

# Text and Fonts In a Cross-platform Multi-lingual World

## Part One: ASCII and Her Friends

“The problem with standards is that there are so *many* of them.”  
— *Unknown*

Language is one of the hardest things with which a computer has to deal. Numbers are simple; text is hard. That should come as no surprise to the faculty and students in the Humanities Division for whom the daily task of simply typing non-English text into a word processor can sometimes be fraught with computational peril: characters that appear on the screen but that don't print out; files from colleagues at other institutions that open up on the screen as a stream of seeming gibberish; word processing programs that require five keystrokes just to produce a simple “ü”.

How this leaning tower of Babel came to be, and how you can best cope with it is the topic of this article. We'll look back in time to see how the problem began, explore a few simple concepts (like the differences between a character and a glyph and how keyboard mapping works), and then take a look at how some modern languages have collided with computer technology.

### A Brief History of ASCII

The problem began with ASCII: the American Standard Code for Information Interchange. Before ASCII was devised, textual information created on one make of computer (a machine manufactured by, say, Megadata International) would almost certainly not be understood by another computer (say, one manufactured by Digibusiness Incorporated). ASCII was an attempt to solve that communications problem, and it was a good idea at the time (that time being the early 1960's). Unfortunately, things have changed a bit since then.

ASCII was (and still is) a simple code that assigned specific numbers to each textual character: for example, “L” was assigned the number 76, “P” was 80, lower-case “m” was 109, and a comma was 44. All in all, ASCII defined 128 different characters, enough to accommodate the upper- and lower-case letters, the numerals 0-9, the common punctuation marks, and some special control codes that ordered early computer terminals to do things like clear the screen or begin displaying text on a new line. Unfortunately for the rest of the world, the alphabet in question was the Latin alphabet as used by Americans (the “A” in ASCII). Even the British were out of luck: ASCII had a number for “\$” (36 if you're interested) but not one for “£”.

Now, 128 is an interesting number, at least as far as computers are concerned. Remember that computers work with “bits”: 1's and 0's. That is, at their innermost silicon souls, computers really only “understand” these two numbers. To make numbers bigger than 1 you have to add more bits. For example, with two bits you can have four separate combinations of 1's and 0's (00, 01, 10, 11) and so you can express all the numbers between 0 and 3 (don't ask about negative numbers; that's too metaphysical for this discussion). With seven bits you can have exactly 128 different possible combinations of 0's and 1's. Thus, ASCII is what computer science folk refer to as a “7-bit character set”: you only need seven bits of precious computer memory to store any ASCII character. Perfect. . .if you are an American writing in English.

The ASCII 7-bit character set standard came along just in time for the invention of the Internet mail standard. Although most computers back then (and even now) tend to subdivide their memories into 8-bit chunks called “bytes”, seven bits for each character was great as far as the developers of e-mail were concerned: the fewer bits per character, the less time it would take to send a message, and the less “bandwidth” (the amount of information a connection can contain at any one time) each message would consume. Even now, three decades later, the 7-bit character standard lives on, at least in some Internet mail systems, as the following warning message from Pegasus Mail’s preferences window cheerfully tells us: “8-bit data is formally illegal in Internet Mail and although many systems permit it, you should be aware that its use can potentially cause considerable problems for Internet sites using older mail systems and gateways.”

America was not alone in adopting a 7-bit character set standard either: for example, the Russian government developed their own 7-bit character standard at about the same time. Called GOST-13052, this character set used the same numbers for most punctuation and for numerals as did ASCII, but substituted the letters of the Russian alphabet for the Latin letters—with one interesting twist: the upper-case Cyrillic characters had numbers corresponding to lower-case Latin characters, and vice-versa. (We’ll revisit the various Cyrillic character sets and their tangled history later in this article; more information is available in Roman Czyborra’s “Cyrillic Charset Soup” article at <http://czyborra.com/charsets/cyrillic.html>.)

### **The Key to Keyboards**

ASCII is just one part of the multi-lingual text problem. Keyboards are another. And, once again, historical accident and American industrial predominance are at the heart of the matter.

The keyboard that we all have come to know and tolerate is often called the “QWERTY” keyboard, named after the first six letters of its top row of letter keys. The seemingly bizarre key layout that makes the learning of touch-typing such an adventure stems from earlier mechanical limitations: the keyboard was designed so that early typewriters would not get jammed by a fast typist. Although mechanical typewriters are rather few and far between these days, the QWERTY key layout is still with us on computer keyboards.

The keys on a computer keyboard are really just buttons that, when pressed, send electrical signals to the computer. These signals are interpreted as numbers by the computer, and each key has its own number (or “keycode”). The letters and other symbols on the keys, in fact, are merely decoration; it is software that decides which keycodes correspond to which character codes. Thus, any time you type a letter on a keyboard, you are (at least) two steps removed from the letter that you typed: the electrical signal generated by the key is converted to a number by the computer, and the computer software responsible for such matters then looks that keycode number up in a table (called a “keyboard mapping table”) to see which character code corresponds to that keycode. For example, the letter “A” on the keyboard may generate the keycode numbered 14, which in the keyboard mapping table corresponds to character 65 (the ASCII number assigned to “A”). Change the mapping and the same keycode will map to an entirely different character code.

This is really a very clever scheme, providing a great deal of flexibility. By simply changing the keyboard mapping table you can get a different keyboard layout without having to build a brand-new keyboard. (Of course, changing the keyboard mapping

won't change the letter that is physically silkscreened on top of the key itself. . .clever software can only take you so far.)

Simply put, keyboard mapping tables tell the computer which keycode numbers correspond to which character numbers; character encoding schemes (such as ASCII or GOST-13052) tell the computer which character numbers correspond to which (human readable) characters. Even more simply (and less precisely): mapping tables help you write, encoding schemes help you read.

### **Extending ASCII By A Bit**

The original ASCII, as noted above, was a 7-bit character set. But most computer memory is divided into 8-bit chunks, called “bytes.” Why eight? Without going into the twisty details, eight (or any other number that is a power of two, like sixteen or thirty-two) is as natural a subdivision in binary arithmetic as ten (or any power of ten, like a hundred or a thousand) is in decimal arithmetic. Computers use binary arithmetic. Hence, dividing computer memory into chunks divisible by the number two makes engineers' and programmers' lives easier.

With seven bits, you can have 128 possible combinations. Add just one bit, and you double the number of combinations to 256. Since eight is such a computer friendly number, it makes sense to store each ASCII character in an 8-bit byte. And rather than waste that extra bit, you can extend ASCII to accommodate another 128 characters. Which is just what happened back in the early 1980's, when the introduction of the IBM Personal Computer brought the world an extended ASCII character encoding (also known as a “code page”), which was designated CP437.

IBM's code page 437 contained all the ASCII characters assigned to the character numbers they had always had, and it added a number of non-American-English characters (mostly accented Latin characters), a collection of line-drawing characters that could be used to make charts and tables, and some less commonly used mathematical symbols. Other code pages followed: there were Eastern European code pages, Cyrillic code pages, Icelandic, Hebrew, Thai, and Greek code pages. Apple Computer, starting in 1984, added to the fun, creating its own character set encodings to extend ASCII; the first of which, still with us today, is called MacRoman. Needless to say, all these code pages were identical for the first 128 characters and varied widely among the next 128, although most IBM code pages retained at least some of the line-drawing characters in the same character positions between code pages.

With extended ASCII code pages, a workable, if clumsy, system was beginning to develop that could support most alphabetic languages—at least, inside the computer: displaying these languages or printing them out was another matter.

### **Fonts and Glyphs**

A keyboard mapping table can tell the computer that the “A” key corresponds to the ASCII “A” character, but neither the mapping table nor the code page is responsible for actually showing you that character. An “A” (or any other character) is something like a Platonic entity: it is the idea of an “A” and not the “A” itself. The character as actually drawn or displayed, called a “glyph” by digital font designers, comes from somewhere else. But where?

In the early IBM PCs and similar computers, the actual character shapes were stored on a chip that was part of the computer's display adapter (the circuit board to which the monitor was attached). If you wanted to write, for example, in Icelandic, your computer would need a keyboard mapping table to create an Icelandic keyboard layout, an

Icelandic code page, and a new chip for your computer's display adapter that contained the Icelandic glyphs. Furthermore, these glyphs had to be stored in such a way that they matched the layout of the Icelandic code page you were using. You also needed a word processor that was advanced enough to work with alternate keyboard mappings and character sets. And if you wanted to print the Icelandic document, you needed a printer with printing elements (such as a daisy wheel) that contained the Icelandic glyphs, a printer driver compatible with your word processor that could match up the printer's Icelandic glyphs with the Icelandic code page, and the hardy spirit of an intrepid adventurer. Not a great state of affairs if the next paper you wanted to write was in Polish.

Enter Apple's Macintosh. Rather than using a display adapter containing a chip with a frozen set of display glyphs, this machine was a graphical computer: everything on the screen, text or not, was drawn by software. The Macintosh popularized the idea of software *fonts* (a font is the collection of glyphs that make up a complete character set). By moving the display of glyphs to software, by making the computer, rather than a specialized circuit board, responsible for drawing glyphs on the screen and at the printer, the Macintosh made it possible for digital documents to display multiple typefaces, and ultimately, multiple character sets. A cottage industry of font designers arose, and the desktop publishing business was born.

### **Babel Revisited, Russian Style**

But all was not skittles and beer. The Macintosh was not the only font-savvy computer in the world; the IBM PC was slowly becoming a graphical computer, too. Although Apple had control of its font standards and a rudimentary plan for international language support, IBM had its standards as well. Meanwhile, Microsoft (who supplied the software that powered the IBM PC) was developing Windows with its own set of font standards, and various international and national standards organizations and boards began springing up, creating their own font standards for various languages. Common ground was not easy to come by beyond the realm of the by-now-antiquated ASCII.

An overview of how Cyrillic encoding standards have come and gone over the past couple of decades illustrates the tangle of character set and font standards that has enmeshed most alphabetic languages. (Again, much of the following is drawn from Roman Cyzborra's wonderfully detailed history of Cyrillic character sets at <http://czybgorra.com/charsets/cyrillic.html>).

In 1974 GOST (the Soviet standardization agency, which has survived into the Russian Federation [see <http://www.gost.ru/homepage.nsf>]) produced two code pages: KOI-7 and KOI-8. The two sets of encodings supplanted the early GOST-13052 encoding (described above), and were published as the GOST 19768-74 standard.

GOST's KOI-7 was another 7-bit character set which included only the upper-case Latin characters (numbered just as ASCII had them) and the upper-case Cyrillic characters (assigned to numbers that were used in ASCII for lower-case Latin characters). It was KOI-8 that really broke new ground by becoming the first 8-bit Cyrillic code page. Unfortunately, not all vendors implemented GOST's KOI-8 exactly as specified, and an unofficial standard called KOI8-B, which added a couple of characters, became prevalent by the late 1980's.

Meanwhile, ECMA (a European standards agency <http://www.ecma.ch/>) was trying to bring order out of chaos to a variety of alphabetic languages with its ISO-8859 series of character-set encodings. ECMA's Cyrillic offering tried to retain compatibility with KOI-8 while adding characters from Ukrainian, Serbian, and other languages. The

result, commonly called ECMA-Cyrillic, was published in draft form in 1987 under the more formal designation ISO-IR-111. ECMA-Cyrillic, however, was never adopted as an official standard.

Why? Because GOST was busy coming up with yet another standard. The new standard, GOST-19768-87, was particularly noteworthy because it was *completely incompatible* with KOI-8. This new standard, while making a decade's worth of Russian software potentially obsolete, did have two seemingly significant advantages over KOI-8: it was backed by the full faith and credit of the Soviet government (which was soon to become history), and it stored the Russian alphabet in proper Russian alphabetical order, making it easier for programmers to write sorting routines.

ECMA then returned serve by publishing *its* new standard, ISO-8859-5, in 1988. This standard matched the GOST-19768-87 standard as far as Russian character ordering was concerned, but varied from it in its assignment of non-Russian characters. ISO-8859-5 went on to become the official ECMA standard for Cyrillic, and it is still the official Cyrillic standard today. Unfortunately, most people don't use it.

Instead, a number of other unofficial and incompatible Cyrillic standards have come to dominate the computational landscape. Andrew Chernov created a variation of KOI-8 known as KOI8-R, used largely for e-mail on the Internet. KOI8-U, a Ukrainian variation, is also in use, as is KOI8-Unified, developed by Peter Cassetta, which attempts to provide some degree of compatibility among the different KOIs.

Finally, American companies have gone their own way, disregarding both the official ISO-8859-5 standard and the unofficial, though widely used, KOI variants. Apple Computer, for example, built its own standards with the MacCyrillic and MacUkrainian character sets and associated fonts; similarly, Microsoft came up with its proprietary encoding called Windows-1251. Surprisingly, GOST (which officially supports the ISO-8859-5 standard) uses the Windows-1251 encoding on its own web pages!

What a world.

### **Preview: Unicode, E-Mail, and the Web**

And so, at the end of the 1980's, the plethora of emerging and contradictory text encoding standards had made the art of multi-lingual text processing only more bewildering than it was when ASCII first reared its 7-bit head nearly twenty years earlier. Was there any relief in sight?

Possibly. As the decade closed, a consortium of computer manufacturers and standards organizations began work on a unified encoding standard for all the world's languages; they called their slowly emerging brain-child "Unicode." At roughly the same time in a laboratory in Switzerland, Tim Berners-Lee sat down at his NeXT workstation and began to develop a textual protocol called Hyper-Text Mark-up Language (HTML). HTML, in turn, led to the World Wide Web, which in just a few short years made cross-platform, multi-lingual computing crucially important to tens of thousands of businesses and turned the once-academically-oriented Internet into the biggest shopping mall in the history of civilization.

## **Part Two: It's A Web World After All**

“Do you want it good, or do you want it Tuesday?”  
—Anonymous Hollywood writer

In part one, we reviewed the history of ASCII, the “standard” digital text encoding system developed by computer workers in the early 1960’s, and saw how it became the cracked foundation of a leaning tower of Babel. We also examined the connection between keyboards and character codes, explored the relationship between characters and glyphs, and presented as a cautionary example the twisty saga of how Slavic and ASCII tried to peacefully co-exist on the digital frontier. Then came e-mail and the World Wide Web...

### **The Way We Were**

Throughout the 1970s and most of the 1980s, personal computers in the Humanities were used mostly for creating information, not for exchanging it. Incompatible or incomplete text encoding standards proliferated, but, even though the standards were created to allow text to be transferred from computer to computer, most humanist computer users were still islands scattered in a wide unnavigable sea. Networks of personal computers were relatively rare; modems were an exotic add-on. If a scholar wanted to send a colleague her latest monograph, she usually printed it out and mailed it. Printouts sufficed for most journal submissions (though a fair number would reject dot-matrix printer output and require “letter quality” output). A few forward-thinking journals would also accept electronic submissions (usually specifying the acceptable kind of disk, the word processor format, the software version, and so on), but electronic submissions were only an experiment. These were the Good Old Days.

### **The E-Mailstrom**

Even then, there was e-mail on the Internet. It was, in fact, one of the earliest applications of the Internet. At first, e-mail was an ad hoc affair, crudely cobbled together and just good enough to be useful to the pioneers who were creating the Internet, helping them keep each other up-to-date on the topic of their mutual interest: the inter-network that they were building under the aegis of the United States government’s Advanced Research Projects Agency (ARPA).

By the late 1970s, e-mail had become important enough to require more formal standardization. True to its semi-anarchic, collaborative roots, the Internet community issued a Request For Comments (the procedure by which many Internet standards came into being), and on 21 November 1977 RFC 733, the Standard for the Format of ARPA Network Text Messages (which the curious may find at <http://metalab.unc.edu/pub/docs/rfc/rfc733.txt>), was issued. Among other things, RFC 733 clearly defined what an e-mail message was: “A message consists of headers and, optionally, a body (i.e. a series of text lines). The text part is just a sequence of lines containing ASCII characters...” ASCII was to be the basis for e-mail, and by default, American English became e-mail’s lingua franca. Given that the Internet was, at the time, the ARPA network, and was funded by the United States government, choosing ASCII as the basis of e-mail was, if nothing else, a politically sound move.

Internet-based e-mail was not the only game in town, of course. By the mid-1980s, proprietary e-mail systems were being created for internal corporate or campus use on local networks. Most such systems didn’t use the Internet (which is a big network that

connects smaller networks together), and data compatibility between these proprietary mail systems was not a major issue. So what if Mercantile Widgets' e-mail system was incompatible with Heartland University's e-mail system? Mercantile's employees couldn't send e-mail to Heartland faculty, and Heartland faculty probably didn't think that they had anything to say to Mercantile. In academia, e-mail was much more important to university administrators than to the average working professor, and as long as the e-mail system suited the internal administrative needs of the campus, compatibility with the rest of the world was not that important.

That all changed, of course, when the Internet suddenly became a mass-market phenomenon in the early 1990s. In the course of a few short years, a huge number of local corporate and academic networks began to interconnect through the Internet. Proprietary e-mail systems, many of which sported specialized, useful, non-Internet-standard capabilities, suddenly had to be modified to work with standard Internet e-mail. And standard Internet e-mail was built on top of ASCII, the old 7-bit character standard designed for handling standard American English and not much more.

This was a serious problem. By the late 1980s, most computers had adopted an 8-bit character standard, which, as we've seen, allowed for characters sets containing 256 characters: enough for the ASCII character set and another alphabet to co-exist in a single code-page [see part One of this series]. But the Internet e-mail standard only supported 7-bit characters: the code-page work-around that allowed at least basic multi-lingual word-processing would not work for e-mail.

To be sure, many Internet e-mail systems did support 8-bit characters, but not all of them did. That was the rub: the Internet worked by breaking up large amounts of data (like, say, an e-mail message) into little self-contained packets and then passing those packets from one computer to another across the network until they eventually reached their destination. There was no way to specify the path that each packet followed because the path was chosen depending on network conditions at the time the packet was handed off to the next computer. Even if most computers in the chain could handle 8-bit data, when a packet was sent to a machine that could only handle 7-bit data, that machine would only send 7-bit data to the next machine. Any 8-bit characters (such as "ñ" or "ü") would be lost. (Today, packets travel between routers — a router being a specialized device designed to direct packets across the network — and nearly all routers can handle 8-bit information flawlessly, but back when the standards were created, many computers on the Internet acted as their own routers. Nonetheless, some antiquated e-mail systems are still in service that may strip 8-bit data down to 7-bits.)

The race was on to "fix" the Internet e-mail standard. Another request for comments went out, and in September of 1993, the Network Working Group issued RFC 1521, MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies (available at <<http://metalab.unc.edu/pub/docs/rfc/rfc1521.txt>>). In the introduction to that document, the authors stated the obvious:

RFC 822 [a close successor to RFC 733 ] is inadequate for the needs of mail users whose languages require the use of character sets richer than US ASCII.... Since RFC 822 does not specify mechanisms for mail containing audio, video, Asian language text, or even text in most European languages, additional specifications are needed.

What MIME did for writers of non-English e-mail was to create a way to specify which character set the text part of the message contained. This attempt to wrest some multi-lingual order out of the ASCII void, however, was quickly followed by a series of revisions to MIME as Internet usage soared to undreamed-of heights and lacunae were found in the MIME standard. For example, the original MIME standard explained how to specify the character set (from the plethora of ISO character sets) that a message

contained, but was vague about how to specify that a message contained text in several character sets (say, an English message quoting both Russian and Hebrew authors).

MIME drove around the 7-bit pothole on the Internet with something called “base64 content-transfer-encoding”, a bizarre (to non-programmer eyes) way of turning 8-bit information into ASCII characters. The result was a textual message that looked like complete gibberish, but that easily could be decoded to recover the actual information. Unfortunately, if a mail client did not support base64 content-transfer-encoding, the message would not be decoded, and the recipient would receive complete gibberish. RFC 1521, though, also gave us “quoted-printable content-transfer-encoding.” This was a compromise that kept 7-bit characters in human readable form and only encoded the 8-bit characters. Such messages would be more or less readable even if not decoded (although small patches of seeming nonsense, such as “=0A” and “=0D”, would appear sprinkled throughout the message).

With RFC1521 and its successors, developers of e-mail clients (those programs that let you read and write e-mail) began racing to support the latest MIME capabilities in their products, and network service managers and e-mail system developers began scrambling to make sure that their systems could handle the new MIME standards.

That race is still being run today, and the runners are sprinting even faster just to keep pace with changing standards. In just a handful of years, e-mail’s rapidly increasing base of users and uses has changed what e-mail is and what it can do. But the standards that support innovations like styled text or multi-lingual support are fairly new, and there are many e-mail users who run e-mail clients that are older than the current standards. So, what does this mean for the multi-lingual humanist scholar?

It means that even if you can write and send e-mail containing English, Arabic, and French text, you cannot assume your recipient will be able to read it if he isn’t running the same e-mail software you are...and even then you can’t be certain. It means that you may find yourself constantly changing settings in your e-mail software depending on which standard was used to encode the particular message you’re trying to read. It means that you must know a good deal of swiftly changing technical lore to use the rapidly changing e-mail medium.

These are not the Good Old Days, but they are not without interest.

### **Arachnophilia**

Today, most people think that e-mail and the World Wide Web are pretty much the same thing: not surprising, since the major web browsers (Netscape Communicator and Internet Explorer) include e-mail clients. They are, in fact, very different creatures.

The Web as we know it is less than ten years old. First proposed by Tim Berners-Lee in 1989, it actually reached the light of Internet day in 1991 — fourteen years after the first major formal specification of e-mail. What Berners-Lee and his fellows at CERN, a high-energy particle research center in Switzerland, wanted was a simple solution to an aggravating problem. In 1990, Berners-Lee wrote a “Proposal for a HyperText Project” <<http://www.w3.org/Proposal.html>>, in which he outlined the problem he was hoping to solve: “At CERN, a variety of data is already available: reports, experiment data, personnel data....spinning around on computer disks continuously. It is however impossible to ‘jump’ from one set to another in an automatic way.”

Berners-Lee’s solution was to create a “hyper text mark-up language” (called HTML) which would encode how sections of one document could link to another document. HTML was a pure text solution: both the data that humans could read, and the data containing the “tags” that specified links, were stored as ordinary text — specifically ISO

Latin-1 encoded text (an extended version of ASCII) for content and only printable ASCII characters for link tags. The original specification for HTML version 1.0 was only a few pages long, and it only provided support for “anchors” (i.e. links), section headings, paragraph markers, and bulleted lists. That’s it: no graphics, no text styles, no color. But, from this humble beginning the World Wide Web was spun.

As HTML 1.0 emerged, a group at the National Center for Supercomputing Applications (NCSA) in Illinois began to take the next step. Eric Bina and Marc Andreessen at NCSA started work on Mosaic, a program that would display HTML documents but would add something near and dear to the hearts of scientists: images. While the debate about HTML’s future enhancements continued among the few people on the nascent HTML mailing list, Andreessen went ahead and unilaterally created the HTML tags for displaying images, along with some other features, and shipped Mosaic off. The HTML standard, now approaching version 2, still only supported ASCII text, however.

Andreessen left NCSA soon after Mosaic shipped in 1994 and founded a company in California called Netscape. That same year, a comet slammed into the planet Jupiter, and images of this cosmic event energized the Internet, as web browsers turned to the various research sites that had pictures of the once in a millennium event. The World Wide Web was quickly growing, and browser makers, among them the newly formed Netscape and the giant Microsoft corporation, furiously began adding features to their products: background images, frames, fonts, tables. The recent HTML 2 standard became almost totally irrelevant as everyone focussed on adding new multimedia features not envisioned in that standard. Multi-lingual support in the web browser environment became an afterthought, relying on a single tag that allowed an HTML document to specify a single encoding standard: one could specify ISO Latin 1 (the default) or maybe ISO 8859-5 (Slavic) or one of a few others, but a page that mixed Slavic and English and Arabic and Japanese was almost impossible to create — completely impossible if one wanted to adhere to agreed-upon standards. In the race to get browsers to market, everyone wanted their pet features on Tuesday; getting them good would just take too long, and market share would pass them by.

The pace of browser development became so frenetic that HTML 3.0 was never even adopted as a standard: by the time it was ready, browser technology had left it in the dust as it sped down the information superhighway. Eventually, the HTML 3.2 standard codified the HTML tags that had been invented and put in use by both Netscape and Microsoft, but it did little to bring any order to the browser market, and did almost nothing to ease the plight of multi-lingual web-page authors.

### **Hebrew Headaches**

It takes more than a character-set standard to make a web page. Take the case of Hebrew. Though Hebrew does have an ISO character standard (ISO 8859-8), that standard only deals with the binary number assigned to each character: writing direction is not explicitly part of the standard. Web page authors have had to fall back on the informational RFC 1556, “Handling of Bi-directional Texts in MIME” <http://metalab.unc.edu/pub/docs/rfc/rfc1556.txt>, for guidance.

This document describes three ways that bi-directional text can be specified: implicit, explicit, and visual. Implicit control of text direction means that the displaying software (such as a web browser or e-mail client) decides what the text direction is by a complex method that examines the type of character (such as its character set encoding) and the type of characters around it and uses that information to decide how to display the text.

Explicit control requires embedding directional information directly in the text (usually using characters codes that don't ordinarily display). Visual control is the simplest technique: it assumes that all text is written left-to-right and then, for languages like Hebrew, it requires that the authors simply compose their texts in reverse order. Visual control is the simplest to implement (the burden is on the writer and not the software) and is the method usually employed for MIME-encoded texts.

Sometimes choice is liberating, and sometimes choice can create conflict and confusion: Microsoft, for example, exercised freedom of choice and chose the implicit control method in its Hebrew version of Windows and in its web browsers. As a result, those pages designed to be read with Microsoft's software may not appear correctly in browsers whose developers chose the visual control method of text direction. Furthermore, some Hebrew web pages don't bother to include the one lone HTML tag that specifies the character set for the page, relying instead upon font tags (which only work in HTML 3.2 compatible browsers) to specify a Hebrew font — and woe to the poor user who doesn't have the right Hebrew font installed.

Hebrew web authors couldn't wait for the standards to be good, because they wanted their pages to be available by Tuesday. As a result, the poor Hebrew-literate web user finds herself tinkering with her fonts and browser settings almost every time she goes to a new Hebrew web-site.

### **Preview: HTML 4.0 and Unicode**

In December of 1997, version 4.0 of the HTML standard was released. It was the first version of the standard to abandon the old ISO 8859 encoding schemes for character sets and to adopt ISO 10646: Unicode. And with Unicode came the promise of a brave new multi-lingual digital world. Or maybe not. We'll cover the story of Unicode and what it means for multi-lingual word processing and web pages in the next section.

## **Part Three: The Writes of Man**

“Double your pleasure, double your fun.”  
—Chewing gum commercial

In part one, we began the chronicles of ASCII, the digital character set standard that, through no real fault of its own, has endlessly complicated multi-language text processing. Part two traced how these complications manifested themselves in the histories of e-mail and of the World Wide Web. Now, as the millennium winds down (ending either this December, or a year from this December, depending on how you count), a new character-set standard called Unicode promises to change everything. But will it?

### **A Short Review**

A short reprise of terms and concepts is probably in order at this point. First, the *character set*. A character set assigns numbers to characters (everything in a computer boils down to numbers sooner or later). Thus, a “P” has its own number, which is not the same as the number assigned to a “d” or the one assigned to an “E”.

Next is the *glyph*. A glyph is the actual drawing of the character, how it looks. A group of glyphs associated with a particular character set is often called a font. For example, a “P” in the Times New Roman font looks quite different from a “P” in the Arial font. Both are the letter “P” and both have the same number in the character set, but they are different glyphs. A character is an abstract entity; a glyph is its particular visual manifestation.

Then there is the *keyboard mapping*, also referred to as the *input method*. The keyboard mapping determines what you have to type in order to input a character. In many cases, this is trivial: press the key labeled “P” on the keyboard and you have successfully typed a “P”. In some cases, though, the input method may be more complicated: on the Macintosh, for example, to type an “ö”, you have to hold down the Option key, press the “u” key, release the Option key and then press the “o” key. (Windows users needn’t be smug; they, too, often have to engage in even more complex keyboard gymnastics).

So, when you write something with a computer, you need a character set, a font assigned to those characters, and an input method. To read something on the computer, you don’t have to worry about the input method, but you still need a character set and a font.

End of review; on with the show.

## Writing Systems

Just because you have a character set for your particular language and a font designed to display that set does not necessarily mean that you are ready to start producing text. Human beings are very creative creatures, and they have come up with a variety of ways to turn their myriad spoken languages into visual symbols. As we saw in the last installment of this series, *writing direction* can vary from language to language: for example, Italian goes from left to right, but Hebrew is written from right to left. Other languages may arrange their texts in vertical columns on the page. Still others (thankfully, very few) alternate the writing direction from one line to the next: this is called *boustrophedonic* writing (from the Greek, meaning to turn like an ox while plowing). Software designed to display the rich variety of human textual expression must take writing direction into account.

Then there are *diacritics*: the distinguishing marks that in some way modify the pronunciation or meaning of the character to which they are attached. Some character sets may treat a character combined with a diacritic as a separate, unique, character: “ö”, for example, is a unique character in the ISO 8859-1 character set (the one used for English and many western European languages), entirely distinct from both the “o” and the “.”. In other languages, diacritic markings may be considered (at least as far as character sets are concerned) to be distinct characters, such as the vowel markings in Arabic or Hebrew. How diacritics are incorporated into a character set affects how things like alphabetical sorting work.

There are other typographical conventions like text justification, which can vary from language to language and from character set to character set. In typeset English text, justified text is created by slightly widening the distance between words and letters. In languages like Arabic, however, justified text is created by using one or more *kashidas* (a kashida is an extension bar that connects the glyphs of a word). Software that must properly display a variety of languages has to deal with different justification methods.

Then there are quasi-textual materials like mathematical formulas, which employ superscripts, subscripts, symbols that enclose other symbols, and other subtleties that can give even the most accomplished typesetter (digital or traditional) nervous twitches.

All of these conventions—justification, diacritics, writing direction, and so on—together make up what are called *writing systems*. We humans must love writing systems because we made so many of them.

### 60,000 Characters in Search of an Author

We've seen how accented Latin characters (such as those found in French or German) “broke” the 128-character seven-bit ASCII encoding system, spawning various schemes for encoding, processing, and displaying the accented, non-American characters. Adding a single bit to the ASCII code was how most of those schemes worked: adding one bit doubled the number of characters that could be encoded. That simple doubling trick was powerful enough to accommodate (more or less) most of the world's alphabets: languages as diverse as Arabic and Inuit could employ character set encodings that included the standard ASCII characters along with their own—that is, as long as those languages had alphabets that could be expressed in 128 or fewer characters. The writing systems used by roughly half the world's people, however, do not.

One of the most significant of those eight-bit-unfriendly writing systems is *Han*, a writing system established during China's Han Dynasty (which flourished more or less from 206 BCE to 220 AD). The Han ideographic characters are used to write Chinese (where they are called *hanzi*), Japanese (where they are called *kanji* or *kanzi*), and Korean (where they are called *hanja*).

While each character in an alphabetic writing system tends to represent a unit of sound, in the Han writing system each ideograph represents a unit of meaning. A writing system designed around units of meaning has some advantages over phonetic alphabets: the spoken form of a word does not matter, but only the meaning, so the same written symbol can mean the same thing in languages that do not sound at all alike. Such a writing system also has its disadvantages when compared to phonetic alphabets: there are a lot more clearly distinct *meanings* that humans can make than there are clearly distinct *sounds*. Although most alphabets contain fewer than one hundred characters, the set of Han ideographs numbers in the tens of thousands.

With that many characters, Han does not lend itself to mechanization (back when most writing was done with brush and ink, this was not much of an issue, but try designing a keyboard with tens of thousands of keys...). Nor is it particularly easy for a writer to learn that many characters. Consequently, some languages that use Han have also developed phonetic writing systems. In Japan, in addition to kanji, there are the phonetic alphabets, *katakana* and *hiragana*. Korean, too, has a phonetic alphabet in addition to hanja: *hangul*, which was invented in 1445 by King Sejong during the Yi Dynasty.

Unfortunately, these phonetic alphabets have not displaced the Han ideographs—both are in common use, meaning that writers in these languages (and their computers) have to deal regularly with not one but several writing systems.

The history of Cyrillic (see Part One) illustrated how even a single alphabet could give rise to a plethora of incompatible encoding systems. The character sets that have evolved over the past few decades to deal with the mix of writing systems used for Chinese, Japanese, and Korean make Cyrillic's history look like a model of simplicity. (For an overview of these character sets, called CJK [for Chinese-Japanese-Korean] in

the information processing community, see Ken Lunde's invaluable online document, CJK.INF <<ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/cjk.inf>>.)

Because most of the world's computing systems have relied upon seven- or eight-bit chunks for storing, retrieving, and transmitting information, many of the various CJK character sets have tried to express themselves in seven- or eight-bit chunks as well. Fitting as many as sixty thousand characters into seven or eight bits, however, is mathematically impossible: we all know that seven bits can only express 128 values, and eight bits only gives us 256. To make the impossible possible, many CJK encoding schemes make use of a trick called an *escape* sequence.

Here's how the escape trick works. You divide your character set into equal sized tables, each of which contains fewer than 256 characters. You then choose one special character, called the *escape character*, to signal that you are shifting from one table of characters to another. Let's say the character "P" is the 80th character in one table (let's call it table 1), and the Hiragana letter "wa" is the 80th character in another table (let's call this one table 17). Finally, let's say that character number 28 is the escape character. When the computer interprets a sequence of numbers as characters it will start by using table 1. When it gets to a character represented by the number 80, it will treat that character as a "P." Sometime later in the sequence of numbers, the computer encounters the number 28—our escape character. The escape character is immediately followed by the number 17: the combination of the escape character followed by the number 17 tells the computer to make table 17 the active character table. From this point on, whenever the computer encounters a character numbered 80 it will interpret that character as Hiragana "wa"—until it encounters the escape character again.

The escape trick is very powerful, but it has its problems, too. Here's one example. Suppose you copy a small portion of text from a large document that has been encoded using an escape scheme. You then paste that text somewhere else. If the escape sequence that identifies the active character set does not happen to be included in the text that you copied, the computer will not be able to figure out which character set it should use to interpret the text that you pasted. The result usually will be gibberish. This problem (and variations on it) happen so often in Japanese text processing that the Japanese have a special word for it: *mojibake*.

## Come Together

The more closely one explores all the characters sets currently in use on the world's computer systems, a simple question emerges: Wouldn't it be wonderful if there were a scheme that would unify all the world's disparate character sets into one character set that accommodated all languages? It turns out such a scheme has been under development for the last ten years. It's called Unicode.

The Unicode Consortium <<http://www.unicode.org/>>, the organization that develops and controls the Unicode character set, began in 1991 as a collaboration between major software and hardware manufacturers, internationally-minded academics, and engineers interested in bringing some order to the chaos of multi-lingual text processing. At roughly the same time, the International Organization for Standardization (ISO) <<http://www.iso.ch/>> began developing a similar character set standard to unify all the character sets that it had previously promulgated.

At first, these two groups were at loggerheads. One of the big issues dividing them was bits. Unicode felt that the tyranny of the eight-bit character set had to end, and that devoting sixteen bits to each character would allow nearly all of the world's languages to use one common character set—after all, sixteen bits could handle over 65,000 unique

characters. ISO, on the other hand, felt that eight-bit characters were too deeply ingrained in the computational environment ever to be completely rooted out, and they developed a scheme that used multiple *octets* (an octet is simply a unit of eight bits): some portions of their character set would use eight-bit characters, some would use sixteen bits, and some would use even more. After much skirmishing and bickering, both Unicode and ISO combined their efforts so that the ISO 10646 character set would match the Unicode Consortium's character set.

And who won the battle of the bits? No one, really: ISO was right about how ingrained eight-bit (and seven-bit) text was, but they also took Unicode's point that it was often more computationally convenient for programmers to store text characters in sixteen bit units. So the grand compromise was achieved: standardized methods would be developed that allowed text to be stored in a variety of ways—seven-bit encodings (called UTF-7), eight-bit encodings (called UTF-8), and sixteen-bit encodings (UTF-16).

Simple procedures for translating between these schemes were also devised. Thus, mail systems that, for example, only supported seven-bit characters could still transmit the entire array of Unicode characters, and software designers could also simply encode all their texts in 16-bit format if it made sense to do so. What was invariant was the character set, where each of the world's characters had one and only one number, a number not used by any other character. Mojibake could become a thing of the past in the Unicode/ISO-10646 world. (And if you really believe that, there's this bridge I'd like to sell you....)

The Unicode character set is big: the current version of it contains character codes for Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, modern Korean Hangul, and a unified Han. Also included are punctuation marks, diacritics, mathematical symbols, and technical symbols. Soon to be added are codes for Cherokee, Ethiopic, Syriac, and even Braille. The standard also includes some non-character set information, such as standardized methods for handling bi-directional text.

There's a lot that the standard leaves out, too. Unicode is silent about glyphs, leaving them up to typeface designers and software developers. Nor does it address input methods, letting hardware manufacturers and programmers decide how the user will enter characters. Nor does Unicode specify sorting order, although it does try to encode its characters in ways that might make sorting more convenient for programmers to implement.

What Unicode does, finally, is provide THE fundamental character set, a set to which all other character sets, from America's venerable ASCII to Japan's Shift-JIS, can refer, and in terms of which all (or nearly all) existing character sets can be defined. It is, in short, a character set Rosetta Stone.

### Writing the Future

Although we are approaching the millennium, we haven't yet achieved nirvana. The multilingual menagerie of character sets, font systems, and input methods, with all their attendant complexities and maddening incompatibilities, will be with us for some time to come. Standards may become obsolete, but they seldom die: as long as there is data stored in KOI-8 format somewhere, there will be a need for software that can handle the KOI-8 standard.

We can never change the past, and if we are not to lose the texts we've created in digital technology's infancy, we will need to remember how those texts were created and processed and stored. As we move ahead we must never forget that all solutions are

interim, and that no standard, no software company, no hardware vendor, is any more enduring than Shelley's statue of Ozymandias, round the decay of whose colossal wreck, boundless and bare, the lone and level sands stretch far away.

—Michael E. Cohen, October 1998–April 1999