Character sets and Unicode in Firebird

Source: Firebird Conference 2009 in Munich; Stefan Heymann.

Character set definition becomes increasingly important as the world of database programming spreads more and more across national borders. Today it is often necessary for applications to also meet the requirements of other countries. The problem of multilingual interfaces is just one aspect of internationalization. A modern application needs to handle the particularities specific to individual countries such as, for example, sorting order (collation). In the German language the umlauts ä, ö and ü are integrated in the alphabet using the letter combinations ae, oe and ue. At the same time there are also special characters in the French language, which are not used in the German language such â, á and à.

There are completely different problems with versions whose characters are not known in the European character sets, for example Korean or Chinese. These character sets also often contain many more characters, which cannot be incorporated in the 8 bit character sets, as the technical upper limit lies at 256 (=28) different characters. For this reason Firebird/InterBase® implements character set support.

We will begin with an introduction to the concept of character sets and Unicode, and show you how these concepts fit together.

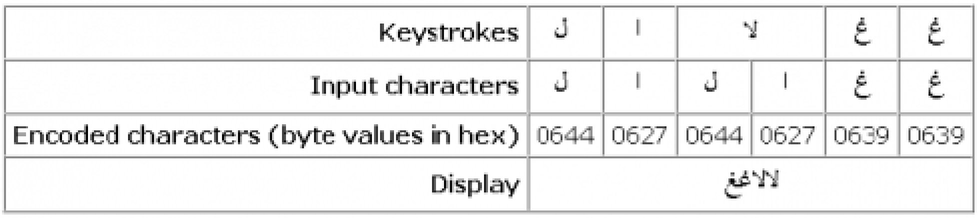
Characters and glyphs

There is a difference between glyphs and characters. A glyph is something you can see with your eyes (it can be the way a character is visually represented). For example:

Latin upper case A:

A character is a more abstract concept. The rendering of characters as glyphs is the job of the rendering machine (Postscript, GDI, TrueType, web browser etc.). What we are talking about in the context of databases is the characters, not the glyphs. A character set is a definition that defines characters and gives them an integer number.

It it important to know is that glyphs are not simply a string of rectangles displayed left to right in all languages, as we are used to in Latin scripts that we use for English, German, French etc. You can have characters that go from right to left (Arabic, Hebrew), or top to bottom as in Japanese and Chinese. You can also have the effect that several characters “melt” into one glyph, so it’s not as easy as we, who are used to Latin language systems, think.



Keystrokes	ل	ا	لا		غ	غ
Input characters	ل	ا	ل	ا	غ	غ
Encoded characters (byte values in hex)	0644	0627	0644	0627	0639	0639
Display	لاغغ					

ASCII

American Standard Code for Information Interchange, ASCII which is standardized in ISO 646. It is defined as 7 bit numbers ranging from 0 to 127 (00-7F) with 32 invisible control characters (NUL, TAB, CR, LF, FF, BEL, ESC, …), with the positions 0 to 31. It includes the Latin letters A-Z, a-z, digits 0-9, punctuation characters, .,:;?!

As the name implies it is optimized for the English language, so there are no accents, umlauts etc.

ASCII for Europe was introduced in the times of MSDOS, where, for example, some characters got reassigned for German or French. For example:

```
[ = Ä
\ = Ö
] = Ü
```

This meant of course that you couldn't use these special characters any more and you had to match the settings in the printer with that of the system. And it was impossible to mix several languages, so a sentence like this, a German sentence with French words in it:

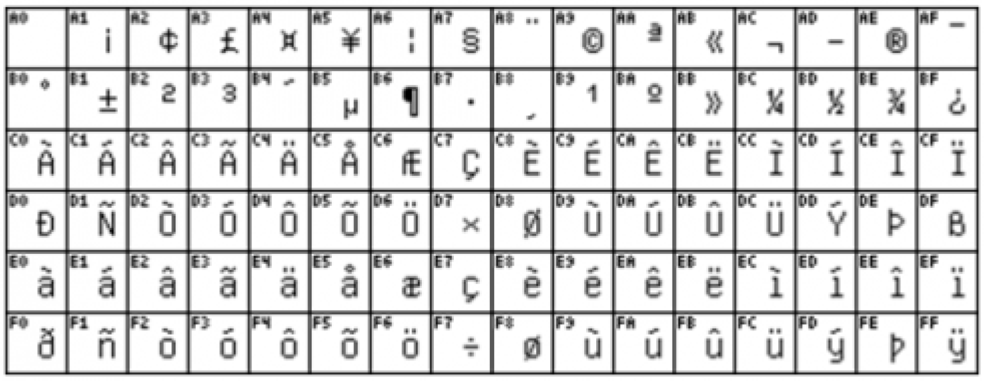
“Amélie knackt gerne die Kruste von Crème Brulé mit dem Löffel.”

was impossible to show onscreen or print on a printer. And so it died together with DOS.

The next idea was to use the 8th bit which had so far remained unused. So, with the additional 128 characters, you now have the complete code range from 0 to 255 (00 … FF) to decode characters. With this system a lot of different character sets have evolved. The most common ones are ISO 8859-x (ASCII + 160…255) and ISO 8859-1 (Latin-1) – for Western European languages. They use the ASCII base from 0-127 and define new characters for 160 onwards. The most common one is called Latin-1, it's the dash 1 character set which defines enough characters to be useful for all Western European languages. In Windows, there is Windows code page 1252, which is based on ISO8859-1 and defines a few additional characters (128…159).

back to top of page

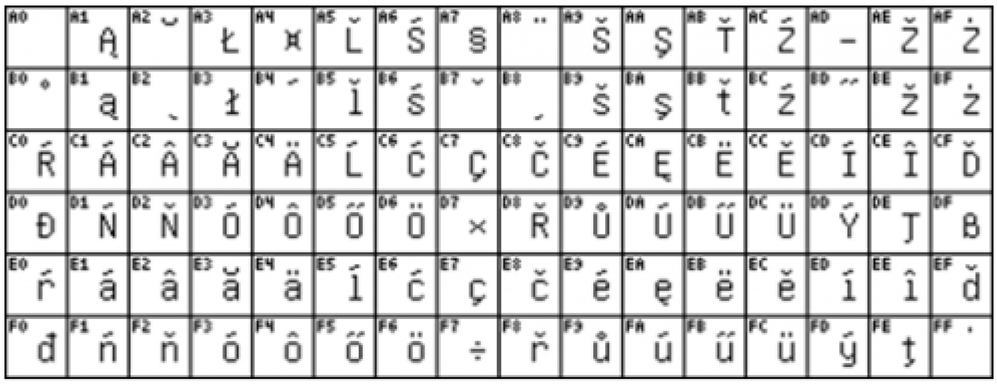
ISO8859-1 / Latin-1



A0	A1 ¡	A2 ¢	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 ¹	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

Latin-1 covers most West European languages, such as French (fr), Spanish (es), Catalan (ca), Basque (eu), Portuguese (pt), Italian (it), Albanian (sq), Rhaeto-Romanic (rm), Dutch (nl), German (de), Danish (da), Swedish (sv), Norwegian (no), Finnish (fi), Faroese (fo), Icelandic (is), Irish (ga), Scottish (gd), and English (en), incidentally also Afrikaans (af) and Swahili (sw), thus in effect also the entire American continent, Australia and much of Africa.

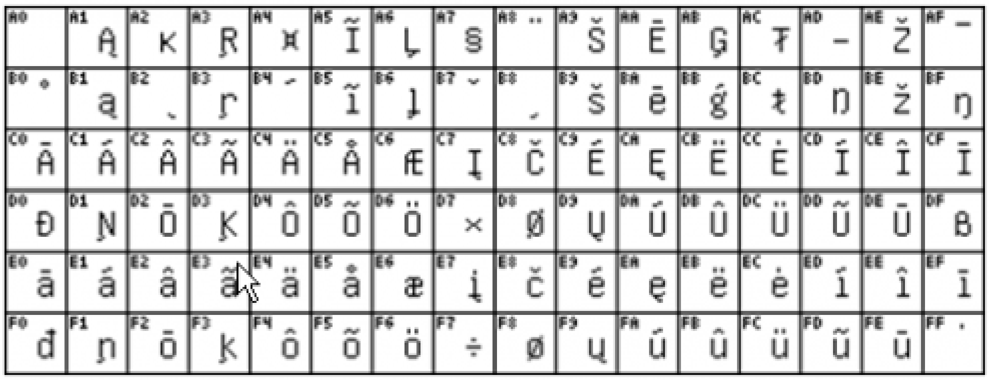
ISO8859-2 / Latin-2



A0	A1 Ą	A2 ˘	A3 Ł	A4 ¤	A5 Ľ	A6 Ś	A7 §	A8 ¨	A9 Š	AA Ş	AB Ť	AC Ź	AD -	AE Ž	AF Ż
B0 °	B1 ą	B2 ˛	B3 ł	B4 ´	B5 ľ	B6 ś	B7 ˇ	B8 ¸	B9 š	BA ş	BB ť	BC ź	BD ˝	BE ž	BF ż
C0 Ŕ	C1 Á	C2 Â	C3 Ă	C4 Ä	C5 Ĺ	C6 Ć	C7 Ç	C8 Č	C9 É	CA Ę	CB Ë	CC Ě	CD Í	CE Î	CF Ď
D0 Đ	D1 Ń	D2 Ň	D3 Ó	D4 Ô	D5 Ő	D6 Ö	D7 ×	D8 Ř	D9 Ů	DA Ú	DB Ű	DC Ü	DD Ý	DE Ţ	DF ß
E0 ŕ	E1 á	E2 â	E3 ă	E4 ä	E5 ĺ	E6 ć	E7 ç	E8 č	E9 é	EA ę	EB ë	EC ě	ED í	EE î	EF ď
F0 đ	F1 ń	F2 ň	F3 ó	F4 ô	F5 ő	F6 ö	F7 ÷	F8 ř	F9 ů	FA ú	FB ű	FC ü	FD ý	FE ţ	FF ˙

Latin-2 covers the languages of Central and Eastern Europe: Czech (cs), Hungarian (hu), Polish (pl), Romanian (ro), Croatian (hr), Slovak (sk), Slovenian (sl), Sorbian.

ISO8859-4 / Latin-4

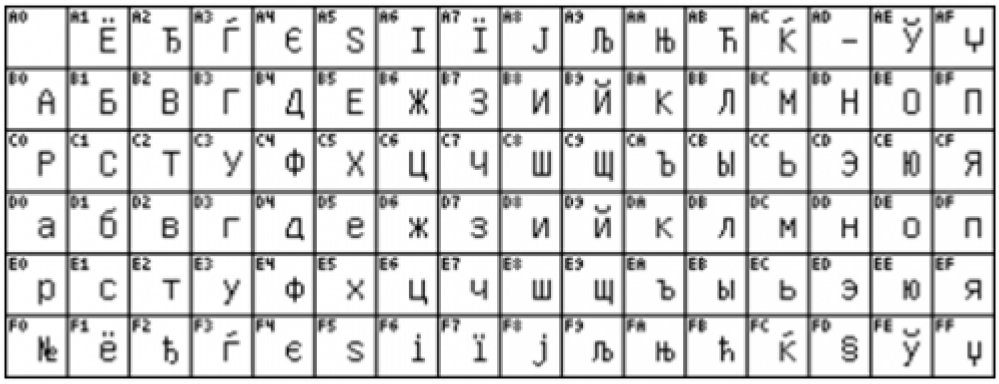


A0	A1 Ą	A2 ĸ	A3 Ŗ	A4 ¤	A5 Ĩ	A6 Ļ	A7 §	A8 ¨	A9 Š	AA Ē	AB Ģ	AC Ŧ	AD -	AE Ž	AF ¯
B0 °	B1 ą	B2 ˛	B3 ŗ	B4 ´	B5 ĩ	B6 ļ	B7 ˇ	B8 ¸	B9 š	BA ē	BB ģ	BC ŧ	BD Ŋ	BE ž	BF ŋ
C0 Ā	C1 Á	C2 Â	C3 Ã	C4 Ä	C5 Å	C6 Æ	C7 Į	C8 Č	C9 É	CA Ę	CB Ë	CC Ė	CD Í	CE Î	CF Ī
D0 Đ	D1 Ņ	D2 Ō	D3 Ķ	D4 Ô	D5 Õ	D6 Ö	D7 ×	D8 Ø	D9 Ų	DA Ú	DB Û	DC Ü	DD Ũ	DE Ū	DF ß
E0 ā	E1 á	E2 â	E3 ã	E4 ä	E5 å	E6 æ	E7 į	E8 č	E9 é	EA ę	EB ë	EC ė	ED í	EE î	EF ī
F0 đ	F1 ņ	F2 ō	F3 ķ	F4 ô	F5 õ	F6 ö	F7 ÷	F8 ø	F9 ų	FA ú	FB û	FC ü	FD ũ	FE ū	FF ˙

Northern Europe and Baltic Estonian (et), Latvian (lv, Lettish) and Lithuanian (lt), Greenlandic (kl) and Sami and Lappish.

back to top of page

ISO8859-5 / Cyrillic



A0	A1 Ё	A2 Ђ	A3 Ѓ	A4 Є	A5 Ѕ	A6 І	A7 Ї	A8 Ј	A9 Љ	AA Њ	AB Ћ	AC Ќ	AD -	AE Ў	AF Џ
B0 А	B1 Б	B2 В	B3 Г	B4 Д	B5 Е	B6 Ж	B7 З	B8 И	B9 Й	BA К	BB Л	BC М	BD Н	BE О	BF П
C0 Р	C1 С	C2 Т	C3 У	C4 Ф	C5 Х	C6 Ц	C7 Ч	C8 Ш	C9 Щ	CA Ъ	CB Ы	CC Ь	CD Э	CE Ю	CF Я
D0 а	D1 б	D2 в	D3 г	D4 д	D5 е	D6 ж	D7 з	D8 и	D9 й	DA к	DB л	DC м	DD н	DE о	DF п
E0 р	E1 с	E2 т	E3 у	E4 ф	E5 х	E6 ц	E7 ч	E8 ш	E9 щ	EA ъ	EB ы	EC ь	ED э	EE ю	EF я
F0 №	F1 ё	F2 ђ	F3 ѓ	F4 є	F5 ѕ	F6 і	F7 ї	F8 ј	F9 љ	FA њ	FB ћ	FC ќ	FD §	FE ў	FF џ

Turkish.

back to top of page

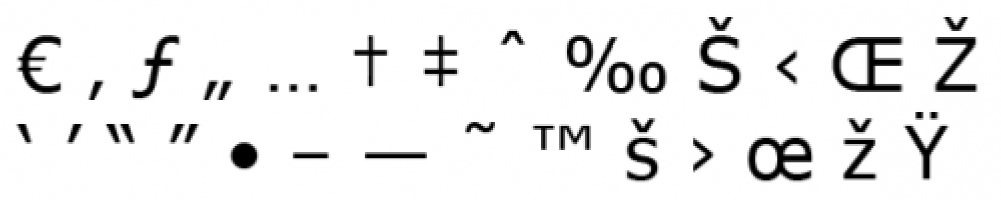
Windows character sets ("code pages")

The Windows code pages are partly congruent, have the same definitions as the ISO character sets, but not always, so you cannot rely on that. The ISO characters have no characters in this range 128-159; Windows defines additional visible (non-control) characters, for example, hyphens n length and m length (vs. Dash -), typographical quotation marks, etc. Windows character sets are officially registered at IANA. These are the most common Windows code pages:

874	Thai
932	Japanese
936	Simplified Chinese
949	Korean
950	Traditional Chinese
1250	Central European
1251	Cyrillic
1252	Western European
1253	Greek
1254	Turkish
1255	Hebrew
1256	Arabic
1257	Baltic
1258	Vietnamese

Windows-1252

Windows 1252 is the corresponding code page for Western European languages, so if you’re using German, French, Spanish Windows, this is the character set that Windows is using. It has additional characters in the 128…159 range:



€ ‚ ƒ „ … † ‡ ˆ ‰ Š ‹ Œ Ž
‘ ’ “ ” • – — ˜ ™ š › œ ž Ÿ

The € sign was introduced in Windows 2000.

back to top of page

Multi-byte character sets MBCS

This uses multiple bytes to define one character, which is often used for Eastern Asian languages (CJK which is the abbreviation for Chinese, Japanese, Korean). The string length is different from the length of the character chain, so it makes the calculation of string length and substrings more difficult. There are a few functions in Firebird to help with the processing:

- BIT_LENGTH: length of a string in bits (!).
- CHAR_LENGTH/CHARACTER_LENGTH: length of a string in characters.
- OCTET_LENGTH: length of a string in bytes.

back to top of page

Unicode

Unicode is the concept to go beyond the limit of these 1 byte character sets. The idea is to have one single character set for all languages that you can think of, including languages that we have never heard of. So the Unicode definition is a thick book with all kinds of definitions for all kinds of characters from all over the world. There was even an initiative to apply for the inclusion of Star Trek’s Klingon language with its special characters! The goal is that there should be no code overlaps between languages, and it must be hardware and operating system independent. The standardization is also ISO; ASCII is ISO646 and Unicode is ISO10646.

Unicode started with a code base that was 16 bits per character wide, but for quite some years now it is 32 bits per character wide. The Unicode standard has the ability to code 1,114,112 different characters; currently only a small fraction of that is used. But with every update of the standard more characters are being added. The current version in October 2009 is 5.2.0. Unicode only defines characters, not glyphs. So it’s still up to the rendering machines to create pretty glyphs from the character strings produced. It is practically equal to ISO/IEC10646.

Character definition

Unicode defines a numerical Code Point which is usually noted in hexadecimal with at least 4 digits and if it is more than 16 bits long, it can be up to 5 digits (scalar value) and an Identifier (description) for every character. For example:

0041	**Latin capital letter A**
00E4	Latin small letter a with a dieresis
0391	Greek capital letter alpha
05D0	Hebrew letter alef
0950	Devanagari om
1D56C	Mathematical bold fraktur capital A

The code space ranges from 0 to 10FFFF; the usual definition is hexadecimal with preceding U+ and at least 4 digits, for example: U+0020, U+0041, U+1D56C. The character names are also defined so they can only consist of letters, digits, hyphens and white space.

Unicode coding

Unicode itself is just a definition of characters and their numerical value code point. It does not define how these are stored in memory. There are several standards to store a Unicode string in memory. So there are various codings around:

- 8-bit (UTF-8)
- 16-bit (UCS-2, UTF-16)
- 32-bit (UCS-4, UTF-32)
- PunyCode for international domain names
- UTF-7, UTF-1 etc.

UCS-2 & UTF-16

UCS-2 is the classical Unicode representation. You have 16 bits per character, but you can only code an area from 0000…FFFF – also called the Basic Multilingual Plane (BMP). Since Unicode 3.1, UCS-2 is not able to code all the Unicode range that is defined, so this was replaced by UTF-16. Unicode is often used as a synonym for UCS-2 which is wrong and can lead to misunderstanding.

UTF-16 also uses 16 bits per character, but all characters above FFFF must be coded as a “surrogate pair”, and occupy two contiguous 16-bit words; these two 16 bits are stored directly after each other in the memory which also makes it more difficult to calculate string length, substrings and so on. The complete code space can be coded, it is used by Windows 2000 and later, and is also used by Delphi 2009 upwards (UnicodeString type). The bad thing in Delphi 2009 is that the definition of the string type is hard coded to UnicodeString, which loses the backwards compatibility, because the old definition was String=ANSIString.

The problem with UCS-2 and UTF-16 is that the low and high byte and the low and high word ordering, so on little endian machines this is different than on big endian machines. So we must differentiate between big endian and little endian representations (differentiate in UTF-16BE and UTF-16LE in metadata). There’s always a BE and an LE flavor. And when you have a text and you know it’s UTF-16 there should be a byte order mark (BOM) at the beginning of the text, consisting of the character U+FEFF, which you can use to see what kind of endianness you have. U+FEFF is the Unicode byte order mark, it should be the very first character in the text. U+FFFE is and will always be an invalid code point, so if you find this in your code you know you have to switch everything around.

UCS-2 versus UTF-16

UTF-16 is backwards compatible. A correct UCS-2 string is also a correct UTF-16 string. Sometime the word Unicode is used as a (bad) synonym for these, and it’s not clear which one.

The types in Delphi are WideString, wchar_t is for C languages. Windows NT3 and NT4 use UCS-2 and since Windows 2000 Windows itself uses UTF-16.

UTF-8

UTF-8 is also widely used to encode Unicode. Here 8-bit strings are used. The US-ASCII characters which also the first 128 characters of the Unicode definition stay untouched. Other characters occupy 2-4 contiguous bytes. The complete code space can be coded. There is the same problem as with multi-byte character sets with the calculation of string lengths, substrings etc.

UTF-8 coding

As already mentioned, the US-ASCII characters stay untouched, so bit 7 is, as in ASCII, always reset to 0: 0xxxxxxx. All other characters of the complete Unicode code space are sequences of bytes, with the 7th bit set. The first byte of such a sequence shows as many leading bits set as the whole sequence so when the first byte is set, it is starting a sequence of 2 bytes: 1xxxxxxx.

Recognize the type of byte:

- Complete character: 0xxxxxxx
- Part of a sequence: 1xxxxxxx
- Sequence head: 11xxxxxx
- Sequence tail: 10xxxxxx

So, when you have UTF8 text you can grab every single byte out of it and recognize what it is. When the first is a zero, it’s probably in the US-ASCII range, when it’s a 1 it’s part of a sequence, when it’s 11 it is the head of a sequence, and 10 shows it’s somewhere in the tail of a sequence.

The bits after the type bits remain for coding the coding point. Here is an illustration of the German ä (Latin small letter “a” with dieresis:

```
00E416 = 111001002

  00000 – 00007F:   0xxxxxxx
  00080 – 0007FF:   110xxxxx 10xxxxxx
  00800 – 00FFFF:   1110xxxx 10xxxxxx 10xxxxxx
 100000 – 1FFFFF:   11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

 110xxxxx   10xxxxxx
    ooo11     100100
 ________   ________

 11000011   10100100    bin
```

```
C3     A4         hex
195        164        dec
Ã      ¤        Latin-1#
```

UCS-4 & UTF-32

UCS-4 and UTF-32 use 32 bits for every character. So every Unicode code point occupies one 32-bit word in the memory. It is fat, but it's very handy (1 word = 1 character), because it's easy to calculate the length of strings and substrings, it does however waste a lot of memory.

- Endianness issues (UTF-32BE, UTF-32LE, BOM)
- Complete code space can be coded
- No practical differences between UCS-4 and UTF-32

back to top of page

Plain text

When someone gives you a text file and says that it is plain text file what is it really? According to Joel Spolsky, the software developer, “There ain't no such thing as plain text”. For every string, for every text (file, email, attachment, download etc.) the encoding must be known. You must have metadata, what kind of character set it is, what Unicode representation you have.

Transcoding / Transliteration

It is possible to transcode or transliterate texts between these various character sets, which is often done by going through a Unicode station, eg. Windows-1252 → Unicode → ISO 8859-1. There are translation tables on the Unicode website (https://www.unicode.org/) where for every character set they define which Unicode code point corresponds with the characters in other character sets. You must know that when transcoding to a different character set or you can lose characters. For example, ك becomes ? in a Western language character set, or characters can be transformed, eg. Ç can become C in character sets where there is no Ç.

Sorting

Sorting rules also apply for searching. There are cultural differences when sorting, for example, when you need to sort alphabetically for a telephone directory or similar, the German Ä can be treated as A (as in the English language), AE (in German) or as a separate character after Z (in Swedish).

Unicode defines algorithms and delivers tables for sorting.

Case mappings

Case mappings are not available in every language. When you want to transfer from lower case to upper case or vice versa, they are not always reversible. In English we know that a lower case “i” with a dot converts to an “I” without a dot in upper case. In Turkish they have a character “i” without a dot above it. When converted to upper case, this produces an “I”. When however you convert an “i” with a dot, you get an upper case “i” with a dot. So you must know the source and/or language of the characters you wish to convert. In German they have the “ß” which only exists as a lower case character. In upper case it is converted to two characters: “SS”. So case mappings are language dependent and need to be considered with care.

Comparisons, sorting

When you need to compare or sort it is often important to compare independent of case or independent of accents.

- Case insensitive:
    - Firebird = FIREBIRD ?
    - river = RiVeR ?
    - Fluß = FLUSS ?
        - a B c ↔ B a c

If aBc is sorted case-sensitively the B is listed first because upper-case characters are code pointed before lower-case characters.

- Accent insensitive:
    - Amélie = Amelie ?
        - a é i o ù ↔ a i o é ù

back to top of page

Firebird and character sets

Default character set

A default character set for all string columns can be defined together with CREATE DATABASE:

```
CREATE DATABASE ‘EMPLOYEE.GDB’
   DEFAULT CHARACTER SET ISO8859_1;
```

You still then have the option to override this default character set in all fields:

```
CREATE TABLE PERSONS (
   PERS_ID INTEGER NOT NULL PRIMARY KEY,
   LAST_NAME VARCHAR(50),
   FIRST_NAME VARCHAR(50),
   CZECH_NAME VARCHAR(50) CHARACTER SET ISO8859_2
);
```

VAR/VARCHAR fields

In Firebird every CHAR or VARCHAR field/column has the default character set applied by Firebird. This character set will be used for storage. You can define the character set for individual fields and domains at the time of creation:

```
CREATE TABLE PERSONS (
   PERS_ID INTEGER NOT NULL PRIMARY KEY,
   LAST_NAME VARCHAR(50) CHARACTER SET ISO8859_1,
   FIRST_NAME VARCHAR(50) CHARACTER SET ISO8859_1
);
```

So for every field you can use a different character set.

Text blobs

Character sets also apply to text blobs. It is stored in the metadata in the same way and Firebird takes care that the correct character set is also used for text blobs:

```
CREATE TABLE PERSONS (
   PERS_ID INTEGER NOT NULL PRIMARY KEY,
   LAST_NAME VARCHAR(50),
   FIRST_NAME VARCHAR(50),
   RESUME BLOB SUB_TYPE TEXT
);
```

back to top of page

Client character set

When you connect to your Firebird database with the client, you define a client connection character set, a client character set. And this is the character set that your client application uses for the communication with the server.

So what the server does for example, when your client character set is UTF-8 and the field in the database is some ISO field, it transliterates it. All strings are transliterated to/from this client character set, always using Unicode as the large base, so everything is transliterated to Unicode and from there to the specified database character set:

Server CS ↔ Unicode ↔ Client CS

Firebird 1.5 and earlier versions do not have this functionality and return the error: Unable to transliterate between *character sets*.

When you use the same character set on the client as on the column on the server, there is of course, no need to do this transliteration.

How to define the client character set

When connecting to a Firebird database you always need to define the character set that you want to use on the client side.

IBObjects

IB_Connection1.Charset := 'ISO8859_1';

IBX

```
IbDatabase1.Params.Add ('lc_ctype=ISO8859_1');
```

PHP

```
$db = ibase_connect ($Name, $Usr, $Pwd, "ISO8859_1");
```

Declaring character sets in XML and HTML (IANA charset definitions)

Declaring character sets in XML

Every XML document or external parsed entity or external DTD must begin with an XML or text declaration like this:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

In the encoding attribute, you must declare the character set you will use for the rest of the document.

You should use the IANA/MIME-Code from Character Set Overview.

Declaring character sets in HTML

In the head of an HTML document you should declare the character set you use for the document:

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
 ...
```

```
</head>
```

Without this declaration (and, by the way, without an additional DOCTYPE declaration), the W3C Validator will not be able to validate your HTML document.

IANA character set definitions

The Internet Assigned Numbers Authority IANA maintains a list of character sets and codes for them. This list is:

IANA-CHARSETS Official Names for Character Sets, https://www.iana.org/assignments/character-sets

back to top of page

Collations

Collations define the sort ordering for things like ORDER BY. The collations also define the casing rules for things like UPPER() and LOWER(). So you can use:

```
SELECT *
FROM …
WHERE UPPER (NAME COLLATE DE_DE) = :SEARCHNAME
ORDER BY LASTNAME COLLATE FR_CA
```

In this case, this will be treated with a DE German (as opposed to Austrian or Swiss German) collation, and the last name is to be treated with a Canadian French collation.

Every character set in Unicode defines the collations that are possible. These are built into Firebird. With every version there are new collations being developed, and these are published in the Firebird Release Notes (e.g. Firebird 2.1, Firebird 2.0).

Specifying a collation for a column

When you define a column you can also define the collation that you want to use, so you don't need to define the collation when you do your ORDER BY. Starting with Firebird 2.5 there will be a default collation for the default character set.

E.g.

```
 CREATE TABLE PERSONS (
    PERS_ID INTEGER NOT NULL PRIMARY KEY,
    LAST_NAME VARCHAR(50) COLLATE DE_DE,
    FIRST_NAME VARCHAR(50) COLLATE DE_DE,
 ) ;
```

There is also a collation option for numerals (only in Unicode collation) in Firebird 2.5:

```
NUMERIC-SORT={0 | 1}
```

The default, 0, sorts numerals in alphabetical order:

```
1
10
100
2
20
```

The parameter, 1, sorts numerals in numerical order:

```
1
2
10
20
100
```

back to top of page

UPPER(), LOWER()

With Firebird 1.0 + 1.5 UPPER() only works correctly if there is a collation defined for the parameter field. Without a specified collation, there is no uppercasing of letters outside the a…z range.

Since Firebird 2 the awareness of Unicode has increased, so in Firebird 2.x UPPER() will return uppercased characters for all characters, no collation required. Although without a specified collation you still don't know which language collation is applied. Firebird 2 also introduced the LOWER() function.

back to top of page

Searches

Case-insensitive searches

WHERE LIKE, WHERE STARTING WITH, WHERE

In this case a collation should be defined so that it works correctly. The collation can be defined directly in the WHERE clause. Examples:

```
SELECT * FROM PERSONS
WHERE UPPER (LAST_NAME) LIKE ‘’
```

```
SELECT * FROM PERSONS
```

```
WHERE UPPER (LAST_NAME) STARTING WITH  ‘STARK’
```

```
SELECT * FROM PERSONS
WHERE UPPER (LAST_NAME COLLATE DE_DE) LIKE ‘%Ä%’
```

Indexed case-insensitive searches

An important thing is, if you need to do quick searches; case-insensitive searches on indexed fields. Up to and including Firebird 1.5 it was common practise to create a shadow field. So I have the field NAME and I have the shadow field NAME_UPPER with the same type and collation:

```
CREATE TABLE PERSONS (
  NAME  VARCHAR (50) COLLATE DE_DE,
  NAME_UPPER  VARCHAR (50) COLLATE DE_DE) ;
```

Then create an index on the shadow field:

```
CREATE INDEX IDX_PERSON_NAME ON PERSONS (NAME_UPPER);
```

and create a trigger that ensures that this shadow field is filled with the upper case value of the NAME field:

```
CREATE OR REPLACE TRIGGER BIU_PERSONS FOR PERSONS
BEFORE INSERT OR UPDATE AS
BEGIN
  NEW.NAME_UPPER = UPPER (NEW.NAME);
END;
```

When you need to search, then the search is performed on the shadow field, NAME_UPPER:

```
SELECT * FROM PERSONS
WHERE NAME_UPPER = ‘LÖW’
```

Starting with Firebird 2.0 there are the new expression indices, so neither the shadow field nor the trigger is no longer needed.

```
CREATE TABLE PERSONS (
  NAME VARCHAR (50) COLLATE DE_DE,
  CITY VARCHAR (50)) ;
```

Just create the index on a computed field, in this case an UPPER operation:

```
CREATE INDEX IDX_PERSON_NAME ON PERSONS
  COMPUTED BY (UPPER (NAME));
```

When a select, a search, is carried out Firebird can match this and use the index created.

```
SELECT * FROM PERSONS
```

```
WHERE UPPER (NAME) = ‘FIREBIRD’
```

It is also possible to use collations in this and case-insensitive searches:

```
CREATE INDEX IDX_PERSON_NAME ON PERSONS
  COMPUTED BY (UPPER (CITY COLLATE DE_DE));

SELECT * FROM PERSONS
WHERE UPPER (CITY COLLATE DE_DE)) = ‘MÜNCHEN’
```

back to top of page

Firebird & Unicode

When you define a Firebird database to use Unicode everything is stored internally as UTF8. There is an old character set, UNICODE_FSS, which is an old implementation of UTF8 that accepts malformed strings, so there are rules for UTF8 which UNICODE_FSS does not enforce, and there are other issues with the maximum string length. UTF8 can use up to 4 consecutive bytes to store a complete Unicode code point, and UNICODE_FSS uses only a maximum of 3 bytes which is incorrect.

Starting with Firebird 2.0 UTF8 is the correct implementation. Unicode is used to transliterate between character sets. The Unicode collation is now implemented for comparisons and casings.

Unicode collations for UTF8

The two standard collations for UTF8 are UCS_BASIC and UNICODE.

- UCS_BASIC – sorts in Unicode code point order.
- UNICODE – uses the Unicode collation algorithm. This algorithm is defined in the Unicode standard.

```
SELECT * FROM T ORDER BY C COLLATE UCS_BASIC;
```

A B a b á

SELECT * FROM T ORDER BY C COLLATE UNICODE;

a A á b B

back to top of page

Special character sets in Firebird

There are a few special character sets in Firebird:

- **NONE:** No character set applied. With this character set setting, Firebird is unable to perform conversion operations like UPPER() correctly on anything other than the standard 26 Latin letters.

- **OCTETS:** same as NONE. Cannot be used as client connection character set. Space character is #x00.
- **ASCII:** US-ASCII (English only).
- **UNICODE_FSS:** an old version of UTF8 that accepts malformed strings and does not enforce correct maximum string length. Replaced by UTF8 in Firebird 2.0.

UNICODE_FSS is still used for system tables in Firebird. This is apparently for legacy reasons and will probably be changed in Firebird 3 so that Firebird also uses UTF8 for the system tables.

So which character set should I use?

We would recommend that you don't use DOS, dBase and Paradox for anything other than support of legacy applications. WINxxxx code pages are extensions of the corresponding ISOxxxx implementations, but may lead to problems on non-Windows systems. So if you have a Linux or Mac system you have to transliterate everything on the client. On the other side the ISOxxxx character sets are missing a few characters of the WINxxxx character sets (e.g. typographic dash signs), so you need to be prepared to handle that.

If you expect to store various languages, go for Unicode UTF8. The support for UTF8 on the client side is becoming better with Delphi 2009 upwards, so this will be the future. Using UTF8 you are prepared for all future eventualities, should your application become so successful that you are suddenly required to export it.

Please refer to Convert your Firebird applications to Unicode, for our Database Technology article concerning the conversion of existing applications to Unicode.