Text Encodings

- if an input stream is to be interpreted as text, the incoming bytes must be converted to characters
- figure on the right: predominant ASCII mapping (128 code points)
- in the past, the big variety of text encodings frequently caused surprises
- sample UTF-8 text parsed as CP-1252:
  Hier gÃ¤be es gute GrÃ¼nde fÃ¼r die Nutzung von Unicode!

Source: msdn.microsoft.com/en-us/library/60ecse8t(v=vs.80).aspx
Unicode Initiative

- Unicode provides an encoding that embraces the characters of most languages of the world, and numerous additional symbols:
  - range of the Unicode encoding: 0x0000-0x10FFFF (16 × 65536 characters):
    - codes that fit into the first 16 bits (denoted U+0000–U+FFFF) have been assigned to encode the most widely used languages and their characters (*Basic Multilingual Plane*, BMP).
    - codes U+00–U+7F have been assigned to match the ASCII encoding
  - Unicode 6.0 will introduce numerous new decorative symbols:
    - 🌴 🌼 🐘 💌 ❤️ 🌱 🌟 🌡️ 😊 😞
  - **UTF** (the *Unicode Transformation Format*) defines how a Unicode character is mapped into units of 32, 16 and 8 bits
Unicode 6.0

- for character nerds: buy decodeunicode (Hermann Schmidt Verlag, 2011)
UTF: Unicode Transformation Format

UTF-32

• characters in the range U+0000–U+10FFFF are mapped to the corresponding 32-bit value 0x00000000–0x0010FFFF

• example transformation for the two characters **Hi**: 
  
  00000000 00000000 00000000 01001000
  00000000 00000000 00000000 01101001

• example transformation for the two characters **日本**: 
  
  00000000 00000000 01100101 11100101
  00000000 00000000 01100111 00101100

• advantage: fixed-length encoding

• what may be disadvantages of this encoding?
UTF-16

- characters are mapped to the following mapping scheme:
  U+000000-U+00FFFF: ________ ________
  U+010000-U+10FFFF: 110110__ ________ 110111__ ________
- for values ≤ 0x00FFFF, replace spaces with the character code
- otherwise, subtract 0x010000 and fill spaces with resulting 20-bit value
- example transformation for the two characters Hi:
  00000000 01001000 00000000 01101001
- example transformation for the two characters 日本:
  01100101 11100101 01100111 00101100
- advantage: takes less space than UTF-16

Do you think this can be further optimized?
UTF-8

- today’s most prevalent encoding; popular because:
  - bytes (octets) are the basic processing unit of hardware
  - text-processing software today is built to deal with 8 bit character encodings

- mapping scheme:
  
  U+000000-U+00007F: 0_______
  U+000080-U+0007FF: 110_____ 10______
  U+000800-U+00FFFF: 1110____ 10______ 10______
  U+010000-U+10FFFF: 11110___ 10______ 10______ 10______

- example transformation for the two characters Hi:
  01001000 01101001

- example transformation for the two characters 日本:
  11100110 10010111 10100101 11100110 10011100 10101100
Advantages of UTF-8

- length of a byte sequence can be computed by counting the *leading 1-bits* (single-byte characters have a leading 0-bit)
- *character boundaries* are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream)
- encoding does not affect (binary) *sort order*
- traditional 7-bit ASCII texts take *little space*
- text processing software which was originally developed to work with ASCII remains functional

⚠️ once again... are there any downsides around UTF-8?
XML: Text Declarations

- a well-formed XML document may start off with an optional header, the *text declaration*
- example: `<xml version="1.0" encoding="UTF-8"/>
- it includes the **XML version** and the **document encoding**
- defined XML versions so far: **1.0; 1.1** (only few implementations)
- the encoding specifies how bytes in the input stream are to be interpreted
- most popular encodings:
  - **US-ASCII**: 128 characters; most-upper bit of all characters is always 0
  - **ISO-8859-1**: character set with most popular Western characters
  - Unicode (**UTF-8, UTF-16, UTF-32**): provides the full range of characters; slowly rules out all other formats
BOM: Byte Order Marks

- some text editors and tools prefix text files with \textit{byte order marks}
- BOMs are mainly used for Unicode formats; they indicate how characters are stored in a file:

  
<table>
<thead>
<tr>
<th>Encoding</th>
<th>BOM</th>
<th>Character 0x3F (?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>EF BB  BF</td>
<td>00111111</td>
</tr>
<tr>
<td>UTF-16 (BE)</td>
<td>FE FF</td>
<td>00000000 00111111</td>
</tr>
<tr>
<td>UTF-16 (LE)</td>
<td>FF FE</td>
<td>00111111 00000000</td>
</tr>
</tbody>
</table>

- if no BOM exists, encoding must be \textit{guessed} by the application

Let’s switch back to the XML context...

- the \textit{text declaration} provides a generic way to specify the text encoding
- encoding is still guessed when reading the first bytes \footnote{...why?}
**Outlook: XQuery and Unicode**

- all XQuery expressions and functions work with Unicode – encoding issues are history!
- functions exist to convert strings to code points, and vice versa:
  ```
  string-to-codepoints('abc')
  codepoints-to-string((97, 98, 99))
  ```
- BaseX provides additional functions to explicitly convert input and output to specific encodings:
  ```
  docs.basex.org/wiki/File_Module
  docs.basex.org/wiki/Conversion_Module
  ```
- the following sample query returns the hexadecimal byte representation of a specific string and encoding:
  ```
  file:read-text('input.txt', 'ISO-8859-1') !
  convert:string-to-hex(., 'UTF-16LE')
  ```