

# Unicode en UTF-8

## *Samenvatting*

Dit verhaal behandelt de beginselen van Unicode en UTF-8, en de algemene principes van *character encoding*. Aan bod komen ASCII, Latin1 (ISO-8859-1) en Latin9 (ISO-8859-15), evenals UCS-2, UCS-4 en UTF-16. Vereiste voorkennis: talstelsels (hexadecimaal, binair).

## 1. Encoding

Sinds 1968 bestaat de ASCII-standaard (officieel: American Standard Code for Information Interchange ANSI-X3.4). Die beschrijft hoe tekstletters (en leestekens en cijfers) gecodeerd worden als getallen. Populair gezegd: als je zo'n getal naar een afdrukkapparaat stuurt (beeldscherm, printer) en dat apparaat is "ASCII-compatible", dan weet het apparaat welk letterbeeldje je voor dat getal te zien wilt krijgen. Een ASCII-keyboard weet welk getalletje je in een file wilt laten terechtkomen als je een toets aanslaat waarop een bepaald letterbeeld is geschilderd.

We noemen zo'n afspraak die getallen aan characters koppelt, een *encoding*. Een encoding kun je weergeven als een tabel met twee kolommen, waarin je naast elk getal het afgesproken character ziet afgebeeld.

Voor de rest van het verhaal is het het meest duidelijk als je steeds centraal stelt hoe (met welk getal) een bepaalde letter *in een file* is opgeslagen. Pas nadat je dát duidelijk besloten hebt, ga je je afvragen:

- hoe krijg ik zo'n getal vanaf een toetsenbord in mijn file, c.q. is er een andere manier om het in de file te krijgen (bijvoorbeeld met een muis een letterbeeldje in een keuze-menu aanklikken);
- hoe maak ik mijn beeldscherm duidelijk dat hij, als ik hem zo'n getal stuur, die bepaalde letter moet laten zien;

- hoe maak ik mijn printer duidelijk dat hij, als ik hem zo'n getal stuur, die bepaalde letter op papier moet zetten.

Merk op dat bij de laatste twee punten niet wordt gesproken over een eventuele *fontkeuze*. Het gaat er dus om dat je een 'a' of 'b' enz. wilt zien, los van de vraag of dat een krul-letter, of een strakke letter, enz. moet worden. We definiëren daarvoor twee jargontermen:

- een *character* is een symbool (letter), onafhankelijk van zijn vormgeving in font en size. Voorbeelden van *characters* zijn: “de hoofdletter A”, “het procent-teken”, “de kleine letter ë”, “het cijfer negen”, “het Euro-symbool”. Een alfabet bestaat uit characters. Het wiskunde-schrift en het muzieknoden-schrift bestaan ook uit characters.
- een *glyph* is een symbool *inclusief* een specifieke vormgeving, maar grootteverschillen (point-sizes) onderscheidt men niet apart. Een font bestaat uit glyphs: één glyph per ondersteund character. Voorbeelden van *glyphs* zijn: “Times-Roman kleine letter a”, “Courier Italic cijfer 8”, “Adobe Sonata muziekschrift kwart-noot”.

Uiteraard kan een font niet glyphs voor oneindig veel characters bevatten. In theorie zou je een font kunnen maken dat glyphs voor tienduizenden characters bevat, maar in de praktijk omvat een font minder dan honderd, tot hooguit een paar duizend glyphs (momenteel groeit dat wel erg hard). Dus je loopt wel eens tegen het probleem aan dat een bepaald font geen glyph voor een bepaald character heeft. Bij de invoering van de € hebben veel computergebruikers dat gemerkt.

Een fontkeuze moet je van te voren apart met je beeldscherm/printer afspreken; moderne apparatuur heeft een heleboel keuze-mogelijkheden. Nádat je een keuze voor font en size aan je afdrukkapparaat hebt gegeven, begin je te vertellen welke characters je wilt zien, en dan krijg je *de glyphs van* die characters te zien. Bij toetsenborden speelt fontkeuze natuurlijk helemaal géén rol.

De ASCII-standaard is waarschijnlijk de meest elementaire en meest verbreide standaard waarmee je in het computervak te maken krijgt. Een character krijgt in ASCII een getalswaarde (de *character-code*) in het gebied van 32 tot en met 126. Dat heeft een aantal gevolgen:

- Eén character-waarde past in 7 bits. Als je besluit om die in één byte te stoppen dan kun je de 8e bit ongebruikt (0) laten.
- Het gebied van 32 tot en met 126 geeft je 95 waarden. Daarmee kun je dus 95 characters aanduiden. Hoofdletters, kleine letters en cijfers kosten al  $26+26+10=62$  waarden. Dus blijven er nog 33 waarden over voor leestekens (inclusief spatie).
- De getallen 0 t.m. 31 en het getal 127 hebben ook een betekenis gekregen, maar leveren geen afdruk-symbool. Het zijn z.g. stuurcodes, zoals *newline* ga naar volgende regel, *carriage return* wagenterugloop, *tab*, *backspace*, *delete*. enz. In UNIX geeft het commando `man ascii` een afdruk van de tabel waarin zowel de codes van de afdrubbare characters (“printables”) als de stuurcodes (“non-printables”) goed te zien zijn.

- De ASCII-standaard gaat uit van de 26 letters van het Amerikaanse alfabet. Daarin zit *geen enkele* letter met accent (officieel: letter met diakritisch teken)<sup>1</sup>.

Er bestaat binnen ASCII wel een mogelijkheid om enkele leestekens *te vervangen door* enkele letters met accenten. Dat paste bijvoorbeeld vrij aardig op de behoeften van de Duitse taal, dus dat kwam je vroeger op Duitse apparatuur nog wel eens tegen: *in plaats van* [!]{} kreeg je dan ÄÖÜäöü. Merk op dat hier nadrukkelijk staat: *in plaats van*, en niet *naast*.

Omdat hiermee de betreffende leestekens onbereikbaar worden gemaakt, is deze ASCII-variantie lastig, en inmiddels zo goed als uitgestorven.

- Momenteel veel gebruikt voor letters met diacritische tekens zijn de z.g. ASCII-extensies. Dat zijn geen *vervangingen* maar *uitbreidingen*. Ze verhogen de bovengrens van 127 naar 255; daarvoor wordt de 8e bit in gebruik genomen.
- Er zijn meerdere extensies ontworpen. Een voorbeeld is de *Roman8* encoding, die je nog tegenkomt bij oudere Hewlett-Packard database software. Erg bekend is ook de *CodePage850* encoding: een oud IBM-ontwerp, ingebracht in de oer-PC architectuur, en veel gebruikt in MS/DOS (en ook wel Windows) omgevingen. Onder Windows wordt ook vaak de *Windows-1252* encoding gebruikt. Printer-experts kennen de *PostScript StandardEncoding*. Maar er zijn er véél meer.
- Het meest gebruikt wordt de z.g. Latin1 extensie, die ook wel ISO-8859-1 heet. Latin1 is ontworpen voor de behoeften van de grote Westeuropese talen, van IJslands tot Duits. Het dekt niet de behoeften van de Oosteuropese talen (zoals Pools) of talen die wel het Latijnse alfabet gebruiken maar niet Europees zijn (Turks, Maleis, Vietnamees). Het is ook onvolledig voor enkele zeer kleine Westeuropese talen (Inuit, Sami). Een ernstig knelpunt is dat Latin1 geen Euro-symbool bevat. Daarom is er inmiddels een opvolger-mèt-€: ISO-Latin9, ook genaamd ISO-8859-15.
- De ISO-8859-xx serie bevat zo'n twintig verschillende definities, zowel bedoeld voor bovengenoemde talen met Latijns alfabet, als ook voor sommige talen met andere alfabetten: Cyrillisch, Hebreeuws... De 8859-xx subnummering telt voor alle alfabetten, terwijl de Latin-alfabetten binnen die verzameling ook een aparte nummering hebben. Dat verklaart waarom ISO-8859-15 hetzelfde is als Latin9.
- Merk op dat, omdat de bovengrens van de codewaarden op 255 is gezet, één codegetal nog steeds in één byte past. Voor de getallen beneden de 128 hebben deze extensies *altijd* compatibiliteit met de basis-ASCII tabel gehandhaafd.

De uitbreidingsruimte met getallen van 127 tot en met 255 levert wel ruimte voor veel nieuwe tekens, maar er zijn nog veel méér wensen. Dit knelpunt in ruimte blijft bestaan zo lang je wilt vasthouden aan de verhouding één byte = één character. Maar: als je dat principe overboord zet krijg je een aardverschuiving die ontzettend veel bestaande software en hardware op z'n kop gaat zetten. Daarom is tientallen jaren lang tegen die beslissing aangehikt.

---

1. In PC-kringen bestaat een wijdverbreide (maar foute) gewoonte om de term ASCII te gebruiken voor charactersets waarin wél letters met accenten zitten.

De druk om veel meer characters *tegelijk* ter beschikking te krijgen is in deze moderne tijd te groot geworden om tegen te houden: een personeelsadministratie wil mensen met een correct gespelde Franse, Poolse of Turkse achternaam kunnen bedienen, en webpagina's komen soms zelfs in niet-Latijnse alfabetten op je af.

Merk op dat we hier *twee* stappen nemen: het personeels-voorbeeld blijft nog binnen het Latijnse alfabet, maar we willen wel allerlei variaties (Frans, Pools, Turks enz.) aankunnen. De volgende, véél grotere (!!), stap is de beslissing om een tekst in elk alfabet van waar ook ter wereld correct weer te geven op je scherm. Binnen de Europese Unie heb je al officieel te maken met drie alfabetten (Latijns en Grieks, en in Bulgarije het Cyrillisch). Mensen met een multiculturele achtergrond of met wereldwijde handelscontacten willen webpagina's in het Japans, Arabisch, enzovoort vanuit Nederland kunnen raadplegen. Is het je al eens opgevallen dat Google dit soort pagina's met een mix van alfabetten perfect voor elkaar heeft, mits je een moderne browser (en bijbehorend uitgebreid font) hebt?

Het noemen waard is nog de EBCDIC-encoding, die je tegenkomt op IBM mainframe computers. EBCDIC is een alternatief voor ASCII, maar was al ontworpen voordat de ASCII-standaard bestond. EBCDIC bevat veel minder leestekens dan ASCII, maar bevat ook een code voor dollarcent ¢ en voor het teken ¬ (*not*), die beide niet in ASCII voorkomen. Daarom zijn één-op-één vertalingen tussen ASCII en EBCDIC nooit helemaal sluitend te krijgen. Binnen EBCDIC bestaat een aantal kleine variaties, waarmee mainframe-beheerders te maken kunnen krijgen.

ASCII en EBCDIC zijn vrij korte encodingtabellen, omdat is afgesproken dat de getallen beperkt blijven qua grootte (bovengrens 127). Latin1, CodePage850, Roman8 en *veel* andere zijn uitgebreidere encodings, met bovengrens 255.

Ter illustratie drukken we hier de ISO-Latin1 (ISO-8859-1) ASCII-uitbreiding af:

A0	nsp	A1	ı	A2	ç	A3	£	A4	¤	A5	¥	A6		A7	§
A8	¨	A9	©	AA	ª	AB	«	AC	¬	AD	-	AE	®	AF	-
B0	°	B1	±	B2	²	B3	³	B4	´	B5	µ	B6	¶	B7	·
B8	,	B9	ı	BA	º	BB	»	BC	¼	BD	½	BE	¾	BF	¿
C0	À	C1	Á	C2	Â	C3	Ã	C4	Ä	C5	Å	C6	Æ	C7	Ç
C8	È	C9	É	CA	Ê	CB	Ë	CC	Ì	CD	Í	CE	Î	CF	Ï
D0	Ð	D1	Ñ	D2	Ò	D3	Ó	D4	Ô	D5	Õ	D6	Ö	D7	×
D8	Ø	D9	Ù	DA	Ú	DB	Û	DC	Ü	DD	Ý	DE	Þ	DF	ß
E0	à	E1	á	E2	â	E3	ã	E4	ä	E5	å	E6	æ	E7	ç
E8	è	E9	é	EA	ê	EB	ë	EC	ì	ED	í	EE	î	EF	ï
F0	ð	F1	ñ	F2	ò	F3	ó	F4	ô	F5	õ	F6	ö	F7	÷
F8	ø	F9	ù	FA	ú	FB	û	FC	ü	FD	ý	FE	þ	FF	ÿ

## 2. Unicode, UCS-2, UCS-4

Unicode (officieel: *ISO-10646 Universal Character Set*) is de meest uitgebreide encoding die er bestaat. Daarvoor heeft men alle characters uit bijna alle levende alfabetten ter wereld in een (lange!) rij gezet, aangevuld met zoveel mogelijk leestekens, accenten en andere symbolen (bijv. Dingbats, wiskunde, enz.) voor zover

ze een relatie met tekst-weergave hebben. Daarbij is wel afgesproken dat de gebruikte getallen beneden de 65536 moesten blijven, want dan past het getal nog in twee bytes (16 bits). Deze beperking lukte alleen maar door de klassieke Chinese tekens niet mee te nemen. Modern Chinees, Japans, Koreaans, plus Arabisch, Hindi, Hebreeuws, Grieks, enz. enz., ruim 20 alfabetten in totaal, pasten wel allemaal samen, en er paste nog een enorme hoeveelheid leestekens, Dingbats, wiskundige symbolen, enz. bij. Formeel heet deze 16-bits (dus 2-bytes) nummering de UCS-2, maar in de rest van dit verhaal zullen we de termen “Unicode” en UCS-2 niet al te scherp uit elkaar houden. Op papier weergegeven heeft de Unicode standaard de omvang van een flink telefoonboek<sup>2</sup>.

Inmiddels is er een “extended-Unicode” bijgekomen, met getallen die wèl boven de 65536 komen, ten behoeve van o.a. dode talen, extreem kleine taalgroepen, en exotische alfabetten. De formele naam van deze “extended-Unicode” is UCS-4; het is een 31-bits nummering. De term BMP *Basic Multilingual Plane* wordt gebruikt om het oude gedeelte (nummers onder 65536) binnen het nieuwe, extended, geheel aan te duiden.

### 3. Van Unicode naar UTF-8

Unicode is, zoals al gezegd, een poging om alle characters die men ter wereld kon bedenken in een rij met afgesproken volgnummering te zetten. Bijvoorbeeld: het euro-symbool € heeft Unicode 20AC gekregen (20AC hexadecimaal is 8364 decimaal).

De allerlaagste Unicode-nummers zijn in overeenstemming gehouden met ASCII (0-127) en met Latin1 (128-255). Maar let op: ASCII heeft de waarden 0-127 in één byte (dus hex 0x00-0x7F), en Unicode heeft voor dezelfde symbolen de waarden 0-127 in twee bytes (dus hex 0x0000-0x007F). Evenzo heeft Latin1 de waarden 128-255 in één byte (hex 0x80-0xFF) en heeft Unicode, voor dezelfde characters, twee bytes (hex 0x0080-0x00FF).

De gedachte was om uiteindelijk alle software en hardware (grafische kaarten, printers...) Unicode-compatible te maken. Vroeger was het ondenkbaar om voor elk character twee bytes te gebruiken: geheugen, diskruimte en datacom-capaciteit waren daarvoor veel te krap. Tegenwoordig is het qua capaciteit geen probleem meer, maar de bestaande investering in hard- en software maakt het nu toch problematisch.

Daarom is een conversie ontwikkeld om Unicode waarden (0-65535) te vertalen naar een encoding die strikt upward-compatible met ASCII is (maar niet met Latin1 !!). Die encoding heet<sup>3</sup> UTF-8.

---

2. The Unicode Standard, Version 5.0, uitgever: Addison Wesley Professional, ISBN 0-321-48091-0, of kijk op [www.unicode.org](http://www.unicode.org)

3. De afkorting UTF stond oorspronkelijk voor *Unix Transformation Format*. Inmiddels is de verklaring herdoopt in *Universal Transformation Format*. Het oorspronkelijke idee komt van Ken Thompson (UNIX-architect) en Rob Pike, in samenwerking met de X/Open Group.

De ASCII-characters blijven één byte groot. Dus een oude ASCII-file, gestuurd naar een nieuwe, UTF-8-aware printer, komt probleemloos op papier. De truc is hier dat elke UTF-8-byte waarin de 8e bit op 0 staat, per definitie een ASCII-compatibele byte is.

Bytes waarvan de 8e bit op 1 staat (dus niet ASCII-compatible is), komen alleen voor in groepjes van twee, drie of vier stuks als volgt (tel de puntjes! ze stellen de vrije bit-posities voor):

- 110..... 10..... twee bytes die bij elkaar horen  
Unicode-waarden tot 2047 (hex 07FF) worden in de vrije 11 bit-posities gezet. In dit gebied zitten bijna alle ASCII-extensies van het Latijnse alfabet (inclusief Latin1), het fonetisch alfabet, een flinke partij diacritische tekens (accenten), en de moderne vormen van het Grieks, Cyrillisch, Armeens, Hebreeuws en Arabisch.
- 1110.... 10..... 10..... drie bytes die bij elkaar horen  
Alle Unicode-waarden vanaf 2048 (hex 0800) worden in deze 16 vrije bit-posities geplaatst. Vanwege de beperking tot 65535 past dit dus altijd.
- 11110... 10..... 10..... 10..... vier bytes die bij elkaar horen  
Deze vorm levert 21 bits vrije ruimte, en is bedoeld voor de *extended* Unicode, dus UCS-4 nummers boven 65535. Het schema gaat zelfs verder naar vijf en zes bytes, maar het zal in de praktijk nog heel lang duren voordat die nodig zijn.
- Een aparte spelregel zegt dat een character dat in een korte notatievorm past, niet in een langere vorm gezet mag worden.

Als voorbeeld kijken we naar het paragraaf-teken §. In Latin1 is dat A7 hex, en in Unicode dus 00A7 hex. Voor conversie naar UTF-8 constateren we dat 00A7 in het gebied tot 07FF ligt, en we dus de twee-bytes UTF-8 vorm moeten nemen. De 11 meest rechtse bits uit 00A7 (binair 0000 0000 1010 0111) moeten we dan invullen op de open plekken van 110..... 10..... Het resultaat: 11000010 10100111, en dat is hex C2A7.

Als we deze procedure op alle characters uit de Latin1-verzameling loslaten zien we het volgende:

Unic	UTF8	Unic	UTF8	Unic	UTF8	Unic	UTF8	
00A0	C2A0	nsp	00A1	C2A1	ı	00A2	C2A2	ç
00A4	C2A4	¤	00A5	C2A5	¥	00A6	C2A6	ı
00A8	C2A8	¨	00A9	C2A9	©	00AA	C2AA	ª
00AC	C2AC	¬	00AD	C2AD	-	00AE	C2AE	®
00B0	C2B0	º	00B1	C2B1	±	00B2	C2B2	²
00B4	C2B4	´	00B5	C2B5	µ	00B6	C2B6	¶
00B8	C2B8	,	00B9	C2B9	¹	00BA	C2BA	º
00BC	C2BC	¼	00BD	C2BD	½	00BE	C2BE	¾
00C0	C380	À	00C1	C381	Á	00C2	C382	Â
00C4	C384	Ä	00C5	C385	Å	00C6	C386	Æ
00C8	C388	È	00C9	C389	É	00CA	C38A	Ê
00CC	C38C	Ì	00CD	C38D	Í	00CE	C38E	Î
00D0	C390	Ð	00D1	C391	Ñ	00D2	C392	Ò
00D4	C394	Ô	00D5	C395	Õ	00D6	C396	Ö
00D8	C398	Ø	00D9	C399	Ù	00DA	C39A	Ú
00DC	C39C	Û	00DD	C39D	Ý	00DE	C39E	Ë
00E0	C3A0	à	00E1	C3A1	á	00E2	C3A2	â
00E4	C3A4	ä	00E5	C3A5	å	00E6	C3A6	æ
00E8	C3A8	è	00E9	C3A9	é	00EA	C3AA	ê
00EC	C3AC	ì	00ED	C3AD	í	00EE	C3AE	î
00F0	C3B0	ð	00F1	C3B1	ñ	00F2	C3B2	ò
00F4	C3B4	ô	00F5	C3B5	õ	00F6	C3B6	ö
00F8	C3B8	ø	00F9	C3B9	ù	00FA	C3BA	ú
00FC	C3BC	ü	00FD	C3BD	ý	00FE	C3BE	þ
						00FF	C3BF	ÿ
00A3	C2A3	£	00A7	C2A7	§	00AB	C2AB	«
00AF	C2AF	-	00B3	C2B3	³	00BB	C2BB	»
00BF	C2BF	¿						
00C3	C383	Ã	00C7	C387	Ç	00CB	C38B	Ë
00CF	C38F	Ï	00D3	C393	Ó	00DB	C39B	Û
00DF	C39F	ß						
00E3	C3A3	ã	00E7	C3A7	ç	00EB	C3AB	ë
00EF	C3AF	ï	00F3	C3B3	ó	00FB	C3BB	û
00FF	C3BF	ÿ						

Als je een tekst van Latin1 omzet naar UTF-8 (zie bijvoorbeeld *man iconv(1)*) dan blijven dus alle ASCII-characters één byte lang, maar de non-ASCII characters worden twee bytes. Maak je vervolgens de fout om die UTF-8-encoded tekst naar een Latin1-ingesteld beeldscherm of printer te sturen, dan komen alle ASCII-characters er onbeschadigd uit, maar de Latin1 characters worden ontvangen als *twee* bytes, elk met de 8e bit aan. Ze worden dan door het apparaat geïnterpreteerd als *twee* Latin1 characters. En die zijn niet hetzelfde als het ene oorspronkelijke!

In bovenstaande tabel kun je zien dat de characters uit de Latin1-verzameling in UTF-8 als hoogste byte allemaal een 0xC2 of 0xC3 hebben. Als een afdrukkapparaat niet weet dat het UTF-8 is, en die interpreteert als zelfstandige Latin1-characters, dan zie je die bytes als de tekens Â resp. Ä. Dus je kunt herkennen dat een UTF-8-gecodeerde file naar een Latin1 (of Latin9) afdrukkapparaat wordt gestuurd, als je steeds op de plek waar je één character-met-accent verwacht te zien, nu paren van characters ziet, waarvan het eerste Â of Ä is, en het tweede wisselend is.

Je moet wel goed opletten want de effecten kunnen verwarring wekken. Bijvoorbeeld het currency-symbool ¢ is 0xA4 in Latin1. In UTF-8 wordt dat 0xC2A4 en als je dat (abusievelijk) display't op een Latin1-apparaat dan zie je dus Â¢. Min of meer per ongeluk verschijnt hier de ¢ opnieuw. Alle characters in het eerste deel van de tabel hierboven vertonen datzelfde effect.

Een tekst die origineel in UTF-8 is geproduceerd (dus niet ooit vanuit Latin1 is omgezet naar UTF-8) zal nog een ander typisch fenomeen laten zien op een Latin1-afdrukapparaat. In Unicode zijn als “preferred quotes” aangewezen: 2018 (enkele quote, openen), 2019 (enkele quote, sluiten), 201C (dubbele quote, openen) en 201D (dubbele quote, sluiten). In UTF-8 worden dat drie-byte codes, resp. E28098, E28099, E2809C en E2809D. Voor een Latin1-apparaat zijn 80,98,99,9C en 9D geen printable bytecodes. Dus uit de UTF-8 codering van deze quotes is in Latin1 alleen de E2-byte bruikbaar, en die wordt als een â weergegeven.

Als je de fout andersom maakt, dus een Latin1 (of Latin9) gecodeerde file naar een UTF-8 ingesteld afdrukapparaat stuurt, dan vallen meestal de characters-met-accenten volledig weg. Of het afdrukapparaat geeft een vraagteken, of een vraagteken-in-een-blob, of zoiets. De reden is dat in UTF-8 deze characters bytereeksen moeten vormen, waarvan de eerste begint met bits 110 en de tweede (en eventueel volgende) met bits 10. Een losstaand Latin1/9 character is dus nooit een geldig UTF-8 character. Als je aaneengesloten reeksen Latin1/9 characters in je file hebt zitten, zou door een toevalstreffer een geldige UTF-8 reeks kunnen ontstaan.

## 4. Latin1 en Latin9

Voor de volledigheid kijken we nog even naar Latin9, een veelgebruikt alternatief voor Latin1. In paragraaf 1 is al uitgelegd dat Latin9 ook ISO-8859-15 wordt genoemd.

We tonen het verschil in de vorm van een tabel. We gaan uit van een één-byte code, en vertellen welk character daarbij hoort in Latin1 en in Latin9. Alleen de characters die verschillen tussen Latin1 en Latin9 worden genoemd.

In het rechterdeel gaan we verder met de Latin9-characters: we laten hun Unicode waarden zien, en hun UTF-8 waarden (beiden hexadecimaal):

<i>dec</i>	<i>hex</i>	<i>Latin1</i>	<i>Latin9</i>
164	A4	¤	€
166	A6	¦	Š
168	A8	¨	š
180	B4	´	Ž
184	B8	,	ž
188	BC	¼	Œ
189	BD	½	œ
190	BE	¾	Ÿ

<i>char</i>	<i>Unicode</i>	<i>UTF-8</i>
€	20AC	E282AC
Š	0160	C5A0
š	0161	C5A1
Ž	017D	C5BD
ž	017E	C5BE
Œ	0152	C592
œ	0153	C593
Ÿ	0178	C5B8

## 5. Troubleshooting

Als je een bepaalde letter (of lettercode) intypt, en je ziet een 'verkeerd' resultaat op je scherm of printer weergegeven, dan kan dat veel oorzaken hebben. Om dat uit te zoeken herhalen we de goede raad uit de eerste paragraaf:

- Je moet éérst helder krijgen welke encoding je verwacht in de file, c.q. in de software waarmee je werkt. Daarna splits je het probleem in twee stappen:

(1) komt vanuit mijn toetsaanslag het goede codegetal in de file terecht, en  
(2) geeft mijn terminal of printer weer wat verwacht mag worden als dat codegetal uit de file komt.

Een aanpak in de trant van: als ik 'dit' intyp dan zie ik 'dat' op mijn scherm, is een oorlog op twee fronten. Die verlies je altijd! Je moet de representatie in de file centraal stellen, en van daaruit twee kanten op werken: naar de input en naar de output. □

## 6. Toch meer characters

■ Deze paragraaf geeft voorbeelden van manieren waarop men in het verleden heeft proberen te ontsnappen aan de beperking van maximaal 255 verschillende characters *tegelijk* binnen eenzelfde pagina. Dat zijn dus onvolledige (maar veelgebruikte) oplossingen voor het probleem dat pas door Unicode/UTF-8 volledig is opgelost.

Strikt genomen gaat deze paragraaf dus niet over Unicode/UTF-8 zelf, maar levert ze slechts interessante en leerzame achtergrondinformatie. □

Onvolledige uitbreidingstechnieken ontstonden waar compatibiliteit met bestaande software niet van belang was, omdat een heel nieuwe omgeving werd opgezet. In alle gevallen stapte men voor een aantal characters over naar een weergave in de file waarbij meerdere bytes werden gebruikt voor één zo'n speciaal character. Voorbeelden zijn:

- De HTML-taal voor webpagina's (versies ouder dan 4.0)  
De oude HTML was in principe ISO-Latin1 (ISO-8859-1) compatible. Elk Latin1 codegetal werd door een browser afgebeeld als het corresponderende Latin1-character (glyph uit het op dat moment ingestelde font). Mensen die vanaf hun keyboard alle Latin1-symbolen konden intypen, hoefden geen speciale trucs uit te halen om ze in hun file te krijgen, en om elke browser de juiste glyph te laten afbeelden. Maar dat gaf problemen met mensen die niet zulke uitgebreide mogelijkheden op hun keyboard hadden (bijna niemand heeft de volledige Latin1-set op zijn keyboard ter beschikking).

Dus was als truc verzonnen dat een &-teken een speciaal effect start. Wanneer in een tekst een & staat, wordt gekeken of vlak daarachter een bekend keyword staat, en daarachter weer een ; Bijvoorbeeld: &pound; voor £, of &ntilde; voor ñ. Alle characters uit de Latin1-extensie hebben zo'n keyword-naam gekregen. Dus nu kon iedereen via zijn (basis-ASCII) keyboard duidelijk maken welke Latin1 characters hij graag zag verschijnen. De &keyword; aanduidingen worden letterlijk zo in de webpagina-file opgeslagen.

Een doordenkertje is natuurlijk dat je moet compenseren voor het feit dat je de & hebt weggehaald van zijn oorspronkelijke betekenis. Want realiseer je goed: de & komt zelf niet uit de Latin1-extensie, maar uit de basis-ASCII verzameling. Dus werd daarvoor de naam &amp; ingevoerd.

In de nieuwe HTML-versies is het principe dat je Latin1-characters ook letterlijk mag intypen en in je file mag opslaan (dus £ en ñ naast &pound; en &ntilde;) verlaten. Om toch oude (Latin1) files te mogen blijven gebruiken moet je nu expliciet in zo'n file aangeven dat je dat doet:

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
```

Het is overigens aan te bevelen om *altijd* in een webpagina expliciet aan te geven welke charset je gebruikt. Als het er niet in staat dan moet de browser een gok doen, en dat kan helemaal verkeerd uitpakken.

- Tekstverwerkers zoals WordPerfect en Word  
Ze gaan ook uit van meerdere bytes (in de file) op de plaats waar een speciaal, niet-Latin1 character wordt bedoeld. In tegenstelling tot bij HTML zijn die groepjes bytes niet als leesbare keywords ontworpen. Dus krijg je een probleem met inbrengen via een keyboard. Dat is opgelost door dergelijke symbolen met een muis te laten aanklikken in een apart keuzevenster. Of, voor de mensen die graag apenkunstjes uit hun hoofd leren, door te werken met nietszeggende key-combinaties zoals *option+g* voor een ©
- De PostScript printertaal  
PostScript behandelen we nu als voorbeeld omdat het een totaal andere aanpak laat zien. Die aanpak begint met “fonts”, die door vormgevers zijn getekend. Elke glyph in elk font heeft van de vormgever een *naam* gekregen. Voor de meestgebruikte fonts (Latijnse alfabet) bestaan vaste namen, maar voor speciale fonts (bijvoorbeeld muziekschrift Sonata) werden de namen speciaal verzonnen. Voorbeelden van dergelijke namen zijn: /Aring voor Å of /Z voor de letter Z of /florin voor f, of /mezzoforte uit het Sonata-muziekschrift. Maar er ligt geen enkele getalswaarde vast, dus je begint in feite helemaal zonder encoding. De bedoeling is dan dat je de *namen* van de glyphs die je uit een bepaald font wilt gebruiken op een rijtje zet (maximaal 256 stuks), en daar getallen tussen 0 en 255 aan koppelt. Je moet dus per gebruikt font een eigen encoding-tabel opbouwen, en die hoeft helemaal niet op de ASCII of Latin1 tabel te lijken (het is wel onhandig om daarvan af te wijken). Als je meer dan 256 symbolen uit eenzelfde font gelijktijdig nodig hebt, mag je meerdere encodingtabellen maken. Elke tabel moet je een eigen naam geven, en je moet je tabel(len) invoegen in het begin van je printjob. Daarna kun je, midden in je job, net zo gemakkelijk encoding-tabellen schakelen als je fonts schakelt: het is gewoon een kwestie van het juiste schakelcommando geven, met de juiste naam erbij.  
  
Voor het gemak kent PostScript wel een aantal voorgedefinieerde encodingtabellen: voor gewone Latijnse letters (de *StandardEncoding*), voor Latin1 (de *ISOLatin1Encoding*), voor Symbolen-fonts (de *Symbol Encoding*), enz.
- Er zijn meer voorbeelden van dergelijke ontsnappings-systemen te vinden: in de hoek van electronic-mail berichten, of bij printertalen zoals HP-PCL.

## 7. UTF-16

Het oorspronkelijke Unicode-ontwerp hoopte om alle characters te kunnen nummeren in een reeks tot 65536. Nieuwe software zou strikt met twee bytes per character moeten gaan werken, en alle ASCII-software moest worden uitgefaseerd.

UTF-8 is ontworpen omdat die strikte twee-byte aanpak toch wel erg veel investeringen in bestaande, ASCII-compatible hard- en software zou vernietigen.

UTF-16 dient een heel ander doel, namelijk het uitwisselen/transporteren van Unicode-files. Daar is een knelpunt ontstaan toen de Unicode-bovengrens van 65535 te krap bleek te zijn, en die “extended” werd.

Als je in een file elk character in twee bytes codeert (dus het UCS-2 formaat gebruikt; zie paragraaf 2), dan loop je vast als je toevallig eens een character nodig hebt uit de *extended* nummering. Dat kun je dan niet onderscheiden. Daarvoor zou je alles met vier bytes (UCS-4) moeten coderen, en wordt je file dubbel zo groot. Omdat extended characters slechts zelden gebruikt worden (tenzij je toevallig specialist in dode talen bent), is vier bytes teveel verspilling. UTF-16 is een tussenoplossing.

In een UTF-16 encoded file heeft elk Unicode character met een nummer in het gebied beneden 65536, dat nummer gewoon in twee bytes geplaatst. In paragraaf 2 hebben we gezien dat dit de UCS-2 encoding heet. Maar in dat gebied zijn twee “gaten” vrijgehouden: D800–DBFF en DC00–DFFF. Het eerste gat bestaat dus uit de tweetallige waarden 110110..... en het tweede gat uit 110111..... In bitjes bekeken heeft elk gat 10 vrije bits.

De waarden van “extended-Unicode” kunnen in theorie tot 31 bits lang worden, maar zo ver zijn de definities nog lang niet ingevuld. Zelfs de laagste 20 bits zijn nog lang niet ingevuld. UTF-16 zegt: wil je een Unicode waarde boven 65535 verwerken, dan ga er maar van uit dat die maximaal 20 bits beslaat. Noteer dus die waarde in 20 bits, hak dat in tweeën, plaats de eerste 10 bits in het eerste “gat”, en de tweede 10 bits in het tweede “gat”. Dat levert in totaal vier bytes op.

UTF-16 is dus een techniek die rekening houdt met het feit dat extended Unicode (UCS-4) nummers tot meer dan 16 bits kunnen oplopen, terwijl jij een transportmethode wenst die tegen UCS-2 aanleunt, dus zoveel mogelijk twee-byte verpakkingen gebruikt. UTF-16 definieert hoe je in een UCS-2 encoded file toch een té groot nummer ertussen kunt smokkelen, zonder te moeten overstappen op een vier-byte verpakking (UCS-4) voor élk nummer.

Het kan een zinvolle keus zijn om teksten *in* een database op te slaan in UTF-16, en ze bij het erinzetten/eruithalen te vertalen van/naar UTF-8. Populair gezegd: UTF-16 is geschikt voor disks, en UTF-8 voor terminals en printers. UTF-16 produceert veel nul-bytes, dus is minder geschikt voor string-manipulatie software.

## 8. Nawoord

Het doel van UTF-8 is dus om bestaande hardware, zoals printers, displays en keyboards te kunnen blijven gebruiken. Eigenlijk is het een compatibiliteits-hack. Unicode is veel eenvoudiger dan UTF-8. Unicode is ook minder nadelig voor mensen uit het Midden- en Verre Oosten, voor wie UTF-8 drie bytes per character kost, en Unicode slechts twee.

Het belangrijkste nadeel van UTF-8 is dat de relatie tussen aantal bytes (opslagruimte) en aantal characters (weergaveruimte) niet in een oogopslag te bepalen is.

## 9. Samenvatting van de definities

We geven tenslotte een korte beschrijving van de belangrijkste begrippen uit dit rapport. De beschrijving is praktisch gehouden, en er is niet geprobeerd om tot in finesses exact te formuleren.

- **Unicode**  
Een systeem om alle symbolen uit levende talen een uniek volgnummer te geven. De nummers 0-127 corresponderen met de ASCII nummers, de nummers 128-255 corresponderen met Latin1. In eerste instantie ging men uit van een bovengrens van 64K.
- **UCS-2**  
De encoding-techniek waarbij elk Unicode nummer uit het gebied tot 64K wordt weergegeven in twee bytes. Merk op dat als je een ASCII of Latin1 file omzet naar UCS-2, hij precies twee keer zo groot wordt. De helft van alle bytes in de UCS-2 file wordt in dat geval binair 0. UCS-2 is opgevolgd door UTF-16.
- **Extended Unicode**  
Doorgroei van de Unicode nummering naar nummers boven de 64K, om ook symbolen uit oude talen, zeer kleine taalgroepen, exotica, enz. enz. te kunnen herbergen. De symbolen uit de “gewone” Unicode passen dus wel in UCS-2, die van de Extended Unicode niet meer.
- **UCS-4**  
De encoding-techniek waarbij elk Unicode nummer wordt weergegeven in vier bytes. Je kunt hiermee dus *alle* Unicode symbolen weergeven, maar als je een ASCII-file omzet naar UCS-4 wordt hij vier keer zo groot.
- **BMP: het Basic Multilingual Plane**  
In de Unicode-nummering beneden de 64K heeft men twee gaten van 1K elk vrijgelaten. Met BMP bedoelt men de 62K nummers die buiten die gaten liggen.
- **UTF-16**  
Een techniek om getallen uit het Extended Unicode gebied boven de 64K te mappen in de twee gaten onder de 64K. Daarmee wordt effectief een uitbreiding van UCS-2 gecreeerd. *Alle* Unicode waarden kun je nu coderen in een bestand dat maar twee bytes nodig heeft voor de gewone Unicode symbolen, terwijl Extended symbolen daar dan op een herkenbare manier als vier-byte groepjes tussen zitten.
- **UTF-8**  
Een techniek om alle Unicode-waarden op een zodanige manier om te rekenen dat de eerste 127 waarden (dus het gebied dat gelijk-op gaat met ASCII) één byte krijgen toegewezen. Dat betekent dat een ASCII file automatisch ook een UTF-8 file is, en ASCII-compatibele hard- en software gewoon in gebruik kan blijven. Unicode waarden vanaf 128 krijgen twee tot zes bytes, dus een Latin1-file is *niet* compatibel met UTF-8.
- **ASCII**  
Een techniek om een beperkt aantal latijnse letters, cijfers en leestekens te coderen in waarden tot 128. Letters met accenten horen daar *niet* bij. Voor letters met accenten, en speciale taalgebonden letters, bestaan ASCII-extensies die waarden van 128 tot 255 uitdelen. Nadeel daarvan is dat er zoveel verschillende extensies zijn, omdat er veel meer wensen zijn dan de uitbreiding tot 255 kan bevatten.