# Base Language - Feature #4761

## I18N phase 3

07/15/2020 12:08 PM - Greg Shah

| | | | |
|---|---|---|---|
| **Status:** | New | **Start date:** | |
| **Priority:** | Normal | **Due date:** | |
| **Assignee:** | | **% Done:** | 0% |
| **Category:** | | **Estimated time:** | 0.00 hour |
| **Target version:** | | | |
| **billable:** | No | **vendor_id:** | GCD |

**Description**

**Related issues:**

| | |
|---|---|
| Related to Base Language - Feature #3753: I18N additions | **Closed** |
| Related to Base Language - Feature #6451: I18N phase 4 | **New** |
| Related to Base Language - Feature #6431: implement support to allow CPINTERN... | **WIP** |

## History

**#1 - 07/15/2020 03:06 PM - Greg Shah**

#4763 replace any hard coded dependencies on charset processing
#4762 localize system error messages
#4768 finish COPY-LOB (some of these items are I18N related)
#4378 properly handle clob/lonchar assignment, especially the implicit codepage conversion
#4764 make modifications to FWD so that all I18N testcases work properly
#5085 uppercasing and string comparisons are incorrect for some character sets
#4766 fix CHR and ASC
#4765 MAP/NO-MAP support
#3817 CURRENT-LANGUAGE/translation manager conversion/resource bundle support
#3871 handle codepage translation at database import
#4389 escape support for supplementary unicode characters
#5276 resolve issues with multi-byte character processing
#5491 custom locale/collation implementations
#5538 fix readByte() for DirStream, ProcessStream, and others
#5570 properly support keyboard input for multi-byte charsets
#6145 invalid characters in query

Future Work (tasks deferred for later, not needed now):

#6268 Characters encoding issues with UTF-8 for emoji and confusable symbols.

Some customer-specific discussions which require the above items can be seen in #4044 and #4379.

**#2 - 07/15/2020 03:07 PM - Greg Shah**

*- Related to Feature #2135: implement COPY-LOB language statement added*

**#3 - 07/15/2020 03:07 PM - Greg Shah**

*- Related to deleted (Feature #2135: implement COPY-LOB language statement)*

**#4 - 07/15/2020 03:10 PM - Greg Shah**

*- Related to Feature #3753: I18N additions added*

**#5 - 10/13/2020 10:43 AM - Greg Shah**

From Marian in regard to the work on #4384:

> Some of those are still on hold mostly because of codepage convert issues - all methods that takes a codepage as input fails now because of the copy-lob or codepage-convert functionality in FWD which doesn't seem to render the same results as in 4GL.

I think it is time to fix these items. Can you please open a new task for this and document the problems?

> Also in String (OpenEdge.Core) they seems to use an utf-8 codepage for storage and then depending on the codepage specified for the String instance the value gets converted to that codepage as needed - this also break all of our tests where a different encoding than UTF-8 is set :(

At conversion time, we need to define the source file (used for procedures, includes and .df files) encoding using the source-charset configuration value. In the most recent FWD builds, this is defaulted to ISO-8859-1/LATIN1/Windows 1252 when the application is converted as GUI (OPSYS is WIN32) and it defaults to UTF-8 for UNIX/Linux. The idea here is that we read in the files with the correct legacy encoding and then all of our output (mostly .java but also some XML and SQL scripts) is written as UTF-8. Using that approach, all possible characters can be handled nicely AND it generally is quite easy to compile/develop Java code using that model.

At database import time, we currently must match up the original database encoding with the .d export encoding and that must match up with the target database (PostgreSQL/H2/SQL Server...) encoding. We have #3871 to properly handle the translation of .d files in one encoding into a different encoding at the target database. This is intended to easily allow customers to take their LATIN1 data export and create a UTF-8 PostgreSQL instance as a result. Of course, any valid encoding conversion would also be supported.

At runtime we have our most complex set of issues. And although the JVM can run with a default encoding that is not UTF-8, it is most common to have UTF-8 as the encoding no matter what the local language is in use. We may in fact have many dependencies on this, after 15 years of development. Our approach is intended to work like this:

- The JVM encoding is UTF-8, which means all internal text processing is handled in Unicode. This is similar to setting CPINTERAL to UTF-8.
- Any input or output operation can be using a different encoding. This will match the 4GL approach of having CPSTREAM to define I/O encoding, CPTERM for ChUI terminals and so forth. When the source/target encodings are different, we should be handling the codepage conversion as part of the input or output processing.
- In practical terms, since we always run with UTF-8 as CPINTERNAL, then the other CP* values are set differently, then there should be an implicit conversion.
- By default, all the CP* values are UTF-8 but we can set these in the directory.xml to match the original application's environment.
- Any explicit codepage conversion should work as expected, to the degree that we have finished implementation.

Any deviations from this should be documented as issues to resolve.

**#6 - 03/22/2021 08:59 AM - Ovidiu Maxiniuc**

Although the CP conversion seems to work in FWD, it is actually incorrect. The appearance of being fine is because only Latin-1 is normally used in programming code. And for these characters, the code will match. Also there is no BOM for them.

See I18nOps.codePageConvertWorker(String text, String srcCP, String tarCP). What is wrong is that the conversion from one CP to another is done using an intermediary byte array. In this approach, the source extract its bytes using its srcCP. Then the target tries to extract some value from those bytes, using its own tarCP. It's like translating from English to Greek just by replacing a with alpha, b with beta and so on. This is more visible when the size of the character is different, like UTF-8 and utf-16: extracting the string bytes in UTF-16 (2 bytes/char) then attempting to extract something in UTF-8 (1 byte/char, for latin-1 part) will result in a double-length target. And there is is the BOM, too.

The problem here is that FWD always keep the character/Strings internally in Java native UTF-16 (as a char array). When a special encoding is needed, the getBytes() will provide it. However, the resulting bytes array only makes sense when read in that that specific CP. The advantage: all strings are compatible, and any operation does not need CP conversion.

The 4GL, OTOH, seems to me to store the strings internally using the specified CP. The cpinternal looks more like a default encoding for LONGCHAR variables. Apparently also used by all character variables. The fields use the CP configured at table/db creation. To understand the stored bytes of a text variable you need to also know the CP it was encoded with, otherwise it is just meaningless. To concatenate such strings, they must be first converted to a common CP. The advantage is that it is easier for IO operations which are usually in the respective CP.

As conclusion, the CP conversion in FWD will just leave the internal data unchanged. We just need to verify if there are inconvertible characters and issue errors as they would occur if a true CP change takes place. The only thing we need to take care is when the value is serialized or converted to binary form in which cases the so the configured CP is used instead of the internal one.

**#7 - 03/22/2021 09:36 AM - Greg Shah**

> The problem here is that FWD always keep the character/Strings internally in Java native UTF-16 (as a char array).

I would like to maintain this core approach, if we can do so without compatibility issues.  The huge advantage is that the vast majority of the processing in the JVM can be unchanged.  We make the following assumptions:

1. **Core Design** - All input from or output to sources/targets that are in a different codepage must be properly handled between UTF-16 and that I/O codepage.
2. **Internal Collation** - All internal collation operations (e.g. comparison operators GT, GTE, LT, LTE and the COMPARE() function) are done as if CPINTERNAL is UTF-16 for String instances.
3. **CPINTERNAL**
   - The CPINTERNAL cannot be set to any other value.
   - All users of the same server have the same CPINTERNAL.

Considering that all possible characters can be encoded in UTF-16, I think the core design may be OK if we can get the rest right.  Issues that we need to handle/consider:

- Core Design
  - We have known bugs to address.
  - One thing I don't think we've considered yet is handling supplemental characters (characters that take 2 UTF-16 char instances to represent).
- Internal Collation
  - We could honor a different CPINTERNAL value for the 4GL comparison operators and COMPARE().
  - This could also be handled in any other places of the runtime that do string collation.  We'd have to consider where these may be.
  - Is there any conflict here with having the JVM default character encoding as UTF-8?  Does this just affect I/O (and get overridden by our

own conversion routines)?
- CPINTERNAL
    - I worry that some applications will have hard coded dependencies that assume CPINTERNAL is set to something else.  How do we handle such cases?
    - If we were to allow this to be customized, what would it affect?  Can we isolate those parts and implement this while still leaving the JVM String instances in UTF-16?

> When a special encoding is needed, the getBytes() will provide it. However, the resulting bytes array only makes sense when read in that that specific CP. The advantage: all strings are compatible, and any operation does not need CP conversion.

I agree that all direct access to the getBytes() needs to be handled differently.

> The 4GL, OTOH, seems to me to store the strings internally using the specified CP. The cpinternal looks more like a default encoding for LONGCHAR variables. Apparently also used by all character variables. The fields use the CP configured at table/db creation. To understand the stored bytes of a text variable you need to also know the CP it was encoded with, otherwise it is just meaningless. To concatenate such strings, they must be first converted to a common CP. The advantage is that it is easier for IO operations which are usually in the respective CP.

I think the 4GL implementation is not that far from ours, we just have a fixd CPINTERNAL and much more of a bias to Unicode.  They also have the assumption that the core text processing in the 4GL session (a.k.a. the "AVM") has all data in that CPINTERNAL.  Any input or output is converted from or to that I/O codepage that is configured.  This is the same core design.

They have the ability to change the CPINTERAL for a given 4GL session, which we don't yet have.  To implement such a thing, we would have to find all the places in our code which are dependent upon CPINTERAL and ensure we handle the encoding correctly.  I think the number of such places is very large.  Any location that processes text data from or for the application is affected.  I don't think it is just the character and longchar types.  It starts with Unicode string literals in the converted code and continues through any location where we do string processing in or on behalf of the converted code.  I can even imagine lots of places where we create temporary String instances inside expressions (with no intermediate local variable).  Mapping all of that to a custom CPINTERAL seems a lot of work (and very error prone).

**#8 - 04/06/2021 11:54 AM - Ovidiu Maxiniuc**

One of the cause of the exception seems to be the chr function implementation. It was patched to match its description from manual:

> The CHR function is double-byte enabled. For a value greater than 255 and less than 65535, it checks for a lead-byte value. If the lead-byte value is valid, the AVM creates and returns a double-byte character.

If we take into consideration character 194, which is less than 255. It should be represented on a single byte in UTF-8. But, if we execute:

```
MESSAGE CHR(194) CHR(194, "ISO8859-1") CHR(194, "UTF-8") LENGTH(CHR(194, "UTF-8"), "CHARACTER").
```

It will print: ┬ ┬ ├é 2 in text/ChUI mode (Windows) and Â Â Ã 2 in GUI. There are multiple observations here:

- ChUI uses the extended ASCII set (webopedia) while GUI uses true 8 bit ISO8859-1 set (wikipedia), so the result may be quite different;
- 194 <= 255, so it should have been represented on a single byte according to the manual. However, it is clearly visible that the character is NOT represented this way. The two bytes used are 195 (0xC3) and 130 (0x82).
- I expected the result of CHR() to be a single (double-byte) character. But LENGTH function, with "CHARACTER" as second parameter returns 2 not 1. I was expecting the value 2 only in "RAW" mode (binary), meaning the result is a single double-byte character. It seems to be a pair single-byte characters instead.

This is confusing.

**#9 - 04/06/2021 11:55 AM - Ovidiu Maxiniuc**

I reached the conclusion that we should configure the FWD to work using the same CPINTERNAL the client had assumed their application will run. This is because functions like CHR and ASC will default their source parameter to that CP. Here is an example from one of our customer: they need the character § (Unicode \u00A7) to be displayed / stored in database. Looking at ISO8859-1 table you can see that this is encoded as decimal 167. So they used CHR(167) in their code. And everything goes as expected as long as OE is started without -cpinternal parameter.

If FWD is to duplicate this behaviour we must return "§" string when CHR(167)/I18nOps.chr(new integer(167)) is invoked. But this is equivalent to CHR(167, SESSION:CPINTERNAL, SESSION:CPINTERNAL) in OE which uses by default ISO8859-1. In FWD that is UTF-8. And here is the problem, because the UTF-8 code for '§' is a double byte C2 A7 (which in decimal is 49831) not the single byte A7. We cannot guess and return '§' because the code 167 is not its representation in UTF-8 (well, in UTF-8 interpretation of OE).

And this: CHR(49831, "ISO8859-1", "UTF-8") is what must be written to get the § printed on the default OE process. Also, CHR(49831) is what the customer must write if they want the § character to be displayed in an OE process which was started with -cpinternal UTF-8 parameter.

Note: OE will successfully encode and decode also 3-byte characters like € (Unicode 20AC) using ASC, returning 14844588 (0xE282AC) even if only double-byte is specified in manual.

Here is a little summary:

| Character | Unicode | OE ISO8859-1 | OE ISO8859-15 | OE UTF-8 |
|---|---|---|---|---|
| A | 65 / \\u0041 | 65 / 0x41 | 65 / 0x41 | 65 / 0x41 |
| § | 167 / \\u00A7 | 167 / 0xA7 | 167 / 0xA7 | 49831 / 0xC2_A7 |
| € | 8364 / \\u20AC | N/A<br>(character does not exist in this CP) | 164 / 0xA4 | 14844588 / 0xE2_82_AC |

As a conclusion, we should always default to ISO8859-1 as OE does and not UTF-8 as we do now.

**#10 - 04/06/2021 04:19 PM - Greg Shah**

Our core I18N runtime design assumes:

- We use the standard Java UTF-16 internal String encoding for all String instances.
- We handle the codepage conversion when needed as we go into and out of String instances.
- We use <locale>.UTF-8 as the LANG for the JVM.
- We think of CPINTERNAL as the same as our default encoding for the JVM.

Your new details prove that our approach is not correct. The implementation of the CP* values is OK, except for CPINTERNAL. We should support CPINTERNAL differently.

**Minimal Implementation (which will not work)** - We could just improve our CPINTERNAL implementation for 4GL features such as CHR() and ASC(). The idea is that we could treated the source inputs for CHR() and ASC() as being processed using CPINTERNAL as you've shown here. So CHR(x) would interpret x in the legacy CPINTERNAL but it would return back a character instance that has a String instance with the correct chars in standard UTF-16 internal encoding. We've previously planned that this would work because UTF-16 can encode all possible characters that we would ever see.

But this idea does not work 100% of the time for multi-byte characters. It would partially work, in that some use cases would be represented OK. But we get into problems with any 4GL feature that accesses the characters as single-byte values. And the 4GL pretty much forces you to do this since they have very limited multi-byte support.

The problem is probably worse than that because we add our own low level access to the backing data in character and longchar types. This is done in TextOps and a very wide range or UI, database and other base language features. Consider how many places we use getValue() and then access the String directly.

**Proper Implementation** - I think to fix this properly, we must do the following:

- Rework Text, character and longchar to:
  - Store the character data using a String that is encoded using the correct codepage instead of UTF-16.
  - This would be CPINTERNAL by default, but would be overridden in any case where a specific value was provided. For example, when FIX-LONGCHAR() is called or there is a character value returned by CODEPAGE-CONVERT().
  - No direct access to the internal data. An API must be created to safely access the data in a very known way.
  - All locations where the data is accessed or serialized must be made safe.
  - The Externalizable implementation will need to carry the codepage with it for cases where the codepage is non-default.
- All operators (including comparisons, assignment, concatenation...), built-in functions and other 4GL string processing must be made safe. This must maintain the codepage through any temporary/transient copies.
- We might as well honor session-level CPINTERNAL values at the same time. The effort is minimal to do so once we've taken the hit with this approach.
- We will have to check the wider range of UI, database and other usage of these types. These must also be made safe.
  - Mostly this should be moving to the safe external API for data access.
  - We must carefully look at how we are using the internal data even on the client side.
- We will need to write testcases to show how the multi-byte support works. Chapter 8 "Using Multi-byte Code Pages" in the Internationalize ABL document provides some details. The support is limited so that is some good news.

I do worry a bit about how the converted code may interact here. We currently convert all String literals to UTF-8 strings in the Java code. And we pass those in as parameters everywhere. This may mean that we have to more carefully/thoroughly utilize the character type as a wrapper in our runtime code wherever there could be a dependency on the legacy codepage. Generally, I'd like to avoid this but I worry that there are places that will matter.

Overall, I think this design should be enough to handle all the 4GL I18N behavior. I may be missing some impacts or some places in the code that will need attention. Please comment on the idea overall, note any problems you can think of and any missing items in the approach.

Unfortunately, this is a bigger effort than I was wanting at this time. I don't see an alternative.

**#11 - 04/07/2021 09:39 AM - Marian Edu**

Ovidiu Maxiniuc wrote:

- I expected the result of CHR() to be a single (double-byte) character. But LENGTH function, with "CHARACTER" as second parameter returns 2 not 1. I was expecting the value 2 only in "RAW" mode (binary), meaning the result is a single double-byte character. It seems to be a pair single-byte characters instead.

It is a double byte character, only what you see is calling LENGTH on a character which is always considered to be using the internal codepage (CPINTERNAL). Hence assigning the result of CHR or even CDEPAGE-CONVERT to a character variable (or longchar that was not the codepage fixed) is always treated as being encoded using CPINTERNAL. So the double byte character stored in that character is then seen as two single byte characters because the internal codepage is single byte - most probably ISO8859-1.

**#12 - 05/27/2022 10:08 AM - Greg Shah**

*- Related to Feature #6451: I18N phase 4 added*

**#13 - 08/03/2023 10:10 AM - Greg Shah**

*- Related to Feature #6431: implement support to allow CPINTERNAL set to UTF-8 and CP936 added*