EuroForth 2019
Internationalisation - a new approach in Forth

Abstract

The unique capabilities of Forth are harnessed in a technique that greatly improves the efficiency of software internationalisation.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

In software, the word "internationalisation" means writing a program in such a way as to separate the function of the program from its country or language specific appearance. The goal is to make it possible to produce either, different language versions of the program, or in our case, to produce a program that can be dynamically switched between different languages. We need this because the normal user of the program might require a foreign language, and our own support engineer might require English. There are cultural aspects to internationalisation (for example the formatting of date, time and currency), but the main part of the work concerns text.

The word "localisation" refers to the process of adding a new language. The aim of the exercise is to make localisation as easy and foolproof as possible.

We have been doing this for many years now, but recently a new technique using the unique capabilities of Forth, has greatly improved the efficiency of the process.

2. The old method reviewed

The previous technique, first described in Euroforth in 1995, was based around the word:

: P" \ Comp: "text" -- ; Run: -- u

The run time action was easy to describe - it simply returned a number that referred to the phrase that in English was "text".

The compilation action was rather more complex.

First, it checked to see if "text" already existed in the set of phrases for English. If so, it just compiled the number of that phrase as a constant. If the phrase did not exist, then it added "text" to the table of English phrases. Then it compiled the new phrase number as a constant.

At the end of the compilation process, the table of phrases was saved to a text file in a directory called "English". (Originally, phrases were saved in a set of Forth "screens" in non-volatile memory!) There was one line in the file for each phrase.

The process of localisation was then to create a matching phrases file, in another directory named after the alternative language. A text editor was selected which both showed line numbers (equal to the phrase number) and allowed two files to be displayed side by side. This made it relatively easy for the translator.

When the user selected a different language, the alternative phrase file was loaded into memory, in a "bi-string" format with both a leading byte count and a double zero terminator. The language text could then be accessed by either a word CTEXT, that returned the address of the counted string, or a word ZTEXT, that returned the address of the zero terminated string. All window text, including menus, were then updated with the new text. Whenever a new dialog was created, during its initialisation, the text was set for each control that required it.

3. Improvements to the underlying system

Over the years, many improvements were made to the P" based system.

a) The storage of phrases was moved to a database table.
b) In addition to standard text phrases, classes of diagnostic phrases and sequences phrases, customised to the machine being controlled, were introduced.
c) When a new phrase was encountered, all supported languages were also updated, using the "text" phrase preceded by the phase number. This made it easy to identify untranslated phrases.
d) A Translators Aid dialog was developed, which isolated the translator from the method of storage, and prevented him from introducing positional errors.
e) English ceased to be special. New "text" was placed into a base language. This meant that text could be altered in English as well as all other languages, without recompilation.
f) Support for counted strings was dropped.
g) P" returned a pointer to the translated string, instead of a phrase number.

4. Limitations of the old method

a) The main limitation of the old method was that all textual objects needed to have their text set, either initially and on change of language, for main windows and menus; or on initialisation, for dialog boxes. In a large program, this accounted for thousands of lines of code.

b) A further limitation was that the original code was devised for ASCII based text. It was extended for "Shift-JIS" for Japanese, and "GB 18030" for simplified Chinese. However, it was very difficult to apply in many other languages with non-standard characters.

5. Big breakthrough no. 1 - dropping Windows!

Several years ago, we came to the conclusion that Windows was no longer a suitable operating system for use with machine automation. It became company policy that all new programs would be written for Linux. From the point of view of internationalisation, the most important improvement here was moving to a native UTF-8 encoding, instead of having to use the "adapted-ASCII" APIs that had been in use with Windows. No more code pages! This dealt with limitation 4b).

6. Big breakthrough no. 2 - GtkBuilder

When we adopted Linux, we also decided to move to GTK+ for our graphical user interfaces.

In my presentation at Euroforth 2017, I described how a control element named in the GTK graphical design program "Glade", could appear automatically as a Forth VALUE, *without any other coding!* This discovery, only possible in Forth, enabled hundreds of lines of code to be removed from a typical program.

It was only later that we realised that a very similar technique could be used to handle text internationalisation, *without any application specific coding*. This would therefore deal with limitation 4a) and remove many hundreds more lines of code.

7. Review of the GtkBuilder process

In order to appreciate the automated internationalisation technique, it is necessary to understand a little bit about how GTK works.

a) The windows, dialog boxes, menus, and all other graphical elements, are designed using a visual tool called "Glade". The information is saved in XML format.

b) Each component is referred to by a name. Components such as labels, titles and menu entries are also given initial text during the design process.

b) The application loads the XML files using functions of an object called GtkBuilder. During the loading process, each component is created and receives a reference number, similar to a "handle" in Windows.

c) When the application uses a graphical component, for example, to set the text of a label, it must use the reference number to identify the component.

d) Any time after loading, a list of all components can be accessed using a GtkBuilder function.

e) We read in XML files not just during execution, but also during compilation, at an early stage. We go through the list and create a Forth VALUE for every component, with the same name as was used in the design tool, and which returns the reference number. This means that during the rest of the Forth compilation process, all components can be referred to by their Glade / Forth VALUE names.

f) Note one big difference from Windows. In Windows, dialog boxes are created and destroyed as necessary. When using GtkBuilder, everything is created when the XML files are loaded, and items such as dialog boxes are shown and hidden, not created and destroyed. Therefore, all graphical components are always accessible.

8. Automating the internationalisation process

When an application is initialising, before any graphical elements have been displayed, but after we have loaded our language phrase table from the database, we need to set the correct phrase into each textual component. We call a word SETPHRASES - simplified listing below:

```
: SETPHRASES { | pslist pobject -- } \ Set language phrases
  PBUILDER gtk_builder_get_objects -> pslist         \ Make list of objects
  pslist g_slist_length 0 ?DO                        \ For all objects
    pslist I g_slist_nth_data -> pobject             \ Get data
    pobject gtk_label_get_type
    g_type_check_instance_is_a IF                    \ It's a label
      pobject SET-LABELPHRASE
    THEN
..... Other object types .....
  LOOP
  pslist g_slist_free                                \ Free list
;
```

This gets the list of objects and checks each one to see if it's a type of object that needs some text (e.g. a label). If so, it calls the function to set text for that object type.

```
: SET-LABELPHRASE { plabel | plabeltxt -- } \ Set language phrase for label
  plabel gtk_label_get_label -> plabeltxt          \ Get text, incl. markup
  plabel plabeltxt SET-WIDGETPHRASE -> plabeltxt   \ Associate with a phrase
  plabel plabeltxt gtk_label_set_label             \ Set language text
;
```

The get_ and set_ functions in GTK are different for each object type, so a different SET-xxxPHRASE word is needed. But all object types use the same base object, so a common word can be used.

```
: SET-WIDGETPHRASE { pwidget ptext | pphrase -- ptext' } \ Widget to phrase
  ptext IF                                    \ Text defined
    ptext C@ TRANSCHAR = ptext                \ Marked for translation
    1+ C@ 0<> AND IF                          \ Not null
      1+ PHRASE? -> pphrase                   \ Get phrase number
      pphrase -1 = IF                         \ Phrase does not already exist
        ptext 1+ ADDPHRASE -> pphrase         \ Add to database
      THEN
      pwidget pphrase SETPHRASE               \ Associate phrase with widget
      pphrase LANGPHRASE                      \ Return language phrase
    ELSE                                      \ This text not to be translated
      ptext                                   \ Return unchanged
    THEN
  ELSE
    ZNULL                                     \ Return a null string
  THEN
;
```

Not all text object should be translated - some might contain symbols. So we mark text that needs translating with a preceding character - usually a hash sign.

Other words such as PHRASE? , ADDPHRASE and LANGPHRASE already exist from our implementation of P".

The interesting bit is SETPHRASE.

```
: SETPHRASE { phandle pphrase -- } \ Associate phrase number with widget
  phandle Z" Phrase" pphrase g_object_set_data
;
```

Because every component in GTK is an object, and all objects can carry a set of key-to-pointer associations, we simply create a key called "Phrase" and link it to the phrase number, which is pretending to be a pointer.

Now we can look at what happens when a user selects a different language. First, we load the set of phrases for the new language, then we call CHANGEPHRASES, which is very similar to SETPHRASES as shown above, except that for each object type, it calls a change function, instead of a set function, for example:

```
: CHANGE-LABELPHRASE { plabel -- } \ Change language phrase for label
  plabel GETPHRASE ?DUP IF                      \ A phrase number defined
    plabel SWAP LANGPHRASE gtk_label_set_label \ Set text in current lang
  THEN
;
```

GETPHRASE simply looks up the phrase number using the key name of the object.

```
: GETPHRASE ( handle---phrase ) \ Get phrase number of widget
  Z" Phrase" g_object_get_data
;
```

Note that none of the above is application specific. Any program that uses GtkBuilder and Forth can be internationalised, simply by including the appropriate files, and adding just just half a dozen words to the application code.

9. Other internationalisation issues

Besides the purely textual issues, we have to resolve some cultural issues, such as formatting of dates.

For example, if we do an SQL query with a date in the result, we need to format that date in a language specific way, before displaying it in, say, a GtkLabel.

The first thing that is needed, is to set the program locale on change of language.

```
2 0 CallProc: on_French_clicked { pbutton puser -- } \ Button "French" click
  ITASK                                  \ Init every thread or callback
  Z" French" CHANGE-LANG                 \ Set all GTK text
  LC_ALL Z" fr_FR.utf8" SetLocale DROP   \ Set program locale
;
```

Now take the date from the SQL result set. First, place it into a Linux "broken down" time structure. Then call the Linux date / time formatting function.

```
: SQLDATE>LOCALE { zsqldate | tm[ tm ] -- z$ } \ Localise SQL date
  zsqldate tm[ SQLDATE>TM          \ Put date into tm structure
  PAD 50 Z" %x" tm[ strftime DROP  \ Format onto PAD
  PAD
;
```

10. An unresolved problem - GtkCalendar

I've just put this in to prove that Forth programmers can still read C code, if they really have to!

There is no easy way to make the GtkCalendar control change language programmatically, because of the way the widget has been written.

It's not a compound widget - the whole calendar is drawn as a single element.
It is locale sensitive, but unfortunately the day and month names for the locale are obtained from the operating system during gtk_calendar_init , which is called only when the first instance of GtkCalendar is first created.

This means that ***any*** calendar control always displays the day and month names that correspond to the locale that was current when the ***first*** calendar control was created. They can't be changed later, even if all calendar controls have been destroyed and recreated.

Possible solutions:
a) Register this as a GTK bug and see if we can persuade the developers to fix it.
I am hoping that by including this note in a conference paper, this might nudge them!
b) Rebuild GTK, fixing the issue ourselves. I can easily fix the code, but we have no experience in the recompilation.
c) Make our own calendar control.
d) Accept the limitation. Remember the locale in use, and save it when closing the application. Then re-select the locale, before doing gtk_init.

In the GTK source code, the problem could be fixed by moving the lookup of
default_abbreviated_dayname[i] and default_monthname[i]
from get_calendar_init to gtk_calendar_draw.

11. Some other enhancements

Several other nice labour-saving things can be done by adding just a few words to the basic SETPHRASES function, for example:

a) If the object is a button, then an icon can be automatically added to the button, according to the base phrase.

```
: SET-BUTTONIMAGE { pbutton | plabeltxt pimage pfilename -- } \ Set image
  pbutton gtk_button_get_label -> plabeltxt    \ Get text of button label
  plabeltxt IF                                 \ Any text set
    plabeltxt C@ IF                            \ A non null string
      ICONSDIR plabeltxt Z+ Z" .png" Z+        \ Construct path to png
      -> pfilename
      pfilename ZCOUNT FILEEXIST? IF           \ There is a matching png
        pfilename gtk_image_new_from_file      \ Create an image
        -> pimage
        pbutton pimage gtk_button_set_image    \ Put image into button
        pbutton TRUE                           \ Always show
        gtk_button_set_always_show_image
      THEN
    THEN
  THEN
;
```

Then, to make *every* button in the program that has, for example, the base phrase "Cancel", display the same cancel icon, all one has to do is place a file "Cancel.png" in the icons directory.

b) If the object is a dialog box, we can automatically make it transient to the main window. This is not possible in Glade, if the main window is defined in a different XML file from the dialog.

```
: SET-DIALOGTRANSIENT { pdialog -- } \ Set transient parent for dialogs
  pdialog MWINDOW gtk_window_set_transient_for
;
```

8. Conclusion

The process of internationalisation has been completely automated. When making a new application, it is only necessary to include the required phrase handling files, and add a couple of words to the initialisation code. A huge amount of work has been saved. This technique is only possible in Forth, with its unique ability to control the compilation action.

NJN
30/7/19