BCI 2000 Localization

Jürgen Mellinger

February 3, 2006

Contents

1 Introduction **2**

2 Strategies **2**

3 Implementation **3**

3.1 Interface to the operator user 3

3.2 Interface to the programmer 4

3.3 Further implementation details 5

1 Introduction

As BCI2000 modules are being used in various countries, there emerges some need to adapt user interface elements to regional customs. Most notably, text that gives information to the subject requires translation into the subject's native language, as often a knowledge of English cannot be assumed, and if so, reading text in a foreign language is a distracting factor that makes the already demanding task of operating a BCI even more difficult.

This document explains the approach taken to manage localized (translated) user interface strings in BCI2000 application modules, trying to meet the following goals:

- **Flexibility:** For the end user (operator), it should be possible to add translations into a language that is not provided inside an application module, and to modify translations he does not feel to be appropriate. This should be achievable without making changes to the source code and recompiling the application module.

- **Separability:** For the user, switching languages should be done at a central place, i.e. by changing a single parameter. For the programmer, adding localization capabilities to existing modules should be possible with minimal changes to existing code. When writing new application modules, there should be no need to consider localization issues in advance.

- **Documentation:** Documenting and providing a collection of existing translations into various languages should be possible inside a module's source code.

2 Strategies

For a number of text values that have always been set from parameters (such as the Speller module's `Goal` parameter that holds the text the user is told to spell in copy spelling mode), localization is not an issue because the user may just change the value as it seems appropriate.

The remaining text items, according to the way they are specified, fall into two categories:

1. C string literals in `*.cpp` source files, and

2. text fields in GUI resource files (such as Borland `*.dfm` files).

For both categories, translations are kept in a single matrix type parameter that uses string labels, i.e. row and column titles. Each column title represents the native English version of a text; row titles represent languages for which translations exist. The user chooses amongst languages by specifying the desired language in a second parameter; the BCI2000 framework will then use this table to look up a text's translation, matching the text itself against column labels, and the target language against row labels. If no match is found, it will leave the text unchanged.

For string literals in `*.cpp` files, the strategy is to simply put a function layer around the string, i.e. to send it through a function that checks for a translation and exchanges the text if there is a match. When introducing localizability into existing code, this implies only a very small amount of changes compared to, e.g., introducing a separate parameter for each string. In the former case, one needs only wrap the string literal into a function call in-place; in the latter case, one would have to add a parameter line to a possibly remote filter class constructor, read that parameter from inside that filter's `Initialize()` call, and put it into the object that actually holds the string which might require introduction of additional accessor functions.

For strings specified in GUI resource files, the strategy is to supply a function that, very generally, examines string properties of GUI objects, replacing them with their localized versions if applicable. This function needs to be called explicitly from inside the `Initialize()` function of the `GenericFilter` descendant that determines the GUI object's behavior (in most cases, this is the application module's `TTask` class).

3 Implementation

The implementation is centered around a `Localization` class declared in `Application/shared/Localization.h`.

3.1 Interface to the operator user

The `Localization` class adds two parameters to the system that govern its localization behavior:

- `Language` defines the language to translate strings into; if its value matches one of the `LocalizedStrings` row labels, translations will be

taken from that row; otherwise, strings will not be translated. A value of `Default` results in all strings keeping their original values.

- `LocalizedStrings` defines string translations. Strings that don't appear as a column label will not be translated. Also, strings with an empty translation entry in `LocalizedStrings` will not be translated.

3.2 Interface to the programmer

The `LocalizedStrings` parameter is empty by default. Although a user may add translations into desired languages to the empty matrix by hand – using the operator module's matrix editor –, translations will preferably be provided in a filter constructor by listing them as in the following example:

```
#include "Localization.h"
...
TTask::TTask()
{
  ...
  LANGUAGES "Italian",
            "French",
  BEGIN_LOCALIZED_STRINGS
   "Yes",
            "Si",
            "Oui",
   "No",
            "No",
            "Non",
  END_LOCALIZED_STRINGS
  ...
}
```

If the `LocalizedStrings` matrix was empty before the `TTask` constructor gets called, it will have these entries after execution of the constructor:

	Yes	No
Italian	Si	No
French	Oui	Non

There may be any number of translation tables inside filter constructors, with their entries being added to the existing ones, or overriding entries that already exist.

Once entered, the translations contained in `LocalizedStrings` are applied via two mechanisms:

- The function `LocalizableString()` takes a string as an argument and returns the appropriate entry from `LocalizedStrings`, or the unmodified string if no entry can be found. E.g., instead of

  ```
  TellUser( "Well done!" );
  ```

  one would write

  ```
  #include "Localization.h"
  ...
  TellUser( LocalizableString( "Well done!" ) );
  ```

  to have a translation for "Well done!" looked up in `LocalizedStrings`.

- The function `ApplyLocalizations()` takes a pointer to a GUI object (usually a VCL `TForm*`) and translates all localizable text contained within it. This function must be called during `GenericFilter::Initialize` for each GUI object generated from a resource file.

3.3 Further implementation details

- You should not use `LocalizableString()` on string constants used before the first call to `GenericFilter::Initialize()` or for initializing static or global objects of any kind because localization information used will not be available at global initialization time, and local static variables, once initialized, will not be updated appropriately.

- Language names are case-insensitive. You may use any string for a language name but as a convention we suggest its most common English name, as in `Italian, Dutch, French, German`, with international country abbreviations as optional regional qualifiers as in `EnglishUS, EnglishGB, GermanA, GermanCH` if necessary.

Agrave	À	Aacute	Á	Acirc	Â	Atilde	Ã
Auml	Ä	Aring	Å	AElig	Æ	Ccedil	Ç
Egrave	È	Eacute	É	Ecirc	Ê	Euml	Ë
Igrave	Ì	Iacute	Í	Icirc	Î	Iuml	Ï
Ntilde	Ñ	Ograve	Ò	Oacute	Ó	Ocirc	Ô
Otilde	Õ	Ouml	Ö	Oslash	Ø	Ugrave	Ù
Uacute	Ú	Ucirc	Û	Uuml	Ü	Yacute	Ý
szlig	ß	agrave	à	aacute	á	acirc	â
atilde	ã	auml	ä	aring	å	aelig	æ
ccedil	ç	egrave	è	eacute	é	ecirc	ê
euml	ë	igrave	ì	iacute	í	icirc	î
iuml	ï	ntilde	ñ	ograve	ò	oacute	ó
ocirc	ô	otilde	õ	ouml	ö	oslash	ø
ugrave	ù	uacute	ú	ucirc	û	uuml	ü
yacute	ý	yuml	ÿ				

Table 1: HTML names for international characters

- Encoding of non-ASCII characters follows the UTF8 convention. To ensure platform independent readability of source code files, there are macros that define HTML character names to their UTF8 encoded strings. This allows to write

  ```
  "Sm" oslash "rrebr" oslash "d"
  ```

  for “Smørrebrød” (cf. table 1).