AN INTRODUCTION TO CHARACTER ENCODING ISSUES IN THE MOBILE WEB

Eduardo Casais
areppim AG
Bern, Switzerland

1. A LINGERING ISSUE

Character encoding – the binary representation of every symbol in documents delivered to mobile terminals – is often treated as an afterthought in mobile Web development. Many developers simply rely upon ISO-8859-1; not a bad choice, as this encoding efficiently supports all important Western European languages, has long been available in the mobile and fixed Internet, is widespread among low-end phones, and is the default encoding in the HTTP standard. Astute software engineers prefer UTF-8; this encoding supports Unicode, and hence the widest range of languages and associated glyphs – great for multilingual "world" applications. It is also a default for several application formats, most popular in the WWW, and available in newer mobile terminals. Even in Japan, i-mode gateways may take care of the complex mapping from UTF-8 to Shift_JIS-only capable terminals.

Producing content in one of these two major encodings and configuring a WWW server to advertise the content type and encoding properly generally suffices for mainstream applications. There can be complications however – especially when dealing with advanced functions or developing for exotic markets – and it is preferable to be aware of them. The article examines the situation in the context of mobile browsing.

2. THE DOCUMENT CHARACTER SET

Let us briefly recapitulate some concepts. A **character set** is a repertoire of abstract symbols (e.g. lowercase a with acute accent, uppercase alpha). Each character is mapped to a **code point** in a numeric space (resp. 0x00E1 and 0x0391 in the ISO-10646 space, or 225 in ISO-8859-1 and 193 in ISO-8859-7). Characters may correspond to several code points (as with Arabic letters, for which several forms must be distinguished). Finally, each code point is represented as bits and bytes depending on the **character encoding** scheme. Each of the 15 ISO-8859 code spaces has just one single byte encoding. ISO-10646 has two possible encodings: UCS-2 and UCS-4, using 2 and 4 bytes respectively. Unicode, whose code space is equivalent to ISO-10646, has, among others: UTF-8 (1 to 4 bytes), UTF-32 (4 bytes, with endian orderings), UTF-16 (2 or 4 bytes, with endian orderings), GB18030 (optimized for Chinese characters, 1, 2 or 4 bytes). Shift_JIS is a multi-byte character encoding, with sequences to access both Japanese code spaces JIS-X-0201 and JIS-X-0208. Many encoding schemes, including UTF-8, GB18030, all ISO-8859, and to a large extent Shift_JIS, comprise US-ASCII as a compatible subset – the basis for many protocols and formats in the Internet.

Web applications, as embodied in markup documents, operate within a specific character set. In practice, one must take care of this in two situations:

- When using numeric **character references** (such as `&#225;` for acute-accented a, or `\E1` in WCSS). The numeric value must then identify a legal code point of the document character set.
- When embedding non-standard characters, called **pictograms**, in the document. These characters must correspond to available code points in a part reserved for user-defined symbols of the document character set.

Since its origins, HTML specifies its document character set to be ISO-10646, with some US-ASCII control characters left unused. The standard further defines a number of special entities (`& > < &quot;`) and character entities (such as `&aacute;` and `&Alpha;`) that must be rendered by user-agents; this set corresponds to ISO-8859-1 in HTML 3.2, and has been extended in version 4.0 of the standard. The XML specification stipulates the document character set to be ISO-10646 too, which therefore applies to relevant XML dialects (WML, XHTML basic and XHTML mobile profile). XML just defines the same special character entities as in HTML, with the supplementary `&apos;`. The WAP standard provides special-purpose markup to insert pictograms into pages written in WML and XHTML mobile profile, so that in these cases one need not mess with non-standard symbols in reserved code points. ISO-10646 is also the document character set of CSS and WCSS; symbols can be designated by escape sequences of the form *\NNNNNN* standing for their hexadecimal code point.

One must distinguish between the document character set and the document encoding: it is possible to format an HTML or XML document with a non-Unicode, non-ISO-10646 encoding scheme (say ISO-8859-6), as long as the characters which fall outside the code space of the document encoding scheme (in this case, accented and Greek letters) are represented via appropriate Unicode-compliant numeric entities. The W3C standard does not define a default encoding for HTML documents; the default encoding for XML documents is UTF-8 or UTF-16.

Local standards may depart markedly from the norms set by the WWW Consortium. In particular, the major Japanese mobile operators (DoCoMo, Softbank, KDDI and Willcom) have been developing and documenting their mobile Web environments for a long time. There, the document character set is often conflated with the document encoding, and pictograms (even equivalent ones) are placed at different positions in private areas of the relevant code spaces.

In the case of i-mode, this means that numeric character references apply to code points in the Shift_JIS space if the document is encoded with Shift_JIS, and in the Unicode space if the document is encoded with UTF-8. Pictograms are represented directly as Shift_JIS bytes, by decimal numeric references pointing in the Shift_JIS reserved code space (0xF89F-0xF95E and 0xF9B1-0xF9FC), or by hexadecimal numeric references pointing in the Unicode code space (0xE63E-0xE6BA and 0xE70C-0xE757). In Europe, supported encodings are usually Windows-1252 or ISO-8859-1, and pictograms are represented via decimal numeric references in the ranges 0xE63E-0xE6A5 and 0xE6CE-0xE757 in the Unicode space.

Willcom follows a scheme similar to i-mode, except that the code space reserved for pictograms is different (0xF040 to 0xF14D in Shift_JIS).

KDDI deploys Openwave browsers; hence, pictograms are embedded in Web pages via encoding-independent markup. With other applications (e.g. e-mail), pictograms are inserted directly as special byte sequences. Shift_JIS is the preferred document encoding.

Softbank phones support EUC-JP, ISO-2022-JP, Shift_JIS encodings; newer devices (since the series “W” and “3G”) also support UTF-8, and their browsers handle numeric character references. Pictograms are entered as special byte sequences.

Clearly, a developer must first ascertain the exact document character set manipulated in the target environment, how it differs from standards, its relation to the document encoding, and the mapping of proprietary symbols. This applies to further applications as well: Java, for instance, specifies the application character set to be Unicode, provides a notation for numeric character entities, and lists the encodings to be supported by a compliant terminal. However, many small-footprint versions of the Java run-time have more limited capabilities – for instance many Java-capable Motorola handsets only handle UCS-2 and ISO-8859-1 encodings.

3. THE DOCUMENT CHARACTER ENCODING

Since one can represent every symbol in a document character set by a numeric entity, would it not be straightforward to encode every HTML, WML or XHTML page entirely in US-ASCII, with all non-ASCII characters appearing as numeric references? This approach is technically feasible, but exhibits several shortcomings:

- This kind of formatting reduces the legibility of documents. Furthermore, editors and authoring tools might not have in-built support to manipulate numeric references, forcing one to type `&#225;` explicitly instead of simply á. All the more so, since the notation for numeric references in style sheets differs from the one in the enclosing markup document – it is `\E1` for á, in external style sheets, in those embedded in `<style>` elements or in-line “`style`” attributes.
- Many older mobile phones do not support numeric character references at all.
- The documents thus encoded are vastly bulkier, especially for languages that do not use a Latin script: a single numeric reference requires *at least* 6 bytes to represent one non-ASCII character – mainstream encodings rarely require more than 4. Not only does this reduce transmission performance in wireless networks; it also makes the application run quicker against the limits on page size imposed by end-user devices. Ultimately, it costs more to the end-users.

Applying an efficient encoding interpretable by the end-user device is clearly a better approach. The list of character encodings supported by a terminal are present, as IANA-registered names, in its user agent profile and in the HTTP header field “`Accept-charset`” it sends. Each source of information has its advantages and shortcomings.

- The HTTP field associates quality values, ranging from 0.000 to 1.000, to each character encoding; it is thus easier to order and select the most suitable encoding satisfying the constraints of the application.
- In both the HTTP field and the user agent profile, the absence of an encoding implicitly entails that it is not supported by the device. In HTTP, the q-value 0.000 explicitly indicates that the corresponding encoding is not admissible.
- Contrarily to the HTTP header, the user agent profile is not altered by gateways or transcoding proxies standing between the terminal and the server.

As an example, here are the contents of the HTTP field sent by a Samsung SGH-X660:

```
iso-8859-1, us-ascii, utf-8;q=0.800, iso-10646-ucs-2;q=0.600, iso-8859-2;q=0.500, windows-1250;q=0.500
```

Which are thoroughly transformed after going through a transcoder:

```
iso-8859-1, windows-1252;q=0.3, utf-8;q=0.2, *;q=0.1, iso-8859-2
```

Whereas its user agent profile advertises the following:

```
<prf:CcppAccept-Charset>
      <rdf:Bag>
         <rdf:li>ISO-8859-1</rdf:li>
         <rdf:li>US-ASCII</rdf:li>
         <rdf:li>UTF-8</rdf:li>
         <rdf:li>ISO-10646-UCS-2</rdf:li>
      </rdf:Bag>
</prf:CcppAccept-Charset>
```

Obviously, the preferred character encoding is best determined from an unadulterated HTTP header field – although one should ignore the indication "`*`", which is generally unsafe. There are circumstances where this is impossible though:

- Whenever the server initiates the communication, as in push applications. Information about applicable encodings must then be extracted from the user agent profile (possibly returned by a push gateway in answer to a client capability query submitted by the application server), in particular from characteristics "`Push-Accept-Charset`", or even "`CcppAccept-Charset`" and "`MmsCcppAcceptCharSet`", if push-specific data is incomplete.
- Whenever the terminal neither transmits the HTTP header field "`Accept-charset`", nor has an official user agent profile – such as happens with i-mode phones and at least all phones from Softbank prior to the series "3G". In this situation, one must manage information about supported character encodings in a server-based terminal capability database.

When inspecting the HTTP header, it is good to look in the alternative field "`X-device-accept-charset`" first (in case a transcoder has modified the header), then in "`Accept-charset`", and eventually in "`X-up-devcap-accept-charset`" (for devices accessing the Internet via an old-fashioned Openwave gateway) before falling back on the user agent profile.

Whichever the source of information, the application should perform a normalization to eliminate unwanted variations in encoding names like ucs2, UCS-2, and iso-10646-ucs-2 – for instance by invoking functions analogous to `mb_preferred_mime_name()` in PHP.

4. THE DOCUMENT FONTS

A browser might have all the necessary mechanisms in place to interpret and manipulate Unicode characters, but representing them requires that suitable **fonts** be installed. Browsers rely upon the operating system to render fonts; Thunderhawk (a browser for Windows Mobile) used to be an exception – but its font pack was restricted to symbols defined in ISO-8859-1.

Because of cost, memory and marketing constraints, manufacturers often release the same phone model with a different set of pre-installed fonts in each market. As a consequence, a device accepting universal encodings such as UTF-8 even with a quality of 1.000, may display

content as strange blocks or some other symbol, although retrieved pages are correctly encoded; the nifty German – Hindi – Chinese multilingual on-line dictionary is unusable.

There is only one conclusive way to assess whether specific symbols (and specific symbols of a specific font size) are properly represented on a specific device: perusing the manufacturer documentation for the model variant at hand (i.e. the one released in a specific market), and performing ad-hoc on-line tests. This also applies to pictograms, as devices support more or less extant collections of these icons (thus, i-mode has two levels of support for pictograms).

When accurate documentation is unavailable or on-line testing too cumbersome, but knowledge about font availability is really important, one can resort to the following heuristics:

- One may deduce which fonts are present from the list of non-universal encodings in the user-agent profile or the HTTP header. A phone accepting Shift_JIS or Big5 as encodings quite likely supports the fonts needed to display text written in Japanese, respectively in traditional Chinese.
- The natural **languages** accepted by the user agent may give a hint as to which fonts are pre-installed: if it is ready to receive documents in "`zh-CN`", then it ought to render text written in simplified Chinese. A related technique is used in the Opera browser, which can estimate the character encoding from the document language. The list of languages, identified as per RFC3066, is sent in the HTTP header field "`Accept-language`" (or "`X-up-devcap-accept-language`"), and also appears in the user agent profile as "`CcppAccept-Language`", "`Push-Accept-Language`", or "`MmsCcppAcceptLanguage`".

One rarely goes to such lengths. Most of the time, end-users select the applications and sites appropriate for them and their mobile phone. Only for multilingual applications mixing different scripts on the same page might some form of detection by the application server be in order – or at least an initial warning about platform requirements to the user.

The application provider has to take the font capabilities of low-end or older handsets as a given and adjust the service accordingly. In the case of smartphones and PDA however (Symbian, Windows Mobile, PalmOS, iPhone, etc), the limitations of pre-installed configurations are increasingly overcome by downloading and installing additional commercial or freeware font packages and utilities, or even performing a substitution with TrueType fonts converted from a PC (a popular method to enhance Nokia N-series phones). Whether the device software fully implements advanced typographical properties such as ligatures and bidirectional display is a question that is settled by the terminal documentation and hands-on testing.

Finally, let us remember that keypads or keyboards and user interfaces are tailored for each region. A phone sold in Europe might be able to display Chinese symbols, but input routines required to enter text in the Chinese script, as well as the correct key labels, are provided natively only in the model variant customized for the relevant markets.

5. THE DOCUMENT CHARACTER SET SPECIFICATION

Internet standards define the ways servers advertise the character encoding of a document and the order of precedence of these various mechanisms. They apply to the mobile Web as well.

1. Charset declaration inside the HTTP header.

- Applies to all document formats (HTML, WML, XHTML, CSS, WCSS).

  The document is returned by the WWW server with a proper HTTP header field indicating its type and character encoding, for instance:

  ```
  Content-type: application/xhtml+xml;charset="iso-8859-7"
  ```

This method has the highest precedence. The construction of the HTTP header is driven by the (implementation-dependent) configuration of the WWW server. The server should be parameterized to return a charset value only when the application fails to set it in the HTTP header; otherwise, the server default might always prevail.

2. Charset declaration inside the document.

- XHTML basic, XHTML mobile profile, WML.

  The document must start with an XML declaration specifying the encoding:

  ```
  <?xml version="1.0" encoding="iso-8859-7"?>
  ```

  The "`encoding`" attribute may be left out only if the document is encoded in UTF-8 or UTF-16. Only byte order marks may precede the declaration; junk, such as empty lines or comments before the XML declaration jeopardize the recognition of the encoding and must be eliminated.

- HTML, XHTML basic, XHTML mobile profile, WML.

  The document header repeats the HTTP header in a meta-tag:

  ```
  <html>
      <head>
          <meta http-equiv="Content-Type"
                content="text/html;charset="iso-8859-7" />
          …
      </head>
      …
  ```

  The meta-tag must appear as closely as possible to the beginning of the document, preferably before any comments and other markup. The meta declaration has lower precedence than the XML declaration, but the NetFront browser has been known to give it a higher priority than even the HTTP header.

- CSS, WCSS.

  The document starts with an encoding declaration:

  ```
  @charset "iso-8859-7";
  ```

  It appears at the top of the file, preceded only by a possible byte-order mark.

Internal declarations are more than a fallback in case the HTTP header gets mangled during transmission: they constitute the only portable method to bind the character encoding to a document when it is accessed from the phone cache or the phone local file store.

3. Charset declaration attached to a hyperlink.

- HTML, XHTML basic, XHTML mobile profile.

  The URL explicitly states the character encoding of the destination document:

```
<a href="http://appsrv.mobi/doc.htm" charset="iso-8859-7">…</a>
```

- CSS, WCSS.

  The link explicitly states the character encoding of the external style sheet:

  ```
  <link type="text/css" media="handheld" charset="iso-8859-7"
        rel="stylesheet" href="http://appsrv.mobi/thestyle.css" />
  ```

These features are unavailable in WML, and Japanese HTML and XHTML variants.

4. Auto-detection of the encoding.

   - All document types.

     In the absence of any declaration, the browser may apply heuristics to determine the character encoding of the document. The XML standard specifies an approach to determine the encoding of XML documents, but browsers may rely upon their own proprietary algorithms, for those encodings they accept as input. Mobile developers must follow the recommendations under point 2 above (avoidance of junk) to facilitate the auto-detection of document encodings.

5. Fall back on a default character encoding.

   - CSS, WCSS.

     Style sheets embedded via `<style>` elements or "`style`" attributes inherit the encoding of the enclosing document, as specified by the CSS standard.

   - All XML dialects.

     As per RFC3023, in the absence of any other information, the default encoding is as specified by the XML standard (UTF-8 or UTF-16, with a proper byte order mark) – *except* when the document is presented with a subtype of "text" (e.g. text/xml), in which case the default encoding is US-ASCII.

   - HTML, CSS, WCSS (all documents subtypes of "text", except XML formats).

     When served through HTTP, and in the absence of any other information, the default encoding specified by RFC2616 is ISO-8859-1.

   These norms are often violated in the mobile Web: Japanese terminals almost always resort to Shift_JIS as a default for all documents. On smartphones and PDA, some browsers can be configured by the end-user to force a specific character set on input and output – overriding standard defaults and auto-detection.

Conscientious mobile developers do not leave the decision as to which encodings are actually used to the (potentially starkly divergent) defaults of various components, and implement unambiguous and consistent declarations as explained in points 1, 2 and 3.

6. INTERACTIONS BETWEEN APPLICATIONS

So far, we have analysed encoding issues in the context of requesting and then delivering content to a mobile browser. Three aspects further complicate the matter:

- Replies, i.e. content deliveries from the terminal to the server.

- Interactions between the mobile Web applications and other user agents on the terminal – such as e-mail, PIM, SMS, or MMS.
- The server environment – for instance scripting run-time, database, CMS.

Data flows back from the terminal to the server when the user fills in and submits a form in a web page. In the case of an HTTP GET method, the proper character encoding is first applied to field names and values, which are then further encoded following the URI-escaping scheme of RFC1738 (apart from letters, numbers and a few symbols, characters are represented in %NN notation, e.g. the tilde is %7E). The outcome, a string formatted as application/x-www-formurlencoded, is appended to the request URL, producing something such as this:

```
http://appsrv.mobi/doit?name=H%E9l%E8ne&msg=It+works%21
```

When user Hélène registers the message "`It works!`" to a hypothetical service with the page configured for ISO-8859-1. With UTF-8, the result is:

```
http://appsrv.mobi/doit?name=H%C3%A9l%C3%A8ne&msg=It+works%21
```

POST is the method of choice to send large amounts of data or upload files. Although one can use the same representation of request parameters as for GET methods, the format multipart/form-data exhibits a definite advantage: explicit information about character encoding accompanies the payload. Besides, because binary transfers are possible, the request may be less heavy than URI-escaped strings. Each component of a form is sent as a distinct part in the body of the response, properly encoded according to RFC2388, and with a "`Content-type`" field indicating the type and character encoding of the form field value. WWW and application servers are configured to interpret this information and decode text automatically; one must just make sure that the application ultimately receives data in a character encoding that suits it.

The attribute "`accept-charset`", bound to the `<form>` element (in HTML and XHTML) or to the `<go>` task (in WML), instructs the browser to encode data according to one of the listed character encodings. For instance

```
<form action="http://appsrv.mobi/doit" method="post"
      accept-charset="iso-8859-7 utf-8 utf-16"
      enctype="multipart/form-data">
```

indicates that the application server is ready to accept form data in any of the three mentioned character encodings. If the attribute is absent or unrecognized, the user agent falls back on the encoding used for the Web page itself, or possibly on a browser-specific default – not an infrequent occurrence since this useful attribute is far from being universally supported:

Content	Form attribute "accept-charset" defined	
Markup format	No	Yes
HTML	3.2	4.0, 5.0
XHTML basic	1.0	1.1
XHTML mobile profile	1.0, 1.1	1.2
WML	2.0 (`<form>`)	1.1, 1.2, 1.3, 2.0 (`<go>`)

The special versions of HTML and XHTML designed by Japanese operators seldom implement "`accept-charset`" in forms, since content is supposed to be in Shift_JIS anyway.

The fact that a browser decodes input in a certain range of encodings does not imply that it can produce output in the same range of encodings. In fact, the latter set is usually significantly smaller than the former one. The values in the "`accept-charset`" form attribute and the encoding of the Web page itself must take this further constraint into account. Prime candidate encodings derived from the HTTP header (preferably those with a q-value of 1), the user agent profile and manufacturer's technical manuals are usually universal encodings such as UTF-8 and UCS-2, and dominant local schemes (ISO-8859-1, Shift_JIS, etc).

Interactions with other user agents – for instance via `mmsto:` and `mailto:` URI schemes – raise similar difficulties: MMS readers and e-mail clients have their own restrictions regarding the allowable input and output character encodings – and these might not be exactly the same as for the browser. The situation gets thornier when accessing the wireless telephony application interface: which characters can be stored in the phonebook? Which symbols can be included in an SMS? Unfortunately, these functions, except for MMS, are not subject to a normalized description in the user agent profile, and may not be explained in enough detail in the readily available developers' documentation. Looking at non-Internet norms helps: if the `smsto:` URI scheme seems to behave haphazardly, a check whether it is not actually implementing the default 7-bit encoded alphabet of GSM 03.38 is in order…

Internationalization in service platforms is an issue whose comprehensive exposition is beyond the scope of a short paper. In short, tools that are not natively designed around Unicode hinder the development of internationalized applications. We pinpoint three essential facets:

1. The character set used internally. Modern software systems rely upon Unicode (frequently encoded as UCS-2 or UTF-16), and are thus able to process, compare and sort strings with few restrictions. Other environments carry a legacy of having been originally built for one-byte character sets (i.e. US-ASCII, ISO-8859-1); multi-byte string manipulation routines may exist, but often do not implement all required functions, exhibit inconsistent capabilities with respect to character encodings, or lack internationalization support for crucial services entirely (e.g. sorting). PHP, for instance, is still affected by these shortcomings.
2. The encodings used for information stored persistently. It may be possible to encode and save data (respectively, decode and read it in) in a number of character encodings. Thus, MySQL allows database administrators to specify the character encoding of text attributes at the level of databases, tables, and individual columns, augmented with a language-specific **collating sequence** (i.e. the sorting order stating that "ä" is sorted as "ae" in German, but comes after "z" and "å" in Finnish). The DBMS sorts data according to the collating sequence when a query is executed. Conversely, prudent programmers keep their PHP source code in ASCII – or at least in a single-byte encoding format.
3. The encoding parameters when communicating with client applications. Generally, it is possible to set up the encoding for outbound data, and the encoding which is assumed for inbound data (e.g. via `SET CHARACTER SET` in MySQL; via functions `mb_http_input`, `mb_output_handler`, and `mbstring` run-time configuration variables in PHP). Sometimes, as in PHP, an auto-detection scheme is relied upon when several possible input encodings are expected. Data is automatically converted from the internal character set to the output encoding, and from the input encoding to the internal character set.

Overall, the goal is to ensure compatibility between the character encodings accepted and produced by different units in the service delivery chain. For performance reasons, the selected encodings should actually be the internal character set of the various components.

7. CONCLUDING REMARKS

Ultimately, the most severe constraints regarding character encoding are imposed by market requirements: languages spoken in a country, scripts used to write, capabilities of phones released to customers, format of source material incorporated in Web sites, etc. From this perspective, mobile developers can rarely avoid dealing with well-established regional character sets such as Big5, GB2312, GB18030, KOI8-R, TIS-620, Shift_JIS, or ISO-2022-JP entirely. For generic or multilingual applications, the observations in the introduction apply: ISO-8859-1 is an efficient encoding for Western languages – which, because of the dissemination of the English, French, Portuguese and Spanish languages, is applicable in a large number of countries all around the world – while UTF-8 is well-suited to international services. Whenever possible, one should actually prefer ISO-8859-1 over UTF-8: ISO-8859-1 is a single-byte encoding whose 256 code points all map directly into the first 256 code points of Unicode – thus no decoding is necessary in practice, contrarily to UTF-8, which is a variable multi-byte scheme. Furthermore, UTF-8 requires two bytes rather than one to represent ISO-8859-1-specific symbols – hence ISO-8859-1 holds an advantage regarding transmission over the air too. The table in the appendix, derived from 4295 user agent profiles, shows the relative importance of universal and regional character sets supported by mobile browsers. These statistics are an approximation, since they evidently underestimate the properties of those models that do not publish any user agent profile: many WAP 1 handsets, low-end phones, and a large range of Japanese terminals.

Internet standards (www.ietf.org, www.w3.org) address in detail many questions regarding internationalization, but, as already mentioned, are not undisputedly authoritative because of the prevalence of market-specific solutions, foremost in such Asian countries as Japan and Korea. There, the developers' documentation published by operators constitutes the reference. The W3C site provides a wealth of tutorials, hands-on FAQ, and reference documents about internationalization (www.w3.org/International). Other sites that delve in depth into the concepts and practical difficulties with character sets can be found at www.alanwood.net/unicode and www.cs.tut.fi/~jkorpela/chars/index.html. The various national and international norms (especially www.unicode.org) remain indispensable for those developers who must implement complex encoding, decoding and typesetting utilities.

8. APPENDIX: FREQUENCY OF SUPPORTED CHARSETS

Charsets are sorted according to decreasing frequencies of appearance in user agent profiles, and arranged in regional groups. Those charsets mentioned in less than 0.82% of the profiles are summed up under the category "other"; we observe the presence of a long tail of special encodings for various Asian languages (Tamil and Vietnamese, besides further charsets for Chinese, Japanese, Korean and Thai). A few profiles, marked "none" do not declare any supported charset. ASCII is classified as the lowest common denominator amongst encodings.

Charset	World **universal**, misc.	Europe **West**, Centre, *North*	Europe **Cyrillic**, Greek	Far East **Ch**. Jap. Kor. *Thai, etc*	Near East **Turk**. Hebr. *Arabic*
utf-8	**95.9** %				
us-ascii	87.0 %				
iso-8859-1		**84.9** %			
ucs-2	**65.4** %				
utf-16	**26.8** %				
koi8-r			**10.4** %		
iso-8859-2		9.0 %			
iso-8859-7			8.9 %		
iso-8859-5			**8.3** %		
big5				**8.2** %	
iso-8859-9					**8.2** %
iso-8859-4		*7.9* %			
windows-1252		**7.5** %			
windows-1250		7.0 %			
shift_jis				6.7 %	
windows-1253			6.6 %		
windows-1254					**6.6** %
euc-jp				5.6 %	
iso-2022-cn				**5.3** %	
iso-2022-jp				5.3 %	
gb2312				**5.2** %	
iso-8859-3					**5.2** %
iso-8859-6					*4.9* %
iso-8859-8					4.9 %
iso-8859-10		*4.3* %			
iso-8859-15		**3.9** %			
iso-8859-8-i					3.8 %
windows-1257		*3.7* %			
windows-1256					*3.7* %
windows-1251			**3.7** %		
windows-1255					3.7 %
cp936				**3.6** %	
euc-kr				3.5 %	
gb18030				**3.4** %	
ks-c-5601				3.3 %	
utf-7	**3.3** %				
tis-620				*3.3* %	
ucs-4	**3.2** %				
iso-8859-13		*1.2* %			
iso-8859-14		**1.2** %			
other	0,4 %	0.7 %	0.6 %	2.8 %	0.3 %
none	1.7 %				

REFERENCE

This paper has been published on mobiForge.com at

http://mobiforge.com/developing/story/character-encoding-issues-and-mobile-web

and can be downloaded in PDF format from areppim.com at

http://www.areppim.com/print_charsetv220081208.pdf

ABOUT THE AUTHOR

Eduardo Casais has been working on mobile Web technologies since 1997. He led the development of the protocol stack and of transcoding and content adaptation facilities in the Nokia WAP Server. As co-founder of areppim AG, he is now developing services to access, analyse and display quantitative information graphically on mobile phones.

ABOUT AREPPIM AG

areppim AG develops Internet applications, with an emphasis on the display of quantitative information. The site http://www.areppim.com publishes data on a wide range of topics, presented as intuitive, content-rich charts and often accompanied by concise analyses. Naturally, data can also be accessed with mobile phones at http://mobile.areppim.com through a no-frills, mobile-optimized interface.

ADDRESS

areppim AG
Wankdorffeldstrasse 102
P.O. Box 261
CH-3000 Bern 22
Switzerland

e-mail: info@areppim.com