)                         # Half-width katakana
  }{($x = $1) =~ tr/\x21-\x7E/\xA1-\xFE/,  # From 7- to 8-bit
    ($y = $x) =~ s/([\xA1-\xFE])/\x8E$1/g, # Prefix with SS2
    $y
  }egx;
  $line =~ s/\e\([BHJ]//g;
  print STDOUT $line;
}
```

EUC-JP to ISO-2022-JP Conversion

The following Perl program performs EUC-JP to ISO-2022-JP conversion, and fully supports half-width katakana:

```
#!/usr/local/bin/perl -w

while (defined($line = <STDIN>)) {
  $line =~ s{                        # JIS X 0208:1997
    ((?:[\xA1-\xFE][\xA1-\xFE])+)
  }{\e\$B$1\e\(J}gx;
  $line =~ s{                        # JIS X 0201-1997 half-width katakana
    ((?:\x8E[\xA0-\xDF])+)
  }{\e\(I$1\e\(J}gx;
  $line =~ s/\x8E//g;                # Remove SS2s
  $line =~ tr/\xA1-\xFE/\x21-\x7E/; # From 8- to 7-bit
  print STDOUT $line;
}
```

ISO-2022-JP or EUC-JP to Shift-JIS Conversion

The following Perl program performs ISO-2022-JP or EUC-JP to Shift-JIS conversion, and fully supports half-width katakana. A single program allows this because EUC-JP

can be normalized to ISO-2022-JP using simple eight- to seven-bit code conversion operations:

```
#!/usr/local/bin/perl -w

sub convert2sjis { # For EUC-JP and ISO-2022-JP to Shift-JIS
  my @euc = unpack("C*", $_[0]);
  my @out = ();
  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;
    push(@out, (($hi + 1) >> 1) + ($hi < 95 ? 112 : 176),
      $lo + (($hi & 1) ? ($lo > 95 ? 32 : 31) : 126));
  }
  return pack("C*", @out);
}

while (defined($line = <STDIN>)) {
  $line =~ s{(                    # EUC-JP
    (?:[\xA1-\xFE][\xA1-\xFE])+| # JIS X 0208:1997
    (?:\x8E[\xA0-\xDF])+         # Half-width katakana
  )}{substr($1,0,1) eq "\x8E" ? (($x = $1) =~ s/\x8E//g, $x) :
    &convert2sjis($1)}egx;
  $line =~ s{                     # Handle ISO-2022-JP
    \e\$[\@B]
    ((?:[\x21-\x7E][\x21-\x7E])+)
    \e\([BHJ]
  }{&convert2sjis($1)}egx;
  $line =~ s{                     # Handle ISO-2022-JP half-width katakana
    \e\(I
    ([\x20-\x5F]+)
    \e\([BHJ]
  }{($x = $1) =~ tr/\x20-\x5F/\xA0-\xDF/, $x}egx;
  print STDOUT $line;
}
```

Shift-JIS to ISO-2022-JP Conversion

The following Perl program performs Shift-JIS to ISO-2022-JP conversion, and fully supports half-width katakana:

```
#!/usr/local/bin/perl -w

sub sjis2jis { # For Shift-JIS to ISO-2022-JP and EUC-JP
  my @ord = unpack("C*", $_[0]);
  for ($i = 0; $i < @ord; $i += 2) {
    $ord[$i] = (($ord[$i]-($ord[$i]<160?112:176))<<1)-
      ($ord[$i+1]<159?1:0);
    $ord[$i+1] -= ($ord[$i+1]<159?($ord[$i+1]>127?32:31):126);
  }
  return pack("C*", @ord);
}

while (defined($line = <STDIN>)) {
  $line =~ s{( # JIS X 0208:1997 and half-width katakana
```

```
    (?:[\x81-\x9F\xE0-\xEF][\x40-\x7E\x80-\xFC])+|
    [\xA0-\xDF]+
  )}{
    ($x=$1) !~ /^[\xA0-\xDF]/ ?
    "\e\$B" . &sjis2jis($1) . "\e\(J" :
    "\e\(I" . (($y=$x) =~ tr/\xA0-\xDF/\x20-\x5F/, $y) . "\e\(J"
  }egx;
  print STDOUT $line;
}
```

Shift-JIS to EUC-JP Conversion

The following Perl program performs Shift-JIS to EUC-JP conversion, and fully supports half-width katakana:

```
#!/usr/local/bin/perl -w

sub sjis2jis { # For Shift-JIS to ISO-2022-JP and EUC-JP
  my @ord = unpack("C*", $_[0]);
  for ($i = 0; $i < @ord; $i += 2) {
    $ord[$i] = (($ord[$i]-($ord[$i]<160?112:176))<<1)-($ord[$i+1]<159?1:0);
    $ord[$i+1] -= ($ord[$i+1]<159?($ord[$i+1]>127?32:31):126);
  }
  return pack("C*", @ord);
}

while (defined($line = <STDIN>)) {
  $line =~ s{( # JIS X 0208:1997 and half-width katakana
    (?:[\x81-\x9F\xE0-\xEF][\x40-\x7E\x80-\xFC])+|
    [\xA0-\xDF]+
  )}{
    ($x = $1) !~ /^[\xA0-\xDF]/ ?
    (($y = &sjis2jis($x)) =~ tr/\x21-\x7E/\xA1-\xFE/, $y) :
    (($y = $x) =~ s/([\xA0-\xDF])/\x8E$1/g, $y)
  }egx;
  print STDOUT $line;
}
```

Half- to Full-Width Katakana Conversion

The following Perl program converts half-width katakana to their full-width equivalents. The default behavior expects Shift-JIS encoding as the input, but invoking the program with the `-e` option changes the behavior to expect EUC-JP encoding.

```
#!/usr/local/bin/perl -w

# unkana.pl
#
# (Version with multi-encoding support without using JPerl)
# Written by Ken Lunde (lunde@adobe.com)
# January 3, 1997

require 5.002;
```

```
$euc = "";

if (defined $ARGV[0] && $ARGV[0] eq "-e") {
  $euc = chr(142);
}

if ($euc) { # If EUC encoding
  $encoding = '[\xA1-\xFE][\xA1-\xFE]';
  $symbol_one = chr(161);
  $kana_one = chr(165);
  # Second-byte values (decimal) in EUC
  @two = (161, 163, 214, 215, 162, 166, 242, 161, 163, 165, 167, 169, 227, 229, 231,
    195, 188, 162, 164, 166, 168, 170, 171, 173, 175, 177, 179, 181, 183, 185, 187,
    189, 191, 193, 196, 198, 200, 202 .. 207, 210, 213, 216, 219, 222 .. 226, 228,
    230, 232 .. 237, 239, 243, 171, 172);
} else { # If Shift-JIS encoding
  $encoding = '[\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC]';
  $symbol_one = chr(129);
  $kana_one = chr(131);

  # Second-byte values (decimal) in Shift-JIS
  @two = (64, 66, 117, 118, 65, 69, 146, 64, 66, 68, 70, 72, 131, 133, 135, 98, 91,
    65, 67, 69, 71, 73, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 99, 101, 103,
    105 .. 110, 113, 116, 119, 122, 125, 126, 128 .. 130, 132, 134, 136 .. 141, 143,
    147, 74, 75);
}

# Initialize lookup for kana substitution (stored in %char_hash)
foreach $value (160 .. 223) {
  $char_hash{chr($value)} = chr($two[$value - 160]);
}

# main loop
while ($line = <STDIN>) {
  $line =~ s/([\x00-\x80]+ |                          # ASCII or JIS-Roman
              (?:$encoding)+ |                        # JIS X 0208:1997
              (?:${euc}[\xA0-\xDF])+                  # Half-width katakana
             )/&dostuff($1)/egox;
  print STDOUT $line;
}

sub dostuff {
  my ($str) = @_;

  if ($str =~ /^${euc}[\xA0-\xDF]/o) {                # If half-width katakana
    $str =~ s/((?:$euc\xB3$euc\xDE)+ |                # u + dakuten
               (?:${euc}[\xCA-\xCE]$euc\xDF)+ |       # KSTH-row + dakuten
               (?:${euc}[\xB6-\xC4\xCA-\xCE]$euc\xDE)+ | # H-row
               (?:${euc}[\xA0-\xDF])                  # All other cases
              )/&han2zen($1)/egox;
  }
  return $str; # Returns ASCII/JIS-Roman and JIS X 0208:1997 as is
}
```

```
sub han2zen {
  my ($hkana) = @_;

  if ($hkana =~ /^$euc\xB3$euc\xDE/o) {         # Special "u + dakuten" case
    if ($euc) {
      $hkana =~ s/$euc\xB3$euc\xDE/\xA5\xF4/go;
    } else {
      $hkana =~ s/\xB3\xDE/\x83\x94/g;
    }
  } elsif ($hkana =~ /^${euc}[\xB6-\xC4\xCA-\xCE]${euc}[\xDE\xDF]/o) {
    $prefix = $kana_one;                        # First byte for katakana
    if ($hkana =~ /^${euc}[\xCA-\xCE]$euc\xDF/o) {
      $suffix = 2;                              # Increment value for handakuten
    } else {
      $suffix = 1;                              # Increment value for dakuten
    }
    $hkana =~ s/$euc([\xB6-\xC4\xCA-\xCE])${euc}[\xDE\xDF]/
                pack("n",unpack("n","$prefix$char_hash{$1}") +
                $suffix)/egox;
  } else {
    if ($hkana =~ /^${euc}[\xA0-\xA5\xB0\xDE\xDF]/o) {
      $prefix = $symbol_one;                    # First byte for symbol
    } else {
      $prefix = $kana_one;                      # First byte for katakana
    }
    $hkana =~ s/$euc([\xA0-\xDF])/$prefix$char_hash{$1}/go;
  }
  return $hkana;
}
```

Korean Code Conversion

Although this section does not include any complete Perl programs, the most difficult algorithms for handling Korean encodings are included as workable subroutines that can be used in Perl programs. The main focus of this section is Johab encoding, which requires an algorithm for handling conversion to and from ISO-2022-KR or EUC-KR encodings, which is strikingly similar to that used for handling conversion to and from Shift-JIS encoding. Because handling Johab encoding also requires mapping tables (for handling the 2,350 hangul in KS X 1001:2004) or bit-array manipulation (as an alternative for turning these 2,350 hangul into their Johab equivalents), I do not include complete programs.

ISO-2022-KR or EUC-KR to Johab Conversion

The following Perl subroutine converts strings of two-byte data encoded according to ISO-2022-KR or EUC-KR into Johab encoding:

```
sub convert2johab ($) { # Convert ISO-2022-KR or EUC-KR to Johab
  my @euc = unpack("C*", $_[0]);
  my ($fe_off,$hi_off,$lo_off) = (0,0,1);
  my @out = ();
```

```
  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;
    $fe_off = 21 if $hi == 73;
    $fe_off = 34 if $hi == 126;
    ($hi_off,$lo_off) = ($lo_off,$hi_off) if ($hi < 74 or $hi > 125);
    push(@out, (((($hi + $hi_off) >> 1) + ($hi < 74 ? 200 : 187) - $fe_off),
      $lo + ((($hi + $lo_off) & 1) ? ($lo > 110 ? 34 : 16) : 128));
  }
  return pack("C*", @out);
}
```

Note that any program that includes this subroutine must not apply it to the code ranges provided in Table C-1, which represent modern jamo and hangul syllables. These characters require completely different handling, as explained in Chapter 4.

Table C-1. Unaffected encode ranges—Johab code conversion algorithm

Character class	ISO-2022-KR	EUC-KR
Modern jamo	24 21–24 54	A4 A1–A4 D4
Hangul syllables	30 21–48 7E	B0 A1–C8 FE

Johab to ISO-2022-KR or EUC-KR Conversion

The following Perl subroutine can be used to convert Johab-encoded symbols and hanja into ISO-2022-KR or EUC-KR encoding. This subroutine actually returns ISO-2022-KR–encoded characters, but the further transformation to EUC-KR encoding is a trivial operation.

```
sub johab2ks ($) { # Convert Johab to ISO-2022-KR
  my @johab = unpack("C*", $_[0]);
  my ($offset,$d8_off) = (0,0);
  my @out = ();

  while (($hi, $lo) = splice(@johab, 0, 2)) {
    $offset = 1 if ($hi > 223 and $hi < 250);
    $d8_off = ($hi == 216 and ($lo > 160 ? 94 : 42));
    push(@out, ((((($hi - ($hi < 223 ? 200 : 187)) << 1) -
      ($lo < 161 ? 1 : 0) + $offset) + $d8_off),
      $lo - ($lo < 161 ? ($lo > 126 ? 34 : 16) : 128));
  }
  return pack("C*", @out);
}
```

Like with conversion to Johab encoding, conversion from Johab encoding using the preceding subroutine affects only a limited encoding region, specifically those for encoding symbols, including ancient jamo, and hanja. Two-byte codes whose first byte is within the range 0x84 through 0xD3 are hangul, and are converted through other means. Those two-byte codes whose first byte is within the range 0xD8 through 0xDE and 0xE0 through 0xF9 are handled by this subroutine.

TRON Code Conversion

The standard TRON character set, as described in Appendix E, consists of the JIS X 0208:1997, JIS X 0212-1990, GB 2312-80, and KS X 1001:2004 character sets. Conversion between TRON encoding and those encodings for JIS X 0208:1997 and JIS X 0212-1990 is trivial, and requires minor adjustments to Perl code provided earlier in this chapter. However, converting between TRON encoding and those encodings for GB 2312-80 and KS X 1001:2004 is less trivial, and brings to bear a different code conversion technique, specifically zero-base code conversion.

Zero-base code conversion is a useful technique for dealing with encodings that have different dimensions, but whose character ordering is identical. Put simply, zero-base code conversion transforms one encoding into a single contiguous list of values starting at 0 (zero). For a 94×94 encoding whose encoding range is <21 21> through <7E 7E> (ISO-2022 encoding with 8,836 code points), the result is the range 0 through 8835 (remember that this list begins at 0, not 1). Reversing this effect, but with different parameters, can effectively fit a 94×94 encoding block into a block of different dimensions. Consider the following line of code:

```
$char = (($hi - 33) * 94) + ($lo - 33);
```

First we assume that we are dealing with an encoding that fits within a 94×94 matrix, and whose byte values are in the range 0x21 through 0x7E (decimal 33 through 127). The first byte's value is stored in the variable *$hi*, and the second byte's is in *$lo*. Note how the lowest byte value is subtracted from each byte (`$hi - 33` and `$lo - 33`). Then, the first byte's value is multiplied by the number of code points in the second byte's range (in this case, 94).

TRON and GB 2312-80 Code Conversion

The 8,836 2-byte code points supported by the encodings for GB 2312-80, such as ISO-2022-CN (<21 21> through <7E 7E>) and EUC-CN (<A1 A1> through <FE FE>), fall into the TRON encodings range <21 80> through <67 8F>. While ISO-2022-CN and EUC-CN encodings are based on encodings rows with 94 code points each (that is, 0x21 through 0x7E or 0xA1 through 0xFE), TRON encoding, for GB 2312-80, is based on encodings rows with 126 code points each (0x80 through 0xFD).

The following function, *gb2tron()*, converts ISO-2022-CN– or EUC-CN–encoded GB 2312-80 2-byte characters into TRON encoding:

```
sub gb2tron ($) {                          # EUC-CN or ISO-2022-CN to TRON
  my @euc = unpack("C*", $_[0]);
  my $char;
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;                # Normalize to ISO-2022-CN
    $char = (($hi - 33) * 94) + ($lo - 33); # Normalize to zero-base
    push(@out, (($char / 126) + 33), (($char % 126) + 128));
  }
```

```
  return pack("C*", @out);
}
```

The following function, *tron2euc_cn()*, converts TRON-encoded GB 2312-80 characters back into EUC-CN encoding:

```
sub tron2euc_cn ($) {                          # TRON GB 2312-80 to EUC-CN
  my @euc = unpack("C*", $_[0]);
  my $char;
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $char = (($hi - 33) * 126) + ($lo - 128); # Normalize to zero-base
    push(@out, (($char / 94) + 161), (($char % 94) + 161));
  }
  return pack("C*", @out);
}
```

TRON and KS X 1001:2004 Code Conversion

The 8,836 2-byte code points supported by the encodings for KS X 1001:2004, such as ISO-2022-KR (<21 21> through <7E 7E>) and EUC-KR (<A1 A1> through <FE FE>), fall into the TRON encodings range <B7 80> through <FD 8F>. Whereas ISO-2022-KR and EUC-KR encodings are based on encodings rows with 94 code points each (that is, 0x21 through 0x7E or 0xA1 through 0xFE), TRON encoding, for KS X 1001:2004, is based on encodings rows with 126 code points each (0x80 through 0xFD), as you learned in the previous section.

The following function, *ks2tron()*, converts ISO-2022-KR– or EUC-KR–encoded KS X 1001:2004 2-byte characters into TRON encoding:

```
sub ks2tron ($) {                             # EUC-KR or ISO-2022-KR to TRON
  my @euc = unpack("C*", $_[0]);
  my $char;
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $hi &= 127; $lo &= 127;                   # Normalize to ISO-2022-KR
    $char = (($hi - 33) * 94) + ($lo - 33); # Normalize to zero-base
    push(@out, (($char / 126) + 183), (($char % 126) + 128));
  }
  return pack("C*", @out);
}
```

The following function, *tron2euc_kr()*, converts TRON-encoded KS X 1001:2004 characters back into EUC-KR encoding:

```
sub tron2euc_kr ($) {                           # TRON KS X 1001:2004 to EUC-KR
  my @euc = unpack("C*", $_[0]);
  my $char;
  my @out = ();

  while (($hi, $lo) = splice(@euc, 0, 2)) {
    $char = (($hi - 183) * 126) + ($lo - 128); # Normalize to zero-base
```

```
    push(@out, (($char / 94) + 161), (($char % 94) + 161));
  }
  return pack("C*", @out);
}
```

Unicode Code Conversion

Although conversion between Unicode and legacy encodings is table-driven, conversion between the various Unicode encoding methods—UTF-8, UTF-16, and UTF-32—is purely algorithmic. This section covers conversion between these encoding methods, and does so by providing a complete test-bed program that implements the necessary conversion algorithms as subroutines. Two of the subroutines, specifically *UTF32toUTF8()* and *UTF8toUTF32()*, were originally adapted from code written by Gisle Aas in order to support conversion between UCS-2 and UTF-8 encodings. I should also point out that for UTF16 and UTF-32 encodings, big-endian byte order is assumed.

```
#!/usr/bin/perl -w

use strict;
require 5.003;

my ($i,$o); # Variable for storing the selected input/output encodings
my ($line); # Variable for storing each input line, which is character code

while (@ARGV and $ARGV[0] =~ /^-/) {
  my $arg = shift @ARGV;
  if (lc $arg eq "-h") {
    &ShowHelp;
    exit;
  } elsif (lc $arg =~ /^-i(8|16|32)$/) {
    $i = $1;
    print STDERR "Input encoding is UTF-$i¥n";
  } elsif (lc $arg =~ /^-o(8|16|32)$/) {
    $o = $1;
    die "Identical input/output encodings!¥nExit¥n" if $i eq $o;
    print STDERR "Output encoding is UTF-$o¥n";
  } else {
    die "Invalid option: $arg! Skipping (try ¥"-h¥" for help)¥nExit¥n";
  }
}

if ($ARGV[0] =~ /^((?:[0-9A-Fa-f][0-9A-Fa-f])+)$/) {
  $line = $1;
  &DoConv($line);
} else {
  while(defined($line = <STDIN>)) {
    chomp $line;
    &DoConv($line);
  }
}
```

```
sub DoConv ($) {
  if ($i == 8) {
    if ($o == 16) {
      printf STDOUT "%s¥n",
        uc unpack("H*",&UTF32toUTF16(&UTF8toUTF32(pack("H*",$line))));
    } elsif ($o == 32) {
      printf STDOUT "%s¥n",uc unpack("H*",&UTF8toUTF32(pack("H*",$line)));
    }
  } elsif ($i == 16) {
    if ($o == 8) {
      printf STDOUT "%s¥n",
        uc unpack("H*",&UTF32toUTF8(&UTF16toUTF32(pack("H*",$line))));
    } elsif ($o == 32) {
      printf STDOUT "%s¥n",uc unpack("H*",&UTF16toUTF32(pack("H*",$line)));
    }
  } elsif ($i == 32) {
    if ($o == 8) {
      printf STDOUT "%s¥n",uc unpack("H*",&UTF32toUTF8(pack("H*",$line)));
    } elsif ($o == 16) {
      printf STDOUT "%s¥n",uc unpack("H*",&UTF32toUTF16(pack("H*",$line)));
    }
  }
}

sub UTF16toUTF32 ($) {
  my ($bytes) = @_;

  if ($bytes =~ /^([¥x00-¥xD7¥xE0-¥xFF][¥x00-¥xFF])$/) {
    pack("N",unpack("n",$bytes));
  } elsif ($bytes =~ /^([¥xD8-¥xDB][¥x00-¥xFF])([¥xDC-¥xDF][¥x00-¥xFF])$/) {
    pack("N",((unpack("n",$1) - 55296) * 1024) + (unpack("n",$2) - 56320) + 65536);
  } else {
    die "Whoah! Bad UTF-16 data!¥n";
  }
}

sub UTF8toUTF32 ($) {
  my ($bytes) = @_;

  if ($bytes =~ /^([¥x00-¥x7F])$/) {
    pack("N",ord($1));
  } elsif ($bytes =~ /^([¥xC0-¥xDF])([¥x80-¥xBF])$/) {
    pack("N",((ord($1) & 31) << 6) | (ord($2) & 63));
  } elsif ($bytes =~ /^([¥xE0-¥xEF])([¥x80-¥xBF])([¥x80-¥xBF])/) {
    pack("N",((ord($1) & 15) << 12) | ((ord($2) & 63) <<  6) | (ord($3) & 63));
  } elsif ($bytes =~ /^([¥xF0-¥xF7])([¥x80-¥xBF])([¥x80-¥xBF])([¥x80-¥xBF])/) {
    pack("N",((ord($1) & 7) << 18) | ((ord($2) & 63) << 12) | ((ord($3) & 63) <<  6)
    | (ord($4) & 63));
  } else {
    die "Whoah! Bad UTF-8 data! Perhaps outside of Unicode (5- or 6-byte).¥n";
  }
}
```

```
sub UTF32toUTF8 ($) {
  my ($ch) = unpack("N",$_[0]);

  if ($ch <= 127) {
    chr($ch);
  } elsif ($ch <= 2047) {
    pack("C*", 192 | ($ch >> 6), 128 | ($ch & 63));
  } elsif ($ch <= 65535) {
    pack("C*", 224 | ($ch >> 12), 128 | (($ch >> 6) & 63), 128 | ($ch & 63));
  } elsif ($ch <= 1114111) {
    pack("C*", 240 | ($ch >> 18), 128 | (($ch >> 12) & 63), 128
    | (($ch >> 6) & 63), 128 | ($ch & 63));
  } else {
    die "Whoah! Bad UTF-32 data! Perhaps outside of Unicode (UCS-4).¥n";
  }
}

sub UTF32toUTF16 ($) {
  my ($ch) = unpack("N",$_[0]);

  if ($ch <= 65535) {
    pack("n", $ch);
  } elsif ($ch <= 1114111) {
    pack("n*", ((($ch - 65536) / 1024) + 55296),(($ch % 1024) + 56320));
  } else {
    die "Whoah! Bad UTF-32 data! Perhaps outside of Unicode (UCS-4).¥n";
  }
}

sub fix {
  my ($string) = @_;
  $string =~ s/^  //gm;
  return $string;
}

sub ShowHelp {
  print STDERR &fix(<<ENDHELP);
  UTFConv.pl (for Perl5)
  Written by Ken Lunde (lunde¥@adobe.com)
  Program Copyright 2001 Ken Lunde. All Rights Reserved.

  SWITCHES:
   -i = Input encoding
   -o = Output encoding

  Note that only the values 8, 16, or 32 can be used with these switches,
  and must follow without spaces.
ENDHELP
}
```

The *UTF16toUTF32()*, *UTF8toUTF32()*, *UTF32toUTF8()*, and *UTF32toUTF16()* subroutines perform the work, and the *DoConv()* subroutine acts as a wrapper. When conversion between UTF-8 and UTF-16 encodings is necessary, UTF-32 encoding acts as the

middle ground. In other words, when converting from UTF-8 to UTF-16 encoding, the *UTF8toUTF32()* subroutine is invoked, immediately followed by the *UTF32toUTF16()* subroutine.

Encoding Detection

The following Perl program illustrates an effective way to automatically detect CJKV encodings, using Japanese encodings as an example. The two lines that have been emboldened are those that perform the actual detection.

This program applies encoding detection on every line of the file, and outputs the line prefixed with information about what encoding was detected.

```
#!/usr/local/bin/perl -w

# The function in this program, DetectJPEncoding(), checks the data
# that it is given, and returns various values depending on what
# encoding it detected. The return values are listed in the definition
# of the %codes hash below. You can feed this function as much data as
# you wish (such as single characters, lines, or the entire buffer),
# but the more you give it, the better the chance it will correctly
# return a single encoding (that is, not "Ambiguous"). It currently
# deals with Japanese encodings through the use of encoding templates.

%codes = (
  0 => "ERROR",
  1 => "Shift-JIS",
  2 => "EUC-JP",
  3 => "Ambiguous",                          # Means ASCII, Shift-JIS, or EUC-JP
  4 => "ISO-2022-JP"
);

open(SJS,"<jis.sjs") or die "Cannot open Shift-JIS file!\n";
open(EUC,"<jis.euc") or die "Cannot open EUC-JP file!\n";
open(JIS,"<jis.jis") or die "Cannot open ISO-2022-JP file!\n";
open(OUT,">out") or die "Cannot open output file!\n";

while (defined($line = <SJS>)){
  print OUT $codes{&DetectJPEncoding($line)} . ": " . $line;
}
close(SJS);

while (defined($line = <EUC>)){
  print OUT $codes{&DetectJPEncoding($line)} . ": " . $line;
}
close(EUC);

while (defined($line = <JIS>)){
  print OUT $codes{&DetectJPEncoding($line)} . ": " . $line;
}
close(JIS);
```

```
sub DetectJPEncoding ($) {
  my $data = shift;
  return 4 if $data =~ m{                       # Return if ISO-2022-JP
    \e                                          # Escape character
    (?:
        \$[\@B]                                 # JIS X 0208 series
      | \([BHIJ]                                # ASCII or JIS X 0201-1997
    )
  }x;
  my ($sjs_out,$euc_out) = (0,0);
  my $euc_jp = q{                               # EUC-JP encoding
      [\x00-\x7F]                               # Code set 0
    | \x8E[\xA0-\xDF]                           # Code set 2
    | \x8F[\xA1-\xFE][\xA1-\xFE]                # Code set 3
    | [\xA1-\xFE][\xA1-\xFE]                    # Code set 1
  };
  my $sjs = q{                                  # Shift-JIS encoding
      [\x00-\x7F\xA0-\xDF]                      # ASCII and half-width katakana
    | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # Two-byte range
  };
  $sjs_out = 1 if $data =~ /\A (?:$sjs)+ \Z/ox;
  $euc_out = 2 if $data =~ /\A (?:$euc_jp)+ \Z/ox;

  return ($sjs_out + $euc_out);
}
```

Through the careful use of the encoding templates that are found later in this appendix, the preceding code can be adapted so that it can automatically detect virtually any encoding, within reason. This same code can also be used to check the integrity of a file's encoding.

Repairing ISO-2022-JP Encoding

As discussed in Chapter 4, ISO-2022-JP encoding (and other ISO-2022–based encodings) can be damaged in a number of ways. The "escape" character, for example, can be damaged as follows:

- Converted into a single space (0x20)
- URL transformation
- Converted into Quoted-Printable encoding
- Simply removed from the file

The following Perl program effectively repairs damaged ISO-2022-JP–encoded files, even if they were damaged in multiple ways (which is usually not the case):

```
#!/usr/local/bin/perl -w

# o Converted into a single space (0x20)
# o Converted into URL transformation -- "%1B"
# o Converted into quoted-printable -- "=1B"
#
```

```
# Or they are simply deleted.

while (defined($line = <STDIN>)) {
  $line =~ s{
    (?:\x20|[=%]1[Bb])?             # Optional space or escape
    (
      (?:
        \$ [\@B]                    # $ plus @ or B
        (?:[\x21-\x7E][\x21-\x7E])+ # One or more two-byte characters
      )
      |                             # Or...
      (?:
        \( I                        # ( plus I
        [\x20-\x5F]+?               # One or more half-width katakana
      )
    )
    (?:\x20|[=%]1[Bb])?             # Optional space or escape
    (
      \( [BHJ]                      # ( plus B, H, or J
    )
  }{\e$1\e$2}gx;
  print STDOUT $line;
}
```

Other Useful Transformations

There are other useful programs written in Perl, including libraries designed for CJKV information processing. Chapter 4 provided references to some useful Perl modules and libraries for CJKV data manipulation.

The following sections provide some simple Perl programs for performing a number of common text-processing tasks.

URL Transformation

The following Perl program, a single line of code, is used to encode a string according to URL transformation:

```
$string =~ s/([^0-9A-Za-z])/sprintf("%%%02X",ord($1))/ge;
```

Likewise, the following Perl program effectively reverses the effect:

```
$string =~ s/%([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/ge;
```

Quoted-Printable Transformation

To encode quoted-printable:

```
$string =~ s/([=\x00-\x1F\x80-\xFF])/sprintf("=%02X",ord($1))/ge;
```

To decode quoted-printable:

```
$string =~ s/=([0-9A-Fa-f][0-9A-Fa-f])/chr hex $1/ge;
$string =~ s/=[\n\r]+$//;
```

CJKV Encoding Templates

The following are some encoding specifications that can be used for handling various CJKV encodings. In particular, they are useful in conjunction with automatically detecting CJKV encodings.

EUC-CN and EUC-KR Encodings

```
$euc = q{
    [\x00-\x7F]            # Code set 0 (ASCII or equivalent)
  | [\xA1-\xFE][\xA1-\xFE] # Code set 1 (GB 2312-80 or KS X 1001:2004)
};
```

EUC-TW Encoding

```
$euc_tw = q{
    [\x00-\x7F]                           # Code set 0 (CNS-Roman)
  | [\xA1-\xFE][\xA1-\xFE]                # Code set 1 (Plane 1)
  | \x8E[\xA1-\xF0][\xA1-\xFE][\xA1-\xFE] # Code set 2 (Planes 1-80)
};
```

EUC-JP Encoding

```
$euc_jp = q{
    [\x00-\x7F]                # Code set 0 (ASCII/JIS-Roman)
  | [\xA1-\xFE][\xA1-\xFE]     # Code set 1 (JIS X 0208:1997)
  | \x8E[\xA0-\xDF]            # Code set 2 (Half-width katakana)
  | \x8F[\xA1-\xFE][\xA1-\xFE] # Code set 3 (JIS X 0212-1990)
};
```

GBK and Big Five Plus Encodings

```
$gbk = q{
    [\x00-\x7F]                     # ASCII or equivalent
  | [\x81-\xFE][\x40-\x7E\x80-\xFE] # Two-byte (GBK or Big Five Plus)
};
```

GB 18030 Encoding

```
$gb18030 = q{
    [\x00-\x7F]                                  # ASCII or equivalent
  | [\x81-\xFE][\x40-\x7E\x80-\xFE]              # Two-byte
  | [\x81-\xFE][\x30-\x39][\x81-\xFE][\x30-\x39] # Four-byte
};
```

Big Five Encoding

```
$big5 = q{
    [\x00-\x7F]                     # ASCII/CNS-Roman
  | [\xA1-\xFE][\x40-\x7E\xA1-\xFE] # Big Five
};
```

Big Five Encoding for Hong Kong SCS-2008

```
$big5_hkscs = q{
    [\x00-\x7F]                     # ASCII/CNS-Roman
  | [\x87-\xFE][\x40-\x7E\xA1-\xFE] # Big Five for Hong Kong SCS-2008
};
```

Shift-JIS Encoding

```
$sjs = q{
    [\x00-\x7F]                             # ASCII/JIS-Roman
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # JIS X 0208:1997
  | [\xA0-\xDF]                             # Half-width katakana
};
```

Johab Encoding

```
$johab = q{
    [\x00-\x7F]                             # ASCII/KS-Roman
  | [\x84-\xD3][\x41-\x7E\x81-\xFE]          # Modern hangul
  | [\xD8-\xDE\xE0-\xF9][\x31-\x7E\x91-\xFE] # Symbols and hanja
};
```

UHC Encoding

```
$uhc = q{
    [\x00-\x7F]                             # One-byte
  | [\x81-\xFE][\x41-\x5A\x61-\x7A\x81-\xFE] # Two-byte
};
```

UCS-2 and UTF-16 Encodings

The following encoding template works for UCS-2 encoding in any byte order:

```
$ucs2 = q{
  [\x00-\xFF][\x00-\xFF]
};
```

The following encoding template is for UTF-16 encoding, in big-endian byte order, and with support for the surrogates area:

```
$utf16be = q{
    [\x00-\xD7\xE0-\xFF][\x00-\xFF]              # One code unit
  | [\xD8-\xDB][\x00-\xFF][\xDC-\xDF][\x00-\xFF] # Two code units (Surrogates)
};
```

And, the following encoding template is for UTF-16 encoding, in little-endian byte order, and with support for the surrogates area:

```
$utf16le = q{
    [\x00-\xFF][\x00-\xD7\xE0-\xFF]              # One code unit
  | [\x00-\xFF][\xD8-\xDB][\x00-\xFF][\xDC-\xDF] # Two code units (Surrogates)
};
```

UTF-32 Encoding

The following encoding template is for UTF-32 encoding in big-endian byte order:

```
$utf32be = q{
    \x00[\x00-\x10][\x00-\xD7\xE0-\xFF][\x00-\xFF]
};
```

The following encoding template is for UTF-32 encoding in little-endian byte order:

```
$utf32le = q{
    [\x00-\xFF][\x00-\xD7\xE0-\xFF][\x00-\x10]\x00
};
```

Note how the 2,048 code points for the Surrogates have been excluded from the appropriate byte-value ranges.

UTF-8 Encoding

The following UTF-8 encoding template supports the original one- to six-byte representation of this encoding method:

```
$utf8_old = q{
    [\x00-\x7F]                                                 # One-byte
  | [\xC2-\xDF][\x80-\xBF]                                      # Two-byte
  | \xE0[\xA0-\xBF][\x80-\xBF]                                  # Three-byte
  | [\xE1-\xEF][\x80-\xBF][\x80-\xBF]                           # Three-byte
  | \xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF]                       # Four-byte
  | [\xF1-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]                # Four-byte
  | \xF8[\x88-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]            # Five-byte
  | [\xF9-\xFB][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF]     # Five-byte
  | \xFC[\x84-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF] # Six-byte
  | \xFD[\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF][\x80-\xBF] # Six-byte
};
```

The following UTF-8 encoding template supports the one- to four-byte representation that is now considered standard, and is compatible with UTF-16 and UTF-32 encodings:

```
$utf8 = q{
    [\x00-\x7F]                               # One-byte
  | [\xC2-\xDF][\x80-\xBF]                    # Two-byte
  | \xE0[\xA0-\xBF][\x80-\xBF]                # Three-byte
  | [\xE1-\xEF][\x80-\xBF][\x80-\xBF]         # Three-byte
  | \xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF]     # Four-byte
  | [\xF1-\xF3][\x80-\xBF][\x80-\xBF][\x80-\xBF] # Four-byte
  | \xF4[\x80-\x8F][\x80-\xBF][\x80-\xBF]     # Four-byte
};
```

Multiple-Byte Anchoring

The following Perl program illustrates how to apply and test multiple-byte anchoring. This technique is critical in order to ensure that regex matches are applied according to character, not byte, boundaries.

```
#!/usr/local/bin/perl -w

$search = "\x8C\x95";                          # "剣"
$text1 = "Text 1 \x90\x56\x8C\x95\x93\xB9";  # "Text 1 新剣道"
$text2 = "Text 2 \x94\x92\x8C\x8C\x95\x61";  # "Text 2 白血病"
$encoding = q{                                 # Shift-JIS encoding
    [\x00-\x7F]                                # ASCII
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # Two-byte range
  | [\xA0-\xDF]                                # Half-width katakana
};

print "First attempt -- no anchoring\n";
print " Matched Text1\n" if $text1 =~ /$search/o;
print " Matched Text2\n" if $text2 =~ /$search/o;

print "Second attempt -- anchoring\n";
print " Matched Text1\n" if $text1 =~ /^(?:$encoding)*?$search/ox;
print " Matched Text2\n" if $text2 =~ /^(?:$encoding)*?$search/ox;
```

The following is the result of running the preceding program (assuming we name it *mb-anchor.pl*):

```
% perl mb-anchor.pl
First attempt -- no anchoring
 Matched Text1
 Matched Text2
Second attempt -- anchoring
 Matched Text1
```

Note how anchoring causes correct matching to take place. The text in the variable *$text2* does not contain the search character (<8C 95>), but its byte sequence does occur between two characters (<8C 8C> and <95 61>).

But, unlike the conventional regex anchors used in Perl, such as ^ and $, and other regular expression implementations, these anchors consume characters.

Multiple-Byte Processing

The following program illustrates how to break up data consisting of multiple-byte characters into separate list elements, where each list element contains one character. This particular program doesn't do anything terribly useful, but does check whether each

character consists of one or two bytes, and then prints out two-byte characters, along with their hexadecimal codes.

```
#!/usr/local/bin/perl -w

$encoding = q{                                # Shift-JIS encoding
    [\x00-\x80\xFD-\xFF]                      # ASCII and other one-byte
  | [\xA0-\xDF]                               # Half-width katakana
  | [\x81-\x9F\xE0-\xFC][\x40-\x7E\x80-\xFC] # Two-byte range
};

while (defined($line = <STDIN>)) {
  @enc = $line =~ /$encoding/gox; # One character per element
  foreach $element (@enc) {
    if (length($element) == 2) {  # If two-byte character
      print STDOUT "0x" . ($x = uc unpack("H*",$element), $x);
    } else {                      # All others are one-byte characters
      print STDOUT "$element\n";
    }
  }
}
```

I find this code useful in developing code converters that make use of table-driven conversion, such as conversion between Unicode and legacy encodings.

APPENDIX D
Glossary

The following glossary entries provide definitions, explanations, and sometimes entertaining commentary for terms found throughout this book, so I suggest that you consult it on an as-needed basis. It also makes for good reading if you have nothing else better to do on a cloudy, rainy, or otherwise boring day.

42
: The answer to the *Ultimate Question of Life, the Universe, and Everything*.

50 Sounds array
: 50音配列 (*gojūon hairetsu*). The Japanese keyboard array whose keys follow the arrangement of the 50 Sounds Table. *See also 50 Sounds Table.*

50 Sounds order
: 50音順 (*gojūon jun*). A Japanese collation sequence that follows the ordering from the 50 Sounds Table. *See also 50 Sounds Table.*

50 Sounds Table
: 50音表 (*gojūon hyō*). A table made up of a 5×10 matrix whose total number of possible sounds is 50. Kana characters are set into this table.

A11Y
: Abbreviation for accessibility.

AAT
: Apple Advanced Typography. The newer name for the font portions of Apple's QuickDraw GX technology, which was used in conjunction with ATSUI. *See also ATSUI.*

AD
: Active Duty. Abbreviated form of advertisement. Also, an abbreviation for the Latin *Anno Domino*.

AFDKO
: Adobe Font Development Kit for OpenType. Adobe Systems' suite of commercial-grade, command-line font tools that were specifically tailored to aid font developers to build high-quality OpenType fonts.

AFM
: Adobe Font Metrics. The file format, for PostScript fonts, that encapsulates per-glyph width and bounding box information.

AI
: Artificial Intelligence (人工知能 *jinkō chinō* in Japanese) or Adobe Illustrator.

AIR
: What we breathe. Also, Adobe Integrated Runtime. The name of a cross-platform runtime environment for building rich web applications through the use of Adobe Flash, Adobe Flex, HTML, or Ajax, and that can be deployed as a desktop application. Its code name during development was *Apollo*.

AIX
: Advanced Interactive Executive. IBM's version of the Unix OS.

Algorithmic conversion
: A type of conversion that makes use of mathematical transformations to change the values of the converted objects. *See also table-driven conversion.*

America Online
: An Internet service provider.

ANK
: Alphabet, Numerals, and Katakana. One way to refer to the characters defined in JIS X 0201-1997, specifically JIS-Roman and half-width katakana.

Annex S
: The informative ISO 10646 annex, entitled *Procedure for the Unification and Arrangement of CJK Ideographs*, which describes the rules and principles of Han Unification. *See also Han Unification.*

ANS
: Amiga Nihongo System. The Japanese OS for Amiga computers.

ANSI
: American National Standards Institute.

ANSI X3.4-1986
: Coded Character Set—7-Bit American National Standard Code for Information Interchange. The standard that defines the ASCII character set standard.

ANSI Z39.64-1989
: *See EACC.*

Antelope
: *Antilocapra americana*. North America's fastest land mammal.

AOL
: *See America Online.*

ASCII
: American Standard Code for Information Interchange.

Assume
: An acronym.

AT&T JIS
: Another name for the Japanese instance of the EUC encoding method. *See EUC-JP.*

ATC
: Adobe Type Composer. Refers to a now-obsolete frontend application for creating rearranged fonts for Mac OS, and to the underlying technology for supporting rearranged fonts.

ATM
: Asychronous Transfer Mode, Automated Teller Machine, or Adobe Type Manager. Go figure out which applies to this book! Actually, given the fact that Adobe Type Manager's functionality is now included in Mac OS X and the latest versions of Windows, it seems that this no longer applies, other than for historical purposes.

ATSUI
: Apple Type Services for Unicode Imaging. When considered a Japanese transliteration, it can mean "hot" (熱い *atsui*) or "thick" (厚い *atsui*).

Base64
: A method used for safely transforming non-ASCII characters in email messages or email message headers.

Base Character
: The first—and primary or meaningful—element of a Unicode sequence, such as an Ideographic Variation Sequence (IVS). *See also Ideographic Variation Sequence.*

Basic Multilingual Plane
: *See BMP.*

BATAC
: Badger Attack TAC.

Batang
: 바탕 (*batang*) in Korean. The current and preferred way to refer to serif typeface designs in Korean. *See also dotum.*

BBS
: 電子掲示板 (*denshi keijiban*) in Japanese. Bulletin Board System.

BC

Ballistic Coefficient. Before Christ, from the Latin *Ante Christum*. Also, Base Character. *See Base Character.*

BDF

Bitmap Distribution Format. A popular bitmapped font format developed by Adobe Systems.

Bézier curve

The type of curve used for representing character shape contours in the PostScript page-description language and its supported font formats.

Big-endian

Refers to the byte order, and pertains only to data represented by more than 8 bits, such as the 16- and 32-bit Unicode encoding forms (UTF-16 and UTF-32), short (16 bits in Java), integer (32 bits in Java), float (32 bits in Java), long (64 bits in Java), and double (64 bits in Java) types. True multiple-byte encodings, such as Big Five, Shift-JIS, EUC, and ISO-2022, and the UTF-8 encoding form, are not affected by byte order. Using the decimal value 4660 as an example, when it is represented using 16 bits, its big-endian form is <12 34>, and its little-endian form is <34 12>. *See also byte order.*

Big Five

大五 (*dàwǔ*). The name of the Chinese character set and encoding used extensively in Taiwan. Big Five is not a national standard, but is roughly equivalent to the first two planes of CNS 11643-2007. *See also CNS 11643-2007.*

Big Five Plus

An extension to Big Five that includes the remaining CJK Unified Ideographs in ISO 10646-1:1993, meaning the URO, that are not in Big Five. *See also URO.*

Binary

二进制/二進制 (*èrjìnzhì*) in Chinese, 二進法 (*nishinhō*) in Japanese, and 이진법/二進法 (*ijinbeop*) in Korean. Base two. A numeric notation that uses two possible values, 0 or 1.

Bit

位 (*wèi*) or 位元 (*wèiyuán*) in Chinese, ビット (*bitto*) in Japanese, 비트 (*biteu*) in Korean. Binary digit. The basic units of memory that computers process.

Bitmapped font

A font whose character shapes are defined by arrays of bits. *See also parametric font and outline font.*

Bitnet

Because It's Time NETwork. Also called CREN.

Blowfish

河豚 (*fugu*) in Japanese. The animal whose image graces the cover of this book.

BMP

Basic Multilingual Plane. Unicode's first of 17 planes, consisting of 65,536 code points each. The BMP includes what are considered the most frequently used characters. Also called Plane 0.

BOM

Byte Order Mark. U+FEFF. A Unicode character that serves to indicate the byte order (or endianness) of the Unicode text that follows. For the UTF-16 encoding form, it is represented as <FE FF> in big-endian byte order, or <FF FE> in little-endian. It is also used with the UTF-32 encoding form.

Bopomofo

ㄅㄆㄇㄈ. *See zhuyin.*

Boten

傍点 (*bōten*). A Japanese term that refers to glyph annotations that serve to emphasize characters, similar to the use of underlining in Western text. These annotations usually appear above the character in horizontal writing, or to its right in vertical writing.

BTRON

Business TRON. *See TRON.*

Byte

字节/字節 (*zìjié*) or 位元组 (*wèiyuánzǔ*) in Chinese, バイト (*baito*) in Japanese, and 바이트 (*baiteu*) in Korean. An 8-bit unit.

Byte order
: The order of the bytes in multiple-byte storage units, which often differs depending on the computer architecture. *See also big-endian and little-endian.*

CAE
: Common Applications Environment.

Calligraphic
: *See cursive.*

Candidate
: 候補 (*kōho*) in Japanese. During typical CJKV input that involves ideographs, candidate refers to the names that are associated with keys in a conversion dictionary. Candidates are usually presented as a list from which the user must select. *See also key and name.*

Cangjie
: 倉頡 (*cāngjié*), short for 倉頡輸入法 (*cāngjié shūrùfǎ*). A popular structure-based Chinese input method.

Canonical Equivalent
: Refers to the relationship between two different Unicode code points or sequences. Normalization may cause a Unicode code point or sequence to change into the Canonical Equivalent, which may be a different Unicode code point or sequence. And, as the name suggests, they are to be treated as equivalent. *See also Normalization.*

CCAG
: 國字整理小組 (*guózì zhěnglǐ xiǎozǔ*). Chinese Character Analysis Group.

CCCII
: 中文資訊交換碼 (*zhōngwén zīxùn jiāohuàn mǎ*). Chinese Character Code for Information Interchange.

CCITT
: International Telegraphy and Telephony Consultative Committee. Comité Consultatif International Télégraphique et Téléphonique in French.

CCJK
: Chinese (Simplified), Chinese (Traditional), Japanese, and Korean. The two Cs refers to the two distinct types of Chinese, specifically Simplified and Traditional.

CCS
: *See Coded Character Set.*

CCSID
: Coded Character Set Identifier. IBM terminology that uniquely identifies a coded character set.

ccTLD
: Country Code Top-Level Domain. *See also gTLD and TLD.*

CD
: Compact Disk.

CDL
: Character Description Language. A powerful XML application that was originally designed for describing ideographs, regardless of whether they are encoded. It has been proven to be useful for describing any character or glyph. CDL was developed by Tom Bishop and Richard Cook.

CDRA
: Character Data Representation Architecture. IBM's solution for conversion among different character sets and encodings.

CD-ROM
: Compact Disk Read Only Memory.

Cell
: 位 (*wèi*) in Chinese, 点 (*ten*) in Japanese, and 렬/列 (*ryeol*) or 열/列 (*yeol*) in Korean. In a two-byte encoding, cell refers to the second byte. In a two-dimensional matrix, cell usually represents the values along the horizontal axis. *See also row and Row-Cell.*

CERNET
: China Education and Research NETwork.

CEF
: *See Character Encoding Form.*

CER
: *See Character Entity Reference.*

CES
: *See Character Encoding Scheme.*

CESI
: China Electronics Standardization Institute.

CFF
: Compact Font Format.

CGI
: Common Gateway Interface. The name given to web programs that are executed on the server side (as opposed to the client side). CGI programs are typically written in Perl.

Character
: 文字 (*wénzì*) in Chinese, 文字 (*moji*) in Japanese, and 문자/文字 (*munja*) in Korean. An abstract notion denoting a class of shapes declared to have the same meaning or form.

Character Encoding Form
: *See encoding form.*

Character Encoding Scheme
: *See encoding scheme.*

Character Entity Reference
: An SGML-derived notation that is recognized by HTML and XML, and serves to specify characters by their name. For example, the five-character string `&` is the CER for the ampersand (U+0026). *See also Numeric Character Reference.*

Character set
: 文字集合 (*moji shūgo*) in Japanese. A collection of characters.

Character spanning
: 均等割付 (*kintō waritsuke*). A special case of justification that is done on a much smaller scale than in the West. Character spanning, sometimes known as Japanese justification, is typically used for lists of names whereby varying numbers of characters per name are made flush to the left and right, but not to the margin of the printed page.

Chinese
: 国语/國語 (*guóyǔ*), 汉语/漢語 (*hànyǔ*), or 中文 (*zhōngwén*). The languages spoken in the Chinese locales, such as China, Hong Kong, Singapore, and Taiwan.

Chinese character
: *See ideograph.*

Chữ Hán
: 字漢. Ideographs as used in Vietnam.

Chữ Nôm
: 字喃. Vietnamese-made ideographs.

CIC
: Communication Intelligence Corporation.

CID
: Character IDentifier. The key used to access outline (glyph) data in CIDFont resources.

CITS
: China Information Technology Standardization Committee.

CJK
: Chinese, Japanese, and Korean. CJKV without the V. *See CJKV.*

CJK Compatibility Ideographs
: The name of the ideographs of Chinese origin that are included in Unicode for compatibility purposes, because they would otherwise be unified with an existing CJK Unified Ideograph. CJK Compatibility Ideographs are also subject to Normalization. *See also CJK Unified Ideographs and Normalization.*

CJK Unified Ideographs
: The name of the ideographs of Chinese origin that are included in Unicode, and divided among various blocks, specifically the URO and the multiple Extensions. *See also URO.*

CJK.INF
: The online document that appeared after *Understanding Japanese Information Processing* was published, but before the first edition of this book was published. Discontinued.

CJK-JRG
: *See IRG.*

CJKV-Roman
: A term that collectively refers to the instances of the ASCII character set as defined by the CJKV locales, including GB-Roman, CNS-Roman, JIS-Roman, KS-Roman, and TCVN-Roman. *See also GB-Roman, CNS-Roman, JIS-Roman, KS-Roman, and TCVN-Roman.*

CJKV
: Chinese, Japanese, Korean, and Vietnamese. 中日韩越/中日韓越 (*zhōng rì hán yuè*) in Chinese, 日中韓越 (*nittchūkan'etsu*) in Japanese, and 한중일월/韓中日越 (*han jung il wol*) in Korean. Refers to the languages that use ideographs as part of their writing system.

CJKV Information Processing
: The incredibly clever title of a book that was first published at the beginning of 1999, and is effectively a revision of a book that was originally published in 1993.

CJKV6N
: Abbreviation for CJKVization, in keeping with the tradition used for L10N, I18N, and J10N. *See CJKVization.*

CJKVese
: A collective term that refers to the languages spoken in CJKV locales.

CJKVization
: CJKV6N. The process of adapting software for CJKV markets. *See also CJKV, internationalization, Japanization, localization.*

CLDR
: Common Locale Data Repository.

cmap
: Character Map. The all-lowercase four-character tag for the 'sfnt' table that maps character codes to GIDs that are specified in the 'glyf' or 'CFF' tables.

CMap
: Character Map. The name of the resource that maps character codes to CIDs that are specified in CIDFont resources.

CN
: The two-letter country code for China.

CNNIC
: China Internet Network Information Center.

CNS
: 中國國家標準 (*zhōngguó guójiā biāozhǔn*). Chinese National Standard.

CNS 5205-1989
: The standard that defines CNS-Roman, which is the Taiwanese equivalent of ASCII. *See CNS-Roman.*

CNS 7654-1989
: The Taiwanese version of ISO 2022:1994. *See ISO 2022:1994.*

CNS 11643-2007
: The national character set standard for Taiwan that encodes a staggering 70,739 characters in 13 of its 80 planes (Planes 1 through 7 and 10 through 15). Previous versions were dated 1986 and 1992 with fewer characters. Planes 1 and 2 are roughly equivalent to Big Five, which is used much more frequently. *See also Big Five.*

CNS 14649-1:2002
: Taiwan's version of the ISO 10646 standard, and equivalent to Unicode version 3.0. *See also ISO 10646.*

CNS 14649-2:2003
: Taiwan's version of the ISO 10646 standard, and equivalent to Unicode version 3.1. *See also ISO 10646.*

CNS-Roman
: The Taiwanese equivalent of the ASCII character set and encoding. The name of the standard that defines this character set is called CNS 5205-1989. *See CNS 5205-1989.*

Code Page
: IBM and Microsoft terminology for a character set and encoding combination. CJKV Code Pages, because of the sheer number of characters, rarely take up a single page.

Code position
: The numeric code within an encoding method that is used to refer to a specific character. For two-byte characters, this refers to the row and the cell.

Code space
: コード領域 (*kōdo ryōiki*) in Japanese. The space in which characters can be encoded according to the specifications of a given encoding method. Code positions outside the code space are considered invalid.

Coded Character Set
: A mapping from a set of abstract characters to a set of integers. A character set that is intended to be encoded. All the character sets described in Chapter 3 are valid CCSs. *See also Noncoded Character Set.*

Compound
: 熟語 (*jukugo*) in Japanese. A word consisting of two or more characters.

Compound ideograph
: 会意文字/會意文字 (*huìyì wénzì*) in Chinese, 会意文字 (*kaii moji*) in Japanese, and

회의문자/會意文字 (*hoeui munja*) in Korean. An ideograph that is built from two or more primitive elements, which may be pictographs or simple ideographs. The ideograph 明 is an example, which is composed of 日 and 月. *See also pictograph and simple ideograph.*

CompuServe
: An Internet service provider.

Computer
: 计算机/計算機 (*jìsuànjī*) or 电脑/電腦 (*diànnǎo*) in Chinese, コンピュータ (*konpyūta*) or 計算機 (*keisanki*) in Japanese, and 컴퓨터 (*keompyuteo*) or 계산기/計算器 (*gyesangi*) in Korean.

Control character
: 制御文字 (*seigyo moji*) in Japanese. A character whose purpose is to control printing devices or communication devices as opposed to actually producing visible marks on a screen or printer. Carriage return, for example, is a control character, whereas the letter "A" is a printable character.

Conversion dictionary
: 変換辞書 (*henkan jisho*) in Japanese. The dictionary that is used by an input method to convert input into ideographs. Each entry in this dictionary is a key, along with one or more names associated with it. The number of entries in such dictionaries now measure in the hundreds of thousands. *See also input method, kana-to-kanji conversion, key, and name.*

CP
: Code Page. *See Code Page.*

CPAN
: Comprehensive Perl Archive Network. A huge repository of Perl documentation and software.

CPGID
: Code Page Global Identifier. IBM terminology for Code Page. *See also Code Page.*

CPSI
: Configurable PostScript Interpreter.

CPU
: 中央処理装置 (*chūō shori sōchi*) in Japanese. Central Processing Unit. Usually refers to the computer itself.

CREN
: *See Bitnet.*

CS
: Computer Science or Chinese Simplified.

CSS
: Cascading Style Sheets. A language for controlling the presentation of web pages, to include color, fonts, layout, and other aspects of how web pages display.

CT
: Chinese Traditional.

CTRON
: Central and Communications TRON. *See TRON.*

CTS
: Computerized Typesetting System.

Cubic spline curve
: *See Bézier curve.*

Cursive
: A smoother, handwritten style of a glyph. Hiragana is an example of a cursive script. Also called calligraphic writing.

D
: 点/點 (*diǎn*) in Chinese. An abbreviation (the first letter of *diǎn*) standing for the fourth of five basic stroke types used as building blocks for ideographs. Represents diagonal strokes that are written left to right. *See also stroke.*

Dakuten
: 濁点 (*dakuten*). Refers to the diacritic mark that serves to transform many kana characters into their voiced counterparts. For example, the katakana character *ta* (タ) is transformed into *da* (ダ). Also called *nigori* (濁り *nigori*) and the voiced mark. *See also handakuten.*

Dangling line breaking
: *See push-out-only line breaking.*

Data
: An android member of the Enterprise-D and Enterprise-E crew who constantly endeavored to become more human—created by

cyberneticist Dr. Noonien Soong in the Omicron Theta colony. Also, information.

DBCS
: Double-Byte Character Set. A character set whose characters are represented by two bytes.

DBCS-EUC
: A double-byte character set encoded according to the specification of EUC.

DBCS-Host
: A double-byte character set with an encoding method designed for running on IBM host computers.

DBCS-PC
: A double-byte character set with an encoding method designed for running on PCs.

DEC
: Digital Equipment Corporation.

DEC Kanji
: The Japanese character set and encoding defined by DEC. There are two implementations: DEC Kanji and Super DEC Kanji.

Decimal
: 十进制/十進制 (*shíjinzhì*) in Chinese, 十進法 (*jisshinhō*) in Japanese, and 십진법/十進法 (*sipjinbeop*) in Korean. Base 10. A numeric notation that uses 10 possible values, ranging from 0 to 9.

Design space
: *See em-square.*

Designator sequence
: A sequence used by some ISO-2022–based encodings for indicating the character sets to use when shifting characters are used. Unlike an escape sequence, a designator sequence does not actually change character sets. *See also escape sequence.*

Diachronic
: A linguistics term that refers to linguistic changes that occur between different periods.

Diacritic mark
: A mark or ornament that serves to annotate characters with additional information, usually a variant reading. Diacritic marks are typically found above or below characters. In the West it is common to see accented characters such as á, à, â, ä, ã, å, and ç. Japanese examples include the hiragana characters ば (*ba*) and ぱ (*pa*), which are derived from the basic hiragana character は (*ha*).

Dialect
: A linguistics term that refers to different flavors of a language that are usually spoken in different regions.

Display PostScript
: A special version of PostScript designed for computer monitor output. It was used as standard software on the NeXT platform. Abbreviated as DPS.

DOS
: Disk Operating System.

Dotum
: 돋움 (*dotum*) in Korean. The current and preferred way to refer to sans serif typeface designs in Korean. *See also batang.*

DPI
: Dots-per-inch. A measurement for device resolution.

DPRK
: 조선 민주 주의 인민 공화국/朝鮮民主主義人民共和國 (*joseon minju juui inmin gonghwaguk*). Democratic People's Republic of Korea. The official name for North Korea.

DPS
: *See Display PostScript.*

DTD
: Document Type Definition. SGML and XML terminology that refers to a document's type so that it can be interpreted correctly.

DTP
: Desktop Publishing.

DVD
: Digital Versatile Disc or Digital Video Disc. A popular storage media, suitable for data or video.

Dvorak array
: A Western keyboard array developed by August Dvorak and William Dealey in the 1930s

as an improvement over the QWERTY keyboard array. *See also QWERTY array.*

EACC
: East Asian Character Code. The common reference to ANSI Z39.64-1989, East Asian Character Code For Bibliographic Use. Based on CCCII. *See also CCCII.*

EB
: 電子ブック (*denshi bukku*) in Japanese. Electronic book.

EBCDIC
: Extended Binary-Coded-Decimal Interchange Code. An encoding for the ASCII character set standard developed by IBM for use on IBM-based computers. Used in conjunction with several double-byte character sets and encodings, such as DBCS-Host (IBM), JEF (Fujitsu), and KEIS (Hitachi). Requires eight bits for representation.

EBCDIK
: Extended Binary-Coded-Decimal Interchange Kana Code. A Japanese version of EBCDIC that includes uppercase Latin characters, numerals, symbols, half-width katakana, and control characters. *See also EBCDIC.*

Ecma
: European Computer Manufacturers Association.

Electronic Character Set
: *See Coded Character Set.*

Elvis
: The King of Rock 'n Roll. Also, a *vi* clone written by Steve Kirkendall for which Japanese (*jelvis*) and Korean (Hangul Elvis) versions exist.

em
: Twice the width of an *en*. Refers to the width of the uppercase "M," which is typically the widest glyph in a font. *See also en.*

Em-box
: *See em-square.*

Em dash
: A dash that is the same width as an uppercase "M." *See also em.*

Em-square
: A square space whose height and width roughly corresponds to the width of the letter "M." Also called a mutton. "Em-box" or "design space" are sometimes thought to be better terms because some typeface designs have a nonsquare (rectangular) design space.

Emacs
: An extremely powerful text editor that has been ported to a variety of platforms.

en
: The two-letter language code for English. Also, half of an *em*. *See also em.*

En dash
: A dash that is the half the width of an em dash. *See also em dash.*

Encoding
: 符号化 (*fugōka*) in Japanese. The method of defining the correspondence between numerical character codes and the final printable glyphs. For instance, 0x41 is the ASCII (JIS-Roman) code for the uppercase Latin character "A." U+0041 is the corresponding Unicode code point for the same character.

Encoding form
: Also called *character encoding form*, which Unicode defines as the mapping from a character set definition to the actual code units used to represent the data. Unicode's three encoding forms are UTF-8, UTF-16, and UTF-32. *See also UTF-8, UTF-16, and UTF-32.*

Encoding scheme
: Also called *character encoding scheme*, which Unicode defines as *character encoding form* plus byte serialization. Unicode's seven encoding schemes are UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, and UTF-32LE. *See also encoding form, UTF-8, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, and UTF-32LE.*

Escape character
: エスケープ文字 (*esukēpu moji*) in Japanese. The control character (0x1B or U+001B) that is used as part of an escape sequence. Escape sequences are used in ISO-2022-JP encodings

to switch between one- and two-byte modes. *See also escape sequence, ISO-2022-JP, ISO-2022-JP-1, and ISO-2022-JP-2.*

Escape sequence
: エスケープシーケンス (*esukēpu shīkensu*) in Japanese. A string of characters that contains one or more escape characters, and is used to signify a shift in mode of some sort. In the case of the Japanese character set, it is used to shift between one- and two-byte modes, and to shift between different character sets or different versions of the same character set. *See also shifting characters.*

eTRON
: Entity and Economy TRON. *See TRON.*

EUC
: Extended Unix Code. There are locale-specific instances of EUC encoding, each of which can specify up to four code sets.

EUC-CN
: The instance of EUC encoding for China, which uses two of the four code sets, and supports the GB 1988-89 and GB 2312-80 character sets.

EUC-JP
: The instance of EUC encoding for Japan, which uses all four code sets, and supports the JIS X 0201-1997, JIS X 0208:1997, and JIS X 0212-1990 character sets.

EUC-KR
: The instance of EUC encoding for Korea, which uses two of the four code sets, and supports the KS X 1003:1993 and KS X 1001:2004 character sets. Sometimes called Wansung.

EUC-TW
: The instance of EUC encoding for Taiwan, which uses three of the four code sets, and supports the CNS 5205-1989 and CNS 11643-2007 character set standards.

External character
: *See gaiji, system-defined character, system-specific character, and user-defined character.*

Extension A
: The block of 6,582 additional CJK Unified Ideographs that were added to the BMP in Unicode version 3.0. *See also CJK Unified Ideographs.*

Extension B
: The block of 42,711 additional CJK Unified Ideographs that were added to Plane 2 in Unicode version 3.1. *See also CJK Unified Ideographs.*

Extension C
: The block of 4,149 additional CJK Unified Ideographs that were added to Plane 2 in 2008. *See also CJK Unified Ideographs.*

FAGAT
: Forum of Asian Graphic Arts Technology. A technology forum sponsored by JAGAT. *See also JAGAT.*

Fangsong
: 仿宋 (*fǎngsòng*) or 仿宋体 (*fǎngsòngtǐ*). The Chinese semi-script typeface style.

FAQ
: Frequently Asked Questions. A document that contains answers to frequently asked questions. Documentation often includes a FAQ section whose purpose is to preemptively answer a significant number of common questions, thus serving to minimize the number of questions that are subsequently asked.

FBM
: Fusion Battle Mistress.

FDPC
: 文字フォント開発・普及センター (*moji fonto kaihatsu fukyū sentā*). Font Development and Promotion Center. The now-defunct consortium, which is part of JSA, that developed the Heisei series of Japanese typefaces.

FEP
: Front-End Processor. A common name for early input methods, which were so named from the way they captured keyboard keystrokes before they were sent to the current application. These keyboard keystrokes were processed, converted into a mixture of kana and kanji, and finally sent to the current application. *See also input method.*

FH
: FreeHand.

FIT
: Focus on Integrated Typesetting (飞腾 *fēiténg*; meaning "to soar"). The name of a page-composition system developed by Peking University Founder Group.

Fixed-length encoding
: An encoding method whereby every character in the character set is represented by the same number of bits or bytes. Examples include UCS-2, UCS-4, UTF-32, and EUC complete two-byte format. Actually, ASCII encoding, if used by itself, is fixed-length. *See also modal encoding and non-modal encoding.*

fj
: From Japan. The initial two letters found in the names of Usenet Newsgroups distributed within Japan. These newsgroups were also available outside of Japan.

FM
: Frequency Modulation or FrameMaker.

FM-R
: The name of the PC series of computers produced by Fujitsu.

FMapType
: A PostScript language key (integer) that indicates which mapping algorithm to use when interpreting the sequence of bytes in a string. All PostScript Type 0 (composite) fonts must specify an FMapType.

Font
: The instantiated form of a typeface. Unicode defines "font" as a collection of glyphs used for the visual depiction of character data. *See also typeface.*

Fontworks
: フォントワークス (*fontowākusu*). A major Japanese type foundry. *See also Morisawa, Ryobi, and Shaken.*

FSF
: Free Software Foundation.

FSH
: Fusion Steel Heart.

FSS-UTF
: File System Safe UTF. *See UTF-8.*

FTP
: File Transfer Protocol. A common way to move files between host computers, and sometimes between a host computer and a personal computer.

Full-width
: 全角 (*quánjiǎo*) or 全形 (*quánxíng*) in Chinese, 全角 (*zenkaku*) in Japanese, and 전각/全角 (*jeongak*) in Korean. A character whose shape occupies a space roughly that of a square. Most CJKV characters are considered to be full-width. *See also half-width.*

Furigana
: 振り仮名 (*furigana*). *See ruby.*

G
: ゴジラ (*gojira*). Godzilla. The King. The King of the Monsters. A life form first encountered in 1954 in Japan, and subsequently went through nuclear meltdown in late 1995. Survived by Junior Godzilla. A leaner (and meaner) version of Godzilla appeared in 1998. G sometimes refers to Gamera (ガメラ *gamera*), a giant turtle-like creature. Abbreviation for Glock (the G33, chambered in the potent .357 Sig, happens to be my favorite). Also, an old unit of typographic measurement used strictly for type size, written 号数/號數 (*hàoshù*) in Chinese, 号数 (*gōsū*) in Japanese, and 호수/號數 (*hosu*) in Korean. Unlike other typographic units of measurement, the scale is not absolute but relative, and the larger the value of G, the smaller the relative size. Though similar, the systems used in Japan and China differ in subtle ways. In Japan, for example, 0G (the largest size) is equivalent to 42 points, and 8G is equivalent to 4 points.

G11N
: Abbreviation for globalization. *See globalization.*

Gaiji
: 外字 (*gaiji*). The name given to desired glyphs that are not available in the selected font. Sometimes called *external character*. *See also system-defined character, system-specific character, and user-defined character.*

Gaiji solution
: A solution that makes it possible to interchange documents that contain nonstandard glyphs to systems that do not have such glyphs installed, and allows such glyphs to be properly used, displayed, and printed.

Gairaigo
: 外来語 (*gairaigo*). Means "foreign word," but usually refers to loan words written using katakana.

Gakushū Kanji
: 学習漢字 (*gakushū kanji*). The 1,006 kanji that are formally taught in the Japanese educational system during the first six grades. Originally enumerated 996 kanji in 1977.

GB
: Short for "Guo Biao" (国标 *guóbiāo*), which is, in turn, short for "Guojia Biaozhun" (国家标准 *guójiā biāozhǔn*), and means "National Standard" in Chinese. Also, gigabyte. 1,073,741,824 or $1,024^3$ bytes.

GB 1988-89
: The standard that defines GB-Roman, which is the Chinese equivalent of ASCII. Originally designated GB 1988-80. *See GB-Roman.*

GB 2311-80
: The Chinese version of the ISO 2022:1994 standard. *See ISO 2022:1994.*

GB 2312-80
: The standard that describes the basic Chinese character set as used in China (PRC). Also known as GB0. It enumerates 7,445 characters.

GB 6345.1-86
: The standard that details additions (132 characters) and corrections for GB 2312-80. *See GB 2312-80.*

GB 7589-87
: The standard that enumerates 7,237 additional hanzi. Also known as GB2.

GB 7590-87
: The standard that enumerates 7,039 additional hanzi. Also known as GB4.

GB 8565.2-88
: The standard on which ISO-IR-165:1992 is based. It is based on GB 2312-80, but provides additional characters. *See also GB 2312-80 and ISO-IR-165:1992.*

GB 13000.1-93
: China's version of the ISO 10646 standard, and equivalent to Unicode version 1.1. *See also ISO 10646.*

GB 18030-2005
: The latest version of the most widely implemented character set standard in China. It can be considered an extended version of GBK, and is code-point–compatible with Unicode. Its original version was dated 2000. The 2005 version acknowledged six regional scripts and CJK Unified Ideographs Extension B. *See also GBK.*

GBK
: An extended version of GB 2312-80 that includes additional hanzi to complete the URO, and was later extended to become GB 18030-2000. The "K" in GBK represents the first sound in the Chinese word meaning "extension" (扩展 *kuòzhǎn*). *See also GB 18030-2005 and URO.*

GB/T
: The "T" is short for "Tuijian" (推荐 *tuījiàn*), which means "recommended" (as opposed to "forced" or "mandatory") in Chinese. *See GB for the meaning of GB.*

GB/T 12345-90
: The traditional analog of GB 2312-80—contains 2,180 hanzi not found in GB 2312-80. Most of these are hanzi replacements, but some are placed into additional rows, specifically rows 88 and 89. Also known as GB1. *See also GB 2312-80.*

GB/T 13131-2XXX
: The traditional analog of GB 7589-87. Also known as GB3. Not yet published. *See also GB 7589-87.*

GB/T 13132-2XXX
: The traditional analog of GB 7590-87. Also known as GB5. Not yet published. *See also GB 7590-87.*

GB-Roman
: The Chinese equivalent of the ASCII character set and encoding. The name of the standard that defines this character set is called GB 1988-89. *See GB 1988-89.*

GID
: Glyph IDentifier. The key used to access outline (glyph) data within 'sfnt' resources, specifically in 'glyf' (TrueType) or 'CFF' (PostScript Type 2 charstring) tables.

GIF
: Graphics Interchange Format.

GL
: Graphic Left. Usually refers to an encoding whose bytes have the eighth bit turned off, such as ISO-2022.

Globalization
: Abbreviated as G11N. Equivalent to internationalization, but includes the business and marketing aspects. Globalization is not necessarily the same thing as global domination. *See also internationalization and localization.*

Gloss
: *See ruby.*

Glyph
: A specific instance of a character. A classic example is that "f" and "i" are two separate glyphs, but you can fuse these two characters into a single glyph called a ligature: fi. *See also ligature.*

GNU
: Short for "GNU is Not Unix." A series of Unix-based software that is provided free of charge. GNU software (and other software that seeks protection) falls under the terms of the GNU General Public License, which protects software from being exploited for commercial uses. It ensures that there will always be a large body of software freely available.

Gothic
: ゴシック (*goshikku*) or ゴシック体 (*goshikkutai*) in Japanese and 고딕 (*godik*) or 고딕체/고딕體 (*godikche*) in Korean. The name commonly given to the Japanese typeface style in which horizontal and vertical strokes are of the same relative weight. In Korean, the preferred way in which to refer to sans serif is now *dotum*. This is roughly equivalent to the sans serif typeface style in Western typography. *See also dotum and sans serif.*

GPOS
: Glyph POSitioning.

GR
: Graphic Right. Usually refers to an encoding whose bytes have the eighth bit turned on, such as EUC.

Grep
: Global regular expression print. The standard regex-based pattern-matching utility, which is standard on most Unix and Unix-like systems.

GSUB
: Glyph SUBstitution.

gTLD
: Generic Top-Level Domain. *See also ccTLD and TLD.*

GUI
: Graphical User Interface.

Gukja
: 국자/國字 (*gukja*). *See hanguksik hanja.*

H
: 齿数/齒數 (*chǐshù*) in Chinese, 歯数 (*hasū*) in Japanese, and 치수/齒數 (*chisu*) in Korean. Unit of typographic measurement equivalent to 0.25mm, and its correct usage is strictly for measurements other than type size. Also, 横 (*héng*) in Chinese. An abbreviation (the first letter of *héng*) standing for the first of five basic stroke types used as building blocks for ideographs. Represents horizontal strokes. *See also stroke.*

Half-width
: 半角 (*bànjiǎo*) or 半形 (*bànxíng*) in Chinese, 半角 (*hankaku*) in Japanese, and 반각/半角 (*bangak*) in Korean. A character whose shape occupies a space half that of a square. ASCII characters as used in the West are typically considered half-width. *See also full-width.*

Han Unification
: The effort on the part of the Unicode Consortium to collapse the Chinese, Japanese,

and Korean versions of ideographs down to a common code set by eliminating duplication based on a set of unification rules and principles.

Handakuten
半濁点 (*handakuten*). Refers to the circle-like diacritic mark that serves to transform h-row kana characters into their p-row counterparts. For example, katakana *ha* (ハ) is transformed into katakana *pa* (パ). Also called *maru* (丸 *maru*; literally means "circle.") and the semi-voiced mark. *See also dakuten.*

Hanging line breaking
See push-out-only line breaking.

Hanguksik hanja
한국식 한자/韓國式漢字 (*hanguksik hanja*). Korean-made ideographs.

Hangul
한글 (*hangeul*). The name of the native Korean script. Each hangul syllable is typically composed of two or three hangul elements (called jamo). *See also jamo.*

Hankaku
半角 (*hankaku*). Analogous to half-width. *See half-width.*

Hanja
한자/漢字 (*hanja*). The Korean word for ideograph. *See also ideograph.*

Hanmun Gyoyukyong Gicho Hanja
한문 교육용 기초 한자/漢文教育用基礎漢字 (*hanmun gyoyukyong gicho hanja*) in Korean. The set of 1,800 hanja that all students in Korea are expected to learn.

Hanzi
汉字/漢字 (*hànzì*). The Chinese word for ideograph. *See also ideograph.*

Hei
黑 (*hēi*) or 黑体 (*hēitǐ*). The Chinese name for the equivalent of the sans serif typeface style. 黑 literally means "black" in Chinese. *See also sans serif.*

Heisei
平成 (*heisei*). The name of the current Japanese era, which began in 1989. Also the name of the typefaces that have been produced by developing members of FDPC. *See also FDPC.*

Hexadecimal
十六进制/十六進制 (*shíliùjìnzhì*) in Chinese, 十六進法 (*jūrokushinhō*) in Japanese, and 십육진법/十六進法 (*sipyukjinbeop*) in Korean. Base 16. A numeric notation that uses 16 possible values, specifically 0 through 9 and A through F. The most common notation used in the computer world.

Hiragana
平仮名 (*hiragana*). The cursive Japanese syllabic script. Together with katakana is collectively called kana. *See also kana and katakana.*

HK
Heckler & Koch, the name of a famous German arms manufacturer. My wife's initials. Also, the two-letter country code for Hong Kong.

HKDNR
Hong Kong Domain Name Registration Company. The name of Hong Kong's NIC. *See also NIC.*

HKSCS
See Hong Kong SCS-2008.

HOG
Heavy Ordinance Grade.

Hojo Kanji
補助漢字 (*hojo kanji*). Supplemental kanji. The name given to the kanji contained in JIS X 0212-1990. These kanji are ordered by radical, then by total number of strokes.

Hong Kong
香港 (*xiānggǎng*). The name of a Chinese locale, which became part of China (PRC) in 1997, and is considered an SAR (*Special Administrative Region*). Still considered a separate locale from China.

Hong Kong GCCS
Hong Kong Government Chinese Character Set. A Hong Kong character set that was designed as an extension of Big Five, and was obsoleted by Hong Kong SCS. *See also Hong Kong SCS-2008.*

Hong Kong SCS-2008
: Hong Kong Supplementary Character Set, 2008 Revision. Sometimes further abbreviated as HKSCS or HKSCS-2008. The Hong Kong character set that is designed as an extension of Big Five and as a subset of Unicode. Its current 2008 version includes 5,009 characters, 4,568 of which are hanzi. Previous versions were dated 1999, 2001, and 2004. Superceded Hong Kong GCCS. *See also Hong Kong GCCS.*

HP
: Hewlett-Packard.

HP-UX
: Hewlett-Packard's version of the Unix operating system.

HTML
: HyperText Markup Language. An application of SGML, and the standard language used to write content-specifying documents for the Web.

I18N
: Abbreviation for internationalization. *See internationalization.*

IANA
: Internet Assigned Numbers Authority. *See ICANN.*

IBM
: アイ・ビー・エム (*ai bī emu*) in Japanese. International Business Machines Corporation.

IBM-eucJP
: A specific instance of DBCS-EUC to include ASCII/JIS-Roman and half-width katakana.

IBM-932
: A specific instance of DBCS-PC to include ASCII/JIS-Roman and half-width katakana.

IBM Japanese
: The name of the Japanese character set as defined by IBM. Some implementations include IBM-eucJP and IBM-932.

ICANN
: Internet Corporation for Assigned Names and Numbers.

ICU
: International Components for Unicode.

ID
: IDentification, IDentity, or IDentifier. Usually refers to the user's name as used to access a web service or host computer, but can also be used to identify objects.

Ideograph
: 汉字/漢字 (*hànzì*) in Chinese, 漢字 (*kanji*) in Japanese, 한자/漢字 (*hanja*) in Korean, and chữ Hán (字漢) in Vietnamese. The characters that originated in China and are used in other East Asian locales, such as Hong Kong, Japan, Korea, Singapore, Taiwan, and Vietnam.

Ideographic Rapporteur Group
: *See IRG.*

Ideographic Variation Sequence
: IVS. A registered Unicode sequence consisting of a CJK Unified Ideograph, which serves as a Base Character, followed by Variation Selector, and corresponds to a variant form of the Base Character. *See also Unicode Variation Sequence.*

IDN
: Internationalized Domain Name.

IE
: An abbreviation for Latin *id est*. Also, Internet Explorer.

IEC
: International Electrotechnical Commission. Often associated with ISO standards as ISO/IEC, meaning that the standard is a joint effort between ISO and IEC. *See also ISO.*

IEEE
: Institute of Electrical and Electronics Engineers.

IETF
: Internet Engineering Task Force. A volunteer organization that deals with networking issues on the Internet. Also refers to the documents produced by this organization. These are also called Internet Drafts. They are then called RFCs when they are no longer in draft status. *See also RFC.*

IIJ
: Internet Initiative Japan.

IKIS
: Interactive Kanji Information System. The Japanese character set and encoding developed by Nippon Data General.

IM
: Instant Messenger. Chances are it refers to input method in this book. *See input method.*

IME
: Input Method Editor. *See input method.*

InDesign
: Adobe Systems' page-composition application whose Japanese version provides an unprecedented level of support for high-end Japanese typography.

Inline conversion
: インライン変換 (*inrain henkan*) in Japanese. The ability to handle Japanese input at the cursor position rather than in a dedicated window.

Indexing
: The process of locating the encoded position of a character, thus providing access to it.

INFI
: Busse Combat's proprietary blade steel.

Information interchange
: 信息交换 (*xìnxī jiāohuàn*) or 資訊交換 (*zīxùn jiāohuàn*) in Chinese, 情報交換 (*jōhō kōkan*) in Japanese, and 정보교환/情報交換 (*jeongbo gyohwan*) in Korean. The process of moving information from one hardware or software configuration to another with no loss of data.

Information processing
: 信息处理 (*xìnxī chǔlǐ*) or 資訊處理 (*zīxùn chǔlǐ*) in Chinese, 情報処理 (*jōhō shori*) in Japanese, and 정보처리/情報處理 (*jeongbo cheori*) in Korean. The process of manipulating digitally encoded information at different levels. Japanese code and text processing are forms of information processing.

Input method
: The software that enables users to input characters from a large character set using a limited number of keyboard keys.

Internationalization
: Abbreviated as I18N. The process of architecting software (or hardware) in a flexible manner such that it becomes an easy task to adapt or localize to another country with different languages. Internationalization also makes it possible to use more than one script on computers. There are two main implementations of internationalization: the locale model and the multilingual model. *See also globalization, locale model, localization, and multilingual model.*

Internet
: The name given to the world-wide network of computers.

IP
: Internet Protocol.

IRG
: Ideographic Rapporteur Group. Formerly called the CJK Joint Research Group (CJK-JRG). Its full designation is ISO/IEC JTC1/SC2/WG2/IRG, which is an abbreviated form of ISO/IEC Joint Technical Committee 1, Subcommittee 2, Working Group 2, Ideographic Rapporteur Group.

Iroha
: いろは or 伊呂波 (*iroha*). A Japanese collation sequence based on the same sounds from the 50 Sounds order. *See also 50 Sounds order.*

IRV
: International Reference Version.

ISO
: 国際標準化機構 (*kokusai hyōjunka kikō*) in Japanese. International Organization for Standardization.

ISO 639-1:2002
: A standard that establishes two-letter lowercase language codes, used as the first part of a locale designation.

ISO 639-2:1998
: A standard that establishes three-letter lowercase language codes, used as the first part of a locale designation.

ISO 646:1991
: Identical to CJKV-Roman except for some minor locale-specific differences, such as currency symbols. Equivalent to ASCII. *See also ASCII and CJKV-Roman.*

ISO 2022:1994
: The standard that details the escape sequences used for encoding character sets beyond ISO 646:1991 or ASCII. This standard forms the foundation for ISO-2022 and EUC encodings.

ISO-2022-CN
: An encoding method, based on techniques described in ISO 2022:1994, for handling a mixture of ASCII, GB 2312-80, and CNS 11643-1992 (Planes 1 and 2). Described in RFC 1922.

ISO-2022-CN-EXT
: An encoding method, based on techniques described in ISO 2022:1994, for handling a mixture of ASCII, GB 2312-80, GB/T 12345-90, GB 7589-87, GB/T 13131-2XXX, GB 7590-87, GB/T 13132-2XXX, and CNS 11643-1992 (all planes). Described in RFC 1922.

ISO-2022-JP
: An encoding method, based on techniques described in ISO 2022:1994, for handling a mixture of ASCII and JIS X 0208:1997. Described in RFC 1468.

ISO-2022-JP-1
: An encoding method, based on techniques described in ISO 2022:1994, for handling JIS X 0212-1990. Described in RFC 2237.

ISO-2022-JP-2
: An encoding method, based on techniques described in ISO 2022:1994, for handling JIS X 0212-1990 and other character sets, such as GB 2312-80, KS X 1001:1992, and two parts of ISO 8859. Described in RFC 1554.

ISO-2022-KR
: An encoding method, based on techniques described in ISO 2022:1994, for handling a mixture of ASCII and KS X 1001:1992. Described in RFC 1557.

ISO 3166-1:2006
: A standard that establishes two-letter uppercase country codes, used as the second part of a locale designation.

ISO 6429:1992
: A standard that describes the control character range as used in ASCII.

ISO 8859
: A standard divided in 15 parts that describes extensions to the ASCII character set to handle other European languages.

ISO 8879:1986
: The standard that describes SGML. *See also SGML.*

ISO 9541:1991
: A set of three standards that describe the standard digital font format. Based on the Type 1 font format by Adobe Systems.

ISO 10646
: The standard that details the ISO version of the Unicode character set; kept in sync with Unicode through amendments and new versions. National analogs of this standard include GB 13000.1-93 (China), CNS 14649-1:2002 and CNS 14649-2:2003 (Taiwan), JIS X 0221:2007 (Japan), and KS X 1005-1:1995 (Korea).

ISO 32000-1:2008
: The ISO standard that corresponds to PDF version 1.7. *See also PDF.*

ISO-IR-165:1992
: An extension to the GB 2312-80 character set that combines all other known extensions (specifically, GB 6345.1-86 and GB 8565.2-88). *See GB 2312-80, GB 6345.1-86, and GB 8565.2-88.*

ISO/TR 11941:1996
: The first international standard that documents methods for transliterating Korean text using Latin characters.

ISP
: Internet Service Provider.

ITC
: International Typeface Corporation.

ITRON
: Industrial TRON. *See TRON.*

IVS
: *See Ideographic Variation Sequence.*

J10N
: Abbreviation for Japanization. *See Japanization.*

ja
: The two-letter language code for Japanese. Also, "yes" in German.

JAGAT
: 日本印刷技術協会 (*nihon insatsu gijutsu kyōkai*). Japan Association of Graphic Arts Technology. The association that sponsors FAGAT. *See also FAGAT.*

Jaggies
: The uneven effect when fixed-size bitmapped fonts are scaled to large sizes and subsequently displayed or printed.

JAIN
: Japanese Academic InterNetwork.

Jamo
: 자모/字母 (*jamo*). Hangul elements. Each jamo is equivalent to a character in an alphabet, either a consonant or vowel.

Japan
: 日本 (*nihon* or *nippon*). The country in which the Japanese language is spoken.

JAPAN.INF
: The name of the (now horribly obsolete) online document on which *Understanding Japanese Information Processing* was based. *See also CJK.INF.*

Japanese
: 日本語 (*nihongo*) in Japanese. The language spoken in Japan.

Japanese justification
: *See character spanning.*

Japanese line wrapping
: *See line breaking.*

Japanese punctuation logic
: *See line breaking.*

Japanization
: 日本語化 (*nihongoka*). The localization of software to the Japanese market. *See also localization.*

Jaso
: 자소/字素 (*jaso*). *See jamo.*

Java
: The most populous island of Indonesia, and a popular nickname for coffee. Incidentally, a programming language designed to be truly cross-platform. Salient features include object-oriented–only programming, cross-platform execution, associative arrays, broad Unicode support, and C-like syntax. Also rumored to be 100% secure (but, as all security specialists know, 100% security is a myth). Developed by Sun Microsystems.

JEF
: Japanese processing Extended Feature. A character set and encoding developed by Fujitsu. JIS C 6226-1978 is a subset.

Jinmei-yō Kanji
: 人名用漢字 (*jinmei-yō kanji*). The 983 kanji, above and beyond Jōyō Kanji, specified by the Japanese government as appropriate for use in writing personal names. *See also Jōyō Kanji.*

JIS
: 日本工業規格 (*nihon kōgyō kikaku*). Its original symbol is ㊐, and its current symbol is ㊎. ジス (*jisu*). Japanese Industrial Standard. The name of the standards established by JISC. Also the name of the encoding method used for the JIS X 0208:1997 and JIS X 0212-1990 character set standards. *See also JISC.*

JIS78
: Short for JIS C 6226-1978. *See JIS C 6226-1978.*

JIS83
: Short for JIS X 0208-1983. *See JIS X 0208:1997.*

JIS90
: Short for JIS X 0208-1990. Can sometimes be confused with JIS X 0212-1990 in some contexts. *See JIS X 0208:1997.*

JIS97
: Short for JIS X 0208:1997. *See JIS X 0208:1997.*

JIS2000
: Short for JIS X 0213:2000, referring to the original version of the JIS X 0213 standard that was released in 2000. *See JIS X 0213:2004.*

JIS2004
: Short for JIS X 0213:2004, referring to the 2004 revision of the JIS X 0213 standard. *See JIS X 0213:2004.*

JIS array
: JIS配列 (*JIS hairetsu*). The most widely used Japanese keyboard array. Specified in the standard JIS X 6002-1985. Like the QWERTY keyboard array in the West, it is also the most inefficient. Also called Old-JIS array.

JISC
: 日本工業標準調査会 (*nihon kōgyō hyōjun chōsakai*). Japanese Industrial Standards Committee. The name of the organization that establishes JIS standards.

JISCII
: Japanese Industrial Standard Code for Information Interchange. An improper reference to the Japanese character set standards established by JIS. More correctly known as simply JIS. *See JIS.*

JIS C 6220-1976
: The original designation of what is now known as JIS X 0201-1997. The name changed on March 1, 1987. *See JIS X 0201-1997.*

JIS C 6225-1979
: The original designation of what is now known as JIS X 0207-1979. The name changed on March 1, 1987. *See JIS X 0207-1979.*

JIS C 6226-1978
: The first double-byte character set. Developed in 1978 by Japanese Industrial Standards. Three revisions followed, first in 1983, second in 1990, and the latest in 1997. *See JIS X 0208:1997.*

JIS C 6226-1983
: The original designation of what is now known as JIS X 0208-1983. The name changed on March 1, 1987. *See JIS X 0208:1997.*

JIS C 6228-1984
: The original designation for what is now known as JIS X 0202:1998. The name changed on March 1, 1987. *See JIS X 0202:1998.*

JIS C 6232-1984
: The original designation for what is now known as JIS X 9051-1984. The name changed on March 1, 1987. *See JIS X 9051-1984.*

JIS C 6233-1980
: The original designation for what is now known as JIS X 6002-1985. The name changed on March 1, 1987. *See JIS X 6002-1985.*

JIS C 6234-1983
: The original designation for what is now known as JIS X 9052-1983. The name changed on March 1, 1987. *See JIS X 9052-1983.*

JIS C 6235-1984
: The original designation for what is now known as JIS X 6003-1989. The name changed on March 1, 1987. *See JIS X 6003-1989.*

JIS C 6236-1986
: The original designation for what is now known as JIS X 6004-1986. The name changed on March 1, 1987. *See JIS X 6004-1986.*

JIS encoding
: Usually equivalent to ISO-2022-JP encoding. The most basic Japanese encoding method that uses escape sequences to shift between one- and two-byte modes. A modal encoding method. *See ISO-2022-JP and modal encoding.*

JIS Level 1 kanji
: JIS第1水準漢字 (*JIS daiichi suijun kanji*). The name given to the 2,965 characters that constitute the first set of kanji in JIS X 0208:1997. Ordered by reading (usually ON reading). *See also JIS X 0208:1997.*

JIS Level 2 kanji
: JIS第2水準漢字 (*JIS daini suijun kanji*). The name given to the 3,390 characters that constitute the second set of kanji in JIS X 0208:1997. The 1978 version (JIS C 6226-1978) had 3,384 such kanji, and the 1983 version (JIS X 0208-1983) had 3,388 such kanji. Ordered by radical, and then by total number of strokes. *See also JIS X 0208:1997.*

JIS Level 3 kanji
: JIS第3水準漢字 (*JIS daisan suijun kanji*). The first set of 1,259 kanji enumerated by JIS X 0213:2004. Also, a name sometimes given to the kanji in JIS X 0212-1990, though the correct reference to JIS X 0212-1990 is Hojo Kanji. *See also Hojo Kanji, JIS X 0212-1990, and JIS X 0213:2004.*

JIS Level 4 kanji
: JIS第4水準漢字 (*JIS daiyon suijun kanji*). The second set of kanji, 2,436 in total, enumerated by JIS X 0213:2004. *See also JIS X 0213:2004.*

JIS order
: The order in which characters appear in the Japanese character set standards published by JSA.

JIS-Roman
: The Japanese equivalent of the ASCII character set and encoding. The name of the standard that defines this character set is called JIS X 0201-1997. *See JIS X 0201-1997.*

JIS sorting
: A sort done in JIS order. *See also JIS order.*

JIS X 0201-1997
: The standard that describes the JIS-Roman and half-width katakana character sets, along with their encodings.

JIS X 0202:1998
: The Japanese version of ISO 2022:1994. *See ISO 2022:1994.*

JIS X 0207-1979
: The Japanese version of ISO 6429:1992. *See ISO 6429:1992.*

JIS X 0208:1997
: The 1997 revision of the basic Japanese character set standard, with no changes to the number or allocation of characters. The previous version, JIS X 0208-1990, enumerated 6,879 characters. Originally issued as JIS C 6226-1978. A 1983 version was also published. Its characters are now considered part of JIS X 0213:2004. *See JIS X 0213:2004.*

JIS X 0212-1990
: The standard that describes the supplement to the Japanese character set standard. 6,067 characters are enumerated, 5,801 of which are kanji.

JIS X 0213:2004
: The 2004 revision of the JIS X 0213:2000 that added 10 kanji, bringing the total number of characters to 4,354. Its characters are in two planes. 1,259 of its kanji are referred to as JIS Level 3 kanji, and the remaining 2,436 are referred to as JIS Level 4 kanji. The original version was dated 2000.

JIS X 0221:2007
: Japan's version of ISO 10646. Previous versions were dated 1995 and 2001. *See ISO 10646.*

JIS X 4051:2004
: The standard that describes Japanese line-layout rules. Originally published in 1993 as JIS X 4051-1993. A 1995 version was also published.

JIS X 4061-1996
: The standard that sets forth the rules for sorting Japanese text.

JIS X 4062:1998
: The standard that establishes an exchange format for Japanese input method conversion dictionaries.

JIS X 4161-1993
: The standard (part 1 of 3) that describes the standard digital font format. Based on the Type 1 font format by Adobe Systems.

JIS X 4162-1993
: The standard (part 2 of 3) that describes the standard digital font format. Based on the Type 1 font format by Adobe Systems.

JIS X 4163-1994
: The standard (part 3 of 3) that describes the standard digital font format. Based on the Type 1 font format by Adobe Systems.

JIS X 6002-1985
: The standard that describes the specifications for the JIS keyboard array. *See JIS array.*

JIS X 6003-1989
: The standard that describes the layout of a kanji tablet, a large input device used to input kanji directly. *See kanji tablet.*

JIS X 6004-1986
: The standard that describes the specificatios for the New-JIS keyboard array. *See New-JIS array.*

JIS X 9051-1984
: The standard that illustrates the 16×16 dot-matrix patterns for the characters specified in JIS X 0208-1983. *See JIS X 0208:1997.*

JIS X 9052-1983
: The standard that illustrates the 24×24 dot-matrix patterns for the characters specified in JIS X 0208-1983. *See JIS X 0208:1997.*

JIS7
: A variation of ISO-2022-JP encoding that encodes half-width katakana using seven bits. *See also ISO-2022-JP.*

JIS8
: A variation of ISO-2022-JP encoding that encodes half-width katakana using eight bits. *See also ISO-2022-JP.*

JLE
: Japanese Language Environment. The name of Sun's extension that provides a Japanese environment.

JLS
: Japanese Language System. The Japanese extensions for SGI's Irix operating system.

Johab
: 조합/組合 (*johap*). Means "combining" in Korean. The name of a Korean encoding method that represents each hangul as a group of three five-bit–encoded jamo. Sometimes considered the opposite of Wansung. *See also Wansung.*

Jōyō Kanji
: 常用漢字 (*jōyō kanji*). The 1,945 kanji designated by the Japanese government as the ones to be used in public documents such as newspapers. Superseded Tōyō Kanji in 1981. *See also Tōyō Kanji.*

JP
: The two-letter country code for Japan.

JPNIC
: Japan Network Information Center.

JSA
: 日本規格協会 (*nihon kikaku kyōkai*). Japanese Standards Association. The publisher of the JIS standards.

JTRON
: Java TRON. *See TRON.*

JUNET
: Japan Unix Network. The original designation for the Internet in Japan. *See also JP.*

K
: Kilobyte. 1,024 bytes.

Kai
: 楷 (*kǎi*) or 楷体 (*kǎitǐ*). The Chinese script typeface style.

Kana
: 仮名 (*kana*). The term that collectively refers to hiragana and katakana. *See also hiragana and katakana.*

Kana-to-kanji conversion
: 仮名漢字変換 (*kana kanji henkan*). The process of converting kana input into a mixture of kana and kanji characters. The most common method of inputting Japanese text.

Kanji
: 漢字 (*kanji*). The ideographs that the Japanese borrowed from the Chinese. These number in the thousands. *See also ideograph.*

Kanji compound
: 漢語 (*kango*). A Japanese word consisting of two or more kanji.

Kanji ligature
: 漢字合字 (*kanji gōji*). A character composed of two or more kanji. Typical examples include ㍻ (平成 *heisei*; the name of a Japanese era) and ㍿ (株式会社 *kabushikigaisha*; meaning "incorporated").

Kanji tablet
: A large tablet containing thousands of individual keys, one for each character. This allows for direct kanji input.

Kanji-in
: 漢字イン (*kanji in*). The name usually given to two-byte character escape sequences as used in ISO-2022-JP encoding. A kanji-in

switches the current *n*-byte-per-character mode into two-byte mode.

Kanji-out
: 漢字アウト (*kanji auto*). The name usually given to one-byte character escape sequences as used in JIS encoding. A kanji-out switches the current *n*-byte-per-character mode into one-byte mode.

KanjiTalk
: 漢字 Talk (*kanji tōku*). The name of the localized Japanese operating system for the Apple Macintosh computer. Later called Mac OS-J. Made obsolete by Mac OS X.

KanjiTalk6 character set
: Apple's Japanese character set standard, used for KanjiTalk6 and earlier. Based largely on JIS X 0208-1983 plus NEC Row 13.

KanjiTalk7 character set
: Apple's Japanese character set standard, used for KanjiTalk 7, which later became known as Mac OS-J. Based largely on JIS X 0208-1990, but with additional characters. This character set was introduced with KanjiTalk version 7.1, and continued to be used until Mac OS X was introduced.

Katakana
: 片仮名 (*katakana*). The square-shaped Japanese syllabary. Usually used for writing recent words of foreign origin. Together with hiragana is collectively called *kana*. *See also kana and hiragana.*

Katakana ligature
: 片仮名合字 (*katakana gōji*). A glyph composed of two or more katakana characters, and for which vertical variants exist. Typical examples include ㍉ (ミリ *miri*, meaning "millimeter"), ㌢ (センチ *senchi*, meaning "centimeter"), ㍍ (メートル *mētoru*, meaning "meter"), ㌔ (キログラム *kiroguramu*, meaning "kilogram"), and ㌖ (キロメートル *kiromētoru*, meaning "kilometer").

KB
: Kilobyte. 1,024 bytes. Usually written as K.

KEIS
: Kanji processing Extended Information System. The Japanese character set and encoding developed by Hitachi.

KEIS78
: The version of KEIS which corresponds to JIS C 6226-1978. *See KEIS.*

KEIS83
: The version of KEIS which corresponds to JIS X 0208-1983. *See KEIS.*

Ken
: My given name that I write as 剣 (*ken*), 劍 (traditional form), or 劍 (variant form) in Japanese. Contrary to popular belief, it is *not* short for *Kenneth*. The name that is recorded on my birth certificate is *Ken Roger Lunde*. *See also Lunde.*

Kermit
: The name of the green frog on the children's television program called *Sesame Street*. Also, a once-popular file transfer protocol.

Key
: The basic text unit that is used to index into a conversion dictionary in order to obtain the names associated with the key. *See also candidate, conversion dictionary, and name.*

KIPS
: Korean Information Processing System. One of the original Korean character sets, which enumerated 2,058 hangul and 2,392 hanja.

ko
: The two-letter language code for Korean.

Kokuji
: 国字 (*kokuji*). Japanese-made ideographs.

Korea
: 한국/韓國 (*hanguk*). The locale or country where the Korean language is spoken. *See DPRK and ROK.*

Korean
: 한국어/韓國語 (*hangukeo*). The language spoken in Korea.

KP
: The two-letter country code for North Korea (*Democratic People's Republic of Korea*).

KPS
: KP Standard. *See KP.*

KPS 9566-97
: The first North Korean character set standard that includes hangul and hanja.

KPS 10721-2000
: A subsequent North Korean character set standard that includes approximately 20,000 hanja.

KR
: The two-letter country code for South Korea (*Republic of Korea*).

KRNIC
: Korea Network Information Center.

KS
: 한국 공업 규격/韓國工業規格 (*hanguk gongeop gyugyeok*). Its symbol is ㉿. Korean Standard.

KS C 5601-1992
: The original designation of what is now known as KS X 1001:2004. The name changed on August 20, 1997. *See KS X 1001:2004.*

KS C 5619-1982
: One of the original Korean character set standards, which enumerated only 51 modern jamo, 1,316 hangul, and 1,672 hanja. Obsoleted by KS X 1001:1992.

KS C 5620-1995
: The original designation of what is now known as KS X 1004:1995. The name changed on August 20, 1997. *See KS X 1004:1995.*

KS C 5636-1993
: The original designation of what is now known as KS X 1003:1993. The name changed on August 20, 1997. *See KS X 1003:1993.*

KS C 5657-1991
: The original designation of what is now known as KS X 1002:2001. The name changed on August 20, 1997. *See KS X 1002:2001.*

KS C 5700-1995
: The original designation of what is now known as KS X 1005-1:1995. The name changed on August 20, 1997. *See KS X 1005-1:1995.*

KS C 5715-1992
: The original designation of what is now known as KS X 5002:1992. The name changed on August 20, 1997. *See KS X 5002:1992.*

KS C 5861-1992
: The original designation of what is now known as KS X 2901:1992. The name changed on August 20, 1997. *See KS X 2901:1992.*

KS-Roman
: The Korean equivalent of the ASCII character set and encoding. The name of the standard that defines this character set is called KS X 1003:1993. *See KS X 1003:1993.*

KS X 1001:2004
: The standard that describes the basic Korean character set that enumerates 8,227 characters. Previously designated KS C 5601-1992. Previous versions were dated 1987, 1989, 1992, 1998, and 2002.

KS X 1002:2001
: The standard that describes the extended Korean character set, which enumerates additional symbols, hangul syllables (in two blocks), and hanja. Previously designated KS C 5657-1991. The original version was dated 1991.

KS X 1003:1993
: The standard that defines KS-Roman, which is the Korean equivalent of ASCII. Previously designated KS C 5636-1993. The original version was dated 1989. *See KS-Roman.*

KS X 1004:1995
: The Korean version of ISO 2022:1994. Previously designated KS C 5620-1995. *See ISO 2022:1994.*

KS X 1005-1:1995
: Korea's version of ISO 10646. It differs from ISO 10646-1:1993 in that it is based on Unicode version 2.0, which includes all 11,172 hangul syllables. Previously designated KS C 5700-1995. *See ISO 10646.*

KS X 2901:1992
: The standard that describes the EUC encoding for Korean text. Previously designated KS C 5861-1992.

KS X 5002:1992
: The standard that illustrates the basic Korean keyboard array, which is based on hangul elements (jamo). Previously designated KS C 5715-1992. Previous versions were dated 1982 and 1985.

Kun reading
: 訓読み (*kun yomi*). The name given to the native Japanese reading for a kanji.

KUTEN
: 区点 (*kuten*) in Japanese. Literally means "ward, point" (or "row, cell"). *See Plane-Row-Cell and Row-Cell.*

Kyōiku Kanji
: 教育漢字 (*kyōiku kanji*). The 881 kanji that were once formally taught during the first six years of school in Japan. Replaced by Gakushū Kanji in 1977. *See also Gakushū Kanji.*

Kyokasho
: 教科書 (*kyōkasho*) or 教科書体 (*kyōkashotai*). The Japanese semi-script typeface style. Literally, it means "textbook" or "textbook-style."

L10N
: Abbreviation for localization. *See localization.*

Latin character
: 拉丁字母 (*lādīng zìmǔ*) in Chinese, ラテン文字 (*raten moji*) or ローマ字 (*rōmaji*) in Japanese, and 로마자 (*romaja*) in Korean. The 52 upper- and lowercase characters of the Latin alphabet.

LaTeX
: A variation of TeX *See TeX.*

Ligature
: A character whose glyph consists of two or more characters fused together. An example is *fi*, which is the ligature form of the letters *f* and *i*. *See also kanji ligature and katakana ligature.*

Line breaking
: 禁則処理 (*kinsoku shori*) in Japanese. The proper handling of CJKV characters at the beginning and at the ends of lines. Punctuation, such as 「, should not terminate a line. Likewise, punctuation, such as 」, should not begin a new line. Also known as Japanese line wrapping and Japanese punctuation logic. *See also push-in-first line breaking, push-out-first line breaking, and push-out-only line breaking.*

Linux
: りぬくす (*rinukusu*) in Japanese and 리눅스 (*rinukseu*) in Korean. A popular open source Unix-compatible OS that runs on PCs and provides outstanding CJKV support.

LISA
: Localization Industry Standards Association. Also, the name of an early computer developed by Apple.

Little-endian
: The opposite of big-endian. *See big-endian.*

Locale model
: A model of internationalization that predefined many attributes that are language- or country-specific, such as the maximum number of bytes per character, date formats, time formats, currency formats, and so on. The actual attributes are located in a library or locale object file that is loaded when required. Locale plays an important part in today's internationalization efforts. *See also internationalization and multilingual model.*

Localization
: 地域化 (*chiikika*) in Japanese. Abbreviated as L10N. The process of adapting software (or a product) such that it conforms to the expectations and conventions of a specific country or region. This often includes translating menus and dialogs into the target language, but sometimes involves more complex changes, such as handling special character encoding methods. Other issues to be addressed are time zones, ways of writing dates and times, currency, culture, customs, and others. *See also globalization, internationalization, and Japanization.*

Lunde
: My family name of Viking origin that can be written 小林 (*kobayashi*) in Japanese. Note that the final "e" is not silent, and is pronounced like the name of the letter. *See also Ken.*

M

See MB.

Machine(-aided) translation

機械 (支援) 翻訳 (*kikai [shien] honyaku*) in Japanese. The process of converting text in one language into another language. Most software to date cannot fully perform this task, and pre- or post-editing by a human is usually required in order to obtain acceptable results.

Mac OS

Macintosh Operating System.

Mac OS X

Mac OS version 10. The latest and greatest version of Apple's OS.

Maru

See handakuten.

MB

Megabyte. 1,048,576 or $1{,}024^2$ bytes.

MBCS

Multiple-Byte Character Set. A character set that contains characters of mixed encoding lengths.

McCune-Reischauer

A Latin-based transliteration system for Korean text that was subsequently adapted by the Korean Ministry of Education in 1984.

MCI Mail

The name of the Internet service offered by MCI, a telecommunications company.

MENKUTEN

面区点 (*menkuten*) in Japanese. Literally means "plane, ward, point" or "plane, row, cell." *See Plane-Row-Cell.*

µITRON

Micro Industrial TRON. Also called MITRON. *See TRON.*

MIME

Multipurpose Internet Mail Extensions.

Mincho

明朝 (*minchō*) or 明朝体 (*minchōtai*). The name commonly given to the Japanese typeface style in which vertical strokes are heavy, and horizontal strokes are thin. This is roughly equivalent to the serif typeface style in Western typography. *See also serif.*

Ming

See Mincho.

MITI

通商産業省 (*tsūshō sangyō shō*). Japan's Ministry of International Trade and Industry.

MITRON

See µITRON (Micro ITRON).

MM

Multiple master or Mincho Medium.

Modal encoding

An encoding method that uses special sequences of one or more characters to signal a change in mode. Mode changes can include shifting between one- and two-byte modes, between different character sets, and between different versions of the same character set. Examples include IBM DBCS-Host, ISO-2022, JEF, KEIS, and UTF-7 encodings. *See also fixed-length encoding and non-modal encoding.*

MOE

Ministry of Education. Written 教育部 (*jiàoyùbù*) in Chinese, 文部省 (*monbushō*) in Japanese, and 교육부/敎育部 (*gyoyukbu*) in Korean.

MOR-CODE II

Morisawa's proprietary character set and encoding.

Morisawa

モリサワ (*morisawa*). A major Japanese type foundry. *See also Fontworks, Ryobi, and Shaken.*

M-style array

M式配列 (*emu shiki hairetsu*). A keyboard array designed by Masasuke Morita for NEC. It not only specifies an ergonomic keyboard design, but also an input method that allows users to select what part converts to kanji and what part does not.

MS

Microsoft Corporation.

MS-DOS

Microsoft Disk Operating System.

MS Kanji
: Another name for Shift-JIS. *See MS and Shift-JIS.*

MSB
: Most Significant Bit. The bit with the most "weight" in an eight-bit sequence (byte). This bit is what distinguishes seven- and eight-bit bytes.

MTA
: Mail Transfer Agent.

MTRON
: Macro TRON. *See TRON.*

MUA
: Mail User Agent. Another way of expressing an email client.

Multilingual model
: A model of internationalization that used a character set whose repertoire contains enough characters to represent most of the world's scripts. No switching between character sets or code pages is required. Today's internationalization model uses such a character set (Unicode) along with locale tagging. *See also internationalization and locale model.*

Multiple-byte character
: A character that is represented by more than one byte.

Myeongjo
: 명조/明朝 (*myeongjo*) or 명조체/明朝體 (*myeongjoche*). The Korean serif typeface style. Sometimes called myungjo. The preferred way in which to refer to serif is now *batang. See also batang and serif.*

Naiji
: 内字 (*naiji*). The opposite of gaiji, specifically characters that are considered to be standard (on your operating system or environment). *See also gaiji, system-defined character, system-specific character, and user-defined character.*

Name
: One or more text strings that are associated with a key in a conversion dictionary. These are presented to the user as a list of candidates from which to choose. *See also candidate, conversion dictionary, and key.*

NCR
: *See Numeric Character Reference.*

NCS
: *See Noncoded Character Set.*

NEC
: 日本電気株式会社 (*nippon denki kabushiki-gaisha*). Nippon Electronics Corporation.

NEC Kanji
: The Japanese character set standard and encoding developed by NEC. *See NEC.*

NEC-JIS
: *See NEC Kanji.*

New-JIS
: 新JIS (*shin JIS*). A common name given to the JIS X 0208-1983 character set standard. Usually refers to the two-byte escape sequence used to designate the JIS X 0208-1990 character set in JIS encoding.

New-JIS array
: 新JIS配列 (*shin JIS hairetsu*). The Japanese keyboard array that was designed to replace the JIS keyboard array. It departs from the JIS array in that there are two kana characters per key. It failed, and the JIS array is still the most commonly used among the Japanese keyboard arrays.

NFC
: Normalization Form C. Canonical Decomposition, followed by Canonical Composition. *See Normalization.*

NFD
: Normalization Form D. Canonical Decomposition. *See Normalization.*

NFKC
: Normalization Form KC. Compatibility Decomposition, followed by Canonical Composition. *See Normalization.*

NFKD
: Normalization Form KD. Compatibility Decomposition. *See Normalization.*

NIC
: Network Information Center.

NIDA
: National Internet Development Agency of Korea. The name of Korea's NIC. *See also NIC.*

NIFTY-Serve
: The name of a Japanese service provider.

Nigori
: *See dakuten.*

NLIO
: Native Language Input/Output.

NNTP
: Network News Transfer Protocol.

Noncoded Character Set
: A character set, such as Japan's Jōyō Kanji, that was designed without regard to how it would be encoded, if at all. *See also Coded Character Set.*

Nonelectronic Character Set
: *See Noncoded Character Set.*

Non-kanji
: 非漢字 (*hikanji*). Characters other than kanji, such as Latin characters, hiragana, katakana, punctuation, and other symbols.

Nonmodal encoding
: An encoding method that does not use special sequences of characters to switch between one- and two-byte modes. *See also fixed-length encoding and modal encoding.*

Nonprinting character
: A character that makes no printable marks on an output device. These include control characters and white space characters, such as a space or tab character.

Normalization
: A standard Unicode process that serves to convert equivalent Unicode code points or sequences into a unified form. There are four Normalization forms: NFC, NFD, NFKC, and NFKD. *See also Canonical Equivalent, NFC, NFD, NFKC, and NFKD.*

North Korea
: *See DPRK.*

Notation
: A method of representing units or other values. In the world of computers, the most commonly used notations are binary (base two), octal (base eight), decimal (base ten), and hexadecimal (base sixteen). *See also binary, decimal, hexadecimal, and octal.*

NTT
: 日本電信電話 (*nippon denshin denwa*). Nippon Telegraph and Telephone.

NTT Kanji
: The name of the Japanese character set and encoding as developed by NTT. *See NTT.*

Numeral
: 数字 (*sūji*). The printed digits ranging from 0 (zero) through 9 (nine).

Numeric Character Reference
: An SGML-derived notation that is recognized by HTML and XML, and serves to specify characters by their code point. For example, the eight-character string `&#x4E00;` is the NCR for the ideograph 一 (U+4E00). *See also Character Entity Reference.*

OASYS
: Office Automation SYStem. The personal word-processor series developed by Fujitsu.

OCR
: 光学的文字認識 (*kōgakuteki moji ninshiki*) in Japanese. Optical Character Recognition. A device that can scan, recognize, and convert printed shapes into meaningful units, such as characters.

Octal
: 八进制/八進制 (*bājìnzhì*) in Chinese, 八進法 (*hasshinhō*) in Japanese, and 팔진법/八進法 (*paljinbeop*) in Korean. Base eight. A numeric notation that uses eight possible values, ranging from 0 to 7.

Octet
: An array of eight bits represented as a single unit (a byte). *See also byte.*

Old-JIS
: 旧JIS (*kyū JIS*). A common name given to the JIS C 6226-1978 character set standard. *See also JIS C 6226-1978.*

Old-JIS array
: *See JIS array.*

On reading
: 音読み (*on yomi*). The Japanese name given to the approximated Chinese reading for a kanji.

Open source
: The name given to the software development methodology or design approach that offers practical accessibility to a software's source code.

OpenType
: A cross-platform and Unicode-*savvy* outline font format jointly developed by Adobe Systems and Microsoft that equally supports PostScript and TrueType outlines.

Operating system
: *See OS.*

Orthography
: 正書法 (*seishohō*) in Japanese. A linguistic term that refers to the writing system of a language.

OS
: Operating System. The software that drives the hardware associated with a computer system.

OSF
: Open Software Foundation.

OTF
: Out-the-front, which refers to a mechanism, usually spring-powered, for deploying a knife blade through an opening in the handle. Also, OpenType Font. The filename extension used for OpenType fonts.

Outline font
: A font whose characters are described mathematically in terms of lines and curves. Outline fonts are often referred to as scalable fonts, because they can be scan converted on demand to bitmaps of any desired size and orientation.

P
: 撇 (*piě*) in Chinese. An abbreviation (the first letter of *piě*) standing for the third of five basic stroke types used as building blocks for ideographs. Represents diagonal strokes that are written right to left. *See also stroke.*

Parametric font
: A font whose shape is described as a series of vectors. This type of font format has scalable properties, but is not as high quality as outline fonts. *See also bitmapped font and outline font.*

Particle
: 助詞 (*joshi*) in Japanese. Grammatical markers used in the Japanese language. They are equivalent to prepositions in English, but unlike English, they come after the noun or phrase they modify. Particles are sometimes called postpositions.

PB
: Petabyte. 1,125,899,906,842,624 or $1{,}024^5$ bytes.

PC
: パソコン (*pasokon*) in Japanese. Personal Computer. In the past, referred to machines that ran MS-DOS. Now, refers to computers for personal use. Also, Politically Correct and Purity Control.

PC-VAN
: Personal Computer Value Added Network. A Japanese ISP based in Japan.

PCF
: Portable Compiled Format. A binary representation of BDF files for use under the X Window System. *See also BDF.*

PDF
: Portable Document Format. The document format that is generated by Adobe Acrobat technology.

Pen input
: ペン入力 (*pen nyūryoku*) in Japanese. An input method that allows the user to enter text and commands with a pen (or stylus) onto a tablet. OCR technology is often used in the process of interpreting handwritten text.

Perl
: An acronym for Pathologically Eclectic Rubbish Lister or Practical Extraction and Report Language. Which one is correct depends on the phase of the moon or the alignment of stars. An ideal programming language for performing complex text-processing tasks.

Salient features include no built-in limits, regular expressions, associative arrays, and C-like syntax. Developed by Larry Wall.

Phonetic ideograph
形声文字/形聲文字 (*xíngshēng wénzì*) in Chinese, 形声文字 (*keisei moji*) in Japanese, and 형성문자/形聲文字 (*hyeongseong munja*) in Korean. An ideograph constructed from at least two radical-like elements. One element is used for its reading, and the other used for its meaning. Together they form a unique character.

PHP
Personal Home Page. The name of a scripting language that was originally designed for producing dynamic web pages.

Pictograph
象形文字 (*xiàngxíng wénzì*) in Chinese, 象形文字 (*shōkei moji*) in Japanese, and 상형문자/象形文字 (*sanghyeong munja*) in Korean. A character whose shape reflects the shape of the object that it represents. An example of such an ideograph is 山, which means "mountain."

Pinyin
拼音 (*pīnyīn*). The most common Latin-based transliteration method for Chinese text.

Plan 9
The name of a classic science fiction film. Also, the multilingual Unix operating system under development at AT&T Bell Laboratories.

Plane
A collection of code points that are grouped by row and cell, and typically represent a two-byte encoding. The term "plane" is significant for a character set standard only if there is more than one of them.

Plane-Row-Cell
An encoding-independent notation for indexing characters in multiple-plane CJKV character set standards, such as CNS 11643-2007 and JIS X 0213:2004. Although Unicode is composed of 17 planes, Unicode Scalar Values are the preferred notation. *See also Unicode Scalar Value.*

PM
Post Meridian or PageMaker.

Point
磅 (*bàng*) or 点/點 (*diǎn*) in Chinese, ポイント (*pointo*) in Japanese, and 포인트 (*pointeu*) in Korean. A unit of measure used in typography. A DTP point is exactly 1/72.27 of an inch. PostScript, on the other hand, rounds this figure to 1/72 of an inch.

POSIX
Portable Operating System Interface.

Postposition
See particle.

PostScript
ポストスクリプト (*posutosukuriputo*) in Japanese. The page description language developed by Adobe Systems.

PPP
Point-to-Point Protocol.

PRC
中华人民共和国 (*zhōnghuá rénmín gònghé guó*). People's Republic of China. The official name for China.

Printable character
A character that makes some sort of mark on an output devices. Also called a graphic character.

Pronghorn
See antelope.

Pseudo ruby
擬似ルビ (*giji rubi*) in Japanese. Small characters, usually kanji or Latin characters, that appear above normal-size characters, and serve to annotate them with a reading or meaning. Similar to ruby characters. *See also ruby.*

PUA
Private Use Area. The preferred way in which to refer to the 137,468 code points of Unicode that are reserved for user-defined characters. *See also user-defined character.*

Push-in-first line breaking
追い込み禁則処理 (*oikomi kinsoku shori*) in Japanese. A method of moving characters up to the previous line in order to prevent

prohibited characters from ending or beginning a line. Also known as wrap-up line breaking. *See also line breaking, push-out-first line breaking, and push-out-only line breaking.*

Push-out-first line breaking
: 追い出し禁則処理 (*oidashi kinsoku shori*) in Japanese. A method of moving characters down to the next line in order to prevent prohibited characters from ending or beginning a line. Also known as wrap-down line breaking. *See also line breaking, push-in-first line breaking, and push-out-only line breaking.*

Push-out-only line breaking
: ぶら下がり禁則処理 (*burasagari kinsoku shori*) in Japanese. A method of moving characters up to the previous line in order to prevent prohibited characters from ending or beginning a line. The character or characters that are moved appear to dangle or hang outside of the right margin. Also known as dangling line breaking and hanging line breaking. *See also line breaking, push-in-first line breaking, and push-out-first line breaking.*

Python
: The name of a popular scripting language.

Q
: 级数/級數 (*jíshù*) in Chinese, 級数 (*kyūsū*) in Japanese, and 급수/級數 (*geupsu*) in Korean. Unit of typographic measurement equivalent to 0.25mm. Its correct usage is strictly for type size. Also, the name of an omnipotent being from the contemporary Star Trek series.

Quadratic spline curve
: The type of curve used for representing character shape contours in the TrueType font format.

Quốc ngữ
: 國語. The Latin-based script used in contemporary Vietnam. A Romanization system.

Quoted-Printable
: A method of preserving non-ASCII characters to ensure reliable transmission.

QWERTY array
: The most common keyboard in use today. Its name comes from the first six keys that have 26 letters of the Latin alphabet imprinted on them.

Radical
: 部首 (*bùshǒu*) in Chinese, 部首 (*bushu*) in Japanese, and 부수/部首 (*busu*) in Korean. The building blocks of ideographs of which the most common set contains 214 radicals. Many CJKV character set standards arrange ideographs by radical. For example, the hanzi of GB 2312-80 Level 2 are arranged by radical. Radicals are subcomposed of strokes. *See also stroke.*

RAM
: Random Access Memory.

Regex
: Short for regular expression. *See regular expression.*

Regexp
: Short for regular expression. *See regular expression.*

Regular expression
: "Have a nice day" is one example. Also, a powerful mechanism for searching, ripping apart, shredding, or otherwise manipulating text (or sometimes, binary) data. Software tools such as awk, GNU Emacs, Perl, sed, and vi include regular expression engines. JPerl, Mule, and lookup provide Japanese-capable regular expression engines.

RFC
: Request For Comments. The designation given to the thousands of documents that describe the inner workings of the Internet.

RKSJ
: Roman, (half-width) Katakana, and Shift-JIS. An encoding used by PostScript Japanese fonts.

RMK
: Randall Made Knives.

ROC
: 中華民國 (*zhōnghuá mínguó*). Republic of China. The official name for Taiwan. *See also Taiwan.*

ROK
: 대한 민국/大韓民國 (*daehan minguk*). Republic of Korea. The official name for South Korea.

ROM
: Read Only Memory.

Roman character
: *See Latin character.*

Romanization
: A system of using Latin characters as the primary script for a language. Vietnamese Quốc ngữ is an example of a Romanization system.

Row
: 区 (*qū*) in Chinese, 区 (*ku*) in Japanese, and 행/行 (*haeng*) in Korean. In a two-byte encoding, row refers to the first byte. In a two-dimensional matrix, row represents the values along the vertical axis. *See also cell and Row-Cell.*

Row-Cell
: 区位 (*qūwèi*) in Chinese, 区点 (*kuten*) in Japanese, and 행렬/行列 (*haengryeol*) in Korean. An encoding-independent notation for indexing characters in single-plane CJKV character set standards (and the vendor character sets derived from them). *See also Plane-Row-Cell.*

RRK
: Revised Romanization of Korean. The official method for transliterating or transcribing Korean text through the use of Latin characters.

Ruby
: ルビ (*rubi*). Small characters, usually kana, that appear above normal-size characters, and annotate them with a reading or meaning. Some folks prefer to spell it "rubi," but that conflicts with its true history. Also, the name of my daughter (瑠美).

Ryobi
: リョービ (*ryōbi*). A major Japanese type foundry. *See also Fontworks, Morisawa, and Shaken.*

S
: 竖/豎 (*shù*) in Chinese. An abbreviation (the first letter of *shù*) standing for the second of five basic stroke types used as building blocks for ideographs. Represents vertical strokes. *See also stroke.*

S32S
: Supercalifragilisticexpialidocious.

Sans serif
: French for without serifs. Sans serif characters do not have little feet on them. Helvetica (this is Helvetica) is a widely used sans serif typeface. *See also serif.*

SAR
: Special Administrative Region.

SBCS
: Single-Byte Character Set.

SBCS-Host
: A single-byte character set that is encoded according to EBCDIC. *See EBCDIC.*

SBCS-PC
: A single-byte character set that is encoded according to ASCII. *See ASCII.*

SCIM
: Smart Common Input Method.

Script
: Unicode defines "script" as a collection of letters and other written signs used to represent textual information in one or more writing systems. *See also writing system.*

SDC
: *See system-defined character.*

SDK
: Software Development Kit. A collection of software and documentation that is specifically designed to aid others to develop software for a particular technology.

Serif
: Characters that have little feet to act as guide marks. Derives all the way from days when letters were carved in stone—the serifs were added to provide even height, and the overall style improves legibility. Garamond, which is used as the standard textface in this book, is an example of a serif typeface. *See also sans serif.*

SG
: The two-letter country code for Singapore.

SGI
: Silicon Graphics Incorporated.

SGML
: Standard Generalized Markup Language. Defined in ISO 8879:1986. HTML and XML are derivatives of SGML. *See also HTML and XML.*

Shaken
: 写研 (*shaken*). A major Japanese type foundry. *See also Fontworks, Morisawa, and Ryobi.*

Shift-JIS
: The most common encoding method used on Japanese PCs. So named from how the first byte range of two-byte characters shifts around the encoding range of single-byte half-width katakana. Also called MS Kanji.

Shifting characters
: A sequence of one or more characters that are often used to shift between one- and two-byte modes. *See also escape sequence.*

SI
: Shift-In. 0x0F or U+000F. A control character that often serves as a shifting character in ISO-2022 encoding.

Simple ideograph
: 指事文字 (*zhǐshì wénzì*) in Chinese, 指事文字 (*shiji moji*) in Japanese, and 지사문자/指事文字 (*jisa munja*) in Korean. An ideograph that represents an abstract shape. Examples include characters such as 上 ("up") and 下 ("down").

Simplified ideograph
: A standardized ideograph variant that is written with fewer strokes. *See traditional ideograph.*

SING
: What some people do while showering. Also, Smart INdependent Glyphlet. The name of Adobe Systems' gaiji solution that implements gaiji as single-glyph, font-like files called *glyphlets*, which are designed to be sticky to the documents that use them, thus solving the portability issue that plagues legacy gaiji solutions.

Singapore
: 新加坡 (*xīnjiāpō*). A Chinese locale, located in Southeast Asia, where simplified ideographs are used.

SJIS
: An abbreviation for Shift-JIS. *See Shift-JIS.*

SJS
: Another abbreviation for Shift-JIS. *See Shift-JIS.*

SJTAC
: Satin Jack TAC.

SK72
: The older of Shaken's proprietary character sets.

SK78
: The newer of Shaken's proprietary character sets.

SKIP
: System of Kanji Indexing by Patterns. A method for indexing kanji that divides them geometrically, thus allowing one to find any kanji in less than 30 seconds. It is used by kanji dictionaries written by Jack Halpern, and is implemented in Jim Breen's online kanji dictionaries.

SMI
: エスエムアイ (*esu emu ai*). Sumitomo Metal Industries (住友金属工業 *sumitomo kinzoku kōgyō*). The original developer of Canon EDICOLOR.

SMTP
: Simple Mail Transfer Protocol.

SNF
: Server Natural Format. A binary representation of BDF files for use under the X Window System. *See also BDF.*

SO
: Shift-Out. 0x0E or U+000E. A control character that often serves as a shifting character in ISO-2022 encoding.

Solaris
: The name of Sun Microsystems' operating system.

Song
: 宋 (*sòng*) or 宋体 (*sòngtǐ*). The Chinese serif typeface style. Also, a musical score. *See also serif.*

Sony
: Standard Oil of New York. Er, uh, hmm, I mean a famous Japanese electronics company.

South Korea
: *See ROK.*

SS2
: Single Shift 2. A special character or character sequence (0x8E in EUC, and the sequence <1B 4E> in ISO-2022) used as a prefix to characters beyond the standard character set (invokes code set 2 in EUC, and extended character sets in ISO-2022).

SS3
: Single Shift 3. A special character or character sequence (0x8F in EUC, and the sequence <1B 4F> in ISO-2022) used as a prefix to characters beyond the standard character set (invokes code set 3 in EUC, and extended character sets in ISO-2022).

SSC
: *See system-specific character.*

STAMEQ
: Standards, Metrology and Quality Control. An English translation of TCVN. *See TCVN.*

Stroke
: 画/畫 (*huà*) in Chinese, 画 (*kaku*) in Japanese, and 획/畫 (*hoek*) in Korean. The basic building blocks of radicals and ideographs. A single stroke is traditionally defined as an element drawn with a single uninterrupted movement of the writing implement. Strokes have shapes that represent straight lines, curves, and angles. There are five basic stroke types, each with several subtypes: 横 (*héng*), 竖/竪 (*shù*), 撇 (*piě*), 点/點 (*diǎn*), and 折 (*zhé*). These basic stroke types are abbreviated in Unicode's CJK Strokes block character names as H, S, P, D, and Z, respectively. Unicode encodes 36 CJK Strokes from U+31C0 through U+31E3, which were defined by the IRG, stemming from CDL work. *See also CDL, D, H, IRG, P, S, and Z.*

Supplemental kanji
: *See hojo kanji.*

Syllabary
: A script whose characters are composed of syllables. Hiragana and katakana are examples of syllabaries. *See also syllable.*

Syllable
: A sound sequence consisting of a consonant plus vowel.

Synchronic
: A linguistics term that is used to refer to linguistic changes that exist during the same period.

System-defined character
: A character that is considered standard across OSes.

System-specific character
: A character that, while considered standard on a given OS, is specific to that OS. That is, it may not be generally available across all OSes.

T1C
: Adobe Type 1 Coprocessor. A now-obsolete computer chip designed by Adobe Systems that significantly reduced the time necessary to render glyphs to the screen or printer.

Table-driven conversion
: A type of conversion that uses mapping tables for converting objects. *See also algorithmic conversion.*

Taiwan
: 臺灣 (*táiwān*). One of the Chinese locales in which traditional forms of hanzi are still used. *See also ROC.*

TB
: Terabyte. 1,099,511,627,776 or $1{,}024^4$ bytes.

TBCS
: Triple-Byte Character Set. A character set whose characters are encoded with three bytes.

TBCS-EUC
: A Triple-Byte Character Set encoded according to the specification of EUC.

TCVN
: Tiêu Chuẩn Việt Nam.

TCVN 5712:1993
: A Vietnamese character set standard that defines various encodings for the Latin-based Quốc ngữ script.

TCVN 5773:1993
: A Vietnamese character set standard that enumerates 2,357 chữ Nôm characters.

TCVN 6056:1995
: A Vietnamese character set standard that enumerates 3,311 chữ Hán characters.

TCVN-Roman
: The Vietnamese equivalent of the ASCII character set and encoding, with additional characters necessary for the Latin-based Quốc ngữ script. The name of the standard that defines this character set is called TCVN 5712:1993. *See TCVN 5712:1993.*

TES
: *See Transfer Encoding Syntax.*

TEX
: A popular typesetting language developed by Donald Knuth for which CJKV-capable versions exist.

Thumb-shift array
: 親指シフト配列 (*oyayubi shifuto hairetsu*). The Japanese keyboard array developed by Fujitsu.

TIMTOWTDI
: There Is More Than One Way To Do It. The Perl slogan.

TISN
: Todai (東大 *tōdai*; University of Tokyo) International Science Network.

TLD
: Top-Level Domain. *See also ccTLD and gTLD.*

Tōyō Kanji
: 当用漢字 (*tōyō kanji*). The 1,850 kanji designated by the Japanese government as the ones to be used in public documents such as newspapers. Superseded by Jōyō Kanji in 1981. *See Jōyō Kanji.*

Traditional ideograph
: Refers to the original, sometimes complex, shapes of ideographs. Sometimes thought of as the opposite of simplified ideograph. 國 is an example of a traditional ideograph. Its simplified counterpart, as used in Japan and China, is 国. Korea and Taiwan use traditional ideographs. Japan and China, in general, use simplified forms. *See also simplified ideograph.*

Transfer Encoding Syntax
: A transformation applied to an encoding to allow it to be safely transmitted. Examples include Base64, BinHex, Quoted-Printable, and uuencoding.

Transliteration
: A method for transcribing a script, which can serve as a prounounciation aid for those who are not familiar with the primary scripts of the language.

TRON
: トロン (*toron*). The name of a 1982 Disney science fiction film. Also, The Real-time Operating system Nucleus. A set of interfaces and design guidelines for creating an OS kernel, originally architected in 1984 by Ken Sakamura (坂村健 *sakamura ken*), in order to address the interface between humans and computers. Variations include BTRON (*Business TRON*), CTRON (*Central and Communications TRON*), eTRON (*Entity and Economy TRON*), ITRON (*Industrial TRON*), JTRON (*Java TRON*), MTRON (*Macro TRON*), and μITRON (*Micro Industrial TRON*).

TrueType
: An outline font format jointly developed by Apple and Microsoft.

TrueType Collection
: A special type of TrueType font that includes multiple font instances, and represents an efficient way of encapsulating what would otherwise be multiple font files that share a large number of glyphs.

TTF
: TrueType Font. The filename extension used for TrueType fonts.

TTC
: TrueType Collection. The filename extension used for TrueType Collections.

TW
: The two-letter country code for Taiwan.

TWICS
: Two-way Information Communication System. The name of an ISP based in Japan.

TWNIC
: Taiwan Network Information Center.

Two-stroke input method
: 二ストローク入力方式(*ni sutorōku nyūryoku hōshiki*). A Japanese input method that associates two key-strokes per kanji. In the case of input by association, the two keys have some sort of relationship to the kanji, usually by reading or meaning. There is also input by unassociation, which arbitrarily associates two key-strokes per kanji.

Type 0 font
: Adobe Systems' composite font format. A Type 0 font contains other fonts in a hierarchical fashion, providing access to huge character sets, such as those used in Japan. A CIDFont is a Type 0 font that has an FMapType value of 9.

Type 1 font
: Adobe Systems' format for describing outlines or scalable fonts. Type 1 fonts use a very special and limited subset of PostScript, optimized for compactness and speed. *See also ISO 9541:1991.*

Type 2 charstring
: A lossless compaction of the Type 1 charstring. *See also Type 1 font.*

Type 3 font
: A user-defined PostScript font. Type 3 fonts can use all of the PostScript language to obtain effects—such as grayscale—not available to Type 1 fonts.

Type 4 font
: Adobe Systems' proprietary font format. Provides no benefits over Type 1 in PostScript Level 2 and beyond.

Type 5 font
: Adobe Systems' ROM-based font format.

Type 9 font
: A CIDFont that uses Type 1 glyph procedures. Equivalent to CIDFontType 0.

Type 10 font
: A CIDFont that uses PostScript BuildGlyph procedures. Equivalent to CIDFontType 1.

Type 11 font
: A CIDFont that uses TrueType glyph procedures. Equivalent to CIDFontType 2.

Type 14 font
: The Chameleon ROM font format.

Type 42 font
: Adobe Systems' font format that provides a wrapper for a TrueType font. This allows a TrueType font to be used in much the same way as a Type 1 font. *See also Type 1 font.*

Typeface
: A distinctive design for a set of visually related symbols that can come in a variety of weights (from light to bold) and styles (upright and italic). Examples include Minion, Myriad, and Kozuka Mincho.

UCS
: Universal Character Set. Refers to Unicode and ISO 10646.

UCS-2
: A now-obsolete, fixed-length, 2-byte (16-bit) encoding form for Unicode and ISO 10646. Replaced by the UTF-16 encoding form. *See also UTF-16.*

UCS-4
: A now-obsolete, fixed-length, 4-byte (31-bit) encoding form for Unicode and ISO 10646. Replaced by the UTF-32 encoding form. *See also UTF-32.*

UDC
: *See user-defined character.*

UHC
: *See Unified Hangul Code.*

UI
: Unix International or User Interface.

UJIS
: Short for Unix*ized* JIS. Identical to EUC-JP. *See EUC-JP.*

Unicode
: The name of the international character set and encoding developed by the members of the Unicode Consortium. Joe Becker, the father of Unicode, wrote on 12/04/1991 that Unicode is an acronym for *Unique Nice International Consistent Official Desirable Encoding*.

Unicode Scalar Value
: A notation for specifying a Unicode code point without specify an encoding form. This notation makes use of a U+ prefix to distinguish it from other values.

Unicode Variation Sequence
: A generic way of referring to Unicode sequences that use a Variation Selector (VS). An Ideographic Variation Sequence (IVS) is a specific type of UVS. *See also Ideographic Variation Sequence and Variation Selector.*

Unified Hangul Code
: A character set equivalent to that of Johab, but whose encoding is backward-compatible with EUC-KR. *See EUC-KR and Johab.*

Unix
: The name of the operating system that runs on most workstations.

URL
: Uniform Resource Locator. The standard by which the locations of files on the Web are described.

URO
: Unified Repertoire and Ordering. The name of the original block of CJK Unified Ideographs in Unicode and ISO 10646. In its original form, it included 20,902 characters. Since then, 38 characters have been appended to the URO.

US
: The two-letter country code for The United States of America.

User-defined character
: A character that is added to a character set by an end user. Sometimes confused with a character that is not considered standard on most operating systems or environments. *See also system-specific character.*

USLP
: Unix System Laboratories Pacific.

USV
: *See Unicode Scalar Value.*

UTC
: Unicode Technical Committee.

UTF
: Unicode (or UCS) Transformation Format. A series of encoding forms for Unicode and ISO 10646. *See also UTF-8, UTF-16, and UTF-32.*

UTF-2
: Obsolete. A UTF defined by AT&T Bell Labs (Plan 9) and X/Open for encoding Unicode text as a stream of bytes. Also called FSS-UTF (*File System Safe UTF*), and now referred to as UTF-8. *See also Plan 9 and UTF-8.*

UTF-7
: A variation of Base64 encoding that transforms Unicode encodings—UCS-2, UCS-4, and UTF-16—into a form that can be safely transmitted through 7-bit pathways. Most ASCII characters represent themselves under this encoding.

UTF-8
: The 8-bit encoding form (and encoding scheme) of Unicode and ISO 10646 that uses up to four code units to represent a character. Its original definition, which accommodated UCS-4 encoding, used a variable-length one- through six-byte encoding. Once called UTF-2 and FFS-UTF (File System Safe UTF).

UTF-16
: The 16-bit encoding form of Unicode and ISO 10646. Uses 16-bit code units. Non-BMP characters are encoded through the use of High and Low Surrogates. Requires the BOM to specify byte order. *See also BOM.*

UTF-16BE
: A Unicode encoding scheme that represents the big-endian version of the UTF-16 encoding form. Does not require the BOM. *See also BOM and UTF-16.*

UTF-16LE
: A Unicode encoding scheme that represents the little-endian version of the UTF-16 encoding form. Does not require the BOM. *See also BOM and UTF-16.*

UTF-32
: The 32-bit encoding form of Unicode and ISO 10646. Uses 32-bit code units. Requires the BOM to specify byte order. *See also BOM.*

UTF-32BE
: A Unicode encoding scheme that represents the big-endian version of the UTF-32 encoding form. Does not require the BOM. *See also BOM and UTF-32.*

UTF-32LE
: A Unicode encoding scheme that represents the little-endian version of the UTF-32 encoding form. Does not require the BOM. *See also BOM and UTF-32.*

UTF-FSS
: *See FSS-UTF.*

UUCP
: Unix-to-Unix Copy.

Uudecode
: A Unix utility for decoding a file encoded by *uuencode*. *See also uuencode.*

Uuencode
: A Unix utility for encoding a file (usually a binary file) such that it can pass through networks with only seven-bit paths. Decoding is performed by *uudecode*. *See also uudecode.*

UVS
: *See Unicode Variation Sequence.*

Variation Selector
: The second component of an IVS or UVS. *See also IVS and UVS.*

Vector font
: *See parametric font.*

Vendor-defined character
: *See gaiji.*

vi
: The two-letter language code for Vietnamese. Also, the name of a popular Unix-based text editor.

Vietnam
: Việt Nam. Also written as two words: Viet Nam. The locale where the Quốc ngữ (Latin characters), chữ Hán (ideographs of Chinese origin), and chữ Nôm (Vietnamese-made ideographs) scripts are used.

VIQR
: VIetnamese Quoted-Readable specification.

VISCII
: Vietnamese Standard Code for Information Interchange.

VN
: The two-letter country code for Vietnam.

VNNIC
: Vietnam Internet Network Information Center.

VOLT
: Visual OpenType Layout Tool.

VS
: *See Variation Selector.*

VSCII
: Vietnamese Standard Code for Information Interchange.

Voice input
: 音声入力 (*onsei nyūryoku*) in Japanese. An input method that is driven by the human voice. Such devices must usually be trained to understand the user's voice.

Wade-Giles
: 韋氏 (*wéishì*). Another Latin-based transliteration system for Chinese text.

Wansung
: 완성/完成 (*wanseong*). Means "precomposing" in Korean. Another name for EUC-KR encoding in which each hangul is encoded as a single entity. Considered to be the opposite of Johab. *See also Johab.*

Ward
: *See row.*

Web
: A short form for World Wide Web. *See WWW.*

Whitespace
: Characters that produce empty space, such as the "space" character or the "tab" character.

Wide character
: A character that consists of a larger than normal byte. A byte typically consists of seven or eight bits. A character represented by 16 or 32 bits is considered a wide character.

Windows
: The somewhat generic name of Microsoft's OS. The three most recent instantiations are called Windows 2000, Windows XP, and Windows Vista.

Word processor
: ワードプロセッサ (*wādopurosessa*) or ワープロ (*wāpuro*) in Japanese. A text-processing application that manipulates text in such a way that it is possible to include multiple fonts in a single document. Sufficient formatting capabilities are also quite common.

Wrap-down line breaking
: *See push-out-first line breaking.*

Wrap-up line breaking
: *See push-in-first line breaking.*

Writing system
: Unicode defines this as a set of rules for using one or more scripts to write a particular language. *See also script.*

Wubi
: 五笔 (*wǔbǐ*), short for 五笔输入法 (*wǔbǐ shūrùfǎ*). A popular stroke-based Chinese input method.

WWW
: World Wide Web or simply "web." Those with slow modems or slow Internet connections sometimes refer to it as World Wide Wait.

WYBIWYG
: What You Buy Is What You Get.

WYSIWYG
: What You See Is What You Get.

X-Modem
: A once-popular file transfer protocol.

X Window System
: The name of a very popular Unix windowing system developed at MIT (Massachusetts Institute of Technology). The latest release is called X11R7.

XCCS
: Xerox Character Code Standard.

XHTML
: Extended HTML.

XKP
: Extended Kanji Processing. An initiative for handling external characters in the context of Windows NT.

XML
: Extensible Markup Language. An implementation of SGML for the Web whose strength is the encapsulation of data. *See also HTML and SGML.*

XPG4
: X/Open Portability Guide issue 4.

Y-Modem
: A once-popular file transfer protocol.

YMMV
: Your Mileage May Vary.

Z
: 折 (*zhé*) in Chinese. An abbreviation (the first letter of *zhé*) standing for the fifth of five basic stroke types used as building blocks for ideographs. Represents angled strokes. *See also stroke.*

Z-Modem
: A once-popular file transfer protocol.

Zenkaku
: 全角 (*zenkaku*). Analogous to full-width. *See full-width.*

zh
: The two-letter language code for Chinese.

Zhuyin
: 注音 (*zhùyīn*), short for 注音符号 (*zhùyīn fúhào*). The name of the symbols used to represent standard readings in Chinese. Also known as *bopomofo* (named from its first four sounds: *b*, *p*, *m*, and *f*).

APPENDIX E
Vendor Character Set Standards

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file.[*] As new material becomes available, the PDF file will be updated accordingly.

[*] *http://examples.oreilly.com/9780596514471/cjkvip2e-appE.pdf*

APPENDIX F
Vendor Encoding Methods

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appF.pdf*

APPENDIX G
Chinese Character Sets—China

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appG.pdf*

APPENDIX H
Chinese Character Sets—Taiwan

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appH.pdf*

APPENDIX I
Chinese Character Sets—Hong Kong

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appI.pdf*

APPENDIX J
Japanese Character Sets

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appJ.pdf*

APPENDIX K
Korean Character Sets

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appK.pdf*

APPENDIX L
Vietnamese Character Sets

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file. As new material becomes available, the PDF file will be updated accordingly.*

* *http://examples.oreilly.com/9780596514471/cjkvip2e-appL.pdf*

APPENDIX M
Miscellaneous Character Sets

This appendix is not included in the printed version of this book, and is instead available as a downloadable and printable PDF file.[*] As new material becomes available, the PDF file will be updated accordingly.

[*] *http://examples.oreilly.com/9780596514471/cjkvip2e-appM.pdf*

Bibliography

This bibliography provides a listing of some potentially useful reference works. They are separated into the following categories: books (subcategorized into languages), character dictionaries, standards, periodicals, papers and articles, and RFCs. While all of these references are useful to some extent, it is by no means necessary to obtain them all (in fact, some may be out of print). I have included ISBNs, ISSNs, and part numbers so that ordering these references becomes an easier (or, at least, somewhat possible) task.

Books

The following listings have been broken down into sections for different languages. This will give you a better idea of whether such references would be of value to you. And whether you'll be able to read them!

Books in English

Adobe Systems Incorporated. *Adobe Type 1 Font Format.* Version 1.1. Addison-Wesley. 1990. ISBN 0-201-57044-0. Also available in PDF.

———. *PostScript Language Tutorial and Cookbook.* Addison-Wesley. 1985. ISBN 0-201-10179-3.

———. *PostScript Language Program Design.* Addison-Wesley. 1988. ISBN 0-201-14396-8.

———. *PostScript Language Reference.* Third Edition. Addison-Wesley. 1999. ISBN 0-201-37922-8. Also available in PDF.

———. *PDF Reference.* Sixth Edition. Adobe Portable Document Format Version 1.7. 2006. Available only in PDF.

American Electronics Association. *Software Partners: The Directory of Japanese Software Distributors.* 1992.

Apple Computer. *Guide to Macintosh Software Localization.* Addison-Wesley. 1992. ISBN 0-201-60856-1.

———. *Inside Macintosh: QuickDraw GX Typography.* Addison-Wesley. 1994. ISBN 0-201-40679-9.

Bienz, Tim & Richard Cohen. *Portable Document Format Reference Manual.* Addison-Wesley. 1993. ISBN 0-201-62628-4.

Bringhurst, Robert. *The Elements of Typographic Style.* Version 3.1. Hartley & Marks. 2005. ISBN 0-88179-206-5 (paper) or ISBN 0-88179-205-8 (cloth).

———. *The Solid Form of Language.* Gaspereau Press. 2004. ISBN 0-894031-88-1.

Cabarga, Leslie. *Learn FontLab Fast.* Iconoclassics Publishing. 2004. ISBN 0-9657628-5-8.

Cameron, Debra et al. *Learning GNU Emacs.* Third Edition. O'Reilly Media, Incorporated. 2004. ISBN 0-596-00648-9.

Christiansen, Tom & Nathan Torkington. *Perl Cookbook.* Second Edition. O'Reilly Media, Incorporated. 2003. ISBN 0-596-00313-7.

Clews, John. *Language Automation Worldwide: The Development of Character Set Standards.* Sesame Computer Projects. 1988. ISBN 1-870095-01-4.

Conner, Kiersten & Ed Krol. *The Whole Internet: The Next Generation.* O'Reilly Media, Incorporated. 1999. ISBN 1-56592-428-2.

Connolly, Dan, editor. *XML: Principles, Tools, and Techniques.* World Wide Web Journal, Volume 2, Number 4, Winter 1997. O'Reilly Media, Incorporated. 1997. ISBN 1-56592-349-9.

Daniels, Peter T. & William Bright, editors. *The World's Writing Systems.* Oxford University Press. 1996. ISBN 0-19-507993-0.

Daub, Edward E. et al. *Basic Technical Japanese.* The University of Wisconsin Press. 1990. ISBN 0-299-12730-3.

———. *Comprehending Technical Japanese.* The University of Wisconsin Press. 1975. ISBN 0-299-06680-0.

Felici, James. *The Complete Manual of Typography.* Adobe Press. 2003. ISBN 0-321-12730-7.

Flanagan, David. *Java in a Nutshell.* Fifth Edition. O'Reilly Media, Incorporated. 2005. ISBN 0-596-00773-6.

Flanagan, David & Yukihiro Matsumoto. *The Ruby Programming Language.* O'Reilly Media, Incorporated. 2008. ISBN 0-596-51617-7.

Frey, Donnalyn & Rick Adams. *!%@:: A Directory of Electronic Mail Addressing & Networks.* Fourth Edition. O'Reilly Media, Incorporated. 1994. ISBN 1-56592-046-5.

Friedl, Jeffrey E.F. *Mastering Regular Expressions.* Third Edition. O'Reilly Media, Incorporated. 2006. ISBN 0-596-52812-4.

Gillam, Richard. *Unicode Demystified.* Addison-Wesley. 2003. ISBN 0-201-70052-2.

Gottlieb, Nanette. *Kanji Politics: Language Policy and Japanese Script*. Kegan Paul International. 1995. ISBN 0-7103-0512-5.

Gourley, David et al. *HTTP: The Definitive Guide*. O'Reilly Media, Incorporated. 2002. ISBN 1-56592-509-2.

Graham, Tony. *Unicode: A Primer*. M&T Books. 2000. ISBN 0-7645-4625-2.

Gralla, Preston. *Windows Vista in a Nutshell*. O'Reilly Media, Incorporated. 2006. ISBN 0-596-52707-1.

Guelich, Scott et al. *CGI Programming with Perl*. Second Edition. O'Reilly Media, Incorporated. 2000. ISBN 1-56592-419-3.

Haralambous, Yannis. *Fonts & Encodings*. O'Reilly Media, Incorporated. 2007. ISBN 0-596-10242-9.

Harold, Elliotte Rusty & W. Scott Means. *XML in a Nutshell*. Third Edition. O'Reilly Media, Incorporated. 2004. ISBN 0-596-00764-7.

Hawking, Stephen. *A Brief History of Time*. Bantam Books. 1988. ISBN 0-533-34614-8.

Heisig, James. *Remembering the Kanji I: A Complete Course on How Not to Forget the Meaning and Writing of Japanese Characters*. Third Edition. Japan Publications Trading Company. 1985. ISBN 0-87040-739-2.

———. *Remembering the Kanji II: A Systematic Guide to Reading Japanese Characters*. Japan Publications Trading Company. 1987. ISBN 0-87040-748-1.

Heisig, James with Tanya Sienko. *Remembering the Kanji III: Writing and Reading Japanese Characters for Upper-Level Proficiency*. Japan Publications Trading Company. 1994. ISBN 0-87040-931-X.

Hensch, Kurt. *IBM History of Far Eastern Languages in Computing*. 2004.

Henshall, Kenneth. *A Guide to Remembering Japanese Characters*. Charles E. Tuttle Company. 1995. ISBN 0-8048-2038-4.

Hewlett-Packard. *Japanese Input Method Guide for NLIO 8.0*. 1991. Hewlett-Packard part number B2200-90003.

———. *Native Language I/O Access User's Guide*. 1991. Hewlett-Packard part number B2200-90001 (Japanese) and B2200-90005 (English).

———. *Native Language I/O System Administrator's Guide*. 1991. Hewlett-Packard part number B2200-90002 (Japanese) and B2200-90006 (Japanese).

———. *Kanji Code Book*. 1989. Hewlett-Packard part number 98861-90003.

Huang, Jack & Timothy Huang. *An Introduction to Chinese, Japanese and Korean Computing*. World Scientific Publishing. 1989. ISBN 9971-50-664-5.

IBM Corporation. *Character Data Representation Architecture Reference and Registry*. 1995. IBM part number SC09-2190-00.

———. *AIX Version 3.2 for RISC System/6000: Internationalization of AIX Software—A Programmer's Guide.* Second Edition. 1992. IBM part number SC23-2431.

———. *DBCS Design Guide for DOS/V and MS Windows Programming.* IBM DBCS Technical Coordination Office. 1992. IBM part number DTC 0-0012-0.

Jelliffe, Rick. *The XML & SGML Cookbook: Recipes for Structured Information.* Prentice-Hall. 1998. ISBN 0-13-614223-0.

Kano, Nadine. *Developing International Software for Windows 95 and Windows NT.* Microsoft Press. 1995. ISBN 1-55615-840-8.

Kaplan, Jerry. *Startup: A Silicon Valley Adventure.* Penguin Books. 1994. ISBN 0-14-025731-4.

Karow, Peter. *Typeface Statistics.* URW. 1993. ISBN 3-926515-08-2.

———. *Digital Formats for Typefaces.* URW. 1987. ISBN 3-926515-01-5.

Karp, David A. et al. *Windows XP in a Nutshell.* Second Edition. O'Reilly Media, Incorporated. 2005. ISBN 0-596-00900-3.

Kissell, Joe. *The Nisus Way.* MIS:Press. 1996. ISBN 1-55828-455-9.

Knuth, Donald E. *Digital Typography.* CSLI Publications. 1999. ISBN 1-57586-010-4 (paper) and 1-57586-011-2 (cloth).

Korpela, Jukka K. *Unicode Explained.* O'Reilly Media, Incorporated. 2006. ISBN 0-596-10121-X.

Lunde, Ken. *Prescriptive Kanji Simplification.* PhD Dissertation. University of Wisconsin–Madison. 1994. University Microfilms International order number 9419580.

———. *Understanding Japanese Information Processing.* O'Reilly Media, Incorporated. 1993. ISBN 1-56592-043-0. Made obsolete by this book.

Luong, Tuoc V. et al. *Internationalization: Developing Software for Global Markets.* John Wiley & Sons, Incorporated. 1995. ISBN 0-471-07661-9.

Lutz, Mark. *Programming Python.* Third Edition. O'Reilly Media, Incorporated. 2006. ISBN 0-596-00925-9.

Madell, Tom et al. *Developing and Localizing International Software.* Prentice-Hall. 1994. ISBN 0-13-300674-3.

McFarland, David Sawyer. *CSS: The Missing Manual.* O'Reilly Media, 1996. ISBN 0-596-52687-3.

McFarland, Thomas. *X Windows on the World: Developing Internationalized Software with X, Motif, and CDE.* Prentice-Hall. 1996. ISBN 0-13-359787-3.

McGilton, Henry & Mary Campione. *PostScript by Example.* Addison-Wesley. 1992. ISBN 0-201-63228-4.

Ministry of Culture & Tourism. *The Revised Romanization of Korean.* 2000.

Moye, Stephen. *Fontographer: Type by Design.* MIS:Press. 1995. ISBN 1-55828-447-8.

Musciano, Chuck & Bill Kennedy. *HTML & XHMTL: The Definitive Guide.* Sixth Edition. O'Reilly Media, Incorporated. 2006. ISBN 0-596-52732-2.

Niederst Robbins, Jennifer. *HTML and XHTML Pocket Reference.* Third Edition. O'Reilly Media, Incorporated. 2006. ISBN 0-596-52727-6.

O'Donnell, Sandra Martin. *Programming for the World: A Guide to Internationalization.* Prentice-Hall. 1994. ISBN 0-13-722190-8.

Pogue, David. *Windows Vista: The Missing Manual.* O'Reilly Media, Incorporated. 2006. ISBN 0-596-52827-2.

———. *Mac OS X Leopard: The Missing Manual.* O'Reilly Media, Incorporated. 2007. ISBN 0-596-52952-X.

Powers, Shelley et al. *UNIX Power Tools.* Third Edition. O'Reilly Media, Incorporated. 2002. ISBN 0-596-00330-7.

Pollack, David, editor. *Soft Landing in Japan: A Market Entry Handbook for U.S. Software Companies.* Version 2.0J. American Electronics Association. 1992.

Reid, Glenn. *Thinking in PostScript.* Addison-Wesley. 1990. ISBN 0-201-52372-8.

Robbins, Arnold. *Unix in a Nutshell.* Fourth Edition. O'Reilly Media, Incorporated. 2005. ISBN 0-596-10029-9.

Robbins, Arnold et al. *Learning the vi and Vim Editors.* Seventh Edition. O'Reilly Media, Incorporated. 2008. ISBN 0-596-52983-X.

Sakamura, Ken. *MicroITRON 3.0: An Open and Portable Real-Time Operating System for Embedded Systems—Concept and Specification.* IEEE Computer Society. ISBN 0-8186-7795-3.

Schwartz, Alan. *Managing Mailing Lists.* O'Reilly Media, Incorporated. 1998. ISBN 1-56592-259-X.

Schwartz, Randal L. et al. *Learning Perl.* Fifth Edition. O'Reilly Media, Incorporated. 2008. ISBN 0-596-52010-7.

Searfoss, Glenn. *JIS-Kanji Character Recognition: Featuring the Gaiji Method.* Van Nostrand Reinhold. 1994. ISBN 0-442-01813-4.

Siever, Ellen et al. *Linux in a Nutshell.* Fifth Edition. O'Reilly Media, Incorporated. 2005. ISBN 0-596-00930-5.

Spainhour, Stephen et al. *Perl in a Nutshell.* Second Edition. O'Reilly Media, Incorporated. 2002. ISBN 0-596-00241-6.

Spiekermann, Erik & E.M. Ginger. *Stop Stealing Sheep & Find Out How Type Works.* Adobe Press. 1993. ISBN 0-672-48543-5.

St.Laurent, Simon & Michael Fitzgerald. *XML Pocket Reference.* Third Edition. O'Reilly Media, Incorporated. 2005. ISBN 0-596-10050-7.

Stallman, Richard. *GNU Emacs Manual.* Sixteenth Edition. Free Software Foundation. 2007. ISBN 1-882114-86-8.

Stein, Lincoln. *Official Guide to Programming with CGI.pm: The Standard for Building Web Scripts.* John Wiley & Sons, Incorporated. 1998. ISBN 0-471-24744-8.

Stubblebine, Tony. *Regular Expression Pocket Reference.* Second Edition. O'Reilly Media, Incorporated. 2007. ISBN 0-596-51427-1.

Tuthill, Bill & David Smallberg. *Creating Worldwide Software: Solaris International Developer's Guide.* Second Edition. Prentice-Hall. 1997. ISBN 0-13-494493-3.

Unger, J. Marshall. *Literacy and Script Reform in Occupation Japan: Reading Between the Lines.* Oxford University Press. 1996. ISBN 0-19-510166-9.

———. *The Fifth Generation Fallacy: Why Japan Is Betting Its Future on Artificial Intelligence.* Oxford University Press. 1987. ISBN 0-19-504939-X.

Unicode Consortium, The. *The Unicode Standard: Worldwide Character Encoding.* Version 1.0, Volume 1. Addison-Wesley. 1991. ISBN 0-201-56788-1.

———. *The Unicode Standard: Worldwide Character Encoding.* Version 1.0, Volume 2. Addison-Wesley. 1992. ISBN 0-201-60845-6.

———. *The Unicode Standard, Version 2.0.* Addison-Wesley. 1996. ISBN 0-201-48345-9.

———. *The Unicode Standard, Version 3.0.* Addison-Wesley. 2000. ISBN 0-201-61633-5.

———. *The Unicode Standard, Version 4.0.* Addison-Wesley. 2003. ISBN 0-321-18578-1.

———. *The Unicode Standard, Version 5.0.* Addison-Wesley. 2006. ISBN 0-321-48091-0.

Uren, Emmanuel et al. *Software Internationalization and Localization: An Introduction.* Van Nostrand Reinhold. 1993. ISBN 0-442-01498-8.

Vromans, Johan. *Perl Pocket Reference.* Fourth Edition. O'Reilly Media, Incorporated. 2002. ISBN 0-596-00374-9.

Wall, Larry et al. *Programming Perl.* Third Edition. O'Reilly Media, Incorporated. 2000. ISBN 0-596-00027-8.

Walsh, Norman. *Making TEX Work.* O'Reilly Media, Incorporated. 1994. ISBN 1-56592-051-1.

Welsh, Matt & Matthias Kalle Dalheimer. *Running Linux.* Fifth Edition. O'Reilly Media, Incorporated. 2005. ISBN 0-596-00760-4.

Wong, Clinton. *Web Client Programming in Perl.* O'Reilly Media, Incorporated. 1997. ISBN 1-56592-214-X.

Books in Chinese

陈建平.『常用汉字输入法操作速成』. 福建科学技术出版社. 1996. ISBN 7-5335-1043-7.

国家语言文字工作委员会.『简化字总表』. Second Edition. 语文出版社. 1986. ISBN 7-80006-282-1.

国家语言文字工作委员会汉字处.『现代汉语常用字表』. 语文出版社. 1988. ISBN 7-80006-107-8.

———.『现代汉语通用字表』. 语文出版社. 1988. ISBN 7-80006-167-1.

何根泽 & 何新, editors.『汉字输入快易通』. 电子工业出版社. 1996. ISBN 7-5053-3450-6.

黃大一.『中文字碼—萬碼奔騰, 一碼當先』. Second Edition. 永麒科技股份有限公司. 1992. ISBN 957-9064-00-8.

刘之强, editor.『简化字 繁体字 选用字 异体字对照表』. 上海辞书出版社. 1983.

苏培成 et al., editors.『现代汉字规范化问题』. 语文出版社. 1995. ISBN 7-80006-889-7.

谢世涯.『新中日简体字研究』. 语文出版社. 1989. ISBN 7-80006-222-8.

行政院研究發展考核委員會, editor.『兩岸常用中文資訊名詞對照表及兩岸常用中文資訊內碼對照轉碼表之編擬』. 行政院研究發展考核委員會. 1994. ISBN 957-00-3422-X.

张乐之 et al., editors.『计算机汉字输入与编辑实用手册』. 上海交通大学出版社. 1994. ISBN 7-313-01405-8.

Books in Japanese

アスキー出版技術部責任編集.『日本語 TEX テクニカルブック I』. ASCII Corporation. 1990. ISBN 4-7561-0405-3.

Apple Computer Japan.『Macintosh 漢字 Talk テクニカル・リファレンス』. 技術評論社. 1990. ISBN 4-87408-369-2.

泉均.『ワープロ用語図説辞典』. 山海堂. 1988. ISBN 4-381-08071-8.

遠藤紹徳.『早わかり中国簡体字』. 国書刊行会. 1986.

大木敦雄.『入門 NEmacs』. ASCII Corporation. 1994. ISBN 4-7561-0287-5.

———.『入門 Mule』. ASCII Corporation. 1994. ISBN 4-7561-0300-6.

岡本保.『タイプ・デザインのルール〈ゴシック体漢字編〉』. 富士通アプリコ株式会社. 1993.

———. 『タイプ・デザインのルール〈明朝体漢字編〉』. 富士通アプリコ株式会社. 1993.

Obscure Inc., editors.『デザイン、DTP のためのフォントの鉄則』. MYCOM. 2003. ISBN 4-8399-0892-3.

エツコ・オバタ・ライマン.『日本人の作った漢字』. 南雲堂. 1990. ISBN 4-523-26156-3.

樺島忠夫 et al., editors.『事典日本の文字』. 大修館書店. 1985. ISBN 4-469-01209-2.

川俣晶.『パソコンにおける日本語処理/文字コード ハンドブック』. 技術評論社. 1999. ISBN 4-7741-0780-8.

『誤字俗字・正字一覧表』. テイハン. 1995. ISBN 4-924485-29-2.

共同通信社.『記者ハンドブック』. Ninth Edition. 共同通信社. 2001. ISBN 4-7641-0475-X.

共同通信社情報システム局通信部.『字形と入力』. Second Edition. 共同通信社. 1995.

清兼義弘 & 末廣陽一, editors.『国際化プログラミング―I18N ハンドブック』. 共立出版. 1998. ISBN 4-320-02904-6.

Lunde, Ken.『日本語情報処理』. SOFTBANK Corporation. 1995. ISBN 4-89052-708-7.

斎藤靖 et al.『新 Perl の国へようこそ』. サイエンス社. 1996. ISBN 4-7819-0795-4.

逆井克己.『基本日本語文字組版』. 日本印刷新聞社. 1999. ISBN 4-88884-093-8C.

坂村健.『新版トロンヒューマンインタフェース標準ハンドブック』. Personal Media Corporation. 1996. ISBN 4-89362-141-6.

———.『BTRON1 プログラミング標準ハンドブック』. Personal Media Corporation. 1992. ISBN 4-89362-093-2.

佐渡秀治 & 吉田智子.『Linux/FreeBSD 日本語環境の構築と活用』. ソフトバンク. 1997. ISBN 4-7973-0480-4.

佐藤喜代治 et al.『漢字講座』. 10 Volumes. 大修館書店. 1987–1989.

J-PRESS.『縦組み DTP 制作の現場』. エーアイ出版. 1997. ISBN 4-87193-524-8.

『常用漢字表』. 大蔵省印刷局. 1987. ISBN 4-17-214500-0.

真堂彬 & プロビット.『JIS 補助漢字』. エーアイ出版. 1991. ISBN 4-87193-158-7.

菅野芩.『中国入力方法の話』. 朝日出版社. 1991. ISBN 4-255-91006-5.

長尾真 et al., editors.『情報科学辞典』. 岩波書店. 1990. ISBN 4-00-080074-4.

中西秀彦.『活字が消えた日―コンピュータと印刷』. 晶文社. 1994. ISBN 4-7949-6172-3.

錦見美貴子 et al.『マルチリンガル環境の実現: X Window/Wnn/Mule/WWW ブラウザでの多国語環境』. Prentice-Hall. 1996. ISBN 4-88735-020-1.

西野嘉章.『歴史の文字―記載・活字・活版』. 東京大学総合研究博物館. 1996.

日本エディタースクール.『標準校正必携〈電算植字対応版〉』. Seventh Edition. 日本エディタースクール出版部. 1995. ISBN 4-88888-235-5.

野村保惠.『〈電算植字〉本づくり入門』. 日本エディタースクール出版部. 1995. ISBN 4-88888-231-2.

林四郎 & 松岡榮志.『日本の漢字・中国の漢字』. 三省堂. 1995. ISBN 4-385-35597-5.

原田種成.『漢字小百科辞典』. 三省堂. 1990. ISBN 4-385-13590-8.

Hitachi.『HITAC 文字コード表 (KEIS83)』. 1989. Hitachi part number 8080-2-100-10.

府川充男.『組版原論―タイポグラフィと活字・写植・DTP』. 太田出版. 1996. ISBN 4-87233-272-5.

藤岡康隆 et al.『DTP フォント入門 Macintosh 編』. MdN Corporation. 1999. ISBN 4-8443-5538-4.

古瀬幸広.『ワープロここが不思議―ちょっと知的なワープロ学』. 講談社. 1994. ISBN 4-06-257018-1.

———.『最新ワープロ用語辞典』. 実業之日本社. 1991. ISBN 4-408-10095-1.

———.『ネットワーク通信活用ブック』. 実業之日本社. 1991. ISBN 4-408-10096-X.

三上喜貴.『文字符号の歴史―アジア編―』. 共立出版. 2002. ISBN 4-320-12040-X.

三上吉彦 et al.『マルチリンガル WEB ガイド』. O'Reilly Japan. 1997. ISBN 4-900900-23-0.

文字フォント開発・普及センター.『新フォント関連用語集―フォントと組版に関する用語 解説』. 日本規格協会. 1993.

森浩孝.『パソコン通信ガイドブック』. HBJ Publishing. 1986. ISBN 4-8337-8512-9.

森田正典.『これが日本語に最適なキーボードだ』. 日本経済新聞社. 1992. ISBN 4-532-40014-7.

森田正典 & 丸山和光.『日本語だから速く入力できる』. 日刊工業新聞社. 1988. ISBN 4-526-02310-8.

安岡孝一 & 安岡素子.『文字コードの世界』. TDU. 1999. ISBN 4-501-53060-X.

———.『文字符号の歴史―欧米と日本編―』. 共立出版. 2006. ISBN 4-320-12102-3.

———.『キーボード配列 QWERTY の謎』. NTT 出版. 2008. ISBN 978-4-7571-4176-6.

吉田智子.『UNIX の日本語処理がわかる本― 最新 Wnn 活用ガイド』. 日刊工業新聞社. 1993. ISBN 4-526-03321-9.

吉目木晴彦 et al.『電脳文化と漢字のゆくえ― 岐路に立つ日本語』. 平凡社. 1998. ISBN 4-582-40322-0.

和田義浩 et al.『DTP フォント完全理解!』. Works Corporation. 2002. ISBN 4-948759-42-2.

Books in Korean

Dong-A's Prime Korean-English Dictionary. Second Edition. Doosan Dong-A Company, Limited. 1996. ISBN 89-00-04440-0.

김 경석.『컴퓨터 속의 한글 이야기』. 영진 출판사. 1995. ISBN 89-314-0578-2.

김 진평.『한글의 글자표현』. Second Edition. 미진사. 1997. ISBN 89-408-0109-1.

김 학성 (金學成).『레터링 디자인』. 조형사. 1997. ISBN 89-8307-011-0.

Ideograph Dictionaries

赤塚忠 et al.『旺文社 漢和辞典』. Fifth Edition. 旺文社. 1993. ISBN 4-01-077703-6.

费锦昌 et al., editors.『汉字写法规范字典』. 上海辞书出版社. 1992. ISBN 7-5326-0119-6.

傅永和, editor.『汉字属性字典』. 语文出版社. 1989. ISBN 7-80006-242-2.

Fujitsu Limited.『FACOM JEF 文字コード索引辞書』. 1987. Fujitsu part number 99FR-0012-3.*

Halpern, Jack, editor. *The Kodansha Kanji Learner's Dictionary.* Kodansha International Limited. 1998. ISBN 4-7700-2335-9.

———. *NTC's New Japanese-English Character Dictionary.* NTC. 1993. ISBN 0-8442-8434-3.

———. *New Japanese-English Character Dictionary.* Kenkyusha. 1990. ISBN 4-7674-9040-5.

飛田良文.『国字の字典』. 東京堂出版. 1990. ISBN 4-490-10279-8.

Hitachi.『HITAC 文字パターン辞書/コードブック (KEIS83 拡張文字セット 3)』. 1987. Hitachi part number 8080-2-109.

———.『HITAC 文字パターン辞書/コードブック (拡張文字セット 3)』. 1984. Hitachi part number 8080-2-074-10.

胡双宝.『简化字 繁体字 异体字辨析手册』. 北京大学出版社. 1996. ISBN 7-301-03198-X.

石川忠久 et al.『福武 漢和辞典 新装版』. Benesse. 1997. ISBN 4-8288-0435-8.

覚田正 & 米山寅太郎.『新版 漢語林』. 大修館書店. 1994. ISBN 4-469-03107-0.

———.『大漢語林』. 大修館書店. 1992. ISBN 4-469-03154-2.

———.『新漢語林』. 大修館書店. 2004. ISBN 4-469-03162-3.

蓝德康, editor.『国际标准汉字大字典』. 电子工业出版社. 1998. ISBN: 7-5053-4481-1.

冷玉龙, editor.『中华字海』. 中华出版社. 1994. ISBN: 7-5057-0630-6.

民志書林編輯, editors.『活用玉篇』. 民志書林. 1983. ISBN 89-387-0110-7.

諸橋轍次.『大漢和辭典』. Revised Second Edition. 13 Volumes. 大修館書店. 1994.

* FACOM is usually pronounced in a way that is close to a two-word obscenity.

NEC.『日本電気標準文字セット辞書 基本編』. 1983. NEC part number ZBB10-2.

———.『日本電気標準文字セット辞書 拡張編』. 1983. NEC part number ZBB11-1.

Nelson, Andrew & John H. Haig. *The New Nelson Japanese-English Character Dictionary* 『新版ネルソン漢英辞典』. New Nelson Edition. Charles E. Tuttle Company. 1997. ISBN 0-8048-2036-8.

日外アソシエーツ編集部.『漢字異体字典』. 日外アソシエーツ. 1994. ISBN 4-8169-1249-5.

小川環樹 et al.『角川 必携漢和辞典』. 角川書店. 1996. ISBN 4-04-013300-5.

———.『角川 新字源 改訂版』. 角川書店. 1994. ISBN 4-04-010804-3.

商務印書館編輯部.『辭源』. 商務印書館. 1995. ISBN 7-100-00540-X.

芝野耕司, editor.『JIS 漢字字典』. 財団法人日本規格協会. 1997. ISBN 4-542-20127-9.

———.『増補改訂 JIS 漢字字典』. 財団法人日本規格協会. 2002. ISBN 4-542-20129-5.

新潮社, editor.『新潮日本語漢字辞典』. 新潮社. 2007. ISBN 978-4-10-730215-1.

新村出, editor.『広辞苑』. Fourth Edition. 岩波書店. 1991. ISBN 4-00-080101-5.

Spahn, Mark & Wolfgang Hadamitzky. *The Kanji Dictionary.* Charles E. Tuttle Company. 1996. ISBN 0-8048-2058-9.

蘇培成, editor.『漢字簡繁體字對照字典』. Second Edition. 海峰出版社 & 中信出版社 (co-publishers). 1996. ISBN 962-238-213-4.

田嶋一夫.『最新 JIS 漢字辞典』. 講談社. 1990. ISBN 4-06-123264-9.

竹田晃＆坂梨隆三.『五十音引き講談社漢和辞典』. 講談社. 1997. ISBN 4-06-123269-X.

上柿力.『パソコンワープロ漢字辞典』. ナツメ社. 1987. ISBN 4-8163-0696-X.

上田万年 et al., editors.『新大字典』. 講談社. 1993. ISBN 4-06-123140-5.

Viện Ngôn Ngữ Học. *Bảng Tra Chữ Nôm.* Nhà Xuất Bản Khoa Học Xã Hội. 1976. Permission number 5/KHXH 76.

Vũ Văn Kính. *Tự Diển Chữ Nôm.* Nhà Xuất Bản Đà Nẵng. 1996. Permission number 10/226.

『ワープロ・パソコン最新漢字辞典』. 小学館. 1994. ISBN 4-09-505121-3.

吴伟和 et al., editors.『汉字输入速查手册』. 中国工人出版社. 1996. ISBN 7-5008-1846-7.

許慎.『說文解字』. 中華書局. 100. ISBN 962-231-208-X.

楊子來.『標準中文輸入碼大字典』. 聚賢館文化有限公司. 1996. ISBN 962-436-287-4.

ユニコード漢字情報辞典編集委員会, editors. *Sanseido's Unicode Kanji Information Dictionary* (『ユニコード漢字情報辞典』). 三省堂. 2000. ISBN 4-385-13690-4.

張三植.『大字源』. 集文堂. 1972.

張玉書 et al.『康熙字典』. 中華書局. 1716. ISBN 962-231-006-0.

中文社会科学院语言研究所词典编辑室, editors. 『现代汉语词典』. 商务印书馆. 1995. ISBN 7-100-00044-0.

周冰洋 et al., editors.『常用汉字编码字典』. 宇航出版社. 1990. ISBN 7-80034-102-X.

朱文章, editor.『字詞·成語·辨正辭典』. 文翔圖書股份有限公司. 1986.

邹华清, editor.『汉语大字典』. 13 volumes. 四川辞书出版社 & 湖北辞书出版社. 1986.

Standards

American National Standards Institute. ANSI X3.4-1986 *Coded Character Set—7-Bit American National Standard Code for Information Interchange*. 1986.

———. ANSI Z39.64-1989 *East Asian Character Code for Bibliographic Use*. 1989. ISSN 1041-5653.

Chinese National Standard. CNS 5205-1989 *Information Processing—7-Bit Coded Character Set for Information Interchange* (『資訊處理及交換用七數元碼字元集』). 1989.

———. CNS 11643-1986 *Standard Interchange Code for Generally-Used Chinese Characters* (『通用漢字標準交換碼』). 1986. Obsoleted by CNS 11643-1992.

———. CNS 11643-1992 *Chinese Standard Interchange Code* (『中文標準交換碼』). 1992. Obsoletes CNS 11643-1986. Obsoleted by CNS 11643-2007.

———. CNS 11643-2007 *Chinese Standard Interchange Code* (『中文標準交換碼』). 2007. Obsoletes CNS 11643-1992.

———. CNS 14649-1:2002 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane* (『資訊技術—廣用多八位元編碼字元集 (UCS)—第 1 部: 架構及基本多語文字面』). 2002

———. CNS 14649-2:2003 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 2: Supplementary Planes* (『資訊技術—廣用多八位元編碼字元集 (UCS)—第 2 部: 輔助字面』). 2003

Fujitsu Limited.『富士通文字コード解説書』. 1989. Fujitsu part number 99FR-8010-1.

Hitachi.『KEIS 概説』. 1990. Hitachi part number 6180-3-003.

IBM Corporation. *Coded Character Sets: Implementation.* IBM Standards Program. 1991. IBM part number C-S 3-3220-019 1991-10.

———. *Double-Byte Character Set (DBCS): Terminology and Code Scheme.* IBM Standards Program. 1992. IBM part number C-S 3-3220-102 1992-11.

———. *Extended BCD Interchange Code: EBCDIC.* IBM Standards Program. 1990. IBM part number C-S 3-3220-002 1990-05.

———. *IBM Japanese Graphic Character Set, Kanji, for Open Environment, DBCS-PC (New JIS Sequence).* IBM Standards Program. 1996. IBM part number C-H 3-3220-133 1996-08.

———. *IBM Japanese Graphic Character Set for Extended UNIX Code (EUC): DBCS-EUC.* IBM Standards Program. 1993. IBM part number C-H 3-3220-127 1993-03.

———. *IBM Japanese Graphic Character Set, Kanji: DBCS-Host and DBCS-PC.* IBM Standards Program. 1992. IBM part number C-H 3-3220-024 1992-11.

———. *IBM Korean Graphic Character Set, DBCS-Host and DBCS-PC (For Windows Environment).* IBM Standards Program. 1997. IBM part number C-H 3-3220-030 1997-09.

———. *IBM Korean Graphic Character Set for Extended UNIX Code (EUC), DBCS-EUC.* IBM Standards Program. 1993. IBM part number C-H 3-3220-128 1993-11.

———. *IBM Korean Graphic Character Set, DBCS-Host and DBCS-PC.* IBM Standards Program. 1992. IBM part number C-H 3-3220-125 1992-09.

———. *IBM Simplified Chinese Graphic Character Set, GBK Code, DBCS-Host and DBCS-PC.* IBM Standards Program. 1997. IBM part number C-H 3-3220-020 1997-02.

———. *IBM Simplified Chinese Graphic Character Set for Extended UNIX Code (EUC), DBCS-EUC.* IBM Standards Program. 1994. IBM part number C-H 3-3220-132 1994-06.

———. *IBM Simplified Chinese Graphic Character Set, DBCS-Host and DBCS-PC.* IBM Standards Program. 1993. IBM part number C-H 3-3220-130 1993-11.

———. *IBM Traditional Chinese Graphic Character Set for IBM BIG-5 Code, DBCS-PC.* IBM Standards Program. 1994. IBM part number C-H 3-3220-131 1994-01.

———. *IBM Traditional Chinese Graphic Character Set for Extended UNIX Code (EUC), DBCS-EUC and TBCS-EUC.* IBM Standards Program. 1993. IBM part number C-H 3-3220-129 1993-11.

———. *IBM Traditional Chinese Graphic Character Set, DBCS-Host and DBCS-PC.* IBM Standards Program. 1992. IBM part number C-H 3-3220-126 1992-01.

International Organization for Standardization. *International Register of Coded Character Sets to Be Used with Escape Sequences.* 1996.

———. ISO 639-1:2002 *Code for the Representation of Names of Languages—Part 1: Alpha-2 Code.* 2002.

———. ISO 639-2:1998 *Codes for the Representation of Names of Languages—Part 2: Alpha-3 Code.* 1998.

———. ISO 646:1991 *Information Technology—ISO 7-Bit Coded Character Set for Information Interchange.* 1991.

———. ISO 2022:1994 *Information Technology— Character Code Structure and Extension Techniques.* 1994.

———. ISO 3166-1:2006 *Codes for the Representation of Names of Countries and Their Subdivisions—Part 1: Country Codes.* 2006.

———. ISO 3166-2:2007 *Codes for the Representation of Names of Countries and Their Subdivisions—Part 2: Country Subdivision Code.* 2007.

———. ISO 6429:1992 *Information Technology—Control Functions for Coded Character Sets.* 1992.

———. ISO 7098:1991 *Information and Documentation—Romanization of Chinese.* 1991.

———. ISO 8859 *Information Processing—8-Bit Single-Byte Coded Graphic Character Sets. Fifteen parts.* 1987–2003.

———. ISO 8879:1986 *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML).* 1986.

———. ISO 9541:1991 *Information Technology—Font Information Interchange. Three parts.* 1991–1994.

———. ISO 10179:1996 *Information Technology—Processing Languages—Document Style Semantics and Specification Language (DSSSL).* 1996.

———. ISO 10646-1:1993 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane.* 1993. Obsoleted by ISO 10646-1:2000.

———. ISO 10646-1:2000 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane.* 2000. Obsoletes ISO 10646-1:1993. Obsoleted by ISO 10646:2003.

———. ISO 10646-2:2001 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 2: Supplementary Planes.* 2001. Obsoleted by ISO 10646:2003.

———. ISO 10646:2003 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS).* 2003. Obsoletes ISO 10646-1:2000 and ISO 10646-2:2001.

———. ISO/TR 11941:1996 *Information Documentation—Transliteration of Korean Script into Latin Characters.* 1996.

———. ISO 14755:1997 *Information Technology—Input Methods to Enter Characters from the Repertoire of ISO/IEC 10646 With a Keyboard or Other Input Device.* 1997.

———. ISO 15924:2004 *Information and Documentation—Codes for the Representation of Names of Scripts.* 2004.

———. ISO 32000-1:2008 *Document Management—Portable Document Format—Part 1: PDF 1.7.* 2008.

Japanese Industrial Standards Committee. JIS C 6226-1978 *Code of the Japanese Graphic Character Set for Information Interchange* (『情報交換用漢字符号』). Japanese Standards Association. 1978.

———. JIS X 0201-1997 *7-Bit and 8-Bit Coded Character Sets for Information Interchange* (『7ビット及び8ビットの情報交換用符号化文字集合』). Japanese Standards Association. 1997. Originally designated JIS C 6220-1976, and obsoletes JIS X 0201-1976.

———. JIS X 0202:1998 *Information Processing—ISO 7-Bit and 8-Bit Coded Character Sets—Code Extension Techniques* (『情報技術—文字符号の構造及び拡張法』). Japanese Standards Association. 1998. Originally designated JIS C 6228-1984, and obsoletes JIS X 0202-1984 and JIS X 0202-1991.

———. JIS X 0207-1979 *Code of the Control Character Set for Japanese Graphic Characters for Information Interchange* (『情報交換用漢字符号のための制御文字符号』). Japanese Standards Association. 1979. Obsoletes JIS C 6225-1979.

———. JIS X 0208-1983 *Code of the Japanese Graphic Character Set for Information Interchange* (『情報交換用漢字符号』). Japanese Standards Association. 1983. Originally designated JIS C 6226-1983, and obsoletes JIS C 6226-1978.

———. JIS X 0208-1990 *Code of the Japanese Graphic Character Set for Information Interchange* (『情報交換用漢字符号』). Japanese Standards Association. 1990. Obsoletes JIS X 0208-1983.

———. JIS X 0208:1997 *7-Bit and 8-Bit Double Byte Coded Kanji Sets for Information Interchange* (『7ビット及び8ビットの2バイト情報交換用符号化漢字集合』). Japanese Standards Association. 1997. Obsoletes JIS X 0208-1990.

———. JIS X 0212-1990 *Code of the Supplementary Japanese Graphic Character Set for Information Interchange* (『情報交換用漢字符号—補助漢字』). Japanese Standards Association. 1990.

———. JIS X 0213:2000 *7-Bit and 8-Bit Double Byte Coded Extended Kanji Sets for Information Interchange* (『7ビット及び8ビットの2バイト情報交換用符号化拡張漢字集合』). Japanese Standards Association. 2000. Obsoleted by JIS X 0213:2004.

———. JIS X 0213:2004 *7-Bit and 8-Bit Double Byte Coded Extended Kanji Sets for Information Interchange (Amendment 1)* (『7ビット及び8ビットの2バイト情報交換用符号化拡張漢字集合 (追補 1)』). Japanese Standards Association. 2004. Obsoletes JIS X 0213:2000.

———. JIS X 0221-1995 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane* (『国際符号化文字集合 (UCS)—第1部: 体系及び基本多言語面』). Japanese Standards Association. 1995. Obsoleted by JIS X 0221-1:2001.

———. JIS X 0221-1:2001 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane* (『国際符号化文字集合 (UCS)—第1部: 体系及び基本多言語面』). Japanese Standards Association. 2001. Obsoletes JIS X 0221-1995. Obsoleted by JIS X 0221:2007.

———. JIS X 0221:2007 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)* (『国際符号化文字集合(UCS)』). Japanese Standards Association. 2007. Obsoletes JIS X 0221-1:2001.

———. JIS X 4051-1993 *Line Composition Rules for Japanese Documents* (『日本語文書の行組版方法』). Japanese Standards Association. 1993. Obsoleted by JIS X 4051-1995.

———. JIS X 4051-1995 *Line Composition Rules for Japanese Documents* (『日本語文書の行組版方法』). Japanese Standards Association. 1995. Obsoletes JIS X 4051-1993. Obsoleted by JIS X 4051:2004.

———. JIS X 4051:2004 *Formatting Rules for Japanese Documents* (『日本語文書の組版方法』). Japanese Standards Association. 2004. Obsoletes JIS X 4051-1995.

———. JIS X 4061-1996 *Collation of Japanese Character String* (『日本語文字列照合順番』). Japanese Standards Association. 1996.

———. JIS X 4062:1998 *Format for Information Interchange for Dictionaries of Japanese Input Method* (『仮名漢字変換辞書交換形式』). Japanese Standards Association. 1998.

———. JIS X 4161-1993 *Font Information Interchange—Architecture* (『フォント情報交換—体系』). Japanese Standards Association. 1993.

———. JIS X 4162-1993 *Font Information Interchange—Interchange Format* (『フォント情報交換—交換用式』). Japanese Standards Association. 1993.

———. JIS X 4163-1994 *Font Information Interchange—Glyph Shape Representation* (『フォント情報交換—グリフ形状表現』). Japanese Standards Association. 1994.

———. JIS X 6002-1985 *Keyboard Layout for Information Processing Using the JIS 7-Bit Coded Character Set* (『情報処理系けん盤配列』). Japanese Standards Association. 1985. Originally designated JIS C 6233-1980.

———. JIS X 6003-1989 *Keyboard Layout for Japanese Text Processing* (『日本語文書処理用文字盤配列』). Japanese Standards Association. 1989. Originally designated JIS C 6235-1984.

———. JIS X 6004-1986 *Basic Keyboard Layout for Japanese Text Processing Using Kana-Kanji Translation Method* (『仮名漢字変換形日本文入力装置用けん盤配列』). Japanese Standards Association. 1986. Originally designated JIS C 6236-1986.

———. JIS X 9051-1984 *16-Dots Matrix Character Patterns for Display Devices* (『表示装置用16ドット字形』). Japanese Standards Association. 1984. Originally designated JIS C 6232-1984.

———. JIS X 9052-1983 *24-Dots Matrix Character Patterns for Dot Printers* (『ドットプリンタ用24ドット字形』). Japanese Standards Association. 1983. Originally designated JIS C 6234-1983.

Korean Industrial Standards Association. KS X 1001:1992 *Code for Information Interchange (Hangul and Hanja)* (『정보 교환용 부호 (한글 및 한자)』). Korean Industrial Standard. 1992. Originally designated KS C 5601-1992. Obsoleted by KS X 1001:1998.

———. KS X 1001:1998 *Code for Information Interchange (Hangul and Hanja)* (『정보 교환용 부호 (한글 및 한자)』). Korean Industrial Standard. 1998. Obsoletes KS X 1001:1992. Obsoleted by KS X 1001:2002.

———. KS X 1001:2002 *Code for Information Interchange (Hangul and Hanja)* (『정보 교환용 부호 (한글 및 한자)』). Korean Industrial Standard. 2002. Obsoletes KS X 1001:1998. Obsoleted by KS X 1001:2004.

———. KS X 1001:2004 *Code for Information Interchange (Hangul and Hanja)* (『정보 교환용 부호 (한글 및 한자)』). Korean Industrial Standard. 2004. Obsoletes KS X 1001:2002.

———. KS X 1002:1991 *Code for Information Interchange Supplementary Set* (『정보 교환용 부호 확장 세트』). Korean Industrial Standard. 1991. Originally designated KS C 5657-1991. Obsoleted by KS X 1002:2001.

———. KS X 1002:2001 *Code for Information Interchange Supplementary Set* (『정보 교환용 부호 확장 세트』). Korean Industrial Standard. 2001. Obsoletes KS X 1002:1991.

———. KS X 1003:1993 *Code for Information Interchange (Roman Characters)* (『정보 교환 용 부호 (로마 문자)』). Korean Industrial Standard. 1993. Originally designated KS C 5636-1993.

———. KS X 1004:1995 *Code Extension Techniques for Use with the Code for Information Interchange* (『정보 교환용 부호의 확장법』). Korean Industrial Standard. 1995. Originally designated KS C 5620-1995.

———. KS X 1005-1:1995 *Information technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane* (『국제 문자 부호계 (UCS) 제 1 부 : 구조 및 기본 다국어 평면』). Korean Industrial Standard. 1995. Originally designated KS C 5700-1995.

———. KS X 2901:1992 *UNIX-Hangul Environment* (『유닉스 한글 환경』). Korean Industrial Standard. 1992. Originally designated KS C 5861-1992.

———. KS X 5002:1992 *Keyboard Layout for Information Processing* (『정보 처리용 건반 배열』). Korean Industrial Standard. 1992. Originally designated KS C 5715-1992.

People's Republic of China, The. GB 1988-89 *Information Processing—7-Bit Coded Character Set for Information Interchange* (『信息处理—信息交换用七位编码字符集』). Technical Standards Press. 1990. Obsoletes GB 1988-80.

———. GB 2311-80 *Information Processing—7-Bit and 8-Bit Coded Character Set—Code Extension Techniques* (『信息处理—七位及八位编码字符集—代码扩充技术』). Technical Standards Press. 1980.

———. GB 2312-80 *Code of Chinese Graphic Character Set for Information Interchange Primary Set* (『信息交换用汉字编码字符集—基本集』). Technical Standards Press. 1981.

———. GB 5007.1-85 *24×24 Dot Matrix Font Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 24×24 点阵字模集』). Technical Standards Press. 1985.

———. GB 5007.2-85 *24×24 Dot Matrix Font Data Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 24×24 点阵字模数据集』). Technical Standards Press. 1985.

———. GB 6345.1-86 *32×32 Dot Matrix Font Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 32×32 点阵字模集』). Technical Standards Press. 1986.

———. GB 6345.2-86 *32×32 Dot Matrix Font Data Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 32×32 点阵字模数据集』). Technical Standards Press. 1986.

———. GB 7589-87 *Code of Chinese Ideograms Set for Information Interchange—the Second Supplementary Set* (『信息交换用汉字编码字符集—第二辅助集』). Technical Standards Press. 1987.

———. GB 7590-87 *Code of Chinese Ideograms Set for Information Interchange—the Fourth Supplementary Set* (『信息交换用汉字编码字符集—第四辅助集』). Technical Standards Press. 1987.

———. GB 8565.1-88 *Information Processing—Coded Character Sets for Text Communication—Part 1: General Introduction* (『信息处理—文本通信用编码字符集—第一部分—总则』). Technical Standards Press. 1988.

———. GB 8565.2-88 *Information Processing—Coded Character Sets for Text Communication—Part 2: Graphic Characters* (『信息处理—文本通信用编码字符集—第二部分—图形字符集』). Technical Standards Press. 1988.

———. GB 8565.3-88 *Information Processing—Coded Character Sets for Text Communication—Part 3: Control Functions for Page-Image Format* (『信息处理—文本通信用编码字符集—第三部分—按页成象格式用控制功能』). Technical Standards Press. 1989.

———. GB 12034-89 *32×32 Dot Matrix Fangsongti Font Set and Data Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 32×32 点阵仿宋体字模集及数据集』). Technical Standards Press. 1990.

———. GB 12035-89 *32×32 Dot Matrix Kaiti Font Set and Data Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 32×32 点阵楷体字模集及数据集』). Technical Standards Press. 1990.

———. GB 12036-89 *32×32 Dot Matrix Heiti Font Set and Data Set of Chinese Ideograms for Information Interchange* (『信息交换用汉字 32×32 点阵黑体字模集及数据集』). Technical Standards Press. 1990.

———. GB 12052-89 *Korean Character Coded Character Set for Information Interchange* (『信息交换用朝鲜文字编码字符集』). Technical Standards Press. 1990

———. GB/T 12345-90 *Code of Chinese Ideogram Set for Information Interchange Supplementary Set* (『信息交换用汉字编码字符集一辅助集』). Technical Standards Press. 1990.

———. GB 13000.1-93 *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane* (『信息技术一通用多八位编码字符集 (UCS)一第一部分：体系结构与基本多文种平面』). Technical Standards Press. 1994.

———. GB/T 15834-1995 *Use of Punctuation Marks* (『标点符号用法』). Technical Standards Press. 1996.

———. GB/T 15835-1995 *General Rules for Writing Numerals in Publications* (『出版物上数字用法的规定』). Technical Standards Press. 1996.

———. GB 16794.1-1997 (『信息技术一通用多八位编码字符集 48 点阵字形』). Technical Standards Press. 1998.

———. GB 18030-2000. *Information Technology—Chinese Ideograms Code Character Set for Information Interchange—Extension for the Basic Set* (『信息技术 信息交换用汉字编码字符集 基本集的扩充』). Technical Standards Press. 2000. Obsoleted by GB 18030-2005.

———. GB 18030-2005. *Information Technology—Chinese Coded Character Set* (『信息技术 中文编码字符集』). Technical Standards Press. 2005. Obsoletes GB 18030-2000.

Vietnam Standards Institute. TCVN 5712:1993 *Công Nghệ Thông Tin—Bộ Mã Chuẩn 8-Bit Kí Tự Việt Dùng Trong Trao Đổi Thông Tin (Information Technology—Vietnamese 8-Bit Standard Coded Character Set for Information Interchange).* 1993.

———. TCVN 5773:1993 *Công Nghệ Thông Tin—Bộ Mã Chuẩn 16-Bit Chữ Nôm Dùng Trong Trao Đổi Thông Tin (Information Technology—Nom 16-Bit Standard Code Set for Information Interchange).* 1993.

———. TCVN 6056:1995 *Công nghệ thông tin—Bộ Mã Chuẩn 16-Bit Chữ Nôm Dùng Trong Trao Đổi Thông Tin—Chữ Nôm Hán (Information Technology—Nom 16-Bit Standard Code for Information Interchange—Han Nom Character).* 1995.

X/Open Consortium. *X/Open CAE Specification: Commands and Utilities, Issue 4, Version 2.* X/Open Company Limited. 1994. ISBN 1-85912-034-2. X/Open Document Number C436.

———. *X/Open CAE Specification: File System Safe UCS Transformation Format (UTF-8).* X/Open Company Limited. 1995. ISBN 1-85912-082-2. X/Open Document Number C501.

———. *X/Open CAE Specification: System Interfaces and Headers. Issue 4, Version 2.* X/Open Company Limited. 1994. ISBN 1-85912-037-7. X/Open Document Number C435.

———. *X/Open CAE Specification: System Interface Definitions. Issue 4, Version 2.* X/Open Company Limited. 1994. ISBN 1-85912-036-9. X/Open Document Number C434.

———. *X/Open Guide: Internationalisation Guide. Version 2.* X/Open Company Limited. 1993. ISBN 1-85912-002-4. X/Open Document Number G304.

Xerox Corporation. *Xerox Character Code Standard 2.0.* Xerox Systems Institute. 1990. Xerox part number XNSS 059003.

Periodicals

Chinese and Oriental Languages Information Processing Society (COLIPS). *Journal of Chinese Language and Computing (Communications of COLIPS).* Published quarterly. ISSN 0219-5968.

Computing Japan. LINC Media, Incorporated. Published from 1994 until 1999. ISSN 1340-7228.

Chinese Language Computer Society (CLCS). *International Journal of Computer Processing of Oriental Languages (IJCPOL).* World Scientific Publishing. Published quarterly. ISSNs 0219-4279 (print) & 1793-6748 (online).

The Globalization Insider. Localization Industry Standards Association (LISA). Published monthly. ISSN 1420-3693.

『情報処理』. Information Processing Society of Japan. Published monthly.

『정글』 (*Jungle*). Yoon Design Institute. Published quarterly. Bar-2557.

Language International. John Benjamins Publishing Company. Published bi-monthly from 1989 until 2002. ISSN 0923-182X.

MultiLingual. MultiLingual Computing, Incorporated. Published bimonthly. ISSN 1523-0309.

The Perl Journal. Published quarterly from 1994 until 1999. ISSN 1087-903X.

『最新ワープロ大百科』. 実業之日本社. Published from 1986 until 1993.

SESAME Bulletin. Sesame Computer Projects. ISSN 0950-2025.

The Seybold Report. RISI. Published twice a month. ISSN 1533-9211.

TRONWARE. *Personal Media Corporation. Pubished bi-monthly.*

TUGboat. TEX Users Group. Published three times per year. ISSN 0896-3207.

U&lc Online. International Typeface Corporation. Published quarterly. ISSN 0362-6245.

World Wide Web Journal. O'Reilly Media, Incorporated. Published quarterly. ISSN 1085-2301.

『中文信息』(*Chinese Information Processing*). Chinese Information Processing Society. Published bi-monthly. ISSN 1003-9082.

Articles, Papers, and Technical Notes

Adobe Systems Incorporated. *Glyph Bitmap Distribution Format (BDF) Specification.* Adobe Developer Support. Adobe Systems Technical Note #5005.

———. *Adobe CMap and CIDFont Files Specification.* Adobe Developer Support. Adobe Systems Technical Note #5014.

———. *The Type 1 Font Format Supplement.* Adobe Developer Support. Adobe Systems Technical Note #5015.

———. *Adobe-Japan1-6 Character Collection for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5078.

———. *Adobe-GB1-5 Character Collection for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5079.

———. *Adobe-CNS1-5 Character Collection for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5080.

———. *CID-Keyed Font Technology Overview.* Adobe Developer Support. Adobe Systems Technical Note #5092.

———. *Adobe-Korea1-2 Character Collection for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5093.

———. *Adobe CJKV Character Collections and CMaps for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5094.

———. *Adobe-Japan2-0 Character Collection for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5097. Obsoleted by Adobe Systems Technical Note #5078.

———. *Building CMap Files for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5099.

———. *SING Glyphlet Production: Tips, Tricks & Techniques.* Adobe Developer Support. Adobe Systems Technical Note #5148.

———. *OpenType-CID/CFF CJK Fonts: 'name' Table Tutorial.* Adobe Developer Support. Adobe Systems Technical Note #5149.

———. *CID-Keyed Font Installation for PostScript File Systems.* Adobe Developer Support. Adobe Systems Technical Note #5174.

———. *CID-Keyed Font Installation for ATM Software.* Adobe Developer Support. Adobe Systems Technical Note #5175.

———. *The Compact Font Format Specification.* Adobe Developer Support. Adobe Systems Technical Note #5176.

———. *The Type 2 Charstring Format.* Adobe Developer Support. Adobe Systems Technical Note #5177.

———. *Building PFM Files for PostScript-Language CJK Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5178.

———. *CID-Keyed sfnt Font File Format for the Macintosh.* Adobe Developer Support. Adobe Systems Technical Note #5180.

———. *PostScript Language Extensions for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5213.

———. *Application Support for PostScript CJK Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5640.

———. *Enabling PDF Font Embedding for CID-Keyed Fonts.* Adobe Developer Support. Adobe Systems Technical Note #5641.

———. *CID Font Tutorial. Adobe Developer Support.* Adobe Systems Technical Note #5643.

———. *AFDKO Version 2.0 Tutorial: mergeFonts, rotateFont & autohint.* Adobe Developer Support. Adobe Systems Technical Note #5900.

———. *Special-Purpose OpenType Japanese Font Tutorial: Kazuraki.* Adobe Developer Support. Adobe Systems Technical Note #5901.

Breen, Jim. *A Japanese Electronic Dictionary Project (Part 1: The Dictionary Files), Technical Report 93/13,* Department of Robotics & Digital Technology, Monash University, November 1993.

Dillard, Troy & Ken Lunde. "Japanese Text Processing and Electronic Mail on the IBM PC and Macintosh." Sesame Computer Projects. *SESAME Bulletin*, Summer 1992, Volume 5, Part 2, pages 40–48.

Huang, Jack Kai-tung. *Status and Font Samples of Digitized Chinese (Hanzi) Font Manufacturers in Taiwan, 1993.* October, 1993.

Liu, Yucheng. *Chinese Information Processing.* MS Thesis. University of Nevada, Las Vegas. 1995.

Lunde, Ken. *CJK.INF.* Distributed and maintained electronically from 1995 until 1996.

———. "Cross-Locale CJKV Code Conversion." Thirteenth International Unicode Conference, San Jose, California, September 8–11, 1998.

———. "Accessibility of Unencoded Glyphs." Thirteenth International Unicode Conference, San Jose, California, September 8–11, 1998.

———. "The Design of an Extended Japanese Character Set." Ninth International Unicode Conference, San Jose, California, September 4–6, 1996.

———. "Unicode/CJK Font Support in PostScript." Seventh International Unicode Conference, San Jose, California, September 14–15, 1995.

———. "The History of the Japanese Character Set and its Encoding." *CPCOL*, June 1993, Volume 7, Number 1, pages 85–94.

———. "Electronic Transfer of Japanese." *ATArashii*, September/October 1990, Volume 4, Number 5, pages 19–27.

———. "Using Electronic Mail as a Medium for Foreign Language Study and Instruction." *CALICO Journal*, March 1990, Volume 7, Number 3, pages 68–78.

———. *JAPAN.INF: Electronic Handling of Japanese Text.* Distributed and maintained electronically from 1989 until 1992.

Morita, Masasuke. "Japanese Text Input System." *IEEE Computer*, May 1985, Volume 18, Number 5, pages 29–35.

———. "Development of New Keyboard Optimized from Standpoint of Ergonomics. Work with Computers: Organizational, Management, Stress and Health Aspects." Proceedings of the Third International Conference on Human-Computer Interaction, September 18– 22, 1989, Volume 1, pages 595–603.

Miyazawa, Akira. "Character Code for Japanese Text Processing." *Journal of Information Processing.* 1990, Volume 13, Number 1, pages 2–9.

西村恕彦.「漢字の JIS」.『標準化ジャーナル』. 1978.5, pages 3–8.

野村雅昭.「JIS C 6226 情報交換用漢字符号系の改正」.『標準化ジャーナル』. 1984.3, pages 4–9.

Open Software Foundation, UNIX International, and UNIX System Laboratories Pacific. *OSF, UI, and USL Standardize on Japanese Language Support.* UI-OSF-USLP Joint Announcement. Press release dated December 12, 1991.

Schilke, Steffen. *Japanization—An Introduction to Software Japanization.* Thesis. Summer, 1992.

田嶋一夫.「JIS 漢字表の利用上の問題―漢字処理システムにおける漢字のデザインと管理」.『情報管理』. 1979. Volume 21, Number 10, pages 753–761.

内田富雄.「JIS X 0212 の制定」.『標準化ジャーナル』. 1990.11.

RFCs

Keep in mind that RFCs are easily accessed through the Web, through the use of a convenient template URL whereby the string "XXXX" is replaced by the four-digit RFC number.*

Alvestrand, Harald. *Tags for the Identification of Languages.* RFC 3066. January 2001. Obsoletes RFC 1766. Obsoleted by RFCs 4646 & 4647.

Berners-Lee, Tim & Daniel Connolly. *Hypertext Markup Language—2.0.* RFC 1866. November 1995. Obsoleted by RFC 2854.

Choi, Uhhyung et al. *Korean Character Encoding for Internet Messages.* RFC 1557. December 1993.

Connolly, Daniel & Larry Masinter. *The 'text/html' Media Type.* RFC 2854. June 2000. Obsoletes RFCs 1866 & 2070.

Freed, Ned at al. *Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures.* RFC 2048. November 1996.

Freed, Ned & Nathaniel Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* RFC 2045. November 1996.

———. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types.* RFC 2046. November 1996.

———. *Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples.* RFC 2049. November 1996.

Freed, Ned & Jon Postel. *ICANN Charset Registration Procedures.* RFC 2278. January 1998. Obsoleted by RFC 2978.

Goldsmith, Deborah & Mark Davis. *UTF-7: A Mail-Safe Transformation Format of Unicode.* RFC 1642. July 1994. Obsoleted by RFC 2152.

———. *UTF-7: A Mail-Safe Transformation Format of Unicode.* RFC 2152. May 1997. Obsoletes RFC 1642.

Alvestrand, Harald. *Tags for the Identification of Languages.* RFC 1766. March 1995.

Klensin, John et al. *SMTP Service Extension for 8bit-MIMEtransport.* RFC 1652. July 1994. Obsoletes RFC 1426.

Lee, Fung Fung. *HZ—A Data Format for Exchanging Files of Arbitrarily Mixed Chinese and ASCII Characters.* RFC 1843. August 1995.

Levinson, Ed. *SGML Media Types.* RFC 1874. December 1995.

* *http://www.ietf.org/rfc/rfcXXXX.txt*

Moore, Keith. *Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text.* RFC 2047. November 1996.

Murai, Jun et al. *Japanese Character Encoding for Internet Messages.* RFC 1468. June 1993.

Ohta, Masataka. *Character Sets ISO-10646 and ISO-10646-J-1.* RFC 1815. July 1995.

Ohta, Masataka & Ken'ichi Handa. *ISO-2022-JP-2: Multilingual Extension of ISO-2022-JP.* RFC 1554. December 1993.

Phillips, Addison & Mark Davis. *Tags for Identifying Languages.* RFC 4646. September 2006. Obsoletes RFC 3066.

———. *Matching of Language Tags.* RFC 4647. September 2006. Obsoletes RFC 3066.

Tamaru, Kenzaburo. *Japanese Character Encoding for Internet Messages.* RFC 2237. November 1997.

Vietnamese Standardization Working Group. *Conventions for Encoding the Vietnamese Language—VISCII: VIetnamese Standard Code for Information Interchange—VIQR: Vietnamese Quoted-Readable Specification. Revision 1.1.* RFC 1456. May 1993.

Wei, Ya-Gui et al. *ASCII Printable Characters-Based Chinese Character Encoding for Internet Messages.* RFC 1842. August 1995.

Whitehead, E. James Jr. & Makoto Murata. *XML Media Types.* RFC 2376. July 1998.

Yergeau, François. *UTF-8, a transformation format of Unicode and ISO 10646.* RFC 2044. October 1996. Obsoleted by RFC 2279.

———. *UTF-8, a transformation format of ISO 10646.* RFC 2279. January 1998. Obsoletes RFC 2044. Obsoleted by RFC 3629.

———. *UTF-8, a transformation format of ISO 10646.* RFC 3629. November 2003. Obsoletes RFC 2279.

Yergeau, François et al. *Internationalization of the Hypertext Markup Language.* RFC 2070. January 1997. Obsoleted by RFC 2854.

Zhu, Haifeng at al. *Chinese Character Encoding for Internet Messages.* RFC 1922. March 1996.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

Ken Lunde[*] was born in Madison, Wisconsin, on the 12th day of August of the year 1965, and was subsequently raised by his parents, Vernon and Jeanne, in nearby greater-metropolitan Mount Horeb. His first post–high school educational experience was studying the Russian language for the United States Army Reserve at the Defense Language Institute–Foreign Language Center (DLI-FLC) in Monterey, California, for all but a couple of weeks of 1984. He was a member of the now disbanded 247th Military Intelligence Detachment for all nine years of his military experience. Ken entered The University of Wisconsin–Madison (UW–Madison) in 1985 as a freshman, graduated with a Bachelor of Arts degree in linguistics in August of 1987, received his Master of Arts degree in linguistics in May of the following year, and then finally earned his Doctor of Philosophy degree—in linguistics yet again, but this time with a minor in Japanese—in May of 1994. His Ph.D. dissertation was entitled *Prescriptive Kanji Simplification*. Ken began his career at Adobe Systems in 1991—before even contemplating his dissertation and before writing his first book, entitled *Understanding Japanese Information Processing* (O'Reilly Media, 1993)—and is currently a Senior Computer Scientist in CJKV Type Development.

Ken's Japanese penname is 小林劍 (*kobayashi ken*). The surname *Lunde* is of Viking origin, and means "small woods" or "grove." Perhaps by sheer coincidence—or rather by choice—the Japanese surname 小林 (*kobayashi*) conveys these same meanings. The Japanese given name 劍 (*ken*) was chosen phonemically, and from his fondness for cutlery and other tools with sharp edges. In retrospect, Ken is very pleased that he chose the kanji 劍 for his name because it is an excellent example of an ideograph that has many variant forms in Japan's JIS X 0208:1997 character set standard, specifically 劍, 劒, 劔, 劒, and 釼. His wife even coined 劍, which is an unencoded variant. When Ken is in a nostalgic mood, he sometimes prefers to use the traditional form, specifically 劍 instead of 劍. And, when visiting China he could potentially use the simplified form 剑. The possibilities seem endless. That, after all, is the nature of ideographs.

Ken resides and works in San Jose, California. He is one of the few people who doesn't need to commute far to get to work. That may change when he and his family move to the Mount Shasta area in a few years. His interests include reading, writing, photography, firearm marksmanship, hunting, cooking, fine wine, listening to a wide variety of music, and spending quality time with friends and his family. He very much enjoys the outdoors, and treasures every chance he gets to hunt with his father.

Although Ken is deeply intrigued by Japanese—and obviously CJKV—computing, seafood is, quite astonishingly, not among his most favorite foods. This does not imply that he hates seafood, and on the contrary, he very much enjoys shrimp and a wide variety of fish. It deserves to be pointed out that he has never eaten blowfish. And, speaking of blowfish….

[*] *http://lundestudio.com/*

Colophon

The animal on the cover of *CJKV Information Processing* is a blowfish, also known as a globefish, swellfish, puffer, and porcupine fish. It exists in tropical waters throughout the world. In Japan it is known as *fugu* (河豚 *fugu*), and is a treasured delicacy, usually eaten raw in thin slices. While parts of the blowfish are deliciously narcotic, other parts contain a deadly toxin. Because of this, only specially certified and licensed chefs are allowed to prepare the fish for people to eat. The skin of the blowfish is often used for making lanterns and other decorative items.

The cover image is a 19th-century engraving from the *Dover Pictorial Archive*. The text font is Adobe Systems' Minion Pro, an Adobe Original typeface design; the heading font is their Myriad Pro Condensed, also an Adobe Original; the code font is LucasFont's TheSansMonoCondensed. Unless otherwise noted, Simplified and some Traditional Chinese text (from GB standards) are set in Adobe Systems' Adobe 宋体 Std L (AdobeSongStd-Light) typeface design; most Traditional Chinese text (from CNS and Big Five standards) and Vietnamese text (from TCVN standards) are set in Arphic Technology's 文鼎中明 CNS11643 (MingTiEG-Medium) typeface design; other Traditional Chinese text (from the Hong Kong SCS standard) is set in Adobe Systems' Adobe 明體 Std L (AdobeMingStd-Light) typeface design; Japanese text (from JIS standards) is set in Adobe Systems' 小塚明朝 Pr6N R (KozMinPr6N-Regular) typeface design, an Adobe Original; and Korean text (from KS standards) is set in Adobe Systems' Adobe 명조 Std M (AdobeMyungjoStd-Medium) typeface design. Non-standard glyphs were implemented as SING glyphlets. Most of these typefaces are available for purchase from Adobe Systems, bundled with its applications, or are available from their respective type foundries.

The inside layout was designed by David Futato, and implemented in Adobe InDesign by Ron Bilodeau. The illustrations were created in FreeHand MX by Robert Romano. Additional production work was provided by Rachel Monaghan. Rachel Monaghan reviewed the index. All aspects of book production, from text entry to page composition, including indexing, was done by the author on an Apple MacBook Pro laptop computer running Mac OS X and Adobe InDesign CS3-J.