

RPG and Unicode

Enable RPG programs to Handle
International Data

Barbara Morris

RPG Compiler Development

IBM

IBM i Anywhere
IBM i Everywhere

CCSID concept

First, let's just consider "Data"

What does x'7d' mean?

' apostrophe?

} curly brace?

-7 minus 7?

125 one hundred twenty five?

Yes, all of those. And many more things.

We have to interpret it

By itself, x'7d' doesn't mean anything.

We have to know how to interpret it

' EBCDIC

} UTF-8 and ASCII

-7 1-byte packed decimal

125 1-byte integer

Another interpretation

It could be even be specific to a particular program, where each bit has a different meaning. `x'7d' = b'0111 1110'`

Double occupancy	False
Prepaid	True
Vegetarian	True
Fitness class	True
Buenos Aires side trip	True
Swimming pool access	True
Returning customer	True
VIP	False

Character data

Let's just focus on character data.

' EBCDIC

} ASCII

~~-7 packed decimal~~

~~125 integer~~

~~bit data about a booking~~

But we still have an interpretation problem. EBCDIC vs UTF-8

And in general, it's not just a matter of UTF-8 vs EBCDIC.

Concept: Character set

There are many different character sets

Latin:

а Н Q m Á è ...

Cyrillic:

Д ф ђ Ю Ы Я Г ...

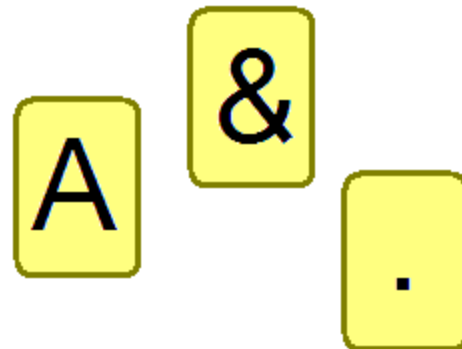
Japanese:

こ よ 日 は で 本 ...

Concept: Character set

Think of a “character set” as the tiles for a game. You get 256 tiles.

- 26 have A-Z
- Another 26 have a-z
- 10 have 0-9
- And there's the accented letters like Á, é, ñ, Ç
- And many characters like this; !@#\$%^&*,©§¶¼½¾



The characters can be ordered in many ways

Let's consider an imaginary character set with just 32 characters:

! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾ A B C D a b c d Á é ñ ç 0 1 2 3 4

There are many many ways to order these characters.

! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾ A B C D a b c d Á é ñ ç 0 1 2 3 4
A B C D a b c d 4 Á é ñ ç 0 1 2 3 ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾
A @ B # C \$ D a b © § ¶ ¼ 4 c d Á é ñ 1 2 3 ! % ^ & * , ½ ¾ ç 0

But they are always the same characters from that set.

The characters can be ordered in many ways

We can give an ID to the orderings

1. ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾ A B C D a b c d Á é ñ ç 0 1 2 3 4
2. A B C D a b c d 4 Á é ñ ç 0 1 2 3 ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾
3. A @ B # C \$ D a b © § ¶ ¼ 4 c d Á é ñ 1 2 3 ! % ^ & * , ½ ¾ ç 0

We have IDs for 3 different orderings for our character set.

Imagine another character set

Here's another imaginary character set with a couple of orderings.

Some of the characters are the same as for the previous set.

4. é ê ë è í î ï ð ß A B C D a b c d Á é ñ ç 0 1 2 3 4 ¿ Đ Ý Þ ® +
5. Á é ñ ç 0 1 2 3 4 ¿ Đ Ý Þ ® + é ê ë è í î ï ð ß A B C D a b c d

We have 2 IDs for our second character set.

Concept: CCSID

1. ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾ A B C D a b c d Á é ñ Ç 0 1 2 3 4
2. A B C D a b c d 4 Á é ñ Ç 0 1 2 3 ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾
3. A @ B # C \$ D a b © § ¶ ¼ 4 c d Á é ñ 1 2 3 ! % ^ & * , ½ ¾ Ç 0
4. é ê ë è í î ï ð ß A B C D a b c d Á é ñ Ç 0 1 2 3 4 ç Đ Ý Þ ® +
5. Á é ñ Ç 0 1 2 3 4 ç Đ Ý Þ ® + é ê ë è í î ï ð ß A B C D a b c d

From just the ID, we can deduce both the character set and the order.

Consider ID "4"

- It is from the second character set
- It is the first ordering from that character set

“4” is the “Coded Character Set ID” or CCSID

Knowing the CCSID lets us interpret the data

1.	!	@	#	\$	%	^	&	*	,	©	§	¶	¼	½	¾	A	B	C	D	a	b	c	d	Á	é	ñ	Ç	0	1	2	3	4	
2.	A	B	C	D	a	b	c	d	4	Á	é	ñ	Ç	0	1	2	3	!	@	#	\$	%	^	&	*	,	©	§	¶	¼	½	¾	
3.	A	@	B	#	C	\$	D	a	b	©	§	¶	¼	4	c	d	Á	é	ñ	1	2	3	!	%	^	&	*	,	½	¾	Ç	0	
4.	é	ê	ë	è	í	î	ï	ì	ß	A	B	C	D	a	b	c	d	Á	é	ñ	Ç	0	1	2	3	4	¿	Đ	Ý	Þ	®	+	
5.	Á	é	ñ	Ç	0	1	2	3	4	¿	Đ	Ý	Þ	®	+	é	ê	ë	è	í	î	ï	ì	ß	A	B	C	D	a	b	c	d	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Let's say the hex value of a byte is x'0B'.

If we know the CCSID, we know the character

CCSID 2:

ñ

CCSID 4:

C

CCSID conversion

CCSID conversion

Let's convert some data from CCSID 2 to CCSID 4, x'011E'

2. A B C D a b c d 4 Á é ñ ç 0 1 2 3 ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		

Step 1. Find out which character matches each hex value in CCSID 2, for our data x'011E'

Hex value in CCSID 2	Character
x'01'	

CCSID conversion

Finding the CCSID 2 characters for our hex data

2. A B C D a b c d 4 Á é ñ ç 0 1 2 3 ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		

Step 1. Find out which character matches each hex value in CCSID 2, for our data x'011E'

Hex value in CCSID 2	Character
x'01'	B

CCSID conversion

Converting to CCSID 4

4. é ê ë è í î ï ð Ñ A B C D a b c d Á é ñ ç 0 1 2 3 4 ı Đ Ý Þ ® +

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F

Step 2. Find the characters in the second character set

Character	Hex value in CCSID 4
B	
½	

CCSID conversion

Converting to CCSID 4

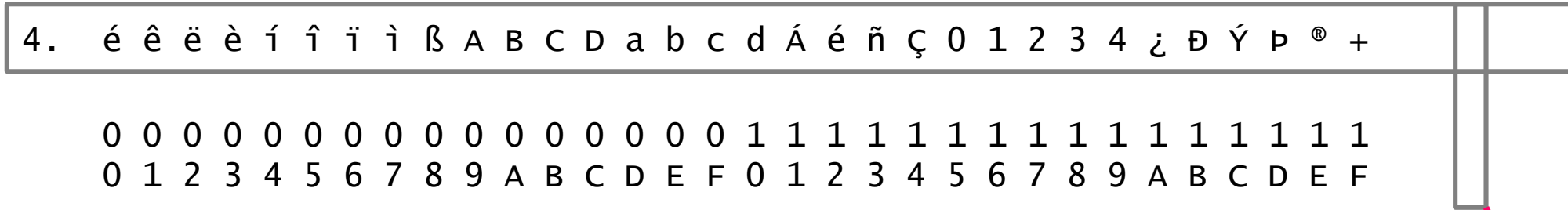
4.	é	ê	ë	è	í	î	ï	ì	ß	A	B	C	D	a	b	c	d	Á	é	ñ	ç	0	1	2	3	4	¿	Đ	Ý	Ɔ	®	+	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Step 2. Find the characters in the second character set

Character	Hex value in CCSID 4
B	x'0A'
½	

CCSID conversion

Converting to CCSID 4



Step 2. Find the characters in the second character set

Character	Hex value in CCSID 4
B	x'0A'
½	

Where does this go?

CCSID conversion

Converting to CCSID 4

4. é ê ë è í î ï ð Ñ A B C D a b c d Á é ñ ç 0 1 2 3 4 ı Đ Ý Þ ® +

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F

Step 2. Find the characters in the second character set

Character	Hex value in CCSID 4
B	x'0A'
½	Does not exist!

CCSID conversion

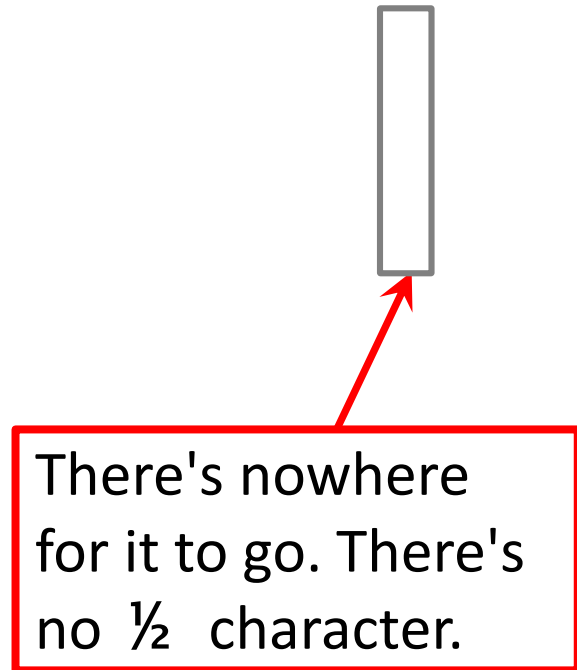
Converting to CCSID 4

4. é ê ë è í î ï ð Ñ A B C D a b c d Á é ñ ç 0 1 2 3 4 ¿ Ð Ý Þ ® +

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F

Step 2. Find the characters in the second character set

Character	Hex value in CCSID 4
B	x'0A'
½	Does not exist!



There's nowhere for it to go. There's no ½ character.

CCSID conversion

We cannot completely convert x'011E' from CCSID 2 to CCSID 4

2. A B C D a b c d 4 Á é ñ ç 0 1 2 3 ! @ # \$ % ^ & * , © § ¶ ¼ ½ ¾
 4. é ê ë è í î ï ð ß A B C D a b c d Á é ñ ç 0 1 2 3 4 ¿ Đ Ý Þ ® +
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F

Hex value in CCSID 2	Character	Hex value in CCSID 4
x'01'	B	x'0A'
X'1E'	½	????

Result: x'B■'

Data has been lost!

Loss of data during CCSID conversion

Each character set has a “replacement character” that represents any character that doesn’t exist in the character set.

When there is no matching character in the target character set, the replacement character is used instead.

If we assume that x'FE' is the replacement character for our CCSID 4, then the result of the previous conversion is x'0AFE'

We converted the value 'B½' to 'B■' where ■ represents the replacement character x'FE'.

Loss of data during CCSID conversion

Let's try to convert the CCSID 4 data back to CCSID 2

Assume the replacement character is also x'FE' in CCSID 2.

Original CCSID 2 data: x'011E' = 'B½'

Converted CCSID 4 data: x'0AFE' = 'B■'

Converted CCSID 2 data: x'01FE' = 'B■'

We cannot get back to the original CCSID 2 data.

Solution: Don't even try to convert to a character set that might not have some characters

Character sets and CCSIDs

Three kinds of character sets

1. Single-byte character set (SBCS)

- One byte per character
- Contains the characters for one type of language
- Examples are Latin (European languages), Cyrillic (Russian)
- Always includes the standard characters A-Z, a-z, 0-9, + = - etc

2. Double-byte character set (DBCS)

- Also called "Graphic"
- Two bytes per character
- Contains the characters for one graphic language
- Examples are Chinese and Japanese

Three kinds of character sets

3. Unicode

- 1-3 bytes per character for UTF-8
- 2-4 bytes per character for UCS-2 and UTF-16
- Contains the characters from all SBCS and DBCS character sets ✓

The Unicode character set will solve our problem. If we have data that might have characters from different languages, using Unicode is the only way to avoid losing data.

In RPG, there are three types of Unicode data:

- UCS-2: data type C with CCSID(13488)
- UTF-16: data type C with CCSID(1200)
- UTF-8: data type A with CCSID(*UTF8) 7.2 +

Several kinds of CCSIDs

Single-byte CCSIDs

- Characters from one SBCS character set

Double-byte CCSIDs

- Characters from one DBCS character set

Mixed-byte CCSIDs

- Characters from one SBCS set and one DBCS set

Unicode CCSIDs

- 1208 is the CCSID for UTF-8: 8 bits (1 byte) is the smallest size
- 1200 is the CCSID for UTF-16: 16 bits (2 bytes) is the smallest size
- 13488 is the CCSID for UCS-2: similar to UTF-16

Hex CCSIDs

- 65535 is the Hex CCSID. Hex data cannot be converted to another CCSID

CCSIDs in RPG

Working with character data in RPG

From now on, I'm going to use the term "string data" to refer to alphanumeric, graphic and UCS-2 data. The term "character" often just means RPG's CHAR (A) data type.

All string data has a CCSID. You can use DSPFFD to find out the CCSID of the data in your files.

You can use the cross reference in your RPG listings to find out the CCSID of your string variables. If alphanumeric data doesn't show a CCSID, then it is the job CCSID.

Working with character data in RPG

RPG defaults:

- Alphanumeric: job CCSID
- Graphic: CCSID is ignored by default
 - Add `CCSID(*GRAPH:ccsid)` to your H spec to support Graphic CCSIDs
- UCS-2: 13488

You can use the CCSID H spec keyword to set different defaults.

For example

```
CCSID(*CHAR:37) CCSID(*UCS2:1200)
```

Working with character data in RPG

You can use the /SET and /RESTORE directives to temporarily set different defaults for your definitions

```
ctl-opt CCSID(*CHAR:37);  
dcl-s company char(20);  
/set ccsid(*char : *JOB RUN)  
    dcl-s city char(20);  
    dcl-s description char(100) ccsid(*utf8);  
/restore ccsid(*char)  
dcl-s library char(10);
```

- "company" has CCSID 37 from the H spec default
- "city" has CCSID *JOB RUN from the /SET default
- "description" has CCSID *UTF8 (1208) from its CCSID keyword
- "library" has CCSID 37 from the H spec default

CCSID conversion in RPG

Implicit CCSID conversion

Normally, RPG automatically does whatever CCSID conversion is needed.

```
dcl-s name varchar(20) ccsid(*jobrun);  
dcl-s temp_name varucs2(20);
```

```
temp_name = name;
```

- "temp_name" has a different CCSID from "name"
- RPG automatically converts the data in "name" from the job CCSID to UCS-2 when assigning to "temp_name"

Explicit CCSID conversion

There are a few scenarios, such as some built-in functions, where RPG does not yet do automatic CCSID conversion

You can explicitly request CCSID conversion using the %CHAR, %UCS2 or %GRAPH built-in functions.

Assume that "description" is defined as UCS-2:

```
p = %scan('? ' : description); // not supported
```

```
p = %scan(%ucs2('? ') : description); // ok
```

Warnings or exceptions for CCSID conversions

We saw that a CCSID conversion may sometimes result in a “substitution” character being placed in the result.

Unicode source data:

The Thai word for “house” is “บ้าน”.

The target is an alphanumeric variable with CCSID 37:

The Thai word for “house” is “■■■”.

- CCSID 37 uses the “Latin” character set, and there are no matching characters for the Thai characters that are in the Unicode variable. Substitution characters are placed in the alphanumeric result.
- The original Thai characters are all converted to the same substitution characters, so their value is lost!

Warnings or exceptions for CCSID conversions

By default, non-error RPG status code 50 is set when the conversion had to use substitution characters.

You have to add code to check whether %status = 50

```
alphaText = unicodeText;  
if %status() = 50;  
    ... there was loss of data
```

Two problems:

- It's awkward to check for status code 50 after every statement with a CCSID conversion
- It's not always easy to tell which statements have CCSID conversions

Get an exception when substitution occurs

6.1 +

CCSIDCVT(*EXCP)

Code H spec keyword CCSIDCVT(*EXCP) to get an exception when a CCSID conversion results in a substitution character.

- Status code 00452

If you are on 6.1 and 7.1 and get this function through a PTF, you will need to add messages RNX0452 and RNQ0452 to your message file. The cover letter of the PTF for the RPG runtime has CLP code for adding the messages. See the RPG Cafe for PTF details.

Get a list of CCSID conversions

CCSIDCVT(*LIST)

Code H spec keyword `CCSIDCVT(*LIST)` to get a list of all the CCSID conversions in the module.

For each conversion, it shows

- The source statements using that conversion
- Whether the conversion might result in substitution characters

If you want both options, code `CCSIDCVT(*EXCP:*LIST)` or `CCSIDCVT(*LIST:*EXCP)`

Sample CCSIDCVT summary

C C S I D C o n v e r s i o n s						
	From CCSID	To CCSID	References			
RNF7361	834	*JOB RUN	15	25		
RNF7357	1200	*JOB RUN	27	921	1073	
	*JOB RUN	1200	28	12	321	426
			552	631		
RNF7359	835	834	41	302	302	
RNF7360	*JOB RUN	834	242	304	305	
* * * *	E N D	O F	C C S I D	C O N V E R S I O N S		* * * *

- RNF7357 Conversion from UCS-2 to Alpha might not convert all data.
- RNF7358 Conversion from UCS-2 to DBCS might not convert all data.
- RNF7359 Conversion from DBCS to DBCS might not convert all data.
- RNF7360 Conversion from Alpha to DBCS might not convert all data.
- RNF7361 Conversion from DBCS to Alpha might not convert all data.

How to use the CCSIDCVT summary

You can use this information for two purposes:

- **Improve performance:** Reduce the number of conversions by changing the data types of some of your variables.
- **Improve reliability:** Eliminate the conversions that have the potential to result in substitution characters.

For example

- You convert UCS-2 variable NAME to alphanumeric variable TEMPNAME
- TEMPNAME is later converted to UCS-2 as part of UCS-2 FULL_NAME
- Consider changing the type of TEMPNAME to UCS-2, to avoid the potential data loss.

Why bother with CCSIDs?

Why bother about the CCSID of alpha data?

Starting in 7.2, you can code the CCSID keyword for character fields:

- All EBCDIC CCSIDs
- ASCII CCSIDs
- The Unicode CCSID UTF-8 (1208, or *UTF8)

But why bother? Isn't the job CCSID good enough?

The job CCSID isn't always the right CCSID

Assume that the getData procedure returns UTF-8 data

Prior to 7.2, you could call some API to convert the data to UCS-2:

```
dcl-s stringA char(10000);  
dcl-s stringC varucs2(10000);  
dcl-pr getData varchar(10000); ...
```

```
stringA = getData ();  
stringC = convert(stringA: %len(stringA): 1208: 13488);
```

But remember not to use the data for ordinary RPG statements because RPG thinks the data is in the job CCSID.

```
if stringA <> *blanks; // BUG!  
    stringC = convert...  
    ...
```

Defining alpha data with a CCSID

7.2+

In 7.2, you can say that the data is UTF-8.

```
dcl-s stringA char(10000) ccsid(*utf8);  
dcl-pr getData varchar(10000) ccsid(*utf8); ...  
  
stringA = getData ();
```

Now you can use the returned value in ordinary RPG statements because RPG knows that it is UTF-8 data.

```
if stringA <> *blanks; // OK!  
    stringC = convert...  
    ...
```

CCSID of externally-described subfields

7.2+

UCS-2 and graphic subfields always get the CCSID of the field in the file.

By default, alpha subfields are defined with the job CCSID for RPG.

If you want the alpha subfields to be defined with the same CCSID as the matching fields in the file, code `CCSID(*EXACT)` for the data structure.

```
dcl-ds ds1 likerec(rec) ccsid(*exact);
```

```
dcl-ds ds2 extname('MYFILE') ccsid(*exact);
```


CCSIDs and RPG I/O

Using CCSID(*EXACT) data structures for I/O

When you specify a data structure in the result field of your I/O operation, the data is copied directly between the I/O buffer and your data structure

If there is invalid data in the record of the file, this allows you to delay the discovery of the problem

```
read rec ds;  
monitor;  
    salary = ds.salary;  
on-error;  
    ... problem with the "salary" field
```

Using CCSID(*EXACT) files for I/O

But what happens if the data structure was defined with CCSID(*EXACT)?
The subfields might have a different CCSID.

The subfields are copied directly between the buffer and the data structure **unless** CCSID conversion is required

Field	Buffer CCSID	Handled ...	DS CCSID
ID	-	Move bytes	-
NAME	Job	Converted	37
SALARY	-	Move bytes	-
STARTDATE	-		-
ADDRESS	Job	Converted	37
DESC	Job	Converted	1208
BONUS	-	Move bytes	-

Avoiding CCSID conversions for database files

7.2+

By default, for a database file

- When you read a record, database converts the alphanumeric data from the field CCSID to the job CCSID
- When you write or update a record, database converts the alphanumeric data from the job CCSID to the field CCSID

Use `DATA(*NOCVT)` for a file to open the file so these conversions do not happen at the database level. Any CCSID conversions will be performed in the RPG program if necessary.

Use H spec `OPENOPT(*NOCVTDATA)` to default this behaviour for all database files.

Using CCSID(*EXACT) with DATA(*NOCVT)

If the data structure is defined with CCSID(*EXACT) and the file is defined with DATA(*NOCVT), the data in the buffer and the data structure will have the same CCSID

Field	Buffer CCSID	Handled ...	DS CCSID
ID	-	Move bytes	-
NAME	37		37
SALARY	-		-
STARTDATE	-		-
ADDRESS	37		37
DESC	1208		1208
BONUS	-		-

Using DATA(*NOCVT) without data structures

If you don't specify a data structure for your I/O operation, the data is always moved between the fields and the buffer individually.

CCSID conversion is automatically done as part of this.


RPG and I/O with CCSID 65535

If the job CCSID is 65535

- By default, a database file is opened with the expectation that the alpha data is converted to the job CCSID
- But no CCSID conversion is done by database for 65535
- The data in the buffer has the same CCSID as the data in the file.

By default, RPG thinks that the data is in the default CCSID of the job.
The default CCSID depends on the Language ID and the Country ID.

Language identifier	:	ENU
Country or region identifier	:	US
Coded character set identifier	:	65535
Default coded character set identifier	:	37



RPG and I/O with job CCSID 65535

If the default job CCSID is the same as the CCSID of all the alpha fields in the file, everything is fine

If all the fields in the file have an EBCDIC CCSID, and all the characters in the data have the same hex values as the characters in the job CCSID, everything is fine

If some fields in the file are UTF-8, RPG will not handle the data correctly (by default)

**Using UTF-8 data with RPG is unusable
if the job CCSID might be 65535
(by default)**

The DATA keyword with CCSID 65535

If DATA(*NOCVT) is coded for the file, the data in the I/O buffer will always have the same CCSID as the fields in the file.

If DATA(*CVT) is coded, then the data in the I/O buffer might have the job CCSID or the CCSID of the fields in the file, depending on the job CCSID at runtime.

The RPG compiler will generate code to handle both situations, so your program will work with a job CCSID of either 65535 or another CCSID.

RPG and I/O with CCSID 65535

To make RPG handle database files correctly when your job CCSID is 65535, do any of the following

- Code CCSID(*EXACT) in the H spec
- Code the DATA keyword for the file, either DATA(*CVT) or DATA(*NOCVT)
- Define the CCSID for all the alpha fields in the RPG program (CCSID keyword on externally-described DS, CCSID keyword on fields)

Doing any one of these will cause RPG to handle the I/O buffers correctly.

RPG and literals

RPG and literals

In the past, RPG did not always handle CCSIDs correctly (and still does not, by default)

Literals are handled as though they have the job CCSID, but they are actually saved in the source file CCSID

```
dcl-pi *n;  
  parm char(1) const;  
end-pi;  
  
if parm = '!';  
  dsply ('parm is exclamation mark!');  
else;  
  dsply ('parm is not exclamation mark!');  
endif;  
  
return;
```

RPG and literals

Here is the hex value of the "if" statement in a source file with CCSID(37):

```
if parm = '!';           ! is x'5A'  
88498994747575  
96071940E0DADE
```

In a source file with CCSID(500):

```
if parm = '|';           ! is x'4F'  
88498994747475  
96071940E0DFDE
```

(My emulator is setup with CCSID 37, so DSPPFM is showing the data as if it were CCSID 37 data)

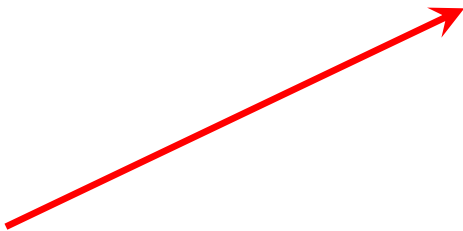
RPG and literals

I compile the two programs. The ! character is saved as x'5A' in TEST37, and as x'4F' in TEST500.

I call the programs:

```
> call bmorris/test37 '!'  
  DSPLY parm is exclamation mark!  
> call bmorris/test500 '!'  
  DSPLY parm is not exclamation mark|
```

Two problems with test500:

- I did pass an exclamation mark!
 - The DSPLY shows | instead of !
- 


To have RPG handle literals correctly

Add CCSID(*EXACT) to the H spec.

```
ctl-opt ccsid(*exact);  
  
dcl-pi *n;  
  parm char(1) const;  
end-pi;  
...
```

I call the new programs. They both work correctly. The TEST500 program understands that the literal must be converted to the job CCSID.

```
> call bmorris/test37 '!'  
  DSPLY parm is exclamation mark!  
> call bmorris/test500 '!'  
  DSPLY parm is exclamation mark!
```



UTF8 in RPG

UTF-8 data

UTF-8 data can have characters which are 1, 2, or 3 bytes long.

For example, the Ø character is two bytes in UTF-8.

```
dcl-s a37    varchar(4) ccsid(37)    inz('Ø');  
dcl-s a1208 varchar(4) ccsid(*utf8) inz('Ø');  
return;
```

In debug (the first two bytes are the varying length)

```
> EVAL a37:x  
    00000    000180    1 byte, x'80'  
> EVAL a1208:x  
    00000    0002c398    2 bytes, x'c398'
```

UTF-8 data

Since some characters may be up to 3 bytes long, UTF-8 data is longer than EBCDIC alphanumeric data.

If you define UTF-8 fields, be sure to define them long enough

UTF-8 data

RPG does not consider the length of each character in string operations

- %len
- substring
- scan
- truncation on assignment

These are all handled by RPG on a byte basis, not a character basis
(This issue has always existed for mixed SBCS/DBCS data)

Please follow (and vote for) IBM Idea 2128 ("*Make %subst and %len operate on characters for UTF-8 and UTF-16*") for information on how RPG may deal with this issue in the future.

<https://ibm-power-systems.ideas.ibm.com/ideas/IBMI-I-2128>

Recommendations

Recommendations

- Code `CCSID(*EXACT)` so that RPG will handle CCSIDs correctly
- Code `CCSIDCVT(*EXCP)` so that conversion failures will cause an exception
- Code `CCSIDCVT(*LIST)`, possibly temporarily, and study the CCSID conversions that are happening in your module
- Be sure to define UTF-8 fields long enough

Useful links

Where to find PTF'd RPG enhancements

IBM i Anywhere
IBM i Everywhere

Regularly check the “welcome” page in the RPG Cafe wiki

http://ibm.biz/rpg_cafe

There is a section for “Enhancements delivered through PTFs”, and the “Announcement” section at the top will have information about the most recent enhancements

Regularly check the “What’s New Since ...” section in the ILE RPG Reference

Where to request RPG enhancements

RPG is part of the RFE (Request for Enhancements) process

- You can submit requirements
- You can vote on requirements that others have requested
 - Votes aren't the only consideration when IBM decides which RFEs to work on, but they are important
 - Be careful not to vote for too many RFEs. Just vote on the ones that you need the most
- Add comments to existing RFEs, to clarify the request, or add additional reasons why the enhancement is important

Here is a link to the current RFEs for the RPG compiler: http://ibm.biz/rpg_rfe

They are under

Brand: Servers and Systems Software

Product family: Power Systems

Product: IBM i

Component: Languages – RPG

Thank you

IBM i Anywhere
IBM i Everywhere