Cuneiform 2024

User Manual

~ Internationalization & Localization Analysis System ~

Blake Madden

Cuneiform 2024

User Manual

Blake Madden

Table of contents

Chapter 1

Overview

`Cuneiform` is a command-line utility and graphical user interface that scans source and resource files to check for various i18n and l10n issues. Additionally, the GUI version provides pseudo-translation support for *gettext* catalogs (*.po files).

1.1 File Support

`Cuneiform` supports static analysis for the following:

- C code
- C++ code ('98 and modern C++)
- Windows resource files (*.rc)

Static analysis and pseudo-translation are available for:

- GNU *gettext* translation files (*.po and *.pot)

Additionally, it offers specialized support for the following frameworks:

- wxWidgets
- Qt
- KDE
- GTK$^+$
- Win32
- MFC

1.2 Static Analysis

The command line and GUI versions provide the following checks:

- Strings exposed for translation[1] that probably should not be. This includes (but not limited to):
  - Filenames
  - Strings only containing `printf()` commands
  - Numbers
  - Regular expressions
  - Strings inside of debug functions
  - Formulas
  - Code (used for code generators)
  - Strings that contain URLs or email addresses
- Strings not available for translation that possibly should be.
- Strings that contain extended ASCII characters that are not encoded. ("Danke schön" instead of "Danke sch\U000000F6n".) Encoding extended ASCII characters is recommended for best portability between compilers.
- Strings with malformed syntax (e.g., malformed HTML tags).
- Use of deprecated text macros (e.g., `wxT()` in wxWidgets, `_T()` in Win32).
- Use of deprecated string functions (e.g., `_tcsncpy` in Win32).
- Files that contain extended ASCII characters, but are not UTF-8 encoded. (It is recommended that files be UTF-8 encoded for portability between compilers.)
- UTF-8 encoded files which start with a BOM/UTF-8 signature. It is recommended to save without the file signature for best compiler portability.
- `printf()`-like functions being used to just format an integer to a string. It is recommended to use `std::to_string()` to do this instead.
- `printf()` command mismatches between source and translation strings. (PO catalogs with C/C++ strings are currently supported.)
- Font issues in Windows resource files (Microsoft, STRINGTABLE resource):
  - Dialogs not using "MS Shell Dlg" or "MS Shell Dlg 2."
  - Dialogs with non-standard font sizes.

Code formatting and other issues can also be checked for, such as:

- Trailing spaces at the end of a line.
- Tabs (instead of spaces).
- Lines longer than 120 characters.
- Spaces missing between "//" and their comments.
- ID variable[2] assignment issues:
  - The same value being assigned to different ID variables in the same source file (e.g., "wxID_HIGHEST + 1" being assigned to two menu ID constants).
  - Hard-coded numbers being assigned to ID variables.
  - Out-of-range values being assigned to MFC IDs (TN020: ID Naming and Numbering Conventions).

[1]Strings are considered translatable if inside of gettext, wxWidgets, Qt, or KDE (ki18n) i18n functions. This includes functions and macros such as `gettext()`, `_()`, `tr()`, `translate()`, `QT_TR_NOOP()`, `wxTRANSLATE()`, `i18n()`, etc.

[2]Variables are determined to be ID variables if they are integral types with the whole word "ID" in their name.

1.3 Pseudo-translation

(available in the GUI version)

Pseudo-translation includes features such as:

- Multiple methods for character replacement (e.g., replacing characters with accented variations or upper casing them).
- Increasing the width of the translations. This is useful for ensuring that strings are not being truncated at runtime.
- Wrapping the translations in braces. This is useful for ensuring that strings are not being pieced together at runtime.
- Appending a unique ID to each translation. This is useful for finding where a translation is being loaded from.

When pseudo-translating, a copy of all string catalogs will be created and have their translations filled with mutations of their respective source strings. These files (which will have a "pseudo_" prefix) can then be compiled and loaded by your application for integration testing.

Note
After processing a folder, the **Analysis Log** tab of the bottom window will display a list of all pseudo-translated resource files that were generated.

Chapter 2

Building

`Cuneiform` requires a C++20 compiler and can be built on Windows, macOS, and Linux.

Cmake scripts are included for building both the command-line and GUI (graphical user interface) versions. *Cmake* 3.25 or higher is required.

For the GUI version, wxWidgets 3.3 or higher is required.

2.1 Command-line Utility

`Cuneiform` can be configured and built with *Cmake*.

On Unix:

Listing 2.1 `Terminal`

```
cmake . -DCMAKE_BUILD_TYPE=Release
cmake --build . --target all -j $(nproc) --config Release
```

On Windows, “CMakeLists.txt” can be opened and built directly in Visual Studio.

After building, “cuneiform” will be available in the “bin” folder.

2.2 GUI Version

wxWidgets 3.3 or higher is required for building the graphical user interface version.

Download wxWidgets, placing it at the same folder level as this project. After building *wxWidgets*, *Cuneiform* can be configured and built with *Cmake*.

On Linux/macOS:

Listing 2.2 `Terminal`

```
# download and build wxWidgets one folder above
cd ..
git clone https://github.com/wxWidgets/wxWidgets.git --recurse-submodules
cd wxWidgets
cmake . -DCMAKE_INSTALL_PREFIX=./wxlib -DwxBUILD_SHARED=OFF \
    -D"CMAKE_OSX_ARCHITECTURES:STRING=arm64;x86_64" \
    -DCMAKE_OSX_DEPLOYMENT_TARGET=10.15 \
    -DwxBUILD_OPTIMISE=ON -DwxBUILD_STRIPPED_RELEASE=ON \
    -DCMAKE_BUILD_TYPE=Release
cmake --build . --target install --config Release

# go back into the project folder and build the GUI version
cd ..
cd Cuneiform/gui
cmake . -DCMAKE_BUILD_TYPE=Release
cmake --build . --target all --config Release
```

Note
On macOS, a universal binary 2 (containing arm64 and x86_64 binaries) will be built with the above commands.

On Windows with Visual Studio, build wxWidgets with the defaults, except `wxBUILD_SHARED` should be set to “OFF” (and `MAKE_BUILD_TYPE` set to “Release” for release builds).

Open “gui/CMakeLists.txt” in Visual Studio, setting the *CMake* setting’s configuration type to “Release” for a release build.

After building, “cuneiform” will be available in the “bin” folder.

Part I

Command-line Utility

Chapter 3

Options

`Cuneiform` accepts the following arguments:

input

The folder (or file) to analyze.

--enable

The following checks can be toggled via the `--enable` argument.

- `allI18N`

Perform all internationalization checks (the default).

- `suspectL10NString`

Check for translatable strings that should not be (e.g., numbers, keywords, `printf()` commands).

- `suspectL10NUsage`

Check for translatable strings being used in internal contexts (e.g., debugging and console functions).

- `suspectI18NUsage`

Check for suspect usage of i18n functions.

- `urlInL10NString`

Check for translatable strings that contain URLs or email addresses.

It is recommended to dynamically format these into the string so that translators do not have to manage them.

- `spacesAroundL10NString`

Check for strings that start or end with spaces. (This is limited to actual space characters, and does not include tabs or newlines.)

These may be strings that are concatenated at runtime, which does not follow best i18n practices. It is recommended to format separate values into a single string via a `printf` formatting function, rather than piecing them together.

Note
An exception is made for strings ending with ":"; here, it is assumed that this is formatting tabular data, rather than piecing a message together.

- `L10NStringNeedsContext`

Check for ambiguous strings that lack a translator comment.

Translator comments can be added to *gettext* functions by either inserting a tagged comment in front of them or by using a variant with a context parameter (e.g., `wxTRANSLATE_IN_CONTEXT()`). For example:

```
ListFiles(wxString::Format(
        /* TRANSLATORS: %s is folder name */ _(L"Searching %s"),
        get_selected_dir()));
```

By default, *gettext* will extract comments beginning with "TRANSLATORS:" in front of a resource function as a comment for the translator.

Strings are considered to be ambiguous if they meet any of the following criteria:

- A single word that contains 32 characters or more
- A single word that is uppercased (e.g., "FACTOR_VAR")
  - Punctuation is ignored for this check
- A single word that contains more than one punctuation mark (e.g., "*Print*")
  - Note that hyphens, slashes, periods, and ampersands are ignored for this check
- Contains three or more `printf` commands
- Contains at least one `printf` command and is less than 16 characters
- `notL10NAvailable`

Check for strings not exposed for translation.

- `deprecatedMacro`

Check for deprecated text macros and functions.

This will detect the usage of functions that are no longer relevant, and provide a suggested replacement.

For example, the `TCHAR` functions and macros (Working with Strings; Dunn) used in Win32 programming (e.g., `_TEXT`, `_tcscmp`) to help target Windows 98 and NT are no longer necessary. `Cuneiform` will recommend how to remove or replace these.

- `nonUTF8File`

Check that files containing extended ASCII characters are UTF-8 encoded.

UTF-8 is recommended for compiler portability.

- `UTF8FileWithBOM`

Check for UTF-8 encoded files which start with a BOM/UTF-8 signature.

It is recommended to save without the file signature for best compiler portability.

This is turned off by default.

- `unencodedExtASCII`

Check for strings containing extended ASCII characters that are not encoded.

This is turned off by default.

- `printfSingleNumber`

Check for `printf()`-like functions being used to just format a number.

In these situations, it is recommended to use the more modern `std::to_[w]string()` function.

This is limited to integral values; `printf()` commands with floating-point precision will be ignored.

- `dupValAssignedToIds`

Check for the same value being assigned to different ID variables.

This check is performed per file; the same ID being assigned multiple times, but within separate files, will be ignored.

This is turned off by default.

- `numberAssignedToId`

Check for ID variables being assigned a hard-coded number.

It may be preferred to assign framework-defined constants (e.g., `wxID_HIGHEST`) to IDs.

This is turned off by default.

- `malformedString`

Check for malformed syntax in strings (e.g., malformed HTML tags).

- `fontIssue`

Check for font issues.

This is performed on Windows *.rc files and checks for dialogs that use unusual font sizes or are not using 'MS Shell Dlg'.(Microsoft, Working with Strings)

- `allL10N`

Perform all localization checks (the default).

- `printfMismatch`

Check for mismatching `printf()` commands between source and translation strings.

This is performed on *gettext* *.po files and will analyze format strings for the following languages:

- C/C++

⚠ Warning
The checks performed here are strict; all `printf()` commands in translations must match their source counterpart exactly. For example, "*%lu*" vs. "*%l*" will emit a warning. Questionable commands such as "*% s*" (space is only meant for numeric formatting) will also emit a warning.

- `acceleratorMismatch`

Check for mismatching keyboard accelerators between source and translation strings. This is performed by checking that both strings contain one '&' followed by an alphanumeric character.

This is performed on *gettext* *.po files.

- `transInconsistency`

Check for inconsistent casing or trailing punctuation, spaces, or newlines between source and translation strings.

This is performed on *gettext* *.po files.

- `allCodeFormatting`

Check all code formatting issues (see below).

> **Note**
> These are not enabled by default.

- `trailingSpaces`

Check for trailing spaces at the end of each line.

- `tabs`

Check for tabs. (Spaces are recommended as tabs may appear differently between editors.)

- `wideLine`

Check for overly long lines.

- `commentMissingSpace`

Check that there is a space at the start of a comment.

--disable

Which checks to not perform. (Refer to options available above.) This will override any options passed to `--enable`.

--log-l10n-allowed

Whether it is acceptable to pass translatable strings to logging functions. Setting this to `false` will emit warnings when a translatable string is passed to functions such as `wxLogMessage`, `g_message`, or `qCWarning`.

(Default is `true`.)

--punct-l10n-allowed

Whether it is acceptable for punctuation only strings to be translatable.

Setting this to `true` will suppress warnings about strings such as "*? - ?*" being available for localization.

(Default is `false`.)

--exceptions-l10n-required

Whether to verify that exception messages are available for translation.

Setting this to `true` will emit warnings when untranslatable strings are passed to various exception constructors or functions (e.g., `AfxThrowOleDispatchException`).

(Default is `true`.)

--min-l10n-wordcount

The minimum number of words that a string must have to be considered translatable.

Higher values for this will result in less strings being classified as `notL10NAvailable` warnings.

(Default is `2`.)

Note
Words in this context are values that contain at least one letter; numbers, punctuation, and syntax commands are ignored. For example, “Printing %s,” “Player 1,” and “Add +” will be seen as one-word strings.

--cpp-version

The C++ standard that should be assumed when issuing deprecated macro warnings (`deprecatedMacro`).

For example, setting this to `2017` or higher (along with enabling the `verbose` flag) will issue warnings for `WXUNUSED`, as it can be replaced with `[[maybe_unused]]`. If setting this to `2014`, however, `WXUNUSED` will be ignored since `[[maybe_unused]]` requires C++17.

(Default is `2014`.)

--fuzzy

Whether to review fuzzy translations.

(Default is `false`.)

-i,--ignore

Folders and files to ignore (can be used multiple times).

Note
Folder and file paths must be absolute or relative to the folder being analyzed.

-o,--output

The output report path, which can be either a CSV or tab-delimited text file. (Format is deteremined by the file extention that you provide.)

Can either be a full path, or a file name within the current working directory.

-q,--quiet

Only print errors and the final output.

-v,--verbose

Perform additional checks and display debug information and display debug information.

-h,--help

Print usage.

Chapter 4

Examples

The following example will analyze the folder “wxWidgets/src” (but ignore the subfolders “expat” and “zlib”). It will only check for suspect translatable strings, and then send the output to “results.txt”.

Listing 4.1 `Terminal`

```
cuneiform wxWidgets/src -i expat,zlib --enable=suspectL10NString -o results.txt
```

This example will only check for `suspectL10NUsage` and `suspectL10NString` and not show any progress messages.

Listing 4.2 `Terminal`

```
cuneiform wxWidgets/samples -q --enable=suspectL10NUsage,suspectL10NString
```

This example will ignore multiple folders (and files) and output the results to “results.txt.”

Listing 4.3 `Terminal`

```
cuneiform src --ignore=easyexif,base/colors.cpp,base/colors.h -o results.txt
```

Chapter 5

Reviewing Output

After building, go into the “bin” folder and run this command to analyze the sample file:

Listing 5.1 `Terminal`

```
$ cuneiform ../samples -o results.txt
```

This will produce a “results.txt” file in the “bin” folder which contains tabular results. This file will contain warnings such as:

- `suspectL10NString`: indicates that the string “*GreaterThanOrEqualTo*” is inside of a `_()` macro, making it available for translation. This does not appear to be something appropriate for translation; hence the warning.
- `suspectL10NUsage`: indicates that the string “*Invalid dataset passed to column filter.*” is being used in a call to `wxASSERT_MSG()`, which is a debug assert function. Asserts normally should not appear in production releases and shouldn't be seen by end users; therefore, they should not be translated.
- `notL10NAvailable`: indicates that the string “*'%s': string value not found for '%s' column filter.*” is not wrapped in a `_()` macro and not available for localization.
- `deprecatedMacro`: indicates that the text-wrapping macro `wxT()` should be removed.
- `nonUTF8File`: indicates that the file contains extended ASCII characters, but is not encoded as UTF-8. It is generally recommended to encode files as UTF-8, making them portable between compilers and other tools.
- `unencodedExtASCII`: indicates that the file contains hard-coded extended ASCII characters. It is recommended that these characters be encoded in hexadecimal format to avoid character-encoding issues between compilers.

To look only for suspect strings that are exposed for translation and show the results in the console window:

Listing 5.2 `Terminal`

```
$ cuneiform ../samples --enable=suspectL10NString,suspectL10NUsage
```

To look for all issues except for deprecated macros:

Listing 5.3 `Terminal`

```
$ cuneiform ../samples --disable=deprecatedMacros
```

By default, `Cuneiform` will assume that messages inside of various exceptions should be translatable. If these messages are not exposed for localization, then a warning will be issued.

To consider exception messages as internal (and suppress warnings about their messages not being localizable) do the following:

Listing 5.4 `Terminal`

```
$ cuneiform ../samples --exceptions-l10n-required=false
```

Similarity, `Cuneiform` will also consider messages inside of various logging functions to be allowable for translation. The difference is that it will not warn if a message is not exposed for translation. This is because log messages can serve a dual role of user-facing messages and internal messages meant for developers.

To consider all log messages to never be appropriate for translation, do the following:

Listing 5.5 `Terminal`

```
$ cuneiform ../samples --log-l10n-allowed=false
```

To display any code-formatting issues, enable them explicitly:

Listing 5.6 `Terminal`

```
$ cuneiform ../samples --enable=trailingSpaces,tabs,wideLine

or

$ cuneiform ../samples --enable=allCodeFormatting
```

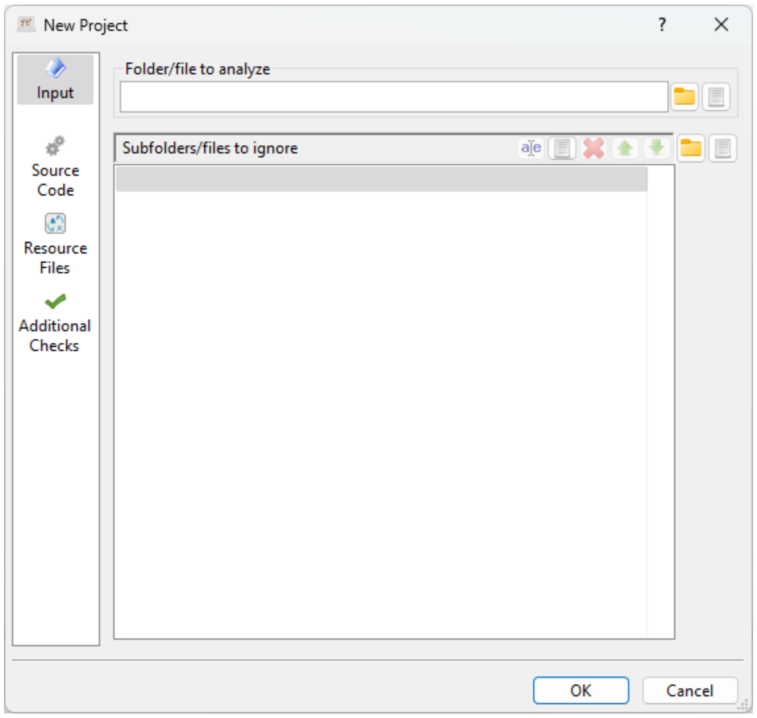
Part II

User Interface

Chapter 6

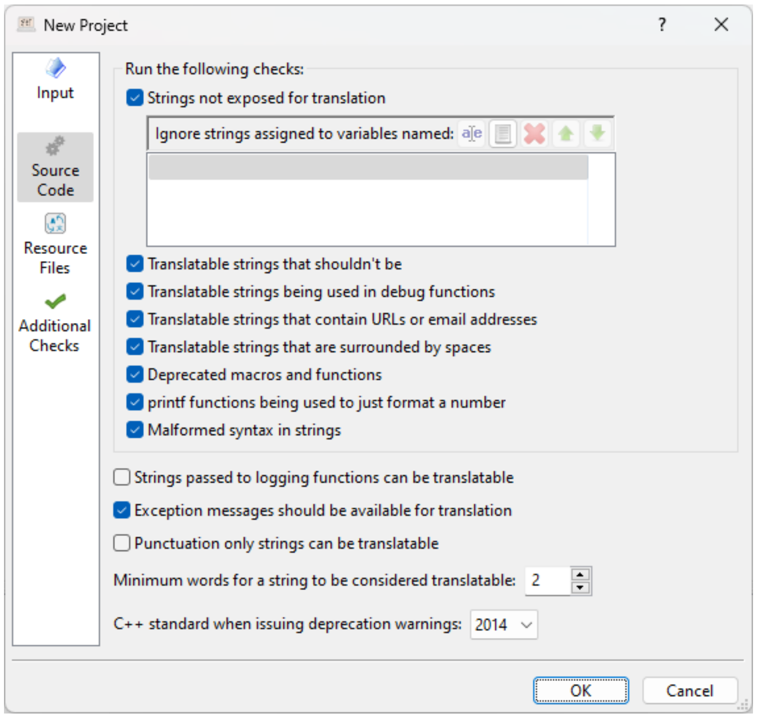
Working with Projects

To create a new project, click the New button on the ribbon. The **New Project** dialog will appear:



First, select which folder (or file) you wish to analysis. If analyzing a folder, you can also optionally select subfolders and files that should be ignored.

Next, select which source code checks to perform from the **Source Code** page.



Run the following checks:

- [x] Strings not exposed for translation

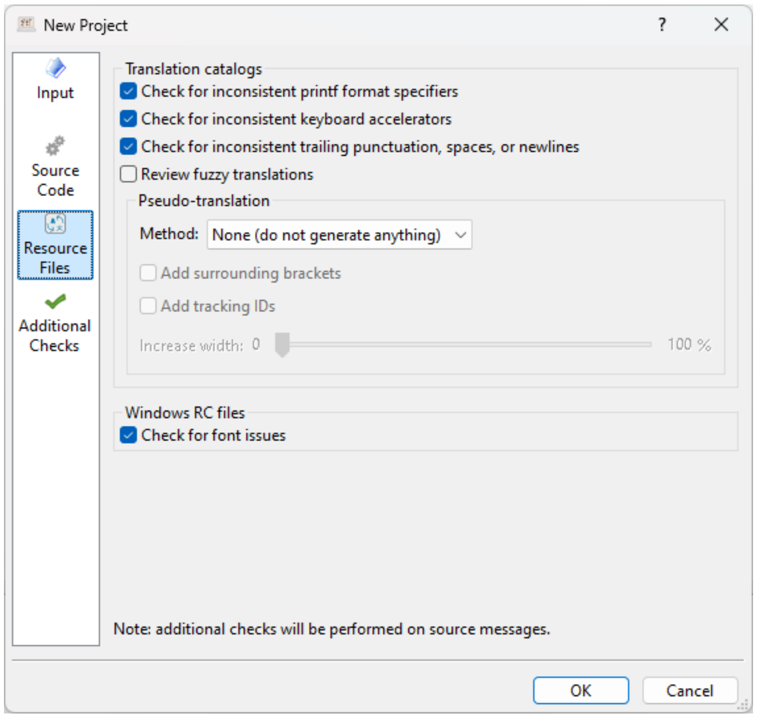
Ignore strings assigned to variables named:

- [x] Translatable strings that shouldn't be
- [x] Translatable strings being used in debug functions
- [x] Translatable strings that contain URLs or email addresses
- [x] Translatable strings that are surrounded by spaces
- [x] Deprecated macros and functions
- [x] printf functions being used to just format a number
- [x] Malformed syntax in strings
- [ ] Strings passed to logging functions can be translatable
- [x] Exception messages should be available for translation
- [ ] Punctuation only strings can be translatable

Minimum words for a string to be considered translatable: 2

C++ standard when issuing deprecation warnings: 2014

Next, options for performing checks on translation catalogs and creating pseudo-translated resources are available on the **Resource Files** page:



Translation catalogs

- [x] Check for inconsistent printf format specifiers
- [x] Check for inconsistent keyboard accelerators
- [x] Check for inconsistent trailing punctuation, spaces, or newlines
- [ ] Review fuzzy translations

Pseudo-translation

Method: None (do not generate anything)

- [ ] Add surrounding brackets
- [ ] Add tracking IDs

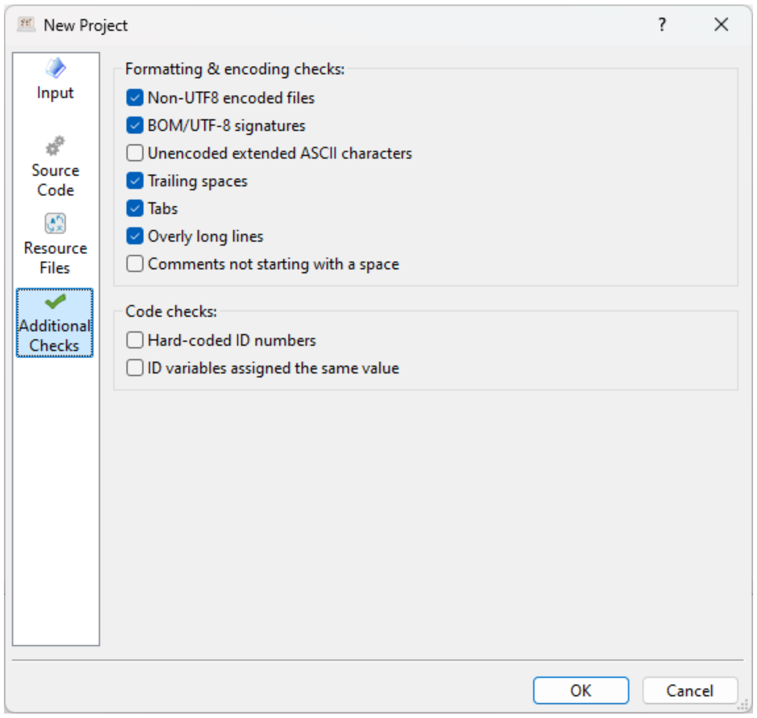
Increase width: 0 100 %

Windows RC files

- [x] Check for font issues

Note: additional checks will be performed on source messages.

Finally, the **Additional Checks** page provides checks unrelated to i18n or l10n, such as code formatting and file encoding issues:



Formatting & encoding checks:

- [x] Non-UTF8 encoded files
- [x] BOM/UTF-8 signatures
- [ ] Unencoded extended ASCII characters
- [x] Trailing spaces
- [x] Tabs
- [x] Overly long lines
- [ ] Comments not starting with a space

Code checks:

- [ ] Hard-coded ID numbers
- [ ] ID variables assigned the same value

Once finished selecting your options, click OK to create the project.

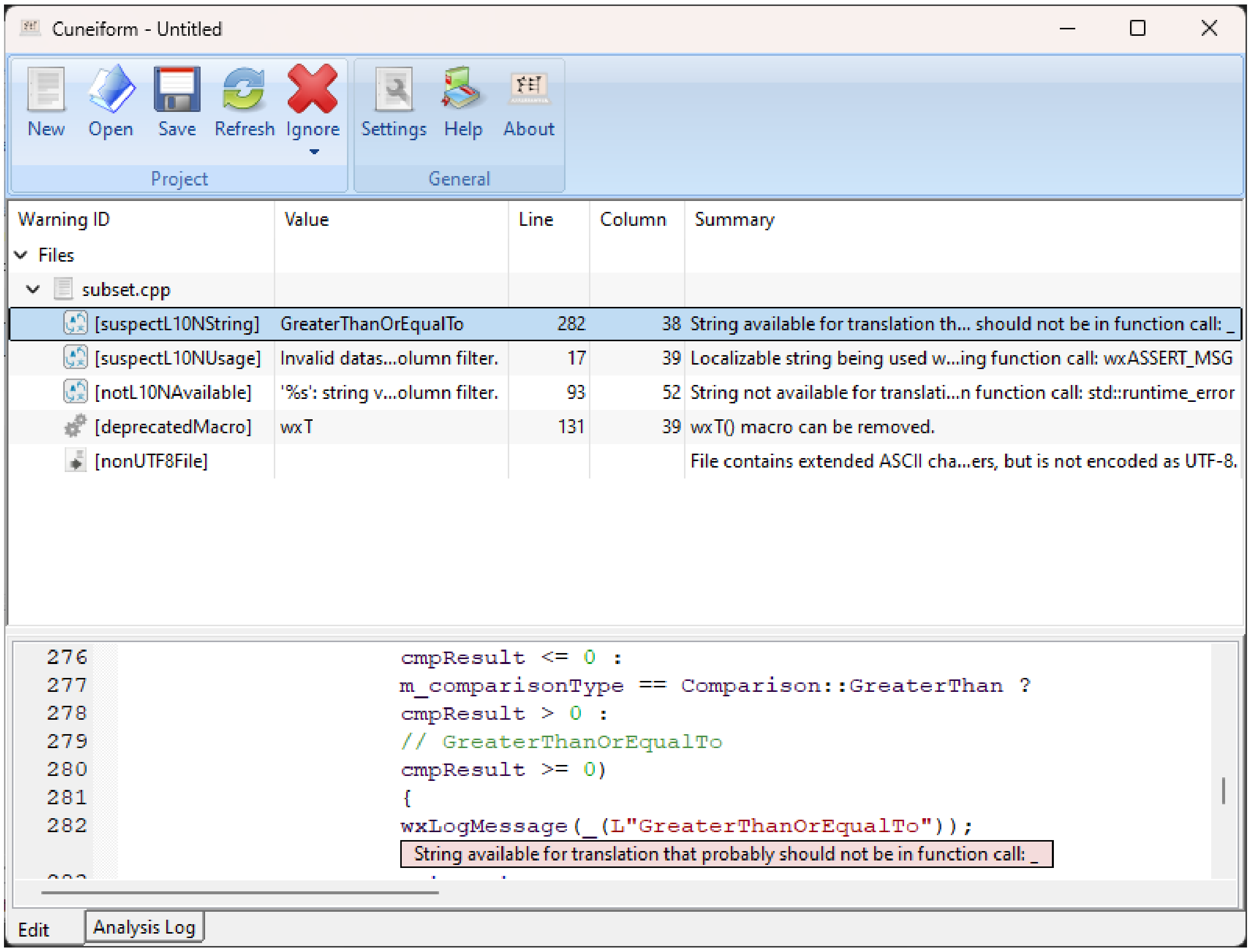
After opening or creating a project, you can reanalyze the folder at any time by clicking Refresh on the ribbon. The **Edit Project** dialog will appear, where you will be able to change any option before re-running the analysis.

When you are finished reviewing a project, click Save to save it. The project file will remember the folder you were reviewing, along with your current settings. To reopen this project, click the Open button on the ribbon and select the project file.

Chapter 7

Reviewing Output

After opening a project, select any warning message in the top window and the associated file will be displayed beneath it. The line where the issue was detected will be scrolled to, along with the warning shown underneath it.



Warning ID	Value	Line	Column	Summary
Files				
subset.cpp				
[suspectL10NString]	GreaterThanOrEqualTo	282	38	String available for translation th... should not be in function call: _
[suspectL10NUsage]	Invalid datas...olumn filter.	17	39	Localizable string being used w...ing function call: wxASSERT_MSG
[notL10NAvailable]	'%s': string v...olumn filter.	93	52	String not available for translati...n function call: std::runtime_error
[deprecatedMacro]	wxT	131	39	wxT() macro can be removed.
[nonUTF8File]				File contains extended ASCII cha...ers, but is not encoded as UTF-8.

```
                    cmpResult <= 0 :
                    m_comparisonType == Comparison::GreaterThan ?
                    cmpResult > 0 :
                    // GreaterThanOrEqualTo
                    cmpResult >= 0)
                    {
                    wxLogMessage(_(L"GreaterThanOrEqualTo"));
                    String available for translation that probably should not be in function call: _
```

From here, you can edit the file in this window to correct any issues. Once you are finished editing, either select another file's warning in the top window or click the Save button on the ribbon. (The latter will also save the project if it has unsaved changes.)

7.1 Ignoring Output

Specific warnings and files can be ignored from the project window.

To ignore a file, select it in the top window and click the Ignore button on the ribbon. From the menu that appears, select the **Ignore '*fileName*'** option (where *fileName* is the file name) and this file will be removed from the project.

To ignore a class of warnings, select a warning for any file in top window and click the Ignore button on the ribbon. From the menu that appears, select the **Ignore '*warningId*'** option (where *warningId* is the warning, such as “[notL10NAvailable]”). All instances of this warning will be removed from all files. Also, this check will no longer be performed when you re-run the analysis.

Note
These options are also available by right clicking any warning and selecting them from the popup menu.

7.2 Exporting Results

To export all warnings, click the Save button on the ribbon and select **Export Results**. From here, you can export the results to either a CSV or tab-delimited text file.

Chapter 8

Editing

When a source file in the loaded in the editor and the editor is selected, the **Edit** section of the ribbon will become enabled. From here, you can edit the source file in the **Editor** and make use of these options from the ribbon.

8.1 Basic Operations

Clipboard operations (e.g., Cut, Copy, and Paste), Undo, Redo, and Select All are all available from this section.

8.2 Insert

From the Insert button, options are available for inserting and converting content within the editor to fix various warnings.

Translator Comment

To fix warnings about strings needing context, place the cursor into the editor in front of a resource loading call (e.g., `_("TABLE")`). Next, click the Insert button on the ribbon and select **Translator Comment...**. Then, select which translator comment style that you wish to use. (*gettext* and *Qt* styles are available.) Enter a comment that will explain to the translators what “TABLE” means in this context, providing guidance for how they should translate it. Click OK and your comment with the comment style that you selected (e.g., “// TRANSLATORS:”) be inserted in front of the function call.

Note
This warning is emitted when the option **Check for ambiguous strings that lack a translator comment** (from the **Source Code** settings) is checked.

Encode Extended ASCII Characters

To fix warnings about extended ASCII characters needing to be encoded, select the text in the editor which contains these characters. Next, click the Insert button on the ribbon and select **Encode Extended ASCII Characters...**. You will then be prompted about how the selection will be re-encoded. Click OK and the selection will be replaced with the re-encoded content.

As an example, if you have the string “Błąd” (Polish for “Error”) in a source file, some legacy compilers may have difficulty processing it (even if encoded as UTF-8). By following the aforementioned steps, this will be re-encoded to “B\U00000142\U00000105d.”

Note
This warning is emitted when the option **Unencoded extended ASCII characters** (from the **Additional Checks** settings) is checked.

Mark Selection for Translation

To mark a string as translatable, select the string in the editor and select **Mark Selection for Translation...** from the Insert button. From the **Mark Selection for Translation** dialog, options are available for choosing which translation function to use, along with providing a domain or context (if applicable). The translation marking functions available are compatible with *gettext*, *wxWidgets*, *Qt*, and *KDE*.

Mark Selection as Non-translatable

To mark a string as non-translatable, select the string in the editor and select **Mark Selection as Non-translatable...** from the Insert button. From the **Mark Selection as Non-translatable** dialog, options are available for which function to wrap the string in, along with providing an optional context and comment. These functions will mark the string as non-translatable and **Cuneiform** will ignore them during an analysis. Additionally, these functions and comments decorating a string will inform developers that the string is not meant for translation and why.

Chapter 9

Settings

9.1 Input

The following options are available when creating or editing a project.

Folder/file to analyze: enter into here the folder or file to analyze.

Subfolders/files to ignore: enter into this list the folders and files to ignore.

Tip
Folder and file paths being ignored must be absolute or relative to the folder being analyzed.

9.2 Source Code

The following options are available for C/C++ source files.

Run the following checks

Strings not exposed for translation: select this to check for strings not exposed for translation.

Ignore strings assigned to variables named: when finding strings not exposed for translation, strings can be ignored if assigned to variables from this list. For example, if a list of color names are assigned to a variable named `colorMode` that you wish to ignore, then add "colorMode" to this list. You can enter variable names (e.g., "`colorMode`") or regular expressions (e.g., "`(RELEASE|DEBUG)check[0-9]?`") here.

Note
Adding a new entry here for a project will also add it to the global settings. This will ensure that you don't have to re-enter these values for future projects.

Translatable strings that shouldn't be: select this to check for translatable strings that should not be (e.g., numbers, keywords, `printf()` commands).

Translatable strings being used in debug functions: select this to check for translatable strings being used in internal contexts (e.g., debugging and console functions).

Translatable strings that contain URLs or email addresses: select this to check for translatable strings that contain URLs or email addresses. It is recommended to dynamically format these into the string so that translators do not have to manage them.

Translatable strings that are surrounded by spaces: select this to check for translatable strings that start or begin with spaces. (This is limited to actual space characters, and does not include tabs or newlines.)

These may be strings that are concatenated at runtime, which does not follow best i18n practices. It is recommended to format separate values into a single string via a `printf` formatting function, rather than piecing them together.

Note
An exception is made for strings that end with a colon followed by a space (":"). It is assumed that this part of tabular data being concatenated and will be ignored.

Suspect i18n function usage: select this to check for suspect usage of i18n functions.

This will check for issues such as:

- String literals being passed as the string argument to `wxGetTranslation()`.
  (This is incorrect, as this function is meant for variables instead of string literals.)
- Usage of the Win32 function `::LoadString()`.
  It is recommended to use other functions (e.g., `CString::LoadString()`) that are not limited to fixed buffer sizes; otherwise, translated strings may become truncated.
- Context arguments passed to l10n functions (e.g., `tr()`) that are longer than 32 characters.
  This may indicate that the context and translatable string arguments are reversed.
- Possible strings meant for translation being passed to ID loading functions (e.g., `qtTrId()`).

Run the following checks (*continued...*)

Check for ambiguous strings that lack a translator comment: select this to check for possibly ambiguous strings that lack a translator comment. *gettext*- and *Qt*-style comments are searched for, which includes formats such as "// TRANSLATORS:" and "//:".

A string will be considered ambiguous if:

- A single word in all caps.
- A single word with multiple punctuation marks embedded in it
- It contains placeholder text (e.g., "###").
- A short string with a `printf` or positional argument (e.g., "%2").
- A longer string with three or more `printf` or positional arguments.

Deprecated macros and functions: select this to check for deprecated text macros and functions. This will detect the usage of functions that are no longer relevant, and provide a suggested replacement.

For example, the `TCHAR` functions and macros used in Win32 programming (e.g., `_TEXT`, `_tcscmp`) to help target Windows 98 and NT are no longer necessary. `Cuneiform` will recommend how to remove or replace these.

printf functions being used to just format a number: select this to check for `printf()`-like functions being used to just format a number. In these situations, it is recommended to use the more modern `std::to_[w]string()` function. This is limited to integral values; `printf()` commands with floating-point precision will be ignored.

Malformed syntax in strings: select this to check for malformed syntax in strings (e.g., malformed HTML tags).

Strings passed to logging functions can be translatable: select this if it should be acceptable to pass translatable strings to logging functions. Setting this to `false` will emit warnings when a translatable string is passed to functions such as `wxLogMessage`, `g_message`, or `qCWarning`.

Exception messages should be available for translation: select this to verify that exception messages are available for translation. Setting this to `true` will emit warnings when untranslatable strings are passed to various exception constructors or functions (e.g., `AfxThrowOleDispatchException`).

Punctuation only strings can be translatable: select this if it should be acceptable for punctuation only strings to be translatable. Setting this to `true` will suppress warnings about strings such as "*? - ?*" being available for localization.

Minimum words for a string to be considered translatable: enter into here the minimum number of words that a string must have to be considered translatable. Higher values for this will result in less strings being classified as `notL10NAvailable` warnings.

Note
Words in this context are values that contain at least one letter; numbers, punctuation, and syntax commands are ignored. For example, "Printing %s," "Player 1," and "Add +" will be seen as one-word strings.

C++ standard when issuing deprecation warnings: enter into here the C++ standard that should be assumed when issuing deprecated macro warnings (`deprecatedMacro`). For example, setting this to `2017` or higher (along with selecting the **Include verbose warnings** option) will issue warnings for `WXUNUSED`, as it can be replaced with `[[maybe_unused]]`. If setting this to `2014`, however, `WXUNUSED` will be ignored since `[[maybe_unused]]` requires C++17.

Include verbose warnings: select this to perform additional checks, including those that may not be related to i18n/l10n issues. This will also include additional information in the **Analysis Log** window.

9.3 Resource Files

The following options are available for resource files (i.e., *.po and *.rc).

Translation catalogs

Check for inconsistent printf format specifiers: select this to check for mismatching `printf()` and positional commands between source and translation strings. (This will review "c-format," "kde-format," and "qt-format" strings in *gettext* files.)

⚠ **Warning**
The checks performed here are strict; all `printf()` commands in translations must match their source counterpart exactly. For example, "*%lu*" vs. "*%l*" will emit a warning. Questionable commands such as "*% s*" (space is only meant for numeric formatting) will also emit a warning.

Check for inconsistent keyboard accelerators: select this to check for mismatching keyboard accelerators between source and translation strings. This is performed by checking that both strings contain one '&' followed by an alphanumeric character.

Check for inconsistent casing or trailing punctuation, spaces, or newlines: select this to check for mismatching trailing punctuation, spaces, or newlines between source and translation strings. This will also emit a warning if the source starts with an uppercase letter and the translation starts with a lowercase letter.

In regards to punctuation, terminating characters such as periods, exclamation points, and question marks are reviewed. For these, warnings will only be emitted if the source string has a terminating mark and the translation does not have either a terminating mark or closing parenthesis. All types of terminating marks are treated interchangeably, as not all languages commonly use question and exclamation marks as much as English.

Review fuzzy translations: select this to review fuzzy translations. This should only be selected if you intend to review translations that still require approval and are likely unfinished.

Translation catalogs (*continued...*)

Pseudo-translation

While analyzing translation catalogs, copies of them can also be created and be pseudo-translated. Later, these catalogs can be loaded by your application for integration testing.

Method

This option specifies how to generate pseudo-translations.

None (do not generate anything): instructs the program to not generate any files.

UPPERCASE: translations will be uppercased variations of the source strings.

European characters: letters and numbers from the source string will be replaced with accented variations. The translation will be clearly different, but still generally readable.

Fill with 'X'es: letters from the source string will be replaced with either 'x' or 'X' (matching the original casing). This will produce the most altered appearance for the pseudo-translations.

During integration testing, these pseudo-translations will be easier to spot, but will make navigating the UI more difficult. This is recommended for the quickest way to interactively find UI elements that are not being made available for translation.

Add surrounding brackets: select this option to add brackets around the pseudo-translations. This can help with identifying truncation and strings being pieced together at runtime.

Add tracking IDs: select this to add a unique ID number (inside or square brackets) in front of every pseudo-translation. This can help with finding where a particular translation is coming from.

Increase width: select how much wider (between 0–100%) to make pseudo-translations. This widening is done by padding the strings with hyphens on both sides.

The default is 40, as "German and Spanish translations can sometimes take 40% more screen space than English" (Boyero).

Note
If surrounding brackets or tracking IDs are being included, then their lengths will be factored into the increased width calculation.

Windows RC files

Check for font issues: select this to check for dialogs that use unusual font sizes or are not using 'MS Shell Dlg'.

Note
Some static analysis options from the **Source Files** section will also be used while analyzing the source strings in these resource files.

9.4 Additional Checks

The following additional options are available for C/C++ source files. These options do not relate to internationalization, but are offered as supplemental checks for code formatting and other issues.

Formatting & encoding checks

Non-UTF8 encoded files: select this to check that files containing extended ASCII characters are UTF-8 encoded. UTF-8 is recommended for compiler portability.

BOM/UTF-8 signatures: select this to check for UTF-8 encoded files which start with a BOM/UTF-8 signature. It is recommended to save without the file signature for best compiler portability.

Unencoded extended ASCII characters: select this to check for strings containing extended ASCII characters that are not encoded.

Trailing spaces: select this to check for trailing spaces at the end of each line.

Tabs: select this to check for tabs. (Spaces are recommended as tabs may appear differently between editors.)

Overly long lines: select this to check for overly long lines.

Comments not starting with a space: select this to check that there is a space at the start of a comment.

Code checks

Hard-coded ID numbers: select this to check for ID variables being assigned a hard-coded number. It may be preferred to assign framework-defined constants (e.g., `wxID_HIGHEST`) to IDs.

ID variables assigned the same value: select this to check for the same value being assigned to different ID variables. This check is performed per file; the same ID being assigned multiple times, but within separate files, will be ignored.

Part III

Additional Features

Chapter 10

Inline Suppression

10.1 Code Blocks

Warnings can be suppressed for blocks of source code by placing `cuneiform-suppress-begin` and `cuneiform-suppress-end` comments around them. For example:

```
// cuneiform-suppress-begin
if (_debug && allocFailed)
    AfxMessageBox("Allocation failed!");
// cuneiform-suppress-end
```

This will prevent a warning from being emitted about “Allocation failed” not being available for localization.

⚠ **Warning**
The comment style of the begin and end tags must match. For example, if using multi-line comments (i.e., `/**/`), then both tags must be inside of `/**/` blocks.

10.2 Individual Strings

To instruct the program that a particular string is not meant for localization, wrap it inside of a `_DT()` or `DONTTRANSLATE()` function call. By doing this, the program will not warn about it not being available for localization. For example, both strings in the following will be ignored:

```
if (allocFailed)
    MessageBox(DONTTRANSLATE("Allocation failed!"));
else
    WriteMessage(_DT("No memory issues"));
```

You can either include the file “donttranslate.h” into your project to add these functions, or you can define your own macro:

```
#define DONTTRANSLATE(x) (x)
#define _DT(x) (x)
```

If using the versions provided in “donttranslate.h,” additional arguments can be included to explain why strings should not be available for translation. For example:

```
const std::string fileName = "C:\\data\\logreport.txt";

// "open " should not be translated, it's part of a command line
auto command = DONTTRANSLATE("open ") + fileName;
// expands to "open C:\\data\\logreport.txt"

// a more descriptive approach
auto command2 = DONTTRANSLATE("open ", DTExplanation::Command) + fileName;
// also expands to "open C:\\data\\logreport.txt"

// an even more descriptive approach
auto command3 = DONTTRANSLATE("open ",
                              DTExplanation::Command,
                              "This is part of a command line, "
                              "don't expose for translation!") +
                fileName;
// also expands to "open C:\\data\\logreport.txt"

// a shorthand, _DT(), is also available
auto command = _DT("open ") + fileName;
```

Note
`_DT()` and `DONTTRANSLATE()` do not have any effect on the code and would normally be compiled out. Their purpose is only to inform `Cuneiform` that their arguments should not be translatable.

Appendix A

Acknowledgements

Open-Source Libraries

The 3rd party, open-source libraries that *Cuneiform* includes (or links with) are listed below (along with their respective licenses).

cxxopts

C++11 library for parsing command line options.

Copyright (c) 2014 Jarryd Beck

MIT License

UTF8-CPP

UTF-8 with C++ in a portable way.

Copyright 2006 Nemanja Trifunovic

Boost Software License 1.0

wxWidgets

A stable and powerful open source framework for developing native cross-platform GUI applications in C++.

Copyright (c) 1998-2005 Julian Smart, Robert Roebling et al

wxWindows Library Licence, Version 3.1

Artwork

Photo by Riccardo Trimeloni on Unsplash

Photo by Fionn Große on Unsplash

References

Boyero, Virginia. “Design and evolution of the localization pipeline in Snowdrop”. *MultiLingual*, vol. 31, no. 2, 2020, p. 37.

cppreference.com. printf, fprintf, sprintf, snprintf, printf_s, fprintf_s, sprintf_s, snprintf_s. en.cppreference.com/w/c/io/fprintf.

Dunn, Michael. The Complete Guide to C++ Strings, Part I - Win32 Character Encodings. www.codeproject.com/Articles/2995/The-Complete-Guide-to-C-Strings-Part-I-Win32-Chara.

Haible, Bruno, and Daiki Ueno. *GNU gettext*, www.gnu.org/software/gettext/manual/gettext.html.

KDE. Development/Tutorials/Localization/i18n Mistakes. techbase.kde.org/Development/Tutorials/Localization/i18n_Mistakes.

———. Translations / i18n. develop.kde.org/docs/plasma/widget/translations-i18n/.

Microsoft. printf_p positional parameters. learn.microsoft.com/en-us/cpp/c-runtime-library/printf-p-positional-parameters?view=msvc-170.

———. STRINGTABLE resource. learn.microsoft.com/en-us/windows/win32/menurc/stringtable-resource.

———. TN020: ID Naming and Numbering Conventions. learn.microsoft.com/en-us/cpp/mfc/tn020-id-naming-and-numbering-conventions?view=msvc-170.

———. Working with Strings. learn.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings.

———. Working with Strings. learn.microsoft.com/en-us/windows/win32/intl/using-ms-shell-dlg-and-ms-shell-dlg-2.

Qt Company, The. Writing Source Code for Translation. doc.qt.io/qt-5/i18n-source-translation.html.

wxWidgets. wxTranslations Class Reference. docs.wxwidgets.org/latest/classwx_translations.html.