

A tutorial on character code issues

Jukka K. Korpella

Päivänsäteenkuja 4 as. 1

FIN-02210 Espoo

Finland

Email : jkorpela@cs.tut.fi

1. The basics

In computers and in data transmission between them, i.e. in digital data processing and transfer, data is internally presented as octets, as a rule. An *octet* is a small unit of data with a numerical value between 0 and 255, inclusively. The numerical values are presented in the normal (decimal) notation here, but notice that other presentations are used too, especially **octal** (base 8) or **hexadecimal** (base 16) notation. Octets are often called *bytes*, but in principle, octet is a more definite concept than **byte**. Internally, octets consist of eight **bits** (hence the name, from Latin but we need not go into bit level here. However, you might need to know what the phrase "first bit set" or "sign bit set" means, since it is often used. In terms of numerical values of octets, it means that the value is greater than 127. In various contexts, such octets are sometimes interpreted as *negative* numbers, and this may cause various problems.

Different conventions can be established as regards to how an octet or a sequence of octets presents some data. For instance, four consecutive octets often form a unit that presents a real number according to a specific standard. We are here interested in the presentation of character data (or string data; a *string* is a sequence of characters) only.

In the simplest case, which is still widely used, one octet corresponds to one character according to some mapping table (encoding). Naturally, this allows at most 256 different characters being represented. There are several different encodings, such as the well-known **ASCII** encoding and the **ISO Latin family** of encodings. The correct interpretation and processing of character data of course requires knowledge about the encoding used.

Previously the **ASCII** encoding was usually assumed by default (and it is still very common). Nowadays **ISO Latin 1**, which can be regarded as an

extension of ASCII, is often the default. The current trend is to avoid giving such a special position to ISO Latin 1 among the variety of encodings.

2. Definitions

The following definitions are not universally accepted and used. In fact, one of the greatest causes of confusion around character set issues is that terminology varies and is sometimes misleading.

character repertoire A set of distinct characters. No specific internal presentation in computers or data transfer is assumed. The repertoire per se does not even define an ordering for the characters; ordering for sorting and other purposes is to be specified separately. A character repertoire is usually defined by specifying **names** of characters and a sample (or reference) presentation of characters in visible form. Notice that a character repertoire may contain characters which *look* the same in some presentations but are regarded as logically distinct, such as Latin uppercase A, Cyrillic uppercase A, and Greek uppercase alpha.

character code A mapping, often presented in tabular form, which defines a one-to-one correspondence between characters in a character **repertoire** and a set of nonnegative integers. That is, it assigns a unique numerical code, a *code position*, to each character in the repertoire. In addition to being often presented as one or more tables, the code as a whole can be regarded as a single table and the code positions as indexes. As synonyms for "code position", the following terms are also in use: *code number*, *code value*, *code element*, *code point*, *code set value* - and just *code*. Note: The set of nonnegative integers corresponding to characters need not consist of consecutive numbers; in fact, most character codes have "holes", such as code positions reserved for **control functions** or for eventual future use to be defined later.

character encoding A method (algorithm) for presenting characters in digital form by mapping sequences of **code numbers** of characters into sequences of **octets**. In the simplest case, each character is mapped to an integer in the range 0 - 255 according to a character code and these are used as such as octets. Naturally, this only works for character **repertoires** with at most 256 characters. For larger sets, more complicated encodings are needed. **Encodings have names, which can be registered.**

Notice that a character code assumes or implicitly defines a character repertoire. A character encoding could, in principle, be viewed purely as a method of mapping a sequence of integers to a sequence of octets. However, quite often an encoding is specified in terms of a character code (and the implied character repertoire). The *logical* structure is still the following:

A character *repertoire* specifies a collection of characters, such as "a", "!", and "ä".

A character *code* defines numeric codes for characters in a repertoire. For example, in the **ISO 10646** character code the numeric codes for "a", "!", "ä", and "%₀₀" (per mille sign) are 97, 33, 228, and 8240. (Note: Especially the per mille sign, presenting %₀₀ as a single character, can be shown incorrectly on display or on paper. That would be an illustration of the symptoms of the problems we are discussing.)

A character *encoding* defines how sequences of numeric codes are presented as (i.e., mapped to) sequences of octets. In one possible encoding for **ISO 10646**, the string a!ä%₀₀ is presented as the following sequence of octets (using two octets for each character): 0, 97, 0, 33, 0, 228, 32, 48.

The phrase *character set* is used in a variety of meanings. It might denote just a character repertoire but it may also refer to a character code, and quite often a particular character encoding is implied too.

Unfortunately the word *charset* is used to refer to an encoding, causing much confusion. It is even the official term to be used in several contexts by Internet protocols, in **MIME** headers.

Quite often the choice of a character repertoire, code, or encoding is presented as the choice of a *language*. For example, Web browsers typically confuse things quite a lot in this area. A pulldown menu in a program might be labeled "Languages", yet consist of character encoding choices (only). A language setting is quite distinct from character issues, although naturally each language has its own requirements on character repertoire. Even more seriously, programs and their documentation very often confuse the above-mentioned issues with the selection of a **font**.

3. Examples of character codes

3.1. Good old ASCII

The name *ASCII*, originally an abbreviation for "American Standard Code for Information Interchange", denotes an old character **repertoire**, **code**, and **encoding**.

Most character codes currently in use contain ASCII as their subset in some sense. ASCII is the safest character repertoire to be used in data transfer. However, **not even all ASCII characters are "safe"!**

ASCII has been used and is used so widely that often the word *ASCII* refers to "text" or "plain text" in general, even if the character code is something else! The words "ASCII file" quite often mean any text file as opposite to a binary file.

The definition of ASCII also specifies a set of **control codes** ("control characters") such as linefeed (LF) and escape (ESC). But the *character repertoire* proper, consisting of the *printable* characters of ASCII, is the following (where the first item is the blank, or space, character) :

~	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	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	[\]	^	_
'	a	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	{		}	~	

The *appearance* of characters varies, of course, especially for some special characters.

A formal view on ASCII The *character code* defined by the ASCII standard is the following: code values are assigned to characters consecutively in the order in which the characters are listed above (rowwise), starting from 32 (assigned to the blank) and ending up with 126 (assigned to the tilde character ~). Positions 0 through 31 and 127 are reserved for **control codes**. They have standardized **names and descriptions**, but in fact their usage varies a lot.

The *character encoding* specified by the ASCII standard is very simple, and the most obvious one for any character code where the code numbers do not exceed 255: each code number is presented as an octet with the same value.

Octets 128 - 255 are not used in ASCII (This allows programs to use the first, most significant bit of an octet as a **parity** bit, for example).

National variants of ASCII There are several national variants of ASCII. In such variants, some special characters have been replaced by national letters (and other symbols). There is great variation here, and even within one country and for one language there might be different variants. The original ASCII is therefore often referred to as *US-ASCII*; the formal standard (by **ANSI**) is *ANSI X3.4-1986*.

The international standard **ISO 646** defines a character set similar to **US-ASCII** but with **code positions** corresponding to US-ASCII characters `@[\]{}` as "national use positions". It also gives some liberties with characters `#$^'~`. The standard also defines "international reference version (IRV)", which is (in the 1991 edition of ISO 646) identical to US-ASCII.

Within the framework of ISO 646, and partly otherwise too, several "national variants of ASCII" have been defined, assigning different letters and symbols to the "national use" positions. Thus, the characters that appear in those positions - including those in US-ASCII - are somewhat "unsafe" in international data transfer, although this problem is losing significance. The trend is towards using the corresponding codes strictly for US-ASCII meanings; national characters are handled otherwise, giving them their own, unique and universal code positions in character codes larger than ASCII. But old software and devices may still reflect various "national variants of ASCII".

The following table lists ASCII characters which might be replaced by other characters in national variants of ASCII. (That is, the code positions of these US-ASCII characters might be occupied by other characters needed for national use.) The lists of characters appearing in national variants are not intended to be exhaustive, just typical *examples*.

Almost all of the characters used in the national variants have been incorporated into **ISO Latin 1**. Systems that support ISO Latin 1 in principle may still reflect the use of national variants of ASCII in some details; for example, an ASCII character might get *printed* or *displayed* according to some national variant. Thus, even "plain ASCII text" is thereby not always portable from one system or application to another.

Subsets of ASCII for safety Mainly due to the "**national variants**" discussed above, some characters are less "safe" than other, i.e. more often transferred or interpreted incorrectly.

In addition to the letters of the English alphabet ("A" to "Z", and "a" to "z"), the digits ("0" to "9") and the space (" "), only the following characters can be regarded as really "safe" in data transmission:

! " % & ' () * + , - . / : ; < = > ?

dec	oct	hex	glyph	official Unicode name	National variants
35	43	23	#	number sign	£ Û
36	44	24	\$	dollar sign	•
64	100	40	@	commercial at	É §Ä à³
91	133	5B	[left square bracket	Ä Æ ° â ¡ ÿ é
92	134	5C	\	reverse solidus	Ö Ø ç Ñ 1/2 •
93	135	5D]	right square bracket	Ä Ü § ê é ï
94	136	5E	^	circumflex accent	Û î
95	137	5F	_	low line	è
96	140	60	`	grave accent	é ä µ ô ù
123	173	7B	{	left curly bracket	ä æ é à ° °
124	174	7C		vertical line	ö ø ù ò ñ f
125	175	7D	}	right curly bracket	å ü è ç 1/4
126	176	7E	~	tilde	ü ¯ ß ° û ì ' _

Even these characters might eventually be *interpreted* wrongly by the recipient, e.g. by a human reader seeing a **glyph** for "&" as something else than what it is intended to denote, or by a program interpreting "<" as starting some special **markup**, "?" as being a so-called wildcard character, etc.

When you need to *name* things (e.g. files, variables, data fields, etc.), it is often best to use only the characters listed above, even if a wider character repertoire is possible. Naturally you need to take into account any additional restrictions imposed by the applicable syntax. For example, the rules of a programming language might restrict the character repertoire in identifier names to letters, digits and one or two other characters.

The misnomer "8-bit ASCII" Sometimes the phrase "8-bit ASCII" is used. It follows from the discussion above that in reality *ASCII is strictly and unambiguously a 7-bit code* in the sense that all code positions are in the range 0-127.

It is a misnomer used to refer to *various* character **codes** which are *extensions of ASCII* in the following sense: the character repertoire contains ASCII as a subset, the code numbers are in the range 0 - 255, and the code numbers of ASCII characters equal their ASCII codes.

3.2. Another example: ISO Latin 1 alias ISO 8859-1

The ISO 8859-1 standard (which is part of the **ISO 8859 family** of standards) defines a *character repertoire* identified as "Latin alphabet No. 1",

commonly called "ISO Latin 1", as well as a *character code* for it. The repertoire contains the **ASCII** repertoire as a subset, and the code numbers for those characters are the same as in ASCII. The standard also specifies an *encoding*, which is similar to that of ASCII: each code number is presented simply as one octet.

In addition to the ASCII characters, ISO Latin 1 contains various accented characters and other letters needed for writing languages of Western Europe, and some special characters. These characters occupy code positions 160 - 255, and they are:

°	ı	ł	£	•	•		§	¨	©	ª	«	¬	®	ˆ
±	²	³	,	μ	¶	·	˙	ı	º	»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ

The first of the characters above appears as space; it is the so-called **no-break space**. Naturally, the appearance of characters varies from one **font** to another.

3.3. More examples: the Windows character set(s)

In **ISO 8859-1**, code positions 128 - 159 are explicitly reserved for **control purposes**; they "correspond to bit combinations that do not represent graphic characters". The so-called **Windows character set** (WinLatin1, or **Windows code page 1252**, to be exact) uses some of those positions for printable characters. Thus, the Windows character set **is not identical with ISO 8859-1**. It is, however, true that the Windows character set is much more similar to ISO 8859-1 than the so-called **DOS character sets** are. The Windows character set is often called "ANSI character set", but this is seriously misleading. It has *not* been approved by **ANSI**. (Historical background: Microsoft based the design of the set on a *draft* for an ANSI standard. **A glossary** by Microsoft explicitly admits this.)

Note that programs used on Windows systems may use a DOS character set; for example, if you create a text file using a Windows program and then use the `type` command on DOS prompt to see its content, strange things may happen, since the DOS command interprets the data according to a DOS character code.

In the Windows character set, some positions in the range 128 - 159 are assigned to printable characters, such as "smart quotes", em dash, en dash, and trademark symbol. Thus, the character repertoire is larger than **ISO Latin 1**. The use of octets in the range 128 - 159 in any data to be processed by a program that expects ISO 8859-1 encoded data is an error which might cause just anything. They might for example get ignored, or be processed in a manner which looks meaningful, or be interpreted as **control characters**.

The Windows character set exists in different variations, or "**code pages**" (CP), which generally differ from the corresponding ISO 8859 standard so that it contains same characters in positions 128 - 159 as code page 1252. (However, there are some more differences between ISO 8859-7 and WIN-1253 (WinGreek)). What we have discussed here is the most usual one, resembling ISO 8859-1. In December 1999, **Microsoft finally registered** it under the name windows-1252. (The name cp-1252 has been used too, but it isn't officially registered even as an alias name).

3.4. The ISO 8859 family

There are several character codes which are extensions to **ASCII** in the same **sense** as **ISO 8859-1** and the Windows character set. ISO 8859-1 itself is just a member of the ISO 8859 family of character codes. Those codes extend the **ASCII** repertoire in different ways with different special characters (used in different languages and cultures). Just as ISO 8859-1 contains ASCII characters and a collection of characters needed in languages of western (and northern) Europe, there is ISO 8859-2 alias ISO Latin 2 constructed similarly for languages of central/eastern Europe, etc. The ISO 8859 character codes are *isomorphic* in the following sense: code positions 0 - 127 contain the same character as in ASCII, positions 128 - 159 are unused (reserved for **control characters**), and positions 160 - 255 are the varying part, used differently in different members of the ISO 8859 family.

The ISO 8859 character codes are normally presented using the obvious encoding: each code position is presented as one octet. Such encodings have several alternative names in the official **registry of character encodings**, but the preferred ones are of the form ISO-8859-*n*.

Although ISO 8859-1 has been a de facto default encoding in many contexts, it has in principle no special role. And in practice, **ISO 8859-15 alias ISO Latin 9 (!)** will probably replace ISO 8859-1 to a great extent, since it contains the politically important symbol for euro.

Notes: ISO 8859-*n* is Latin alphabet no. *n* for *n*=1,2,3,4, but this correspondence is broken for the other Latin alphabets.

The parts of ISO 8859

standard	Name of alphabet	characterization
ISO 8859-1	Latin alphabet No. 1	"Western", "West European"
ISO 8859-2	Latin alphabet No. 2	"Central European", "East European"
ISO 8859-3	Latin alphabet No. 3	"South European", "Maltese & Esperanto"
ISO 8859-4	Latin alphabet No. 4	"North European"
ISO 8859-5	Latin/Cyrillic alphabet	(for Slavic languages)
ISO 8859-6	Latin/Arabic alphabet	(for the Arabic language)
ISO 8859-7	Latin/Greek alphabet	(for modern Greek)
ISO 8859-8	Latin/Hebrew alphabet	(for Hebrew and Yiddish)
ISO 8859-9	Latin alphabet No. 5	"Turkish"
ISO 8859-10	Latin alphabet No. 6	"Nordic" (Sámi, Inuit, Icelandic)
ISO 8859-11	Latin/Thai alphabet	(for the Thai language; draft
(Part 12 has not been defined).		
ISO 8859-13	Latin alphabet No. 7	Baltic Rim
ISO 8859-14	Latin alphabet No. 8	Celtic
ISO 8859-15	Latin alphabet No. 9	"euro"
ISO 8859-16	Latin alphabet No. 10	for Romanian and various other languages

3.5. Other "extensions to ASCII"

In addition to the codes discussed above, there are other extensions to ASCII which utilize the code range 0 - 255 ("**8-bit ASCII codes**"), such as

DOS character codes, or "code pages" (CP)

In **MS DOS** systems, different character codes are used; they are called "code pages". The original American code page was CP 437, which has e.g. some Greek letters, mathematical symbols, and characters which can be used as elements in simple pseudo-graphics. Later CP 850 became popular, since it contains letters needed for West European languages - largely the same letters as **ISO 8859-1**, but in different code positions. Note that DOS code pages are quite different from **Windows character codes**, though the latter are sometimes called with names like cp-1252 (= windows-1252)! For further confusion, Microsoft now prefers to use the notion "OEM code page" for the DOS character set used in a particular country.

Macintosh character code

On the **Macs**, the character code is more uniform than on PCs (although there are some **national variants**). The Mac character repertoire is a mixed combination of ASCII, accented letters, mathematical symbols, and other ingredients.

Notice that many of these are very different from ISO 8859-1. They may have different character repertoires, and the same character often has different code values in different codes. For example, code position 228 is occupied by ä (letter a with dieresis, or umlaut) in ISO 8859-1, by ð (Icelandic letter eth) in HP's **Roman-8**, by ò (letter o with tilde) in DOS code page 850, and per mille sign (‰) in Macintosh character code.

In general, full **conversions** between the character codes mentioned above are not possible. For example, the Macintosh character repertoire contains the Greek letter pi, which does not exist in **ISO Latin 1** at all. Naturally, a text can be converted (by a simple program which uses a conversion table) from Macintosh character code to ISO 8859-1 if the text contains only those characters which belong to the ISO Latin 1 character repertoire. Text presented in **Windows character code** can be used as such as ISO 8859-1 encoded data *if* it contains only those characters which belong to the ISO Latin 1 character repertoire.

3.6. Other "8-bit codes"

All the character codes discussed above are "8-bit codes", eight bits are sufficient for presenting the **code numbers** and in practice the **encoding** (at least the normal encoding) is the obvious (trivial) one where each code position (thereby, each character) is presented as one octet (byte). This means that there are 256 code positions, but several positions are reserved for **control codes** or left unused (unassigned, undefined).

Although currently most "8-bit codes" are **extensions to ASCII** in the sense described above, this is just a practical matter caused by the widespread use of **ASCII**. It was practical to make the "lower halves" of the character codes the same, for several reasons.

The standards **ISO 2022** and **ISO 4873** define a **general framework** for 8-bit codes (and 7-bit codes) and for switching between them. One of the basic ideas is that code positions 128 - 159 (decimal) are reserved for use as **control codes** ("C1 controls"). Note that the **Windows** character sets do not comply with this principle.

To illustrate that other kinds of 8-bit codes can be defined than extensions to Ascii, we briefly consider the **EBCDIC** code, defined by **IBM** and once in widespread use on "**mainframes**" (and still in use). EBCDIC contains all ASCII characters but in quite different **code positions**. As an interesting detail, in EBCDIC normal letters A - Z do not all appear in consecutive code positions. EBCDIC exists in different national variants (cf. to **variants of ASCII**).

3.7. ISO 10646 (UCS) and Unicode

ISO 10646 (officially: ISO/IEC 10646) is an international standard, by **ISO** and **IEC**. It defines UCS, Universal Character Set, which is a very large and growing **character repertoire**, and a **character code** for it. Currently tens of thousands of characters have been defined, and new amendments are defined fairly often. It contains, among other things, all characters in the character repertoires discussed above.

Unicode is a **standard**, by the **Unicode Consortium**, which defines a character repertoire and character code intended to be fully compatible with ISO 10646, and an encoding for it. ISO 10646 is more general (abstract) in nature, whereas Unicode "additional constraints on implementations to ensure that they treat characters uniformly across platforms and applications", as they say in the **Unicode FAQ**. Moreover, Unicode basically corresponds to "Basic Multilingual Plane (BMP)" of ISO 10646 (though there are mechanisms in Unicode to extend beyond BMP); however, other "**planes**" haven't even been defined yet.

The ISO 10646 and Unicode *character repertoire* can be regarded as a *superset* of most character repertoires in use. However, the *code positions* of characters vary from one character code to another.

In practice, people usually talk about Unicode rather than ISO 10646, partly because we prefer names to numbers, partly because Unicode is more explicit about the *meanings* of characters, partly because detailed information Unicode is available on the Web.

Unicode version 1.0 used somewhat different **names** for some characters than ISO 10646. In Unicode version, 2.0, the names were made the same as in ISO 10646. New **versions** of Unicode are expected to add new characters mostly. **Version 3.0**, with a total number of 49,194 characters (38,887 in version 2.1), was published in February 2000.

The ISO 10646 standard has *not* been put onto the Web. It is available in printed form from **ISO member bodies**. But for most practical purposes, the same information is in the Unicode standard.

The "native" Unicode encoding, *UCS-2*, presents each code number as two consecutive octets m and n so that the number equals $256m + n$. This means, to express it in computer jargon, that the code number is presented as a **two-byte integer**. This is a very obvious and simple encoding. However, it can be inefficient in terms of the number of octets needed. If we have normal English text or other text which contains **ISO Latin 1** characters only, the length of the Unicode encoded octet sequence is twice the length of the string in ISO 8859-1 encoding.

It is somewhat debatable whether Unicode defines an encoding or just a character code. However, it refers to code values being presentable as 16-bit integers, and it seems to imply the corresponding two-octet representation. In principle, Unicode requires that "Unicode values can be stored in native 16-bit machine words" and "does not specify any order of bytes inside a Unicode value". Thus, it allows "**little-endian**" presentation where the least significant byte precedes the most significant byte, if agreed on by higher-level protocols.

ISO 10646 can be, and often is, encoded in other ways, too, such as the following **encodings**:

UTF-8

Character codes less than 128 (effectively, the **ASCII** repertoire) are presented "as such", using one octet for each code (character). All other codes are presented, according to a relatively complicated method, so that one code (character) is presented as a sequence of two to six octets, each of which is in the range 128 - 255. This means that in a sequence of octets, octets in the range 0 - 127 ("bytes with most significant bit set to 0") directly represent **ASCII** characters, whereas octets in the range 128 - 255 ("bytes with most significant bit set to 1") are to be interpreted as really encoded presentations of characters.

UTF-7

Each character code is presented as a sequence of one or more octets in the range 0 - 127 ("bytes with most significant bit set to 0", or "seven-bit bytes", hence the name). Most **ASCII** characters are presented as such, each as one octet, but for obvious reasons some octet values must be reserved for use as "escape" octets, specifying the octet together with a certain number of subsequent octets forms a multi-octet encoded presentation of one character.

IETF Policy on Character Sets and Languages (RFC 2277) clearly **favours UTF-8**. It requires support to it in Internet protocols (and doesn't even mention UTF-7). Note that UTF-8 is efficient, if the data consists dominantly of **ASCII** characters with just a few "special characters" in addition to them, and reasonably efficient for dominantly ISO Latin 1 text.

The **implementation** of Unicode support is a long and mostly gradual process. Unicode can be supported by programs on any operating systems, although some systems may allow much easier implementation than others; this mainly depends on whether the system uses Unicode internally so that support to Unicode is "built-in".

Even in circumstances where Unicode is supported in principle, the support usually does not cover *all* Unicode characters. For example, a **font** available may cover just some part of Unicode which is practically important in some

area. On the other hand, for data transfer it is essential to know which Unicode characters the recipient is able to handle. For such reasons, various **subsets** of the Unicode character repertoire have been and will be defined. For example, the *Minimum European Subset* specified by **ENV 1973:1995** is intended to provide a first step towards the implementation of large character sets in Europe. There are also **three Multilingual European Subsets** (MES-1, MES-2, MES-3, with MES-2 based on the *Minimum European Subset*).

In addition to international standards, there are company policies which define various subsets of the character repertoire. A practically important one is Microsoft's "**Windows Glyph List 4**" (**WGL4**), or "PanEuropean" character set.

Unicode characters are often referred to using a notation of the form **U+nnnn** where *nnnn* is a four-digit hexadecimal notation of the code value. For example, U+0020 means the space character (with code value 20 in hexadecimal, 32 in decimal). Notice that such notations identify a character through its Unicode code value, without referring to any particular encoding. There are other **ways to mention (identify) a character**, too.

4. More about the character concept

An "A" (or any other character) is something like a Platonic entity: it is the idea of an "A" and not the "A" itself.

– Michael E. Cohen: *Text and Fonts in a Multi-lingual Cross-platform World*.

The *character* concept is very fundamental for the issues discussed here but difficult to define exactly. The more fundamental concepts we use, the harder it is to give good definitions. (How would you define "life"? Or "structure"?) Here we will concentrate on clarifying the character concept by indicating what it does *not* imply.

4.1. The Unicode view

The **Unicode** standard describes characters as "the smallest components of written language that have semantic value", which is somewhat misleading. A character such as a letter can hardly be described as having a meaning (semantic value) in itself. Moreover, a character such as ú (letter u with acute accent), which belongs to Unicode, can often be regarded as consisting of smaller components: a letter and a diacritic. And in fact the very definition of the character concept in Unicode is the following:

abstract character: a unit of information used for the organization, control, or representation of textual data.

4.2. Control characters (control codes)

The rôle of the so-called *control characters* in character codes is somewhat obscure. Character codes often contain code positions which are not assigned to any visible character but reserved for control purposes. For example, in communication between a terminal and a computer using the **ASCII** code, the computer could regard **octet 3** as a request for terminating the currently running process. Some older character code standards contain *explicit descriptions* of such conventions whereas newer standards just *reserve some positions* for such usage, to be defined in **separate standards or agreements** such as "**C0 controls**" and "**C1 controls**", or specifically **ISO 6429**. And although the definition quoted above suggests that "control characters" might be regarded as characters in the Unicode terminology, perhaps it is more natural to regard them as *control codes*.

Control codes can be used for **device control** such as cursor movement, page eject, or changing colors. Quite often they are used in combination with codes for graphic characters, so that a device driver is expected to interpret the combination as a specific command and not display the graphic character(s) contained in it. For example, in the classical **VT100 controls**, ESC followed by the code corresponding to the letter "A" or something more complicated (depending on mode settings) moves the cursor up. To take a different example, the **Emacs** editor treats ESC A as a request to move to the beginning of a sentence. Note that the ESC control code is logically distinct from the ESC *key* in a keyboard, and many other things than pressing ESC might cause the ESC control code to be sent. Also note that phrases like "**escape sequences**" are often used to refer to things that don't involve ESC at all and operate at a quite different level.

One possible form of device control is changing the way a device interprets the data (octets) that it receives. For example, a control code followed by some data in a specific format might be interpreted so that any subsequent octets to be interpreted according to a table identified in some specific way. This is often called "code page switching", and it means that control codes could be used **change the character encoding**. And it is then more logical to consider the control codes and associated data at the level of fundamental interpretation of data rather than direct device control. The international standard **ISO 2022** defines powerful facilities for using different 8-bit character codes in a document.

Widely used **formatting** control codes include carriage return (CR), line-feed (LF), and horizontal tab (HT), which in **ASCII** occupy code positions 13, 10, and 9. The names (or abbreviations) suggest generic meanings, but the actual meanings are defined partly in each character code definition, partly - and more importantly - by various other conventions "above" the character level. The "formatting" codes might be seen as a special case of device control, in a sense, but more naturally, a CR or a LF or a CR LF pair (to mention the most common conventions) when used in a text file simply indicates a new line. The **HT (TAB) character** is often used for real "tabbing" to some predefined writing position. But it is also used e.g. for indicating data boundaries, without any particular presentational effect, for example in the widely used "tab separated values" (**TSV**) data format.

4.3. A glyph - a visual appearance

It is important to distinguish the character concept from the glyph concept. A *glyph* is a presentation of a particular shape which a character may have when rendered or displayed. For example, the character Z might be presented as a boldface **Z** or as an italic *Z*, and it would still be a presentation of the same character. On the other hand, lower-case z is defined to be a separate character - which in turn may have different glyph presentations.

This is ultimately a *matter of definition*: a definition of a character repertoire specifies the "identity" of characters, among other things. One *could* define a repertoire where uppercase Z and lowercase z are just two glyphs for the same character. On the other hand, one *could* define that italic *Z* is a character different from normal Z, not just a different glyph for it. In fact, in Unicode for example there are several characters which could be regarded as typographic variants of letters only, but for various reasons Unicode defines them as separate characters. For example, mathematicians use a variant of letter N to denote the set of natural numbers (0, 1, 2, ...), and this variant is defined as being a separate character ("double-struck capital N") in Unicode. There are some more **notes on the identity of characters** below.

4.4. What's in a name?

The *names* of characters are **assigned identifiers** rather than definitions. Typically the names are selected so that they contain only letters A - Z, spaces, and hyphens; often uppercase variant is the reference spelling of a character name. The same character may have different names in different definitions of character repertoires. Generally the name is intended to **suggest a generic**

meaning and scope of use. But the **Unicode** standard warns (mentioning **full stop** as an example of a character with varying usage):

A character may have a broader range of use than the most literal interpretation of its name might indicate; coded representation, name, and representative glyph need to be taken in context when establishing the semantics of a character.

4.5. Glyph variation

When a character repertoire is defined (e.g. in a standard), *some* particular glyph is often used to describe the appearance of each character, but this should be taken as an example only. The **Unicode** standard specifically says that great variation is allowed between "representative glyph" appearing in the standard and a glyph used for the corresponding character:

Consistency with the representative glyph does not require that the images be identical or even graphically similar; rather, it means that both images are generally recognized to be representations of the same character. Representing the character U+0061 Latin small letter a by the glyph "X" would violate its character identity.

Thus, the definition of a repertoire is not a matter of just listing *glyphs*, but neither is it a matter of defining exactly the *meanings* of characters. It's actually an exception rather than a rule that a character repertoire definition explicitly says something about the meaning and use of a character.

Possibly some *specific properties* (e.g. being classified as a letter or having numeric value in the sense that digits have) are defined, as in the **Unicode database**, but such properties are rather general in nature.

This vagueness may sound irritating, and it often is. But an essential point to be noted is that **quite a lot of information is implied**. You are expected to deduce what the character is, using both the character name and its representative glyph, and perhaps context too, like the grouping of characters under different headings like "currency symbols".

4.6. Fonts

A repertoire of **glyphs** comprises a *font*. In a more technical sense, as the implementation of a font, a font is a *numbered* set of glyphs. The numbers correspond to **code positions** of the characters (presented by the glyphs). Thus, a font in that sense is character code dependent. An expression like

"Unicode font" refers to such issues and does not imply that the font contains glyphs for *all* **Unicode** characters.

It is possible that a font which is used for the presentation of some character repertoire does not contain a *different* glyph for each character. For example, although characters such as Latin uppercase A, Cyrillic uppercase A, and Greek uppercase alpha are regarded as distinct characters (with distinct code values) in **Unicode**, a particular font might contain just one A which is used to present all of them.

You should never use a character just because it "looks right" or "almost right". Characters with quite different purposes and meanings may well look similar, or almost similar, in some **fonts** at least. Using a character as a surrogate for another for the sake of apparent similarity may lead to great confusion. Consider, for example, the so-called sharp s (es-zed), which is used in the German language. Some people who have noticed such a character in the **ISO Latin 1** repertoire have thought "wow, here we have the beta character!". In many fonts, the sharp s (ß) really looks more or less like the Greek lowercase beta character (β). But it *must not* be used as a surrogate for beta. You wouldn't get very far with it, really; what's the big idea of having beta without alpha and all the other Greek letters? More seriously, the use of sharp s in place of beta would confuse text searches, spelling checkers, indexers, etc.; an automatic converter might well turn sharp s into ss; and some font might present sharp s in a manner which is very different from beta.

4.7. Identity of characters: a matter of definition

The *identity of characters* is defined by the **definition** of a **character repertoire**. Thus, it is not an absolute concept but relative to the repertoire; some repertoire might contain a character with mixed usage while another defines distinct characters for the different uses. For instance, the **ASCII** repertoire has a character called **hyphen**. It is also used as a minus sign (as well as a substitute for a dash, since ASCII contains no dashes). Thus, that ASCII character is a generic, multipurpose character, and one can say that in ASCII hyphen and minus are identical. But in **Unicode**, there are distinct characters named "hyphen" and "minus sign" (as well as different dash characters). For compatibility, the old ASCII character is preserved in Unicode, too (in the old code position, with the name **hyphen-minus**).

Similarly, as a matter of definition, **Unicode** defines characters for **micro sign**, **n-ary product**, etc., as distinct from the **Greek letters** (small mu, capital pi, etc.) they originate from. This is a logical distinction and does not necessarily imply that different glyphs are used. The distinction is important

e.g. when textual data in digital form is processed by a program (which "sees" the code values, through some encoding, and not the glyphs at all). Notice that Unicode does not make any distinction e.g. between the **greek small letter pi** (π), and the mathematical symbol pi denoting the well-known constant 3.14159... (i.e. there is no separate symbol for the latter). For the **ohm sign** (Ω), there is a specific character (in the Symbols Area), but it is defined as being *compatibility* equivalent to **greek capital letter omega** (Ω), i.e. there are two separate characters but they are equivalent. On the other hand, it makes a distinction between greek capital letter pi (Π) and the mathematical symbol n-ary product (\prod), so that they are *not* compatibility equivalents.

If you think this doesn't sound quite logical, you are not the only one to think so. But the point is that for symbols resembling Greek letter and used in various contexts, there are three possibilities in Unicode:

- the symbol is regarded as identical to the Greek letter (just as its particular *usage*)
- the symbol is included as a separate character but only for compatibility and as compatibility equivalent to the Greek letter
- the symbol is regarded as a completely separate character.

You need to check the **Unicode references** for information about each individual symbol. As a rough *rule of thumb* about symbols looking like Greek letters, mathematical *operators* (like summation) exist as independent characters whereas symbols of *quantities and units* (like pi and ohm) are either compatibility characters or identical to Greek letters.

4.8. Failures to display a character

In addition to the fact that **the appearance of a character may vary**, it is quite possible that some program **fails to display a character at all**. Perhaps the program cannot interpret a particular way in which the character is presented. The reason might simply be that some **program-specific way** had been used to denote the character and a different program is in use now. (This happens quite often even if "the same" program is used; for example, Internet Explorer version 4.0 is able to recognize `α` as denoting the Greek letter alpha (α) but IE 3.0 is not and displays the notation literally.) And naturally it often occurs that a program does not recognize the basic **character encoding** of the data, either because it was not properly informed about the encoding according to which the data should be interpreted or because it has not been programmed to handle the particular encoding in use.

But even if a program *recognizes* some data as denoting a character, it may well be unable to display it since it lacks a **glyph** for it. Often it will help if the user manually checks the **font** settings, perhaps manually trying to find a rich enough font. (Advanced programs could be expected to do this automatically and even to pick up glyphs from different fonts, but such expectations are mostly unrealistic at present.) But it's quite possible that no such font can be found. As an important detail, the possibility of seeing e.g. Greek characters on some Windows systems depends on whether "internationalization support" has been installed.

A well-design program will in some appropriate way indicate its inability to display a character. For example, a small rectangular box, the size of a character, could be used to indicate that there is a character which was recognized but cannot be displayed. Some programs use a question mark, but this is risky - how is the reader expected to distinguish such usage from the real "?" character?

4.9. Linear text vs. mathematical notations

Although several character **repertoires**, most notably that of **ISO 10646 and Unicode**, contain **mathematical** and other symbols, the presentation of mathematical **formulas** is essentially not a character level problem. At the character level, symbols like integration or n -ary summation can be defined and their **code positions** and **encodings** defined, and representative **glyphs** shown, and perhaps some usage notes given. But the construction of real formulas, e.g. for a definite integral of a function, is a different thing, no matter whether one considers formulas abstractly (how the structure of the formula is given) or presentationally (how the formula is displayed on paper or on screen). To mention just a few approaches to such issues, the \TeX system is widely used by mathematicians to produce high-quality presentations of formulas, and **MathML** is an ambitious project for creating a markup language for mathematics so that both structure and presentation can be handled.

In other respects, too, character standards usually deal with **plain text** only. Other structural or presentational aspects, such as font variation, are to be handled separately. However, there are characters which would *now* be considered as differing in font only but for historical reasons regarded as distinct.

4.10. Compatibility characters

There is a large number of *compatibility characters* in **ISO 10646 and Unicode** which are variants of other characters. They were included for compatibility with other standards so that data presented using some other **code**

can be converted to ISO 10646 and back without losing information. The Unicode standard says:

Compatibility characters are included in the Unicode Standard only to represent distinctions in other base standards and would not otherwise have been encoded. However, replacing a compatibility character by its decomposition may lose round-trip convertibility with a base standard.

There is a large number of compatibility characters in the **Compatibility Area** but also scattered around the Unicode space. The **Unicode database** contains, for each character, a field (the sixth one) which specifies whether it is a compatibility character as well as its eventual compatibility decomposition.

Thus, to take a simple example, **superscript two** (²) is an **ISO Latin 1** character with its own code position in that standard. In ISO 10646 way of thinking, it would have been treated as just a superscript variant of digit two. But since the character is contained in an important standard, it was included into ISO 10646, though only as a "compatibility character". The practical reason is that now one can convert from ISO Latin 1 to ISO 10646 and back and get the original data. This does not mean that in the ISO 10646 philosophy superscripting (or subscripting, italics, bolding etc.) would be irrelevant; rather, they are to be *handled at another level* of data presentation, such as some special **markup**.

The definition of Unicode indicates our sample character, *superscript two*, as a compatibility character with the *compatibility decomposition* "<super> + 0032 2". Here "<super>" is a semi-formal way of referring to what is considered as typographic variation, in this case superscript style, and "0032 2" shows the hexadecimal code of a character and the character itself.

Some **compatibility characters** have compatibility decompositions consisting of several characters. Due to this property, they can be said to represent *ligatures* in the broad sense. For example, latin small ligature fi (**U+FB01**) has the obvious decomposition consisting of letters "f" and "i". It is still a distinct character in Unicode, but in the spirit of **Unicode**, we should not use it except for storing and transmitting existing data which contains that character. Generally, ligature issues should be handled outside the character level, e.g. selected automatically by a formatting program or indicated using some suitable **markup**.

Note that the *word* ligature can be misleading when it appears in a character name. In particular, the old name of the character "æ", latin small letter ae (**U+00E6**), is latin small ligature ae, but it is *not* a ligature of "a" and "e" in the sense described above. It has no compatibility decomposition.

In *comp.fonts FAQ*, the term *ligature* is defined as follows:

A ligature occurs where two or more letterforms are written or printed as a unit. Generally, ligatures replace characters that occur next to each other when they share common components. Ligatures are a subset of a more general class of figures called "contextual forms."

4.11. Compositions and decompositions

A *diacritic mark*, i.e. an additional graphic such as an accent or cedilla attached to a character, can be treated in different ways when defining a character repertoire. In the **Unicode** approach, there are separate characters called *combining diacritical marks*. The general idea is that you can express a vast set of characters with diacritics by representing them so that a base character is followed by one or more (!) combining (non-spacing) diacritic marks. And a program which *displays* such a construct is expected to do rather clever things in formatting, e.g. selecting a particular shape for the diacritic according to the shape of the base character. This requires Unicode support at **implementation level 3**. Most programs currently in use are totally incapable of doing anything meaningful with combining diacritic marks. But there is some simple support to them in Internet Explorer for example, though you would need a font which contains the combining diacritics (such as **Arial Unicode MS**); then IE can handle simple combinations reasonably.

Thus, in practical terms, in order to use a character with a diacritic mark, you should try to find it as a *precomposed* character. A precomposed character, also called *composite character* or *decomposable character*, is one that has a **code position** (and thereby **identity**) of its own but is in some sense equivalent to a sequence of other characters. There are lots of them in Unicode, and they cover the needs of most (but not all) languages of the world, but not e.g. the presentation of the **International phonetic alphabet** by **IPA** which, in its general form, requires several different diacritic marks. For example, the character latin small letter a with diaeresis (**U+00E4**, ä) is, by Unicode definition, decomposable to the sequence of the two characters latin small letter a (**U+0061**) and combining diaeresis (**U+0308**). This is at present mostly a theoretic possibility. Generally by decomposing all decomposable characters one could in many cases simplify the processing of textual data (and the resulting data might be converted back to a format using precomposed characters).

5. Typing characters

5.1. Just pressing a key?

Typing characters on a computer may appear deceptively simple: you press a key labeled "A", and the character "A" appears on the screen. You also expect "A" to be included into a disk file when you save what you are typing, you expect "A" to appear on paper if you print your text, and you expect "A" to be sent if you send your product by E-mail or something like that. And you expect the recipient to see an "A".

Thus far, you should have learned that the presentation of a character in computer storage or disk or in data transfer may vary a lot. You have probably realized that especially if it's not the common "A" but something more special (say, an "A" with an accent), strange things might happen, especially if data is not accompanied with adequate **information about its encoding**.

But you might still be too confident. You probably expect that on *your* system at least things are simpler than that. If you use your very own, very personal computer and press the key labeled "A" on *its* keyboard, then shouldn't it be evident that in *its* storage and processor, on *its* disk, on *its* screen it's invariably "A"? Can't you just ignore its internal character code and character encoding? Well, probably yes - with "A". I wouldn't be so sure about "Ä", for instance. (On Windows systems, for example, DOS mode programs differ from genuine Windows programs in this respect; they use a DOS character code.)

When you press a key on your **keyboard**, then what actually happens is this. The keyboard sends the code of a character to the processor. The processor then, in addition to storing the data internally somewhere, normally sends it to the display device. Now, the *keyboard settings* and the *display settings* might be different from what you expect. Even if a key is labeled "Ä", it might send something else than the code of "Ä" in the character code used in your computer. Similarly, the display device, upon receiving such a code, might be set to display something different. Such mismatches are usually undesirable, but they are definitely *possible*.

Moreover, there are often *keyboard restrictions*. If your computer uses internally, say, **ISO Latin 1** character repertoire, you probably won't find keys for all 191 characters in it on your keyboard. And for **Unicode**, it would be quite impossible to have a key for each character! Different keyboards are used, often according to the needs of particular languages. For example, keyboards used in Sweden often have a key for the å character but seldom a key for ñ. Quite often some keys have multiple uses via various "**composition**" keys.

5.2. Program-specific methods for typing characters

Thus, you often need program-specific ways of entering characters from a keyboard, either because there is no key for a character you need or there is but it does not work (properly). Three important *examples* of such ways:

- On **Windows** systems, you can (usually - some application programs may override this) produce any character in the **Windows character set** (naturally, in its Windows encoding) as follows: Press down the **Alt key** and keep it down. Then type, using the separate **numeric keypad** (not the numbers above the letter keys!), the four-digit code of the character in decimal. Finally release the Alt key. Notice that the first digit is always 0, since the code values are in the range 32 - 255 (decimal). For instance, to produce the letter "Ä" (which has code 196 in decimal), you would press Alt down, type 0196 and then release Alt. Upon releasing Alt, the character should appear on the screen. In MS Word, the method works only if Num Lock is set. This method is often referred to as *Alt-0nnn*. (If you omit the leading zero, i.e. use *Alt-nnn*, the effect is *different*, since that way you insert the character in code position *nnn* in the **DOS character code**! For example, Alt-196 would probably insert a graphic character which looks somewhat like a hyphen. There are variations in the behavior of various Windows programs in this area, and using those DOS codes is best avoided).
- In the **Emacs** editor (which is popular especially on Unix systems), you can produce any **ISO Latin 1** character by typing first control-Q, then its code as a three-digit **octal** number. To produce "Ä", you would thus type control-Q followed by the three digits 304 (and expect the "Ä" character to appear on screen). This method is often referred to as *C-Q-nnn*. (There are other ways of entering many ISO Latin 1 characters in Emacs, too.)
- Programs often process some **keyboard key combinations**, often involving the use of an **Alt** or Alt Gr key or some other "composition key", by converting them to special characters. In fact, even the well-known shift key is a composition key: it is used to modify the meaning of another key, e.g. by changing a letter to uppercase or turning a digit key to a special character key. Such things are not just "program-specific"; they also *depend on the program version and settings* (and on the keyboard, of course). For example, in order to support the **euro sign**, various methods have been developed, e.g. by Microsoft so that pressing the "e" key while keeping the Alt Gr key pressed down might produce the euro sign - in *some encoding*! But this may require a special "euro update", and **the key combinations vary** even when we consider Microsoft products

only. So it would be quite inappropriate to say e.g. "to type the euro, use AltGr+e" as general, unqualified advice.

The last method above could often be called "device dependent" rather than program specific, since the program that performs the conversion might be a keyboard **driver**. In that case, normal programs would have all their input from the keyboard processed that way. This method may also involve the use of auxiliary keys for typing characters with **diacritic marks** such as "á". Such an auxiliary key is often called *dead key*, since just pressing it causes nothing; it works only in combination with some other key. For example, depending on the keyboard and the driver, you *might* be able to produce "á" by pressing first a key labeled with the acute accent (´), then the "a" key. *My* keyboard has two keys for such purposes: one with the acute accent and the grave accent (`) above it (meaning I need to use the **shift key** for it) and one with the dieresis (¨) and the circumflex (ˆ) above it and the tilde (~) below or left to it (meaning I need to use Alt Gr for it), so I can produce **ISO Latin 1** characters with those diacritics. Note that this does *not* involve any operation on the *characters* "¨ ~ - the keyboard does not send those characters at all in such situations. If I try to enter that way a character outside the ISO Latin 1 repertoire, I get just the diacritic as a separate character followed by the normal character, e.g. " ¨ j". To enter the diacritic itself, such as the **tilde** (~), I may need to press the space bar so that the tilde diacritic combines with the blank (producing ~) instead of a letter (producing e.g. "ã"). Your situation may well be different, in part or entirely. For example, a typical French keyboard has separate keys for those accented characters which are used in French (e.g. "à") and no key for the accents themselves, but there is a key for attaching the circumflex or the dieresis in the manner outlined above.

5.3. "Escape" notations ("meta notations") for characters

It is often possible to use various *"escape" notations* for characters. This rather vague term means notations which are *afterwards* converted to (or just displayed as) characters according to some specific rules by some programs. They depend on the markup, programming, or other *language* (in a broad but technical meaning for "language", so that data formats can be included but human languages are excluded). If different languages have similar conventions in this respect, a language designer may have picked up a notation from an existing language, or it might be a coincidence.

The phrase "escape notations" or even "escapes" for short is rather widespread, and it reflects the general idea of escaping from the limitations of a character repertoire or device or protocol or something else. So it's used

here, although a name like *meta notations* might be better. It is any case essential to distinguish these notations from the use of the ESC (escape) **control code** in **ASCII** and other character codes.

Examples:

- In the **PostScript** language, characters have *names*, such as **A**dieresis for **Ä**, which can be used to denote them according to certain rules.
- In the **RTF** data format, the notation `\c4` is used to denote **Ä**.
- In **T_EX** systems, there are different ways of producing characters, possibly depending on the "packages" used. Examples of ways to produce **Ä**: `\ "A`, `\symbol{196}`, `\char'0304`, `\capitaldieresis{A}`.
- In the **HTML language** one can use the notation `Ä` for **the character Ä**. In the official HTML terminology, such notations are called **entity references (denoting characters)**. It depends on HTML version which entities are defined, and it depends on a browser **which entities are actually supported**.
- In HTML, one can also use the notation `Ä` for **the character Ä**. Generally, in any **SGML** based system, or "SGML application" as the jargon goes, a *numeric character reference* (or, actually, just *character references*) of the form `&#number;` can be used, and it refers to the character which is in code position *n* in the **character code** defined for the "SGML application" in question. This is actually very simple: you specify a character by its index (position, number). But in SGML terminology, the character code which determines the interpretation of `&#number;` is called, quite confusingly, the *document character set*. For HTML, the "document character set" is **ISO 10646** (or, to be exact, a subset thereof, depending on HTML version). A most essential point is that for HTML, the "document character set" is completely independent of the **encoding** of the document! The so-called *character entity references* like `Ä` in HTML can be regarded as symbolic names defined for some numeric character references.
- In the **C programming language**, one can usually write `\0304` to denote **Ä** within a string constant, although this makes the program character code dependent.

As you can see, the notations typically involve some (semi-)mnemonic name or the **code number** of the character, in some **number system**. (The **ISO 8859-1** code number for our example character **Ä** is 196 in decimal, 304 in octal, C4 in hexadecimal). And there is some method of indicating that the

letters or digits are not to be taken as such but as part of a special notation denoting a character. Often some specific character such as the **backslash** `\` is used as an "escape character". This implies that such a character cannot be used as such in the language or format but must itself be "escaped"; for example, to include the backslash itself into a string constant in C, you need to write it twice (`\\`).

In cases like these, the character itself does not occur in a file (such as an HTML document or a C source program). Instead, the file contains the "escape" notation as a character sequence, which will then be *interpreted* in a specific way by programs like a Web browser or a C compiler. One can in a sense regard the "escape notations" as **encodings** used in specific contexts upon specific agreements.

There are also "escape notations" which are to be interpreted by **human readers** directly. For example, when sending E-mail one might use `A"` (letter A followed by a quotation mark) as a surrogate for `Ä` (letter A with dieresis), *or* one might use `AE` instead of `Ä`. The reader is assumed to understand that e.g. `A"` on display actually means `Ä`. Quite often the purpose is to use ASCII characters only, so that the typing, transmission, and display of the characters is "safe". But this typically means that text becomes very messy; the Finnish word *Hämäläinen* does not look too good or readable when written as *Ha"ma"la"inen* or *Haemaelaeinen*. Such usage is based on special (though often implicit) conventions and can cause a lot of confusion when there is no mutual agreement on the conventions, especially because there are so many of them. (For example, to denote letter a with acute accent, `á`, a convention might use the apostrophe, `a'`, or the solidus, `a/`, or the acute accent, `a'`, or something else).

5.4. How to mention (identify) a character

There are also various ways to **identify** a character when it cannot be used as such or when the appearance of a character is not sufficient identification. This might be regarded as a variant of the *"escape notations for human readers"* discussed above, but the pragmatic view is different here. We are not primarily interested in *using* characters in running text but in *specifying* which character is being discussed.

For example, when discussing the **Cyrillic letter that resembles the Latin letter E** (and may have an identical or very similar glyph, and is transliterated as E according to **ISO 9**), there are various options:

- "Cyrillic E"; this is probably intuitively understandable in this case, and can be seen as referring *either* to the similarity of shape *or* to the translit-

eration equivalence; but in the general case these interpretations do not coincide, and the method is otherwise vague too

- "U+0415"; this is a unique identification but requires the reader to know the idea of **U+nnnn notations**
- "cyrillic capital letter ie" (using the official Unicode **name**) or "cyrillic IE" (using an abridged version); one problem with this is that the names can be long even if simplified, and they still cannot be assumed to be universally known even by people who recognize the character
- "KE02", which uses the special notation system defined in **ISO 7350**; the system uses a compact notation and is marginally mnemonic (**K** = *kirillica* 'Cyrillics'; the numeric codes indicate small/capital letter variation and the use of **diacritics**)
- any of the **"escape" notations** discussed above, such as "E=" by **RFC 1345** or "Е" in HTML; this can be quite adequate in a context where the reader can be assumed to be familiar with the particular notation.

6. Information about encoding

6.1. The need for information about encoding

It is hopefully obvious from the preceding discussion that *a sequence of octets can be interpreted in a multitude of ways* when processed as character data. By looking at the octet sequence only, you cannot even know whether each octet presents one character or just part of a two-octet presentation of a character, or something more complicated. Sometimes one can guess the encoding, but data processing and transfer shouldn't be guesswork.

Naturally, a sequence of octets could be intended to present other than character data, too. It could be an image in a bitmap format, or a computer program in binary form, or numeric data in the internal format used in computers.

This problem can be handled in different ways in different systems when data is stored and processed within one computer system. For *data transmission*, a platform-independent method of specifying the general format and the encoding and other relevant information is needed. Such methods exist, although they not always used widely enough. People still send each other data without specifying the encoding, and this may cause a lot of harm. Attaching a human-readable note, such as a few words of explanation in an E-mail

message body, is better than nothing. But since data is processed by programs which cannot understand such notes, the encoding should be specified in a standardized computer-readable form.

6.2. The MIME solution

Media types *Internet media types*, often called *MIME media types*, can be used to specify a major media type ("top level media type", such as **text**), a subtype (such as **html**), and an encoding (such as **iso-8859-1**). They were originally developed to allow sending other than plain **ASCII** data by E-mail. They can be (and should be) used for specifying the encoding when data is sent over a network, e.g. by E-mail or using the **HTTP** protocol on the World Wide Web.

Character encoding ("charset") information The technical term used to denote a **character encoding** in the Internet media type context is "character set", abbreviated "charset". This has caused a lot of confusion, since "set" can easily be understood as **repertoire**!

Specifically, when data is sent in MIME format, the media type and encoding are specified in a manner illustrated by the following example:

```
Content-Type: text/html; charset=iso-8859-1
```

This specifies, in addition to saying that the media type is **text** and subtype is **html**, that the character encoding is **ISO 8859-1**.

The official registry of "charset" (i.e., character encoding) names, with references to documents defining their meanings, is kept by IANA at <http://www.iana.org/assignments/character-sets>.

Several character encodings have alternate (alias) names in the registry. For example, the basic (ISO 646) variant of **ASCII** can be called "ASCII" or "ANSI_X3.4-1968" or "cp367" (plus a few other names); the preferred name in **MIME** context is, according to the registry, "US-ASCII". Similarly, **ISO 8859-1** has several names, the preferred MIME name being "ISO-8859-1". The "native" encoding for Unicode, **UCS-2**, is named "ISO-10646-UCS-2" there.

MIME headers The Content-Type information is an example of information in a *header*. Headers relate to some data, describing its presentation and other things, but are passed as logically separate from it. Possible headers and their contents are defined in **the basic MIME specification**, RFC 2045. Adequate headers should normally be generated automatically by the software which sends the data (such as a program for sending E-mail, or a Web server) and interpreted automatically by receiving software (such as a program for reading E-mail, or a Web browser). In E-mail messages, headers precede the

message body; it depends on the E-mail program whether and how it displays the headers. For Web documents, a Web server is required to send headers when it delivers a document to a browser (or other user agent) which has sent a request for the document.

6.3. An auxiliary encoding: Quoted-Printable (QP)

The MIME specification defines, among many other things, the general purpose "Quoted-Printable" (QP) **encoding** which can be used to present any sequence of **octets** as a sequence of such octets which correspond to **ASCII** characters. This implies that the sequence of octets becomes longer, and if it is read as an ASCII string, it can be incomprehensible to humans. But what is gained is robustness in data transfer, since the encoding uses only "safe" ASCII characters which will most probably get through any component in the transfer unmodified.

Basically, QP encoding means that most octets smaller than 128 are used as such, whereas larger octets and some of the small ones are presented as follows: octet n is presented as a sequence of three octets, corresponding to ASCII codes for the = sign and the two digits of the hexadecimal notation of n . If QP encoding is applied to a sequence of octets presenting character data according to **ISO 8859-1** character code, then effectively this means that most ASCII characters (including all ASCII letters) are preserved as such whereas e.g. the ISO 8859-1 character **ä** (code position 228 in decimal, E4 in hexadecimal) is encoded as =E4. (For obvious reasons, the equals sign = itself is among the few ASCII characters which are encoded. Being in code position 61 in decimal, 3D in hexadecimal, it is encoded as =3D.)

Notice that encoding ISO 8859-1 data this way means that the *character code* is the one specified by the ISO 8859-1 standard, whereas the *character encoding* is different from the one specified (or at least suggested) in that standard. Since QP only specifies the mapping of a sequence of octets to another sequence of octets, it is a pure encoding and can be applied to any character data, or to any data for that matter.

Naturally, Quoted-Printable encoding needs to be processed by a program which knows it and can convert it to human-readable form. It looks rather confusing when displayed as such. Roughly speaking, one can expect most *E-mail* programs to be able to handle QP, but the same does not apply to *newsreaders* (or Web browsers). Therefore, you should normally use QP in E-mail only.

6.4. How MIME should work in practice

Basically, MIME should let people communicate smoothly without hindrances caused by character code and encoding differences. MIME should handle the necessary conversions automatically and invisibly.

For example, when person *A* sends E-mail to person *B*, the following should happen: The E-mail program used by *A* encodes *A*'s message in some particular manner, probably according to some convention which is normal on the system where the program is used (such as **ISO 8859-1** encoding on a typical modern Unix system). The program automatically includes information about this encoding into an E-mail header, which is usually invisible both when sending and when reading the message. The message, with the headers, is then delivered, through network connections, to *B*'s system. When *B* uses his E-mail program (which may be very different from *A*'s) to read the message, the program should automatically pick up the information about the encoding as specified in a header and interpret the message body according to it. For example, if *B* is using a Macintosh computer, the program would automatically convert the message into **Mac's internal character encoding** and only then display it. Thus, if the message was **ISO 8859-1** encoded and contained the Ä (upper case A with dieresis) character, encoded as octet 196, the E-mail program used on the Mac should use a conversion table to map this to octet 128, which is the encoding for Ä on Mac. (If the program fails to do such a conversion, strange things will happen. **ASCII** characters would be displayed correctly, since they have the same codes in both encodings, but instead of Ä, the character corresponding to octet 196 in Mac encoding would appear - a symbol which looks like *f* in italics.)

6.5. Problems with implementations - examples

Unfortunately, there are deficiencies and errors in software so that *users* often have to struggle with character code conversion problems, perhaps correcting the actions taken by programs. It takes two to tango, and some more participants to get characters right. This section demonstrates different things which may happen, and do happen, when just *one* component is faulty, i.e. when **MIME is not used or is inadequately supported by some "partner"** (software involved in entering, storing, transferring, and displaying character data).

Typical minor (!) problems which may occur in communication in Western European languages other than English is that most characters get interpreted and displayed correctly but some "national letters" don't. For example, character repertoire needed in German, Swedish, and Finnish is essentially **ASCII**

plus a few letters like "ä" from the rest of **ISO Latin 1**. If a text in such a language is processed so that a necessary conversion is not applied, or an incorrect conversion is applied, the result might be that e.g. the word "später" becomes "spter" or "spĭter" or "spdter" or "sp=E4ter", to mention just a few possibilities. People familiar with such problems might be able to read the distorted text too, but others may get seriously confused.

7. Practical conclusions

Whenever text data is sent over a network, the sender and the recipient should have a joint **agreement on the character encoding** used. In the optimal case, this is handled by the software automatically, but in reality the users need to take some precautions.

Most importantly, make sure that any **Internet-related software** that you use to send data **specifies the encoding** correctly in suitable headers. There are two things involved: the header must be there and it must reflect the actual encoding used; and the encoding used must be one that is widely understood by the (potential) recipients' software. One must often make compromises as regards to the latter aim: you may need to use an encoding which is not yet widely supported to get your message through at all.

It is useful to find out how to make your Web browser, newsreader, and E-mail program so that you can display the encoding information for the page, article, or message you are reading. (For example, on Netscape use View Page Info; on News Xpress, use View Raw Format; on Pine, use h.)

If you use, say, Netscape to send E-mail or to post to Usenet news, make sure it sends the message in a reasonable form. In particular, **make sure it does not send the message as HTML** or duplicate it by sending it both as plain text and as HTML (select plain text only). As regards to character encoding, make sure it is something widely understood, such as **ASCII**, some **ISO 8859** encoding, or **UTF-8**, depending on how large character repertoire you need.

In particular, **avoid sending data in a proprietary encoding** (like the **Macintosh encoding** or a **DOS encoding**) to a public network. At the very least, if you do that, make sure that the message heading specifies the encoding! There's nothing wrong with using such an encoding within a single computer or in data transfer between similar computers. But when sent to Internet, data should be converted to a more widely known encoding, by the sending program. If you cannot find a way to configure your program to do that, get another program.

As regards to other forms of transfer of data in digital form, such as diskettes, information about encoding is important, too. The problem is typically handled by guesswork. Often the crucial thing is to know which *program* was used to generate the data, since the text data might be inside a file in, say, the MS Word format which can only be read by (a suitable version of) MS Word or by a program which knows its internal data format. That format, once recognized, might contain information which specifies the character encoding used in the text data included; or it might not, in which case one has to ask the sender, or make a guess, or use trial and error - viewing the data using different encodings until something sensible appears.

Jukka Korpela

This text is an abridged version of the author's document at <http://www.cs.tut.fi/~jkorpe/chars.html>, which contains some additional details as well as links to further information on the topics discussed.