IBM i Geographical and National Language Constructs Overview

Mandy Shaw (mandy.shaw@iperimeter.co.uk), August 2012.

Why should I care?

A lot of this stuff needs to be understood even if all your users are in one country and you are using only one national language on your system.

I personally care because I come up against this all the time. UK systems suffer worse than most ('divided by a common language'), especially if they want to support the Euro symbol.

But practically everybody needs to exchange data between IBM i (EBCDIC encoding) and a PC or a Unix or Linux system (ASCII encoding); everyone, especially everyone outside the US, needs to watch their software vendors' use of these constructs; and unnecessary transcoding can affect performance in non-obvious ways.

So please do read on!

Fundamental principles

National language constructs

An individual system often needs to support multiple national languages at the same time, displaying information to each reader in their appropriate language.

Geographical constructs

These include date format (DMY or MDY), decimal point character, default currency symbol, default time zone/daylight saving handling, 12 hour versus 24 hour clock, etc.

A *keyboard type* describes the values that can be entered using each of its keys in each possible mode (CTRL, SHIFT, ALT, etc.)

Many applications also need to know your geographical region so that they can give you location-sensitive content.

Relevant open systems constructs

A *locale* is a C and Unix/Linux construct that defines a set of geographical and national language attributes.

A Java virtual machine also has a set of geographical/national language properties (e.g. **java.region**).

Character sets

A *character set* (formally, in IBM land, a *graphic character set*) is a defined, fixed set of digits, alphabetic characters (lower and upper case), punctuation marks, diacritics such as é, and special characters such as @, %, ~.

A character set may or may not relate to the Latin alphabet.

A particular character may be present in one character set but not in another.

from http://www.linotype.com

Traditional printer fonts provide the oldest and most obvious examples of fixed character sets: if a character wasn't provided by the chosen metal font, the compositor couldn't use it.

But the principle is still exactly the same: if a specific character is absent from your chosen printer font, you can't print it, and if your green screen display (or PC emulation thereof) doesn't know about the Euro symbol, you can't display it.

IBM graphic character set 695	
SC040000	Cent Sign
SC050000	Yen Sign
SC200000	Euro symbol

IBM numbers the character sets it uses in its products: an example fragment is to the left. See http://www-01.ibm.com/software/globalization/cs/cs00695.html for the rest of character set 695. Be aware (we'll need this fact in the next section) that the only difference between character sets **695** and **697** is that 695 has the Euro symbol **€**, while 697 has the 'International Currency Symbol' (ICS) ¤ instead.

Unicode is a unifying mechanism that effectively provides a universal character set.

The character set is completely independent of how it may be *encoded* (for use and/or for storage) in an IT system. This encoding is done via *code pages*, *encoding schemes*, and *coded character set identifiers* (*CCSIDs*). See the next section for details.

How the data is encoded

There is lots of good information on this at http://www-01.ibm.com/software/globalization.

A CCSID (formally, the combination of a code page and an encoding scheme) describes how characters are encoded in bits and bytes. It is designed to go with a particular graphic character set: see right.

Character set 695 principal code pages:	01140 01141 01142 01143 01144 01145 01146 01147 01148 01149

A CCSID will use the *single byte* (*SBCS*) or *double byte* (*DBCS*) encoding scheme, dependent on the number of entries (*code points*) in the code page, which is obviously in turn dependent on the number of characters in the character set.

CCSIDs may be specified in a hierarchy (used, for example, for Unicode), allowing some parts of the character set to be managed via SBCS and others via DBCS. See http://en.wikipedia.org/wiki/CCSID for some good examples.

When discussing these matters on IBM i we frequently treat code pages and CCSIDs as synonymous, but be careful unless absolutely everything is SBCS.

1146	37
9F=€	9F=¤
4A=$	4A=¢
5B=£	5B=$
B0=¢	B0=^
B1=[	B1=£
BA=^	BA=[

The terms *ASCII* and *EBCDIC* refer to classes of code pages that follow traditional non-IBM (space=hex 20) or IBM (space=hex 40) approaches, respectively:

- US EBCDIC (used with character set 697): 37
- UK EBCDIC (used with character set 697): 285
- UK EBCDIC with Euro symbol (used with character set 695): 1146
- Some ASCII code pages: 819, 850, 858, 1252 (see later for details)

Code page 1146 allows for the Euro symbol where code page 285 allows for the ICS; they are otherwise identical.

It will be seen that code pages 285 and 37 (and lots of others) are different encodings of exactly the same set of characters. Should you be asking why they have to be different (and see later for some consequent practical irritations), I don't have an answer for you.

Transcoding

Transcoding is the process of converting a data stream between code pages.

It uses CPU cycles (even when transcoding between EBCDIC code pages). This has been the root cause of some of the more mysterious performance problems I've encountered on IBM i.

Where the two character sets are not the same, the approach taken to missing target characters will vary: leave the character's representation unchanged, use a 'best match', or (worst case) refuse to transcode.

Relevant artefacts on IBM i

Lots of information here: http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/topic/nls/rbagsmst.pdf

Devices and character sets

Each display or printer device (*DEVD) has a character identifier (CHRID), which has two parts: a graphic character set number, identifying the specific character set with which the device can work, and a CCSID, which indicates the byte(s) that the device expects to receive and send for each character.

Examples: (697 285) for UK without Euro, (695 1146) for UK with Euro.

Each display device has a keyboard type: USB, UKB, UKE (the last of these supporting the Euro symbol).

Languages, regions, and locales

A primary language is always present, plus secondary languages as required: 2924=English, 2928=French, etc. 2924 is English in general, and is assumed by IBM i to mean US English.

The default geographical/national language system values are automatically set to match the primary language.

Thus the first thing you have to do on a brand new UK IBM i partition with primary language=2924 is to correct all these system values, taking into account whether or not you need to handle the Euro symbol – see diagram.

QCHRID	QDECFMT	QKBDTYPE	QCURSYM	QDATSEP
00697 00037 to 00697 00285 or 00695 01146	Blank (no change)	USB to UKB or UKE	Dollar ($) to Pound (£)	Slash (/) (no change)

QDATFMT	QTIMSEP	QCCSID	QCNTRYID	QLOCALE	QLANGID
MDY to DMY	Colon (:) (no change)	00037 to 00285 or 01146	US to GB	EN_US to EN_GB	ENU to ENG

All these values are defaults and can be overridden:

- CHRID, KBDTYPE – on the device description;
- CCSID, CNTRYID, LOCALE, LANGID – on the user profile, further overrideable at job level;
- Other values – via edit codes/edit words on DDS-based display and printer files.

The CNTRYID, LANGID and LOCALE settings are then available to applications requiring geographical/national language context information. For example, the java.region Java property is set automatically to reflect the CNTRYID that applies when the Java Virtual Machine is started.

CCSIDs and data storage

Every data store (IFS file, physical file, data area, etc.) has a CCSID attribute which describes how its data is stored.

This CCSID may be set to 65535, which means *don't transcode on storage or on retrieval* (i.e. treat as binary data). (See my recent blog posts on ODBC and JDBC for additional comments on this.)

Database tables created with DDS will by default use CCSID 65535.

Database tables created with SQL can even have specific CCSIDs set *at the column (field) level* on their alphanumeric fields.

Transcoding

The way IBM i handles absent target characters varies according to the character sets involved and their level of mismatch.

The difference between 285 and 1146 being just the interpretation of a single character, transcoding between the two is fully supported: the data does not change. Thus a 285 hex 9F (ICS) becomes a 1146 hex 9F (Euro symbol), and vice versa.

EBCDIC to ASCII

When the hex 9F is transcoded from 285 to the PC ASCII code page 1252 (*PCASCII on CPYTOSTMF etc.), which supports both the ICS and the Euro symbol, the target character will be the ICS (hex 4F); but when the (identically stored) data is transcoded from 1146 to 1252, the target character will be the Euro symbol (hex 80).

If you transcode to the ASCII code page 850 (*STDASCII), you will get a target character of hex CF whether you start from 285 or from 1146, because 850 does not support the Euro symbol.

```
Originally entered character (through UKE keyboard): €
Stored in EBCDIC (285 or 1146): hex 9F, €
Transcoded from 285 or 1146 to 819: hex A4, ¤
Transcoded from 285 or 1146 to 850 (*STDASCII): hex CF, ¤
Transcoded from 285 to 858: hex CF, ¤
Transcoded from 1146 to 858: hex D5, €
Transcoded from 285 to 1252 (*PCASCII): hex 4F, ¤
Transcoded from 1146 to 1252 (*PCASCII): hex 80, €
```

Code page 858 is a follow-on from 850, with the only change being a Euro symbol at hex D5. Transcoding to this gives correct results: from 285 it gives the ICS (hex CF), but from 1146 it gives the Euro symbol (hex D5).

Again, if you transcode to the ASCII code page 819 (which remains the default IBM i FTP ASCII conversion target), you will get a target character of hex A4 (ICS) whether you start from 285 or from 1146, because 819 does not support the Euro symbol.

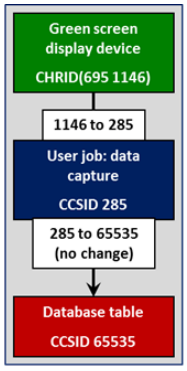
So the moral of the story is … ***Make sure that both code pages support all the characters you want***.

I have found Wikipedia to be a good source re ASCII code pages.

A final warning: do not trust the evidence of your eyes – characters that look the same may be encoded differently, and only one of the encodings may do what you want. Look at the hex values: DSPPFM and press F10 followed by F11, or, for IFS files, use `od -t x <file>` from STRQSH.

And most importantly of all: d***on't forget that the job has a CCSID too***. CPYTOSTMF, for example, transcodes twice, not once. See next section.

Job CCSIDs

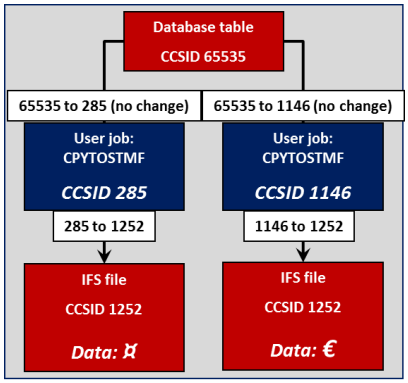


Data gets transcoded on the way in to the job, and then again on the way out.

The job CCSID defaults to the user profile CCSID (but can be overridden via CHGJOB or on SBMJOB).

Objects you create may be affected, e.g. new IFS files pick up the CCSID of the creating job.

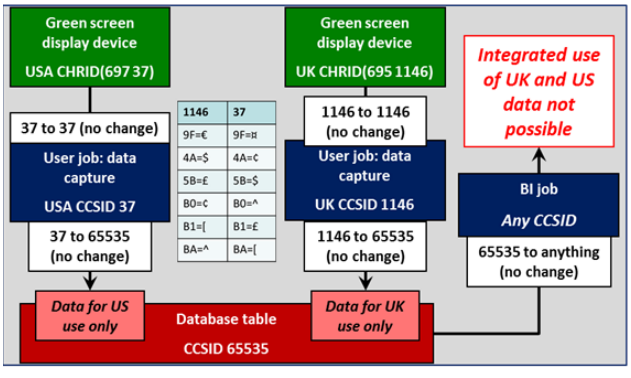
Functioning of applications, and even access to data, may be affected. (Don't use pound signs in your database table names if you want them to work from all job CCSIDs!)



Also, some software vendors enforce particular job CCSIDs – there's probably a reason (though rarely an excuse).

And never forget that any transcode activity that involves change uses CPU cycles.

So the moral here is: ***get your default CCSID right***. This is probably the most important globalisation rule of all.



One final warning. Try to avoid using a database table with CCSID 65535 from different *job* CCSIDs – see left.

This seems like a good idea at the time, until you try to use the table from outside the application, e.g. for business intelligence. You then find that any given row of data is meaningless if you don't know its originating job CCSID.