

## Chapter 6

### Character encoding

This chapter is about how to interpret the characters in an input file – no there ain't such a thing as a plain text file – and how the printed characters are encoded in a font.

#### Handouts and further reading for this chapter

There is very little printed material on this topic. A good introduction is <http://www.joelonsoftware.com/articles/Unicode.html>; after that, <http://www.cs.tut.fi/~jkorpela/chars.html> is a good tutorial for general issues, and <http://en.wikipedia.org/wiki/Unicode> for Unicode.

For the technical details on Unicode consult <http://www.unicode.org/>. An introduction to ISO 8859: [http://www.wordiq.com/definition/ISO\\_8859](http://www.wordiq.com/definition/ISO_8859).

## Input file encoding.

### 6.1 History and context

#### 6.1.1 One-byte character sets; Ascii

Somewhere in the depths of prehistory, people got to agree on a standard for character codes under 127, ASCII. Unlike another encoding scheme, EBCDIC, it has a few nice properties.

- All letters are consecutive, making a test ‘is this a letter’ easy to perform.
- Uppercase and lowercase letters are at a distance of 32.
- The first 31 codes, everything below the space character, as well as position 127, are ‘unprintable’, and can be used for such purposes as terminal cursor control.
- Unprintable codes are accessible through the control modifier (for this reason they are also called ‘control codes’), which zeros bits 2 and 3: hit `Ctrl-[` to get `Esc`.

The ISO 646 standard codified 7-bit ASCII, but it left certain character positions (or ‘code points’) open for national variation. For instance, British usage put a pound sign (£) in the position of the dollar. The ASCII character set was originally accepted as ANSI X3.4 in 1968.

20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
p	q	r	s	t	u	v	w	x	y	z	{		}	~	

#### 6.1.2 Code pages

This left the codes with the high bit set (‘extended ASCII’) undefined, and different manufacturers of computer equipment came up with their own way of filling them in. These standards were called ‘code pages’, and IBM gave a standard numbering to

1. The way key presses generate characters is typically controlled in software. This mapping from keyboard scan codes to 7 or 8-bit characters is called a ‘keyboard’, and can be changed dynamically in most operating systems.

them. For instance, code page 437 is the MS-DOS code page with accented characters for most European languages, 862 is DOS in Israel, 737 is DOS for Greeks.

Here is cp473:

	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	DA	DB	DC	DD	DE	DF	00
E0	E1	E2	E3	E4	E5	E6	E7	E8	EA	EB	EC	ED	EE	EF	01
F0	F1	F2	F3	F4	F5	F6	F7	F8	FA	FB	FC	FD	FE	FF	02

MacRoman:

80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	DA	DB	DC	DD	DE	DF	00
E0	E1	E2	E3	E4	E5	E6	E7	E8	EA	EB	EC	ED	EE	EF	01
F0	F1	F2	F3	F4	F5	F6	F7	F8	FA	FB	FC	FD	FE	FF	02

and Microsoft cp1252:

80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	DA	DB	DC	DD	DE	DF	00
E0	E1	E2	E3	E4	E5	E6	E7	E8	EA	EB	EC	ED	EE	EF	01
F0	F1	F2	F3	F4	F5	F6	F7	F8	FA	FB	FC	FD	FE	FF	02

More code pages are displayed on <http://aspell.net/charsets/codepages>.

html These diagrams can be generated from Unicode mapping tables, which look like

```
=20      U+0020  SPACE
=21      U+0021  EXCLAMATION MARK
=22      U+0022  QUOTATION MARK
...
=A3      U+00A3  POUND SIGN
=A4      U+20AC  EURO SIGN
=A5      U+00A5  YEN SIGN
...
```

The international variants were standardized as ISO 646-DE (German), 646-DK (Danish), et cetera. Originally, the dollar sign could still be replaced by the currency symbol, but after a 1991 revision the dollar is now the only possibility.

### 6.1.3 ISO 8859

The different code pages were ultimately standardized as ISO 8859, with such popular code pages as 8859-1 (‘Latin 1’) for western European,

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
	í	φ	£	¥	¢	§	·	©	≡	«	¬	—	®	—	
B0	ó	±	²	³	´	µ	¶	·	¸	»	¼	½	¾	¿	
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ

8859-2 for eastern, and 8859-5 for Cyrillic.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
	É	Ђ	Ѓ	Ѕ	€	Š	Ї	Ј	Љ	Њ	Ћ	Ќ	—	Ў	Ў
B0	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О
C0	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю
D0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о
E0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю
F0	ђ	ѓ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ	ѕ

These ISO standards explicitly left the first 32 extended positions undefined. Microsoft code page 1252 uses ISO 8859-1.

More useful information about ASCII: <http://jimprice.com/jim-asc.htm>. History of ASCII out of telegraph codes: <http://www.wps.com/projects/codes/index.html>. A history, paying attention to multilingual use: <http://tronweb.super-nova.co.jp/characcodehist.html> History as written

by the father of ASCII: Bob Bemer <http://www.bobbemer.com/HISTORY.HTM>

A good inventory of ISO 8859, Latin-1: <http://www.cs.tut.fi/~jkorpela/latin1/index.html>, with a discussion by logical grouping: <http://www.cs.tut.fi/~jkorpela/latin1/4.html>.

#### 6.1.4 DBCS

Since certain Asian alphabets do not fit in 256 positions, a system called the ‘Double Byte Character Set’ was invented where some characters were stored in one, others in two bytes. This is very messy, since you can not simply write `s++` or `s--` to traverse a string. Instead you have to use functions from some library that understands these encodings. This system is now only of historical interest.

## 6.2 Unicode

The systems above functioned quite well as long as you stuck to one language or writing system. Poor dictionary makers. More or less simultaneously two efforts started that aimed to incorporate all the world’s character sets in one standard: Unicode standard (originally 2-byte), and ISO 10646 (originally 4-byte). Unicode then was extended, so that it has all numbers up to `10FFFFFF`, which is slightly over a million.

### 6.2.1 ISO 10646 and Unicode

Two international standards organizations, the Unicode Consortium and ISO/IEC JTC1/SC2, started designing a universal standard that was to be a superset of all existing character sets. These standards are now synchronized. Unicode has elements that are not in 10646, but they are compatible where it concerns straight character encoding.

ISO 10646 defines UCS, the ‘Universal Character Set’. This is in essence a table of official names and code numbers for characters. Unicode adds to this rules for hyphenation, bi-directional writing, and more.

The full Unicode list of code points can be found, broken down by blocks, online at <http://www.fileformat.info/info/unicode/index.htm>, or downloadable at <http://www.unicode.org/charts/>.

### 6.2.2 BMP and earlier standards

Characters in Unicode are mostly denoted hexadecimally as U+wx<sub>1</sub>yz, for instance U+0041 is ‘Latin Capital Letter A’. The range U+0000–U+007F (0–127) is identical to US-ASCII (ISO 646 IRV), and U+0000–U+00FF (0–255) is identical to Latin 1 (ISO 8859-1).

The original 2-byte subset is now called ‘BMP’ for Basic Multilingual Plane.

From <http://www.hyperdictionary.com/>:

**BMP** (Basic Multilingual Plane) The first plane defined in Unicode/ISO 10646, designed to include all scripts in active modern use. The BMP currently includes the Latin, Greek, Cyrillic, Devangari, hiragana, katakana, and Cherokee scripts, among others, and a large body of mathematical, APL-related, and other miscellaneous characters. Most of the Han ideographs in current use are present in the BMP, but due to the large number of ideographs, many were placed in the Supplementary Ideographic Plane.

**SIP** (Supplementary Ideographic Plane) The third plane (plane 2) defined in Unicode/ISO 10646, designed to hold all the ideographs descended from Chinese writing (mainly found in Vietnamese, Korean, Japanese and Chinese) that aren’t found in the Basic Multilingual Plane. The BMP was supposed to hold all ideographs in modern use; unfortunately, many Chinese dialects (like Cantonese and Hong Kong Chinese) were overlooked; to write these, characters from the SIP are necessary. This is one reason even non-academic software must support characters outside the BMP.

### 6.2.3 Unicode encodings

Unicode is basically a numbered list of characters. When they are used in a file, their numbers can be encoded in a number of ways. To name the obvious example: if only the first 128 positions are used, the long Unicode code point can be truncated to just one byte. Here are a few encodings:

**UCS-2** Obsolete: this was the original ‘native’ two-byte encoding before Unicode was extended.

**UTF-32** Little used: this is a four-byte encoding. (UTF stands for ‘UCS Transformation Format’.)

**UTF-16** This is the BMP.

**UTF-8** A one-byte scheme; details below.

**UTF-7** Another one-byte scheme, but now the high bit is always off. Certain byte values act as ‘escape’, so that higher values can be encoded. Like UTF-1 and SCSU, this encoding is only of historical interest.

There is an important practical reason for UTF-8. Encodings such as UCS-2 are wasteful of space, if only traditional ASCII is needed. Furthermore, they would break software that is expecting to walk through a file with `s++` and such. Also, they would introduce many zero bytes in a file, which would play havoc with Unix software that uses null-termination for strings. Then there would be the problem of whether two bytes are stored in low-endian or high-endian order. For this reason it was suggested to store `FE FF` or `FF FE` at the beginning of each file as the ‘Unicode Byte Order Mark’. Of course this plays havoc with files such as shell scripts which expect to find `#!` at the beginning of the file.

### 6.2.4 UTF-8

UTF-8 is an encoding where the positions up to 127 are encoded ‘as such’; higher numbers are encoded in groups of 2 to 6 bytes. (UTF-8 is standardized as RFC 3629.) In a multi-byte group, the first byte is in the range `0xC0–0xFD` (192–252). The next up to 5 bytes are in the range `0x80–0xBF` (128–191, bit pattern starting with `10`). Note that `8 = 1000` and `B = 1011`, so the highest two bits are always `10`, leaving six bits for encoding).

U-00000000 – U-0000007F	7 bits	0xxxxxxx	
U-00000080 – U-000007FF	11 = 5 + 6	110xxxxx	10xxxxxx
U-00000800 – U-0000FFFF	16 = 4 + 2 × 6	1110xxxx	10xxxxxx 10xxxxxx
U-00010000 – U-001FFFFF	21 = 3 + 3 × 6	11110xxx	10xxxxxx (3 times)
U-00200000 – U-03FFFFFF	26 = 2 + 4 × 6	111110xx	10xxxxxx (4 times)
U-04000000 – U-7FFFFFFF	31 = 1 + 5 × 6	1111110x	10xxxxxx (5 times)

All bites in a multi-byte sequence have their high bit set.

**Exercise 49.** Show that a UTF-8 parser will not miss more than two characters if a byte becomes damaged (any number of bits arbitrarily changed).

IETF documents such as RFC 2277 require support for this encoding in internet software. Here is a good introduction to UTF-8 use in Unix: <http://www.cl.cam.ac.uk/~mgk25/unicode.html>. The history of it: <http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>.

### 6.2.5 Unicode tidbits

#### 6.2.5.1 Line breaking

See <http://www.cs.tut.fi/~jkorpela/unicode/linebr.html> and <http://www.unicode.org/reports/tr14/>

### 6.2.5.2 Bi-directional writing

Most scripts are left-to-right, but Arabic and Hebrew run right-to-left. Characters in a file are stored in ‘logical order’, and usually it is clear in which direction to render them, even if they are used mixed. Letters have a ‘strong’ directionality: unless overridden, they will be displayed in their natural direction. The first letter of a paragraph with strong direction determines the main direction of that paragraph.

أوروبا, برمجيات الحاسوب + انترنت :  
تصبح عالميا مع يونيكود

تسجل الآن لحضور المؤتمر الدولي العاشر ليونيكود الذي سيعقد في 10-12 آذار 1997 بمدينة ماينتس ألمانيا. وسيجمع المؤتمر بين خبراء من كافة قطاعات الصناعة على الشبكة العالمية انترنت ويونيكود حيث سيتم على الصعيدين الدولي والمحلي على حد سواء مناقشة سبل استخدام يونيكود في النظم القائمة وفيما يخص التطبيقات الحاسوبية الخطوط تصميم النصوص والحوسبة متعددة اللغات.  
عندما يريد العالم أن يتكلم فهو يتحدث بلغة يونيكود.

However, when differently directional texts are embedded, some explicit help is needed. The problem arises with letters that have only weak directionality. The following is a sketch of a problematic case.

Memory: he said "I NEED WATER!", and expired.

Display: he said "RETAW DEEN I!", and expired.

If the exclamation mark is to be part of the Arabic quotation, then the user can select the text ‘I NEED WATER!’ and explicitly mark it as embedded Arabic (<RLE> is Right-Left Embedding; <PDF> Pop Directional Format), which produces the following result:

Memory: he said "<RLE>I NEED WATER!<PDF>", and expired.

Display: he said "!RETAW DEEN I", and expired.

A simpler method of doing this is to place a Right Directional Mark <RLM> after the exclamation mark. Since the exclamation mark is now not on a directional boundary, this produces the correct result.

Memory: he said "I NEED WATER!<RLM>", and expired.

Display: he said "!RETAW DEEN I", and expired.

For the full definition, see <http://www.unicode.org/reports/tr9/>.

### 6.2.6 Unicode and oriental languages

‘Han unification’ is the Unicode strategy of saving space in the oriental languages (traditional Chinese, simplified Chinese, Japanese, Korean: ‘CJK’) by recognizing common characters. This idea is not uncontroversial; see [http://en.wikipedia.org/wiki/Han\\_unification](http://en.wikipedia.org/wiki/Han_unification).



## 6.3 More about character sets and encodings

### 6.3.1 Character sets

Informally, the term ‘character set’ (also ‘character code’ or ‘code’) used to mean something like ‘a table of bytes, each with a character shape’. With only the English alphabet to deal with that is a good enough definition. These days, much more general cases are handled, mapping one octet into several characters, or several octets into one character. The definition has changed accordingly:

A ‘*charset*’ is a method of converting a sequence of octets into a sequence of characters. This conversion may also optionally produce additional control information such as directionality indicators.

(From RFC 2978) A conversion the other way may not exist, since different octet combinations may map to the same character. Another complicating factor is the possibility of switching between character sets; for instance, ISO 2022-JP is the standard ASCII character set, but the escape sequence `ESC $ @` switches to JIS X 0208-1978.

### 6.3.2 From character to encoding in four easy steps

To disentangle the concepts behind encoding, we need to introduce a couple of levels:

**ACR** Abstract Character Repertoire: the set of characters to be encoded; for example, some alphabet or symbol set. This is an unordered set of characters, which can be fixed (the contents of ISO 8859-1), or open (the contents of Unicode).

**CCS** Coded Character Set: a mapping from an abstract character repertoire to a set of nonnegative integers. This is what is meant by ‘encoding’, ‘character set definition’, or ‘code page’; the integer assigned to a character is its ‘code point’.

There used to be a drive towards unambiguous abstract character names across repertoires and encodings, but Unicode ended this, as it provides (or aims to provide) more or less a complete list of every character on earth.

**CEF** Character Encoding Form: a mapping from a set of nonnegative integers that are elements of a CCS to a set of sequences of particular code units. A ‘code unit’ is an integer of a specific binary width, for instance 8 or 16 bits. A CEF then maps the code points of a coded character set into sequences of code point, and these sequences can be of different lengths inside one code page. For instance

- ASCII uses a single 7-bit unit
- UCS-2 uses a single 16-bit unit
- DBCS uses two 8-bit units
- UTF-8 uses one to four 8-bit units.

- UTF-16 uses a mix of one and two 16-bit code units.

**CES** Character Encoding Scheme: a reversible transformation from a set of sequences of code units (from one or more CEFs to a serialized sequence of bytes. In cases such as ASCII and UTF-8 this mapping is trivial. With UCS-2 there is a single ‘byte order mark’, after which the code units are trivially mapped to bytes. On the other hand, ISO 2022, which uses escape sequences to switch between different encodings, is a complicated CES.

Additionally, there are the concepts of

**CM** Character Map: a mapping from sequences of members of an abstract character repertoire to serialized sequences of bytes bridging all four levels in a single operation. These maps are what gets assigned MIBenum values by IANA; see section [6.3.3](#)

**TES** Transfer Encoding Syntax: a reversible transform of encoded data. This data may or may not contain textual data. Examples of a TES are base64, uuencode, and quoted-printable, which all transform a byte stream to avoid certain values.

### 6.3.3 A bootstrapping problem

In order to know how to interpret a file, you need to know what character set it uses. This problem also occurs in MIME mail encoding (section [6.3.5](#)), which can use many character sets. Names and numbers for character sets are standardized by IANA: the Internet Assigned Names Authority (<http://www.iana.org/>). However, in what character set do you write this name down?

Fortunately, everyone agrees on (7-bit) ASCII, so that is what is used. A name can be up to 40 characters from us-ascii.

As an example, here is the iana definition of ASCII:

```
name ANSI_X3.4-1968
reference RFC1345,KXS2
MIBenum 3
source ECMA registry
aliases iso-ir-6,ANSI_X3.4-1986,ISO_646.irv:1991,ASCII,ISO646-US,
US-ASCII (preferred MIME name),us,IBM367,cp367,csASCII
```

The MIBenum (Management Information Base) is a number assigned by IANA<sup>2</sup>. The full list of character sets is at <http://www.iana.org/assignments/character-sets>, and RFC 3808 is a memo that describes the IANA Charset MIB.

2. Apparently these numbers derive from the Printer MIB, RFC 1759.

### 6.3.4 Character codes in HTML

HTML can access unusual characters in several ways:

- With a decimal numerical code: `&#32;` is a space token. (HTML 4 supports hexadecimal codes.)
- With a vaguely symbolic name: `&copy;` is the copyright symbol. See <http://www.cs.tut.fi/~jkorpela/HTML3.2/latin1.html> for a list of symbolic names in Latin-1.
- The more interesting way is to use an encoding such as UTF-8 (section 6.2.3) for the file. For this it would be nice if the server could state that the file is  
Content-type: text/html; charset=utf-8  
but it is also all right if the file starts with

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8">
```

Description	Char Code	Entity name
non-breaking space	<code>&amp;#160;</code> -->	<code>&amp;nbsp;</code> -->
inverted exclamation	<code>¡</code> <code>&amp;#161;</code> --> <code>¡</code>	<code>&amp;iexcl;</code> --> <code>¡</code>
cent sign	<code>¢</code> <code>&amp;#162;</code> --> <code>¢</code>	<code>&amp;cent;</code> --> <code>¢</code>
pound sterling	<code>£</code> <code>&amp;#163;</code> --> <code>£</code>	<code>&amp;pound;</code> --> <code>£</code>
general currency sign	<code>¤</code> <code>&amp;#164;</code> --> <code>¤</code>	<code>&amp;curren;</code> --> <code>¤</code>
yen sign	<code>¥</code> <code>&amp;#165;</code> --> <code>¥</code>	<code>&amp;yen;</code> --> <code>¥</code>
broken vertical bar	<code> </code> <code>&amp;#166;</code> --> <code> </code>	<code>&amp;brvbar;</code> --> <code> </code>

It is requirement that user agents can at least parse the `charset` parameter, which means they have to understand us-ascii.

Open this link in your browser, and additionally view the source: <http://www.unicode.org/unicode/iuc10/x-utf8.html>. How well does your software deal with it?

See also section 6.7.1.

### 6.3.5 Characters in email

### 6.3.6 FTP

FTP is a very old ARPA protocol. It knows ‘binary’ and ‘text’ mode, but the text mode is not well defined. Some ftp programs adjust line ends; others, such as `Fetch` on the Mac, actually do code page translation.

### 6.3.7 Character encodings in editors and programming languages

Software must be rewritten to use character encodings. Windows NT/2000/XP, including Visual Basic, uses UCS-2 as native string type. Strings are declared of type `wchar_t` instead of `char`, and the programmer uses `wcslen` instead of `strlen`, et cetera. A literal string is created as `L"Hello world"`.

## 6.4 Character issues in $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ / $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$

### 6.4.1 Diacritics

Original  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is not very good at dealing with diacritics. They are implemented as things to put on top of characters, even when, as with the cedilla, they are under the letter. Furthermore,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  can not hyphenate a word with accents, since the accent introduces a space in the word (technically: an explicit kern). Both problems were remedied to a large extent with the ‘Cork font encoding’, which contains most accented letters as single characters. This means that accents are correctly placed by design, and also that the word can be hyphenated, since the kern has disappeared.

These fonts with accented characters became possible when  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  version 3 came out around 1990. This introduced full 8-bit compatibility, both in the input side and in the font addressing.

### 6.4.2 $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$ input file access to fonts

If an input file for  $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is allowed to contain all 8-bit octets, we get all the problems of compatibility that plagued regular text files. This is solved by the package `inputenc`:

```
\usepackage[code]{inputenc}
```

where `code` is `applemac`, `ansinew`, or various other code pages.

This package makes all unprintable ASCII characters, plus the codes over 127, into active characters. The definitions are then dynamically set depending on the code page that is loaded.

### 6.4.3 $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$ output encoding

The `inputenc` package does not solve the whole problem of producing a certain font character from certain keyboard input. It only mapped a byte value to the  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  command for producing a character. To map such commands to actual code point in a font file, the  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  and  $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$  formats contain lines such as

```
\chardef\i="10
```

declaring that the dotless-i is at position 16. However, this position is a convention, and other people – type manufacturers – may put it somewhere else.

This is handled by the ‘font encoding’ mechanism. The various people working on the  $\mathrm{L}^{\mathrm{A}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$  font schemes have devised a number of standard font encodings. For instance, the `OT1` encoding corresponds to the original 128-character set. The `T1` encoding is a 256-character extension thereof, which includes most accented characters for Latin alphabet languages.

A font encoding is selected with

```
\usepackage[T1]{fontenc}
```

A font encoding definition contains lines such as

```
\DeclareTextSymbol{\AE}{OT1}{29}  
\DeclareTextSymbol{\OE}{OT1}{30}  
\DeclareTextSymbol{\O}{OT1}{31}  
\DeclareTextSymbol{\ae}{OT1}{26}  
\DeclareTextSymbol{\i}{OT1}{16}
```

#### 6.4.4 Virtual fonts

**Exercise 50.** What does an ALT key do?

**Exercise 51.** What is EBCDIC? What is the basic idea?

**Exercise 52.** Find the Unicode definition. Can you find an example of a character that has two functions, but is not defined as two characters? Find two characters that are defined separately for compatibility, but that are defined equivalent.

**Exercise 53.** ISO 8859 has the ‘non-breaking space’ at position A0. How does T<sub>E</sub>X handle the nbsp? How do T<sub>E</sub>X, HTML, Latin-1, MS Word, et cetera handle multiple spaces? Discuss the pros and cons.

Character	Sample Glyphs					
a	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ

Figure 6.1: Different shapes of ‘lowercase roman a’

## Font encoding.

### 6.5 Basic terminology

Terminology of fonts and typefaces is quite confused these days. Traditionally, a typeface was a design, realized in number of fonts, that could be sorted in families, such as roman, and italic. A font would then be a style (medium weight bold italic) in a particular size.

Somewhat surprisingly, once you start throwing computers at this problem, even talking about characters becomes very subtle.

In Unicode, there are abstract characters and characters. They don’t differ by much: an abstract character is a concept such as ‘Latin lowercase a with accent grave’, and a character is that concept plus a position in the Unicode table. The actually visible representation of a character is called a ‘glyph’. According to ISO 9541, a glyph is ‘A recognizable abstract graphic symbol which is independent of any specific design’.

#### 6.5.1 The difference between glyphs and characters

Often, the mapping between character and glyph is clear: we all know what we mean by ‘Uppercase Roman A’. However, there may be different glyph shapes that correspond to the same character.

An abstract character is defined as

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

This definition has some interesting consequences. Sometimes one glyph can correspond to more than one character, and the other way around.

For example, in Danish, the ligature ‘æ’ is an actual character. On the other hand, the ligature ‘fl’, which appears in English texts, is merely a typographical device to make




Character Sequence	Sample Glyph
 	

Figure 6.2: The f-i ligature




Character	Split Glyphs
	 

Figure 6.3: A split character in Tamil

the combination ‘fi’ look better, so one glyph corresponds to two characters.

The opposite case is rarer. In Tamil, a certain character is split, because it is positioned *around* other characters. It can then even happen that one of the split parts forms a ligature with adjacent characters.

A trickier question is how to handle accented letters: is ‘é’ one character or a combination of two? In math, is the relation in  $a \neq b$  *one* symbol, or an overstrike of one over

















Character Sequence	Possible Glyph Sequences
	     
  	     

Figure 6.4: Different interpretations of an accented character glyph

another?

Another problem with ligatures is that a single glyph needs to be displayed, but two glyphs need to be stored to make searching for the string possible.

### 6.5.2 The identity of a character

Another problem in defining a character is whether two glyphs that look the same, or sometimes even *are* the same, should be the same character. For example, uppercase Latin a, uppercase Greek  $\alpha$ , and uppercase Cyrillic a, are all rendered ‘A’. Still, in Unicode they are three distinct characters.

Similarly, in ASCII, there are no separate glyphs for minus, hyphen, and dash. In Unicode, these are three characters. Is the character ‘superscript 2’ a separate glyph, or a typographical variant of the character ‘digit 2’? The latter should be the logical solution, but for compatibility reasons with other standards it is defined as a separate glyph. There are quite a few of these ‘compatibility characters’.

Yet another example is the Greek letter  $\Omega$ , which can be that letter, or the sign for electrical resistance in physics texts. Unicode defines them as two characters, but with identical glyphs.

A capital ‘A’ in Times Roman and in Helvetica are the same character, but what about italic?

All these matters are hard to settle objectively: everything is a matter of definition, or a judgement call. The official Unicode white paper on characters versus glyphs is <http://www.unicode.org/reports/tr17/>.

Here are some of the guidelines of the Unicode project:

- The Unicode Standard encodes characters, not glyphs.
- Characters have well-defined semantics.
- The Unicode Standard encodes plain text.
- And:

The Unicode Standard avoids duplicate encoding of characters by unifying them within scripts across languages; characters that are equivalent in form are given a single code. Common letters, punctuation marks, symbols, and diacritics are given one code each, regardless of language, [...]

### 6.5.3 Diacritics

Unicode, a bit like  $\text{T}_{\text{E}}\text{X}$ , has two ways of dealing with diacritics. It has precomposed accented characters, but it can also compose accented characters by listing accents (yes, plural: transliterated Vietnamese regularly has two accents over a letter one relating to vowel quality, and one to tone) after the base symbol. This mechanism can also deal with languages such as Hangul (Korean) which have composite characters.



## 6.6 Æsthetics

### 6.6.1 Scaling versus Design sizes

Lots of attention is devoted to font scaling, with the implicit assumption that that is the way to get a font to display at different sizes. This is only true to an extent: a small version of a typeface was traditionally of a different design than the same typeface at larger sizes. With metal type, independent designs for the different sizes were of course the only way one could proceed at all, but with photo typesetters and computers the need went away, and with it the realization that independent designs are visually actually a Good Thing. Figure 6.5 shows the difference between a typeface set at its

Ten point type is different from magnified five-point type.

Figure 6.5: A typeface and a smaller version scaled up

‘design size’, and a scaled up smaller version of it.



Figure 6.6: Adobe’s optical masters for a typeface

Adobe incorporated this idea in their Multiple Masters typefaces, which could interpolate between different designs. This technology seems to have been abandoned, but Adobe’s Originals now have so-called ‘Optical masters: four different designs of the same typeface, to be set at different sizes. Adobe labels their purposes as ‘display’, ‘subhead’, ‘text’, and ‘caption’ in decreasing design size; see figure 6.6

Apple developed their own version of multiple design sizes in TrueType GX, released in 1994. The ideas in TrueType GX are incorporated in Apple Advanced Typography (AAT) in OS X, but there few AAT typefaces, and certainly very few non-Apple ones.

## 6.7 Font technologies

### 6.7.1 Unicode in fonts

It is unrealistic to expect any single font to support even a decent fraction of the Unicode character repertoire. However, TrueType and OpenType do support Unicode.

The few fonts that support (almost) the whole of Unicode are called ‘pan-Unicode’. There are only a few of those. However, software these days is pretty sophisticated in gathering together symbols from disparate fonts. Some browsers do this, prompting the user for ‘install on demand’ of fonts if necessary.

### 6.7.2 Type 1 and TrueType

Type 1 (‘Postscript fonts’) was the outline font format developed by Adobe that was adopted by Apple in the mid 1980s. Since it was proprietary (Adobe had release the specifications for Type 3 fonts, but not Type 1), Apple and Microsoft later developed TrueType.

With Type 1 fonts, information is stored in two files, one for shape data and one for hinting and such. With TrueType, all information is in the one font file.

#### 6.7.2.1 *Type1*

Adobe Type 1 fonts are stored in two common formats, .pfa (PostScript Font ASCII) and .pfb (PostScript Font Binary). These contain descriptions of the character shapes, with each character being generated by a small program that calls on other small programs to compute common parts of the characters in the font. In both cases, the character descriptions are encrypted.

Before such a font can be used, it must be rendered into dots in a bitmap, either by the PostScript interpreter, or by a specialized rendering engine, such as Adobe Type Manager, which is used to generate low-resolution screen fonts on Apple Macintosh and on Microsoft Windows systems.

The Type 1 outline files do not contain sufficient information for typesetting with the font, because they have only limited metric data, and nothing about kerning (position adjustments of particular adjacent characters) or ligatures (replacement of adjacent characters by a single character glyph, those for fi, ffi, fl, and ffl being most common in English typography).

This missing information is supplied in additional files, called .afm (Adobe Font Metric) files. These are ASCII files with a well-defined easy-to-parse structure. Some font vendors, such as Adobe, allow them to be freely distributed; others, such as Bitstream, consider them to be restricted by a font license which must be purchased.

### 6.7.2.2 *TrueType* $\Leftrightarrow$ *Type1* conversion

Beware! There is no such thing as a one-to-one reversible conversion. There are several problems:

The outlines are stored in different ways in both formats. In truetype, second-order Bezier curves are used, and in type 1, third-order Bezier curves are employed. One second order Bezier can be transformed into a third-order Bezier, but a third-order Bezier cannot be transformed into one, two or seventeen second-order Beziers—approximations are in order for that conversion. So, type 1 to truetype is problematic, right from the start. For truetype to type 1, there is a snake in the grass, in the form of integer grid rounding (see below).

Both formats require all control points to be integers (whole numbers), falling in a grid. Truetype uses a 2048x2048 grid, type 1 typically a 1000x1000 grid. For the truetype to type 1 direction, one could divide all grid values by two, but then what? Should 183.5 become 183 or 184? The type 1 to truetype direction is easier, at least from this point of view, as we could multiply each grid coordinate by two, so no rounding loss would be involved. However, in the truetype to type 1 direction, the rounding causes additional problems for the new control points needed for the perfect third-order Bezier outlines mentioned above.

Placing ink on paper: the formats have different rules for placing ink on paper in case of outlines that are nested or intersecting. These differences are not caught by many conversion programs. In most cases, the user should not worry about this—only rarely do we have overlapping outlines (I was forced once to have them, for other reasons).

Complexity of the outlines: truetype permits more complex outlines, with more control points. For example, I am sure you have all seen fonts made from scans of pictures of faces of people. Typically, these outlines are beyond the type 1 limit, so this restriction makes the truetype to type 1 conversion impossible for ultra complex fonts.

Encoding: truetype can work with a huge number of glyphs. There are truetype fonts for Chinese and Japanese, for example. In type 1, the number of active glyphs is limited to 256. Again, for most Latin fonts, this is a non-issue.

The remarks about grid rounding also apply to all metrics, the bounding boxes, the character widths, the character spacing, the kerning, and so forth.

Finally, there is the hinting. This is handled very differently in both formats, with truetype being more sophisticated this time. So, in truetype to type 1 conversions of professionally (hand-hinted) fonts, a loss will occur. Luckily, 99% of the truetype fonts do not make use of the fancy hinting possibilities of truetype, and so, one is often safe.

All this to tell people to steer away like the plague from format conversions. And a

plea to the font software community to develop one final format. My recommendation: get rid of truetype, tinker with the type 1 format (well, tinker a lot). More about that ideal format elsewhere.

### 6.7.2.3 Downsampling bitmaps

In principle, given adequate resolution, the screen preview quality of documents set in bitmap fonts, and set in outline fonts, should be comparable, since the outline fonts have to be rasterized dynamically anyway for use on a printer or a display screen.

Sadly, this is not the case with versions of Adobe Acrobat Reader, acroread, and Exchange, acroexch (version 5.x or earlier); they do a poor job of downsampling high-resolution bitmap fonts to low-resolution screen fonts. This is particularly inexcusable, inasmuch as the co-founder, and CEO, of Adobe Systems, is the author of one of the earliest publications on the use of gray levels for font display: [ John E. Warnock, The display of characters using gray level sample arrays, Computer Graphics, 14 (3), 302–307, July, 1980. ]

## 6.7.3 FreeType

FreeType is an Open Source implementation of TrueType. Unfortunately this runs into patent problems, since Apple has patented some of the hinting mechanism. Recently FreeType has acquired an automatic hinting engine.

## 6.7.4 OpenType



OpenType is a standard developed by Adobe and Microsoft. It combines bitmap, outline, and metric information in a single cross-platform file. It has Unicode support, and can use ‘Optical Masters’ (section 6.6.1) multiple designs. It knows about the distinction between code points and glyphs, so applications can render a character differently based on context.

## 6.8 Font handling in T<sub>E</sub>X and L<sub>A</sub>T<sub>E</sub>X

T<sub>E</sub>X has fairly sophisticated font handling, in the sense that it knows a lot about the characters in a font. However, its handling of typefaces and relations between fonts is primitive. L<sub>A</sub>T<sub>E</sub>X has a good mechanism for that.

### 6.8.1 T<sub>E</sub>X font handling

Font outlines can be stored in any number of ways; T<sub>E</sub>X is only concerned with the ‘font metrics’, which are stored in a ‘`tfm` file’. These files contain

- Global information about the font: the `\fontdimen` parameters, which describe the spacing of the font, but also the x-height, and the slant-per-point, which describes the angle of italic and slanted fonts.
- Dimensions and italic corrections of the characters.
- Ligature and kerning programs.

We will look at these in slightly more detail.

#### 6.8.1.1 Font dimensions

The `tfm` file specifies the natural amount of space, with stretch and shrink for a font, but also a few properties related to the size and shape of letters. For instance, it contains the x-height, which is the height of characters without ascenders and descenders. This is, for instance, used for accents: T<sub>E</sub>X assumes that accents are at the right height for characters as high as an ‘x’: for any others the accent is raised or lowered.

The ‘slant per point’ parameters is also for use in accents: it determines the horizontal offset of a character.

#### 6.8.1.2 Character dimensions

The height, width, and depth of a character is used to determine the size of the enclosing boxes of words. A non-trivial character dimension is the ‘italic correction’. A tall italic character will protrude from its bounding box (which apparently does not always bound). The italic correction can be added to a subsequent space.

‘T<sub>E</sub>X has’ versus ‘T<sub>E</sub>X has’

#### 6.8.1.3 Ligatures and kerning

The `tfm` file contains information that certain sequences of characters can be replaced by another character. The intended use of this is to replace sequences such as `fi` or `fl` by ‘fi’ or ‘fl’.

Kerning is the horizontal spacing that can bring characters closer in certain combinations. Compare

‘Von’ versus ‘Von’

Kerning programs are in the `tfm` file, not accessible to the user.

### 6.8.2 Font selection in L<sup>A</sup>T<sub>E</sub>X

Font selection in L<sup>A</sup>T<sub>E</sub>X (and T<sub>E</sub>X) was rather crude in the early versions. Commands such as `\bf` and `\it` switched to boldface and italic respectively, but could not be combined to give bold italic. The New Font Selection Scheme improved that situation considerably.

With NFSS, it becomes possible to make orthogonal combinations of the font family (roman, sans serif), series (medium, bold), and shape (upright, italic, small caps). A quick switch back to the main document font is `\textnormal` or `\normalfont`.

#### 6.8.2.1 Font families

It is not necessary for a typeface to have both serified and serifless (sans serif) shapes. Often, therefore, these shapes are taken from different, but visually compatible typefaces, for instance combining Times New Roman with Helvetica. This is the combination that results from

```
\usepackage{times}
```

Loading the package `lucidabr` instead, gives Lucida Bright and Lucida Sans.

The available font families are

**roman** using the command `\textrm` and the declaration `\rmfamily`.

**sans serif** using the command `\textsf` and the declaration `\sffamily`.

**typewriter type** using the command `\texttt` and the declaration `\ttfamily`.

Typewriter type is usually a monospaced font – all characters of the same width – and is useful for writing about L<sup>A</sup>T<sub>E</sub>X or for giving code samples.

#### 6.8.2.2 Font series: width and weight

The difference between normal and medium width, or normal and bold weight, can be indicated with font series commands:

**medium width/weight** using the command `\textmd` and the declaration `\mdseries`.

**bold** using the command `\textbf` and the declaration `\bfseries`.

#### 6.8.2.3 Font shape

The final parameter with which to classify fonts is their shape.

**upright** This is the default shape, explicitly available through `\textup` or `\upshape`.

**italic and slanted** These are often the same; they are available through `\textit`, `\textsl`, and `\itshape`, `\slshape`.

**small caps** Here text is set through large and small capital letters; this shape is available through `\textsc` and `\scshape`.

## Input and output encoding in L<sup>A</sup>T<sub>E</sub>X.

### 6.9 The fontenc package

Traditionally, in T<sub>E</sub>X accented characters were handled with control characters, such as in `\'e`. However, many keyboards – and this should be understood in a software sense – are able to generate accented characters, and other non-latin characters, directly. Typically, this uses octets with the high bit set.

As we have seen, the interpretation of these octets is not clear. In the absense of some Unicode encoding, the best we can say is that it depends on the code page that was used. This dependency could be solved by having the T<sub>E</sub>X software know, on installation, what code page the system is using. While this may be feasible for one system, if the input files are moved to a different system, they are no longer interpreted correctly. For this purpose the `inputenc` package was developed.

An input encoding can be stated at the load of the package:

```
\usepackage[cp1252]{inputenc}
```

or input encodings can be set and switched later:

```
\inputencoding{latin1}
```

With this, a (part of a) file can be written on one machine, using some code page, and still be formatted correctly on another machine, which natively has a different code page.

These code pages are all conventions for the interpretation of singly octets. The `inputenc` package also has limited support for UTF-8, which is a variable length (up to four octets) encoding of Unicode.

### Projects for this chapter.

**Project 6.1.** What is the problem with ‘Han unification’? (section 6.2.6) Discuss history, philology, politics, and whatever may be appropriate.

**Project 6.2.** How do characters get into a file in the first place? Discuss keyboard scan codes and such. How do modifier keys work? How can an OS switch between different keyboard layouts? What do APIs for different input methods look like?

**Project 6.3.** Dig into history (find the archives of `alt.folklore.computers!`) and write a history of character encoding, focusing on the pre-ascii years. Describe design decisions made in various prehistoric computer architectures. Discuss.