Character Encoding: An Introduction for E-Discovery Professionals

By John Tredennick and Larry Barela
Contents

Character Encoding: An Introduction for E-Discovery Professionals ...... 1
ASCII Gets Extended ............................................................................. 3
Double Byte Languages ......................................................................... 4
Unicode: The Modern Standard.............................................................. 6
UTF-8: The Leading Brand of Unicode................................................... 7
Dealing with Non-Unicode ...................................................................... 8
Language Detection and Encoding are not the Same ......................... 9
Why it Matters to Understand Encoding ........................................... 10
“There is something wrong with your system,” the angry lawyer on the phone said to Laura, the project consultant working on her case. “I am looking at the screen and all I see are a bunch of question marks and boxes,” she continued, getting more exasperated by the minute. “How am I supposed to review these documents if I can’t read the words?”

“Let me see if I can help,” Laura answered, trying to be as calm as possible. “Perhaps your computer is just using the wrong code page to display the text. If so, we can probably fix the problem with a mouse click,” she offered hopefully. “If not, there could be a problem with how your data was collected or processed.”

“A code page?” responded the caller. “What the heck is a code page?”

Our caller’s confusion was not unusual. After all, most of us went to school to study law, not technology. Many lawyers still have little interest in knowing more about technology than how to turn on their computers.

But legal-technology professionals do need to know about code pages and character encoding, particularly as multi-language discovery becomes more common. The good news is that the subject isn’t that difficult. It is just a matter of taking it step-by-step.

In this article, we provide a primer on what you need to know. We start by reviewing the development of ASCII—the basic standard for English-language programs—and discuss its limitations for non-English languages. Later, we introduce you to Unicode, the standard that evolved to address ASCII’s limitations. Finally, we explain why all this is important for you to understand.

ASCII: The Base for English-Language Programs

To get a handle on character encoding and code pages, we need to start at the beginning, with ASCII, which, for our purposes, was the first character encoding set created for computing.

ASCII, pronounced “ask-ee” is the acronym for American Standard Code for Information Interchange. First developed in 1963 and finalized as a standard in 1968, ASCII was a system to encode the basic characters used by computers to communicate with people.
The task was to create a universal way to represent all of the basic characters one needed to use a computer—from writing programming code to drafting a research memo using a word processing program. And, because computers run off binary code (bits and bytes), ASCII likewise needed to be expressed in bits and bytes.

In ASCII, each character you see on a page (and others you can’t see) is presented to the computer not as letters but as a “byte” of code. A byte consists of eight individual “bits” that are either a “1” or a “0.” Thus:

- The letter “A” would be encoded in seven bits as: 100 0001.
- The letter “B” is: 100 0010.
- The letter “C” is: 100 0011.

And so on through the alphabet (large and small letters). ASCII also includes the 10 possible number values (0-9) along with standard punctuation characters ($ % * & + =, etc.). It reserves the first 32 characters to control things such as tabs, line feeds and carriage returns.

You probably noticed that the letters shown above only use seven bits rather than the eight that make up a byte of code. As a historical anomaly, ASCII’s drafters felt that 128 characters would be plenty to represent the letters, numbers and other “control characters” they would need. Yet most computers required eight bits as a minimum unit. So, they used the last bit for error checking. They figured they would never need it for anything else.

The ASCII system worked great in a world that spoke English and has held up well for more than four decades. Over the years, it became the base for text transcripts (“Could I get an ASCII copy of the transcript please?”), the core of most word processing programs, and the heart of most of the programming code used in litigation support applications.

There was only one problem with seven-bit ASCII. Having 128 possible combinations of 1s and 0s works fine if your alphabet only has 26 letters. But
what if you want to compute in French or German or Russian or Hebrew? Even more fun, what if you are one of the billions of Chinese or Japanese or other Asian-language speakers blessed with a language that has tens of thousands of characters?

**ASCII Gets Extended**

Not willing to change their mother language just to use computers, non-English-speaking programmers began developing their own character sets, extending the ASCII standard. Simply moving from seven to eight bits doubled the range of characters to 256, which helped for many languages. Most included the first 128 characters from ASCII but added other characters they needed in the extended range. This extra set of character points was often called “upper” or “extended” ASCII.

At the same time, English-speaking programmers started using the additional characters to support line drawings and horizontal and vertical bars so you could make spiffy drawings on your page. The IBM PC, for example, offered the OEM character set, which included some of the accented characters needed for European languages along with a bunch of line drawing characters that programmers used to make boxes and other rough graphics on those primitive DOS screens.

As you can imagine, this got a bit crazy. People from different countries started creating proprietary code sets to handle each of their languages. For the most part, they reserved the first 128 characters for the original English characters, which meant that English worked everywhere. After that, it was, “Katy bar the door!” There were thousands of code sets to choose from.

That meant your computer program had to know which encoding (code page) was being used. Why? Because different encodings used the same numerical value to express different characters. For example, here is how character 162 in upper ASCII displays using several different encodings:

<table>
<thead>
<tr>
<th>U.S (437)</th>
<th>Greek (1253)</th>
<th>Cyrillic (1251)</th>
<th>Arabic (1256)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ö</td>
<td>À</td>
<td>Ў</td>
<td>ⶀ</td>
</tr>
</tbody>
</table>

What happened if your computer didn’t know which code page was being used? Simply put, you would get gibberish. The computer would render either nonsensical characters from a different language or those bothersome question marks and boxes that bedeviled our user in this story. Why question marks and boxes? Because the characters did not exist in the code set the computer was
using to render them. Lacking a valid character to display, the computer displays question marks or, in some cases, funny boxes.

For these special encodings to work well across borders, computer operating systems had to be able to recognize and handle each one and have the proper fonts loaded to display them. In addition, programmers had to properly label their code sets, which did not always happen. That left your computer confused and left you with those strange characters on your screen.

Double Byte Languages

To make matters even more interesting, some languages had so many characters that they couldn’t begin to fit within the narrow confines of upper ASCII. After all, the addition of the eighth bit to the ASCII standard provided only an additional 128 characters. That may work fine for French or Spanish, but Chinese written language has something like 65,000 symbols. Japanese and Korean also have a whole lot more than 128 symbols between them.

Lacking any other alternative, Asian programmers started using a second byte to express those languages. Adding another byte to describe the written symbols or characters you need changes the picture dramatically. Instead of 256 possibilities, you now have 65,536 options to play with. The number is simply a matter of two to the power of 16. Most of us remember how quickly the numbers add up when you start with two (representing a 0 or 1, which are the two possible states of an individual bit). Two to the eighth power is 256. Two to the 16th power is 65,536.

This resulted in the special “double byte” encodings that you hear about if you work with the Asian languages. “JIS” and “Shift JIS” are two such encodings from Japan. “Chinese Traditional” (aka Big 5) and “Simplified Chinese” are used to express the Chinese language. All are special encodings that will be difficult to view or understand unless your computer recognizes the proprietary code page.

As an illustration, here is a text page that is not properly rendered. You see some English here but also boxes and other strange characters.
The problem is that we have chosen the wrong encoding. Actually, the computer chose the wrong encoding. In this case, Internet Explorer made the mistake in detecting the intended encoding using its “Auto-Select” function.

Here is how the same page looks after changing the browser encoding setting to Chinese Traditional (Big 5). In IE, you do this by right-clicking the page and choosing the encoding option from the ones listed.
As you see, the boxes and question marks are gone. Instead you see well-formed Chinese characters, in exactly the same form as when they were created.

**Unicode: The Modern Standard**

All this special encoding worked well up to a point. But in an age of globalization, these multiple codes created a sort-of technological Babel, confounding our ability to easily share and process data across borders.

In the early days of encoding, people were less concerned with sending documents from country to country. E-mail wasn’t anywhere near the universal communications medium it is today. Google hadn’t been invented and facebooks were still something students passed around college dorms.

By the early- to mid-1990s, however, people started feeling the pinch of all this encoding. A group of visionaries realized that the world needed some kind of universal encoding that could go beyond ASCII and embrace all possible languages. That realization was the impetus for the consortium that developed the Unicode Standard, the modern foundation for handling foreign language documents around the world.

Unicode was a big leap forward. Simply put, the drafters wanted to create a single character set that could be used to express every writing system out there. Having a sense of humor, they even made it big enough to encompass Esperanto (the failed universal language) and a Galactic language such as Klingon.

A lot of people think Unicode is another way to say double byte. That is not really true. Unicode gave a numeric value (hexadecimal actually) to each individual letter or character out there. The letter A is expressed as U+0041. The word “hello” is expressed this way:

U+0048 U+0065 U+006C U+006C U+006F

Since there are a lot more than 65,536 possible characters to express, Unicode can extend well beyond
two bytes. In fact, the consortium has assigned more than 100,000 characters and has the ability to use up to four bytes if needed. With four bytes, the consortium has more than four-billion possible characters to play around with. That will cover English, Chinese, Arabic and a whole lot more.

A key fact to know about Unicode is this: The consortium kept the first 256 characters from the original and extended ASCII standards. By doing so, they made it easy for the Western world to adopt the new standard. All of their old encodings would work just fine. At the same time, they forced the rest of the world to change if they wanted to get on the Unicode bandwagon.

The core point about Unicode is that it can handle such a wide range of characters because it can use more than one byte to express characters. In the years since Unicode was introduced, it has become a global standard and is the encoding used in modern operating systems such as Windows, XML, the .NET framework, JAVA and the Mac OS X, among many others.

**UTF-8: The Leading Brand of Unicode**

The original Unicode offerings were built around two bytes of data, which, as you will recall, supported 65,000 or so possible characters. Some called this Unicode 2 (Unicode Transformation Format) for two bytes or UTF-16 for the 16 bits that are encompassed in two bytes of data.

This caused consternation among English-language programmers. Why? Because they faced the possibility of having to write two bytes of code to express language that only needed one byte. Why waste the extra bytes? Better to stick to ASCII programming, many thought, particularly when they weren’t programming for the international set.

The sentiment was strong enough that the consortium adopted a Unicode variant called UTF-8. This version allowed programmers to use a single byte to express the first 256 characters represented in ASCII. When you needed more, you simply added a second byte along with a signal that the computer should read two bytes for the next character rather than one. That covered the first 65,000 characters. For certain Chinese characters, UTF-8 encoding extends to three bytes in size. As we mentioned earlier, there is the possibility of a fourth byte, but its use is currently reserved.

Today, UTF-8 is the worldwide standard. Because of its versatility and consistency with ASCII, UTF-8 is steadily becoming the preferred encoding for e-
mail, web pages, and other places where multi-language characters are stored or streamed.

Dealing with Non-Unicode

Even so, Unicode has yet to be adopted by everyone. Many Asian programs (including e-mail systems as well as HTML and text pages) are still expressed using proprietary code pages. Japanese e-mail systems often use a proprietary format like Shift-JIS for e-mail text, for example. Others still run older Exchange programs in a non-Unicode format, which can fool people who expect Exchange collections to be in UTF-8 format. Collection and processing of this data can be mangled easily if your computer is not set to recognize the proprietary format being used.

To make matters worse, while Exchange and its PST format is Unicode compliant, for some unknown reason the popular MSG format itself is not. The body of the e-mail messages you extract will typically be in Unicode format (although the administrator can choose otherwise). However, certain metadata fields are not kept in Unicode. Specifically, the Subject, From, To, CC and BCC fields are not kept in Unicode when they are expressed in the MSG format (they are kept in Unicode when stored in Exchange itself).

This is important for e-discovery professionals who have to collect, process, index and make this data viewable. If you extract MSG files from Exchange without setting your computer to handle the local encoding properly, you will likely mangle your data. Viewers will see the body of the e-mail properly but may see those bothersome question marks, boxes and other characters that result from improper code mapping. By this time, it is too late to fix the problem. You have to re-collect or at the least reprocess the data.

At Catalyst, we built an automated processing system that allows our clients and partners to submit raw files (including PSTs, NSFs and loose e-mail) directly to our system for processing. Where Asian data is collected, we encourage our users to specify the locale settings used to collect the data (which hopefully match the settings on the computer being harvested). By doing so, we can automatically route the collected data to servers set to the same locale. That way, non-Unicode data won’t be mangled during processing and can be sent on for indexing and review.

Beyond processing, it is equally important that your search and review platform be able to recognize and deal with non-Unicode files as well as the typical ASCII and Unicode data. Catalyst’s system is based on the FAST platform which
includes special technology to recognize the special encodings used in non-Unicode documents and convert them to Unicode for indexing and search. The process isn’t always perfect because there is no perfect way to recognize which encoding has been used. The hope is that the programmer has specified which encoding is being used in the document, but that doesn’t always happen. Sometimes, the computer simply has to guess.

Thus, when you are dealing with Asian and other languages, you may find you have a problem with non-Unicode documents even if your system is “Unicode compliant.”

Language Detection and Encoding are not the Same

Legal professionals sometimes confuse language detection with encoding. They are not the same. The difference is important.

Let’s start with language detection. A Unicode document, as you now know, may include text from many languages all in a single encoding. After all, that was the purpose for Unicode, to provide a single code page that could express every language.

So, when the computer is ingesting a Unicode document, it has to find a way to determine whether the document is in Spanish, English, French or some other language. You might think that Unicode would solve that problem, given that it assigns characters in each language to specific code points. However, the problem is that the different languages use common letters and often common words, sometimes with different meanings. How do we tell which language is being used?

For example, in English, the word “chat” means to talk informally. In French, chat means cat. If we see just “chat,” how do we know which language it is? Unicode doesn’t help us here because both languages use the common characters. The same is true with Asian as well as other languages. Japanese uses many characters taken from the written Chinese but sometimes gives them different meanings. And Chinese has two written forms: traditional and simplified. Many of the characters are the same but the meanings and usage can be different.

Our system uses a special language detection program from a company called Basis Technology to try and recognize the different languages being used in the documents as they are being processed and indexed. How they do it is a closely guarded secret. We know that the program analyzes the letters and other symbols being used, considers how they are used and whether there are special
characters that are unique to a specific language, and even looks up words in a variety of dictionaries to determine the most likely language.

It helps to have more text when you are trying to determine languages used in a document. If you only have a few Chinese or Japanese symbols, the system might be confused, particularly if the symbols are common to both languages. With a paragraph of text, the system is less likely to be confused.

We ask our users to specify the language in which they are searching when the CJK languages (Chinese, Japanese or Korean) are involved. Unless you know the intended language, you can’t properly tokenize the characters so that they match the pages being searched. (Tokenization, which essentially means breaking up text properly into word units, is a topic for another day.

What about non-Unicode encoding? If we can detect the encoding used in a document, will that tell us the language being used? Unfortunately, the answer is no. Knowing a document’s encoding can help narrow down the possibilities but won’t always tell you the language.

The reason for this is that a single encoding can encompass a number of languages. For example, let’s say we have a non-Unicode document with a code page that is used to construct Arabic, Farsi and Urdu. Assume we detect the encoding correctly, what does that tell us about the language being used? If we consider Arabic alone, it could be a number of languages. Here are a few possibilities: Persian, Urdu, Pashto, Baloch, Malay; Fulfulde-Pular, Hausa, and Mandinka (all in West Africa); Swahili (in East Africa); Brahui (in Pakistan); Kashmiri, Sindhi, Balti, and Panjabi (in Pakistan); Arwi (in Sri Lanka and Southern India), Chinese, Uyghur (in China and Central Asia); Kazakh, Uzbek and Kyrgyz (all in Central Asia); Azerbaijani (in Iran), Kurdish (in Iraq and Iran), Belarusian (among Belarusian Tatars), Ottoman Turkish, Bosniaks (in Bosnia), and Mozarabic.

Which language is being used? That is the province of language detection software. Encoding only helps get you started.

**Why it Matters to Understand Encoding**

In this flat-Earth era of globalization, multi-language documents are becoming a standard fixture of the e-discovery landscape. If you represent a multinational, you will undoubtedly be required to collect and review documents from a number of countries to determine relevance and privilege. If you sue a multinational, you will most likely receive non-English documents that you will have to master. Many
documents will contain several languages. An e-mail, for example, might easily include combinations of Japanese, Chinese and English.

If you haven’t encountered this yet, you will—and soon. The more you know about the process and the pitfalls, the better prepared you will be.

John Tredennick, a former trial lawyer, is founder and CEO of Catalyst Repository Systems. Larry Barela is Catalyst’s vice president of technology and development and CTO. Learn more about Catalyst at www.catalystsecure.com.