Guide to the Qt Translation Tools

Qt 3.0

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

Introduction

Qt provides excellent support for translating applications into local languages. This Guide explains how to use Qt's translation tools for each of the roles involved in translating an application. The Guide begins with a brief overview of the issues that must be considered, followed by chapters devoted to each role and the supporting tools provided.

Chapter 2: Release Manager is aimed at the person with overall responsibility for the release of the application. They will typically coordinate the work of the software engineers and the translator. The chapter describes the use of two tools. The lupdate tool is used to synchronize source code and translations. The lrelease tool is used to create runtime translation files for use by the released application.

Chapter 3: Translators is for translators. It describes the use of the *Qt Linguist* tool. No computer knowledge beyond the ability to start a program and use a text editor or word processor is required.

Chapter 4: Programmers is for Qt programmers. It explains how to create Qt applications that are able to use translated text. It also provides guidance on how to help the translator identify the context in which phrases appear. This chapter's three short tutorials cover everything the programmer needs to do.

Overview of the Translation Process

Most of the text that must be translated in an application program consists of either single words or short phrases. These typically appear as window titles, menu items, pop-up help text (balloon help), and labels to buttons, check boxes and radio buttons.

The phrases are entered into the source code by the programmer in their native language using a simple but special syntax to identify that the phrases require translation. The Qt tools provide context information for each of the phrases to help the translator, and the programmer is able to add additional context information to phrases when necessary. The release manager generates a set of translation files that are produced from the source files and passes these to the translator. The translator opens the translation files using *Qt Linguist*, enters their translations and saves the results back into the translation files, which they pass back to the release manager. The release manager then generates fast compact versions of these translation files ready for use by the application. The tools are designed to be used in repeated cycles as applications change and evolve, preserving existing translations and making it easy to identify which new translations are required. *Qt Linguist* also provides a phrase book facility to help ensure consistent translations across multiple applications and projects.

Translators and programmers must address a number of issues because of the subtleties and complexities of human language:

- A single phrase may need to be translated into several different forms depending on context, e.g. *open* in English might become *öffnen*, "open file", or *aufbauen*, "open internet connection", in German.
- Keyboard accelerators may need to be changed but without introducing conflicts, e.g. "&Quit" in English becomes "Avslutt" in Norwegian which doesn't contain a "Q". We cannot use a letter that is already in use — unless we change several accelerators.
- Phrases that contain variables, for example, "The 25 files selected will take 63 seconds to process", where the two numbers are inserted programmatically at runtime may need to be reworded because in a different language the word order and therefore the placement of the variables may have to change.

The Qt translation tools provide clear and simple solutions to these issues.

Please send comments and suggestions regarding this tutorial to the Qt doc team. Bugs in the tools should be sent to qt-bugs.

Release Manager

Two tools are provided for the release manager, lupdate and lrelease. These tools depend on *qmake* project files. You don't have to use *qmake*, though.

Qt Project Files

lupdate and lrelease depend on information in the application's `.pro` Qt project file. There must be an entry in the `TRANSLATIONS` section of the project file for each language that is additional to the native language. A typical entry looks like this:

```
TRANSLATIONS    = tt2_fr.ts \
                  tt2_nl.ts
```

Using a locale within the translation file name is useful for determining which language to load at runtime. This is explained in Chapter 4: Programmers.

An example of a complete `.pro` file with four translation source files:

```
HEADERS         = main-dlg.h \
                  options-dlg.h
SOURCES         = main-dlg.cpp \
                  options-dlg.cpp \
                  main.cpp
FORMS           = search-dlg.ui
TRANSLATIONS    = superapp_dk.ts \
                  superapp_fi.ts \
                  superapp_no.ts \
                  superapp_se.ts
```

QApplication::setDefaultCodec() makes it possible to choose a 8-bit encoding for literal strings that appear within `tr()` calls. If no encoding is set, `tr()` uses Latin-1.

If you use the QApplication::defaultCodec() mechanism in your application, *Qt Linguist* needs you to set the `DEFAULTCODEC` entry in the `.pro` file as well. For example:

```
DEFAULTCODEC    = ISO-8859-5
```

lupdate

Usage: `lupdate myproject.pro`

This is a simple command line tool. lupdate reads a Qt `.pro` project file and produces or updates the `.ts` translation source files listed in the project file. The translation files are given to the translator who uses *Qt Linguist* to read the files and insert the translations.

Companies that have their own translators in-house may find it useful to run lupdate regularly, perhaps monthly, as the application develops. This will lead to a fairly low volume of translation work spread evenly over the life of the project and will allow the translators to support a number of projects simultaneously.

Companies that hire in translators as required may prefer to run lupdate only a few times in the application's life cycle, the first time might be just before the first test phase. This will provide the translator with a substantial single block of work and any bugs that the translator detects may easily be included with those found during the initial test phase. The second and any subsequent lupdate runs would probably take place during the final beta phase.

The `.ts` file format is a simple human-readable XML format that can be used with version control systems if required.

lrelease

Usage: `lrelease myproject.pro`

This is another simple command line tool. It reads a Qt `.pro` project file and produces the `.qm` files used by the application, one for each `.ts` translation source file listed in the project file. The `.qm` file format is a compact binary format that provides extremely fast lookups for translations.

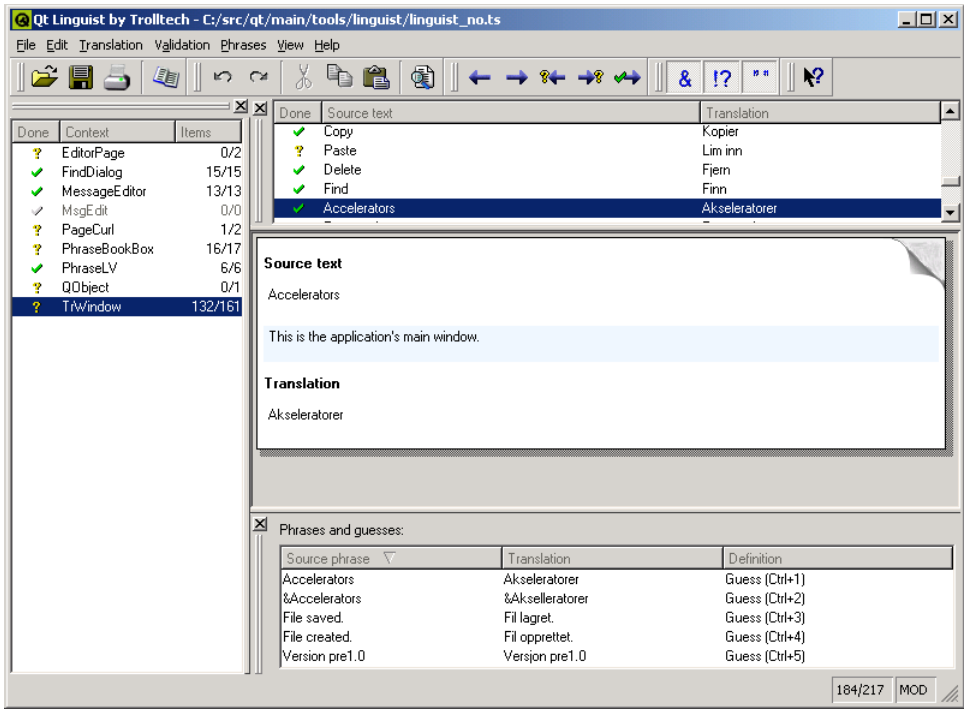
This tool is run whenever a release of the application is to be made, from initial test version through to final release version. If the `.qm` files are not created, e.g. because an alpha release is required before any translation has been undertaken, the application will run perfectly well using the text the programmers placed in the source files. Once the `.qm` files are available the application will detect them and use them automatically.

Note that lrelease will only incorporate translations that are marked as "done". If a translation is missing, or has failed validation, the original text will be used instead.

Missing Translations

Both lupdate and lrelease may be used with `.ts` translation source files which are incomplete. Missing translations will be replaced with the native language phrases at runtime.

Translators



Linguist Main Window

The One Minute Guide to Using Qt Linguist

Qt Linguist is a tool for adding translations to Qt applications. It introduces the concept of a translation "context" which means a group of phrases that appear together on the screen e.g. in the same menu or dialog.

To start, run *Qt Linguist*, either from the taskbar menu, or by double clicking the desktop icon, or type `linguist` (followed by **Enter**) at the command line. Once *Qt Linguist* has started choose **File|Open** from the menu bar and select a `.ts` translation source file to work on.

Qt Linguist's main window is divided into four main areas. The left hand side contains the Context list, the top right is the Source text area, the middle right is the translation area and the bottom right is the phrases and guesses area. We'll describe them in detail later.

Click on one of the contexts in the context list (left hand side) and then click on one of the phrases that appears in the Source text area (top right). The phrase will be copied into the translation area (middle right). Click under the word 'Translation' and type in the translation. Click **Ctrl+Enter** (Done & Next) to confirm that you have completed the translation and to move on to the next phrase that requires translation.

The cycle of entering a translation then pressing **Ctrl+Enter** can be repeated until all the translations are done or until you finish the session. Linguist will attempt to fill the "phrases and guesses" area with possible translations from any open phrase books and any previous translations. Each has a keyboard shortcut, e.g. **Ctrl+1**, **Ctrl+2**,

etc., which you can use to copy the guess into the Translation area. (Mouse users can double click a phrase or guess to move it into the Translation area.) At the end of the session choose **File|Save** from the menu bar and then **File|Exit** to quit.

Qt Linguist's Main Window

Context List

This appears at the left hand side of the main window by default. The first column, 'Done', identifies whether or not the translations for the context have been done. A tick indicates that all the translations have been done and are valid. A question mark indicates that one or more translations have not been done or have failed validation. The second column, 'Context' is the name of the context in which the translation phrases appear. The third column, 'Items' shows two numbers, the first is the number of translations that have been done, and the second is the number of phrases that are in the context; if the numbers are equal then all the translations have been done. Note that a greyed out tick indicates an obsolete translation, i.e. a phrase that was translated in a previous version of the application but which does not occur in the new version.

The contexts are ordered alphabetically. The phrases within each context are in the order in which they appear in the source program and this may not be the order in which they are shown on screen.

The Context List is a dockable window so it can be dragged to another position in the main window, or dragged out of the main window to be a window in its own right. If you move the Context List, *Qt Linguist* will remember its position and restore it whenever you start the program.

Source Text Area

This appears at the top right of the main window by default. The first column, 'Done', signifies the status of the translation. A tick indicates that the phrase has been translated and passed validation. A question mark indicates that the translation has not been done. An exclamation mark indicates that the translation has failed validation. The second column 'Source text' shows the text that must be translated. The third column shows the translation.

Qt Linguist provides three kinds of validation: accelerator, punctuation and phrase. If the source text contains an accelerator i.e. an ampersand, '&' and the translated text does not contain an ampersand the translation will fail the accelerator validation. Similarly, if the source text ends with a particular punctuation mark, e.g. '?', '!' or '.' and the translation ends with a different punctuation mark the translation will fail the punctuation validation. If the source text has a translation in one of the open phrase books that differs from the translation used the translation will fail phrase validation. (See Validation.)

The Source Text Area is a dockable window.

Translation Area

This area appears at the middle right of the main window by default. It is comprised of three vertical sections. The first section is labelled 'Source text' below which the source text appears. The second section contains contextual information on a light blue background that the programmer has added to assist the translator. If no contextual information has been given this section does not appear. The third section is labelled 'Translation' and this is where you enter the translation of the source text.

Phrases and Guesses Area

This area appears at the bottom right of the main window by default. When you move to a new phrase if the phrase is in one of the phrase books that has been loaded the phrase will appear in this area with its translation. If the phrase is the same or similar to another phrase that has already been translated the phrase and translation will be shown in this area. To copy a translation from the phrases and guesses area press **F6** to move to the phrases

and guesses area, use the up and down arrow keys to move to the phrase you want to use and press Enter to copy it. If you decide that you don't want to copy a phrase after all, press Esc. In both cases the focus will return to the Translation area. Alternatively, double click the translation you want to use and it will be copied into the translation area.

The Phrases and Guesses Area is a dockable window.

Common Tasks

Leaving a Translation for Later

If you wish to leave a translation press **Ctrl+L** (Next Unfinished) to move to the next unfinished translation. An unfinished translation is one that either has not been translated at all or one which fails validation. To move to the next phrase press **Shift+Ctrl+L**. You can also navigate using the Translation menu. If you want to go to a different context entirely, click the context you want to work on in the Context list, then click the source text in the Source Text area.

Phrases That Require Multiple Translations Depending on Context

The same phrase may occur in two or more contexts without conflict. Once a phrase has been translated in one context, *Qt Linguist* notes that the translation has been made and when the translator reaches a later occurrence of the same phrase *Qt Linguist* will provide the previous translation as a possible translation candidate in the phrases and guesses area. If the previous translation is acceptable just click the *Done & Next* button (press **Alt+Enter**) to move on to the next unfinished phrase.

If a phrase occurs more than once in a particular context it will only be shown once in *Qt Linguist*'s context list and the translation will be applied to every occurrence within the context. If the same phrase needs to be translated differently within the same context the programmer must provide a distinguishing comment for each of the phrases concerned. If such comments are used the duplicate phrases will appear in the context list. The programmers comments will appear in the translation area on a light blue background.

Changing Keyboard Accelerators

A keyboard accelerator is a key combination that when pressed will cause an application to perform an action. Keyboard accelerators normally come in two forms: Alt key and Ctrl key accelerators.

Alt key accelerators are used for menus and buttons. The underlining signifies that pressing the Alt key with the underlined letter is the same as clicking the menu item with the mouse. For example, most applications have a *File* menu with the "F" in the word "file" underlined. In these applications the file menu can be invoked either by clicking the word "File" on the menu bar or by pressing Alt+F. The accelerator key which is underlined is signified by preceeding it with an ampersand, e.g. &File. If a source phrase appears with an ampersand in it then the translation should also contain an ampersand, preferably in front of the same letter. The meaning of Alt key accelerators can be determined from the phrase in which the ampersand is embedded. The translator may need to change the letter used with the Alt key, e.g. if the translated phrase does not contain the original accelerator letter. Conflicts with other keys, i.e. having two Alt key accelerators using the same letter in the same context, must be avoided. Note that some Alt key accelerators, usually those on the menu bar, may apply in other contexts.

Ctrl key accelerators can exist independently of any visual control. They are often used to invoke actions in menus that would otherwise take several keystrokes or mouse clicks. They may also be used to perform actions that do not appear in any menu or on any button. For example, most applications that have a *File* menu have a submenu item called *New*. In many applications this will appear as "New... Ctrl+N". This menu option could be invoked by clicking *File* then clicking *New* with the mouse. Or you could press Alt+F then press N since these letters are underlined. But the same thing can be achieved simply by pressing **Ctrl+Enter**. Accelerators that use the Ctrl key are shown literally in the source text, e.g. **Ctrl+Enter**. Ctrl key accelerators have no phrase so the translator must rely on the programmer to add a "comment" which appears in the top right hand pane. This comment should

explain what action the Ctrl key accelerator performs. Ideally Ctrl key accelerators are translated simply by copying them by clicking the *Begin from Source* button. However in some cases the letter will not make sense in the target language and must be changed. Whatever letter (or digit) is chosen, the translation should always be in the form "Ctrl+" followed by the letter or digit in upper case. As with Alt key accelerators, if the translator changes the key it must not conflict with any other Ctrl key accelerator.

Later versions of *Qt Linguist* are expected to help the translator avoid accelerator conflicts.

Dealing with Phrases that Contain Variables

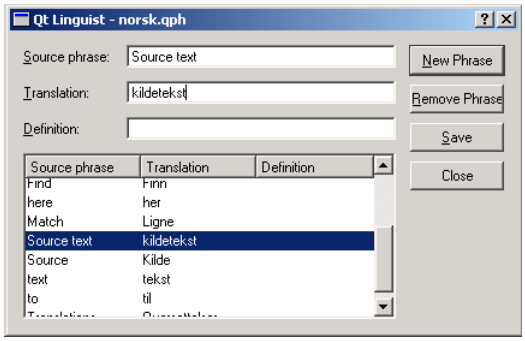
Some phrases contain variables. Variables are placeholders for items of text that are filled in at runtime. They are signified in the source text with a percent sign followed by a digit, e.g. *After processing file %1, file %2 is next in line*. In this example, %1 will be replaced at runtime with the name of the first file to be processed and %2 with the name of the next file to be processed. In the translated version the variables must still appear. For example a German translation might reverse the phrases, e.g. *Datei %2 wird bearbeitet, wenn. Datei %1 fertig ist*. Note that both variables are still used but their order has changed. The order in which variables appear does not matter; %1 will always be replaced by the same text at runtime no matter where it appears in the source text or translation and similarly %2, etc.

Reusing Translations

If the translated text is similar to the source text, click the *Begin from Source* button (press **Alt+T**) which will copy the source text into the translation area.

Qt Linguist automatically lists phrases from the open phrase books and similar or identical phrases that have already been translated in the Phrases and guesses area.

Creating and Using Phrase Books



Phrase Book Dialog

A *Qt Linguist* phrase book is a set of source phrases, target (translated) phrases, and optional definitions. Phrase Books are created independently of any application, although typically one phrase book will be created per application or family of applications.

If the translator reaches an untranslated phrase that is the same as a source phrase in the phrase book, *Qt Linguist* will show the phrase book entry in the *Relevant phrases* panel at the bottom right of the main window. Phrases which have translations that conflict with those given in the phrase book are marked with a question mark in the source text pane. Phrase Books are used to provide a common set of translations to help ensure consistency. They can also be used to avoid duplication of effort since the translations for a family of applications can be produced once in the phrase book and the phrase book used for the majority of translations in each application.

Before a phrase book can be edited it must be created or if it already exists, opened. Create a new phrase book by selecting **Phrase|New Phrase Book** from the menu bar. You must enter a filename and may change the location

of the file if you wish. A newly created phrase book is automatically opened. Open an existing phrase book by choosing **Phrase|Open Phrase Book** from the menu bar.

To add a new phrase click the **New Phrase** button (or press Alt+N) and type in a new source phrase. Press Tab and type in the translation. Optionally press Tab and enter a definition — this is useful to distinguish different translations of the same source phrase. This process may be repeated as often as necessary.

You can delete a phrase by selecting it in the phrases list and clicking Remove Phrase.

Click the *Save* button (press Alt+S) and then click the *Close* button (press Esc) once you've finished adding (and removing) phrases.

When a phrase or set of phrases appears in the phrase book double clicking the required target phrase will copy it to the translation pane at the text cursor position. If you want to *replace* the text in the translation pane with the target phrase, click the translation pane, choose **Edit|Select All** (press **Alt+A**) and then double click the target phrase.

Validation

Qt Linguist provides three kinds of validation on translated phrases.

1. *Accelerator validation* detects translated phrases that do not have an ampersand when the source phrase does and vice versa.
2. *Punctuation validation* detects differences in the terminating punctuation between source and translated phrases when this may be significant, e.g. warns if the source phrase ends with an ellipsis, exclamation mark or question mark, and the translated phrase doesn't and vice versa.
3. *Phrases validation* detects source phrases that are also in the phrase book but whose translation differs from that given in the phrase book.

Validation may be switched on or off from the menu bar's Validation item or using the toolbar buttons. Phrases that fail validation are marked with a question mark in the source text pane. If you switch validation off and then switch it on later, *Qt Linguist* will recheck all phrases and mark any that fail validation.

If any phrase in a context is invalid then the context itself will be marked with a question mark; if all the phrases in a context are done and are valid the context will be marked with a tick.

Note that only phrases which are marked as done (with a tick) will appear in the application. Invalid phrases and phrases which are translated but not marked as done are kept in the translation source file but are not used by the application.

Qt Linguist Reference

File Types

Qt Linguist makes use of three kinds of file:

- `.ts` *translation source files* are human-readable XML files containing source phrases and their translations. These files are usually created and updated by lupdate and are specific to an application.
- `.qm` *Qt message files* are binary files that contain translations used by an application at runtime. These files are generated by lrelease, but can also be generated by *Qt Linguist*.
- `.qph` *Qt phrase book files* are human-readable XML files containing standard phrases and their translations. These files are created and updated by *Qt Linguist* and may be used by any number of projects and applications.

The Menu Bar

Menu Bar

- *File*
  - *Open... Ctrl+O* pops up an open file dialog from which a translation source `.ts` file can be chosen.
  - *Save Ctrl+S* saves the current translation source `.ts` file.
  - *Save As...* pops up a save as file dialog so that the current translation source `.ts` file may be saved with a different name and/or put in a different location.
  - *Release...* pops up a save as file dialog. The filename entered will be a Qt message `.qm` file of the translation based on the current translation source file. The release manager's command line tool lrelease performs the same function on *all* of an application's translation source files.
  - *Print... Ctrl+P* pops up a print dialog. If you click OK the translation source and the translations will be printed.
  - *Recently opened files* shows the `.ts` files that have been opened recently, click one to open it.
  - *Exit Ctrl+Q* closes *Qt Linguist*.

Edit

- – *Undo Ctrl+Z* undoes the last editing action in the translation pane.
  - *Redo Ctrl+Y* redoes the last editing action in the translation pane.
  - *Cut Ctrl+X* deletes any highlighted text in the translation pane and saves a copy to the clipboard.
  - *Copy Ctrl+C* copies the highlighted text in the translation pane to the clipboard.
  - *Paste Ctrl+V* pastes the clipboard text into the translation pane.
  - *Select All Ctrl+A* selects all the text in the translation pane ready for copying or deleting.
  - *Find... Ctrl+F* pops up the Find dialog. When the dialog pops up enter the text to be found and click the *Find Next* button. Source phrases, translations and comments may be searched.
  - *Find Next F3* finds the next occurrence of the text that was last entered in the Find dialog.
- *Translation*
  - *Prev Unfinished Ctrl+K* moves to the nearest previous unfinished source phrase (unfinished means untranslated or translated but failed validation).
  - *Next Unfinished Ctrl+L* moves to the next unfinished source phrase.
  - *Prev Shift+Ctrl+K* moves to the previous source phrase.
  - *Next Shift+Ctrl+L* moves to the next source phrase.
  - *Done & NextCtrl+Enter* mark this phrase as 'done' (translated) and move to the next unfinished source phrase.
  - *Begin from Source Ctrl+B* copies the source text into the translation.

Validation (See the Validation section)

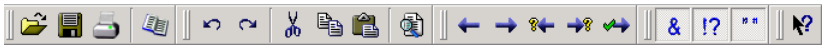
- – *Accelerators* toggles validation on or off for Alt accelerators.
  - *Ending Punctuation* switches validation on or off for phrase ending punctuation, e.g. ellipsis, exclamation mark, question mark, etc.
  - *Phrase Matches* sets validation on or off for matching against translations that are in the current phrase book.
- *Phrase* (See the section Creating and Using Phrase Books for details.)
  - *New Phrase Book... Ctrl+N* pops up a save as file dialog. You must enter a filename to be used for the phrase book and save the file. Once saved you should open the phrase book to begin using it.
  - *Open Phrase Book... Ctrl+H* pops up an open file dialog. Find and choose a phrase book to open.
  - *Close Phrase Book* closes the current phrase book. This will stop any further phrase validation taking place. The same effect can be achieved by switching off phrase validation using the Validation menu or the phrase toolbar button.

- *Edit Phrase Book...* pops up the phrase book dialog where you can add, edit or delete phrases.
- *Print Phrase Book...* pops up a print dialog. If you click OK the phrase book will be printed.

View

- *Revert Sorting* puts the phrases in the source text pane into their original order.
- *Display Guesses* turns the display of phrases and guesses on or off.
- *Views* toggles the visibility of the Context, Source text and Phrase views.
- *Toolbars* toggles the visibility of the different toolbars.

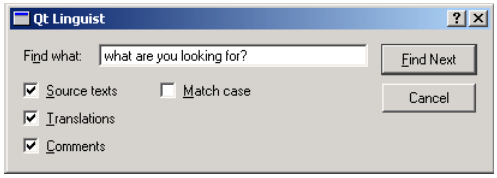
The Toolbar



Toolbar

- Pops up the open file dialog to open a new translation source `.ts` file.
- Saves the current translation source `.ts` file.
- Prints the current translation source `.ts` file.
- Pops up the file open dialog to open a new phrase book `.qph` file.
- Undoes the last editing action in the translation pane.
- Redoes the last editing action in the translation pane.
- Deletes any highlighted text in the translation pane and save a copy to the clipboard.
- Copies the highlighted text in the translation pane to the clipboard.
- Pastes the clipboard text into the translation pane.
- Pops up the Find dialog.
- Moves to the previous source phrase.
- Moves to the next source phrase.
- Moves to the previous unfinished source phrase.
- Moves to the next unfinished source phrase.
- Marks the phrase as 'done' (translated) and move to the next unfinished source phrase.
- Toggles accelerator validation on and off.
- Toggles phrase ending punctuation validation on and off.
- Toggles phrase book validation on or off.

The Find Dialog



The Find Dialog

Choose **Edit|Find** from the menu bar or press **Ctrl+F** to pop up the Find dialog. Press **F3** to repeat the last search. By default the source phrases, translations and comments will all be searched and the search will be case-insensitive. These settings can be changed by checking or unchecking the checkboxes to reflect your preferences.

The Phrase Dialog

This dialog is explained in the Creating and Using Phrase Books section.

Programmers

Support for multiple languages is extremely simple in Qt applications and adds little overhead to the programmer's workload.

Qt minimizes the performance cost of using translations by translating the phrases for each window as they are created. In most applications the main window is created just once. Dialogs are often created once and then shown and hidden as required. Once the initial translation has taken place there is no further runtime overhead for the translated windows. Only those windows that are created, destroyed and subsequently created will have a translation performance cost — although the overhead is still very low.

Creating applications that can switch language at runtime is possible with Qt, but requires a certain amount of programmer intervention and will of course incur some runtime performance cost.

Making the Application Translation Aware

Programmers should make their application look for and load the appropriate translation file and mark user-visible text and Ctrl keyboard accelerators as targets for translation.

Each piece of text that requires translating requires context to help the translator identify where in the program the text occurs. In the case of multiple identical texts that require different translations the translator also requires some information to disambiguate the source texts. Marking text for translation will automatically cause the class name to be used as basic context information. In some cases the programmer may be required to add additional information to help the translator.

Creating Translation Files

Translation files consist of all the user-visible text and Ctrl key accelerators in an application and translations of that text. Translation files are created as follows:

1. Run lupdate initially to generate the first set of `.ts` translation source files with all the user-visible text but no translations.
2. The `.ts` files are given to the translator who adds translations using *Qt Linguist*. *Qt Linguist* takes care of any changed or deleted source text.
3. Run lupdate to incorporate any new text added to the application. lupdate synchronizes the user-visible text from the application with the translations; it does not destroy any data.
4. Steps 2 and 3 are repeated as often as necessary.
5. When a release of the application is needed lrelease is run to read the `.ts` files and produce the `.qm` files used by the application at runtime.

For lupdate to work successfully, it must know which translation files to produce. The files are simply listed in the application's `.pro` Qt project file, for example:

```
TRANSLATIONS    = tt2_fr.ts \
                  tt2_nl.ts
```

See the "lupdate" and "lrelease" sections.

Loading Translations

```
int main( int argc, char **argv )
{
    QApplication app( argc, argv );
```

This is how a simple `main()` function of a Qt application begins.

```
int main( int argc, char **argv )
{
    QApplication app( argc, argv );

    QTranslator translator( 0 );
    translator.load( "tt1_la", "." );
    app.installTranslator( &translator );
```

For a translation-aware application a translator object is created, a translation is loaded and the translator object installed into the application.

```
int main( int argc, char **argv )
{
    QApplication app( argc, argv );

    QTranslator translator( 0 );
    translator.load( QString( "tt2_" ) + QTextCodec::locale(), "." );
    app.installTranslator( &translator );
```

In production applications a more flexible approach, for example, loading translations according to locale, might be more appropriate. If the `.ts` files are all named according to a convention such as *appname_locale*, e.g. `tt2_fr`, `tt2_de` etc, then the code above will load the current locale's translation at runtime.

If there is no translation file for the current locale the application will fall back to using the original source text.

Making the Application Translate User-Visible Strings

User-visible strings are marked as translation targets by wrapping them in a `tr()` call, for example:

```
button = new QPushButton( "&Quit", this );
```

would become

```
button = new QPushButton( tr( "&Quit" ), this);
```

All QObject subclasses that use the `Q_OBJECT` macro have a reimplementation of the `tr()` function.

Although the `tr()` call is normally made directly since it is usually called as a member function of a QObject subclass, in other cases an explicit class name can be supplied, for example:

```
QPushButton::tr( "&Quit" )
```

or

```
QObject::tr( "&Quit" )
```

Distinguishing Identical Strings That Require Different Translations

The lupdate program automatically provides a *context* for every source text. This context is the class name of the class that contains the `tr()` call. This is sufficient in the vast majority of cases. Sometimes however, the translator will need further information to uniquely identify a source text; for example, a dialog that contained two separate frames, each of which contained an "Enabled" option would need each identified because in some languages the translation would differ between the two. This is easily achieved using the two argument form of the `tr()` call, e.g.

```
rbc = new QRadioButton( tr( "Enabled", "Color frame" ), this);
```

and

```
rbh = new QRadioButton( tr( "Enabled", "Hue frame" ), this);
```

Ctrl key accelerators are also translatable:

```
    file->insertItem( tr( "E&xit" ), qApp, SLOT(quit()),
                      QAccel::stringToKey( tr("Ctrl+Q", "Quit") ) );
```

It is strongly recommended that the two argument form of `tr()` is used for Ctrl key accelerators. The second argument is the only clue the translator has as to the function performed by the accelerator.

Helping The Translator With Navigation Information

In large complex applications it may be difficult for the translator to see where a particular source text comes from. This problem can be solved by adding a comment using the keyword *TRANSLATOR* which describes the navigation steps to reach the text in question; e.g.

```
/*  TRANSLATOR FindDialog

    Choose Edit|Find from the menu bar or press Ctrl+F to pop up the
    Find dialog.
*/
```

These comments are particularly useful for widget classes.

Coping With C++ Namespaces

C++ namespaces and the `using namespace` statement can confuse lupdate. It will interpret `MyClass::tr()` as meaning just that, not as `MyNamespace::MyClass::tr()`, even if `MyClass` is defined in the `MyNamespace` namespace. Run-time translation of these strings will fail because of that.

You can work around this limitation by putting a *TRANSLATOR* comment at the beginning of the source files that use `MyClass::tr()`:

```
/* TRANSLATOR MyNamespace::MyClass */
```

After the comment, all references to `MyClass::tr()` will be understood as meaning `MyNamespace::MyClass::tr()`.

Tranlating Text that is Outside of a QObject subclass

Using QApplication::translate()

If the quoted text is not in a member function of a QObject subclass, use either the tr() function of an appropriate class, or the QApplication::translate() function directly:

```
void some_global_function( LoginWidget *logwid )
{
    QLabel *label = new QLabel(
            LoginWidget::tr("Password:"), logwid );
}

void same_global_function( LoginWidget *logwid )
{
    QLabel *label = new QLabel(
            qApp->translate("LoginWidget", "Password:"),
            logwid );
}
```

Using QT_TR_NOOP and QT_TRANSLATE_NOOP

If you need to have translatable text completely outside a function, there are two macros to help: QT_TR_NOOP() and QT_TRANSLATE_NOOP(). These macros merely mark the text for extraction by lupdate. The macros expand to just the text (without the context).

Example of QT_TR_NOOP():

```
QString FriendlyConversation::greeting( int greet_type )
{
    static const char* greeting_strings[] = {
        QT_TR_NOOP( "Hello" ),
        QT_TR_NOOP( "Goodbye" )
    };
    return tr( greeting_strings[greet_type] );
}
```

Example of QT_TRANSLATE_NOOP():

```
static const char* greeting_strings[] = {
    QT_TRANSLATE_NOOP( "FriendlyConversation", "Hello" ),
    QT_TRANSLATE_NOOP( "FriendlyConversation", "Goodbye" )
};

QString FriendlyConversation::greeting( int greet_type )
{
    return tr( greeting_strings[greet_type] );
}

QString global_greeting( int greet_type )
{
    return qApp->translate( "FriendlyConversation",
                            greeting_strings[greet_type] );
}
```

Tutorials

Three tutorials are presented. The first demonstrates the creation of a QTranslator object. It also shows the simplest use of the `tr()` function to mark user-visible source text for translation. The second tutorial explains how to make the application load the translation file applicable to the current locale. It also shows the use of the two-argument form of `tr()` which provides additional information to the translator. The third tutorial explains how identical

source texts can be distinguished even when they occur in the same context. This tutorial also discusses how the translation tools help minimize the translator's work when an application is upgraded.

Tutorial 1: Loading and Using Translations

Tutorial 1 Screenshot, English version

```
TEMPLATE        = app
CONFIG          += qt warn_on
SOURCES         = main.cpp
TRANSLATIONS    = tt1_la.ts
```

tt1.pro

```
/****************************************************************
**
** Translation tutorial 1
**
****************************************************************/

#include <qapplication.h>
#include <qpushbutton.h>
#include <qtranslator.h>


int main( int argc, char **argv )
{
    QApplication app( argc, argv );

    QTranslator translator( 0 );
    translator.load( "tt1_la", "." );
    app.installTranslator( &translator );

    QPushButton hello( QPushButton::tr( "Hello world!" ), 0 );

    app.setMainWidget( &hello );
    hello.show();
    return app.exec();
}
```

main.cpp

This example is a reworking of the "hello-world" example from the first Qt tutorial, with a Latin translation. The *Tutorial 1 Screenshot, English version*, above, shows the English version.

Line by Line Walk-through

```
    #include <qtranslator.h>
```

This line includes the definition of the QTranslator class. Objects of this class provide translations for user-visible text.

```
        QTranslator translator( 0 );
```

Creates a QTranslator object without a parent.

```
translator.load( "tt1_la", "." );
```

Try to load a file called `tt1_la.qm` (the `.qm` file extension is implicit) that contains Latin translations for the source texts used in the program. No error will occur if the file is not found.

```
app.installTranslator( &translator );
```

Add the translations from `tt1_la.qm` to the pool of translations used by the program. No error will occur if the file is not found.

```
QPushButton hello( QPushButton::tr( "Hello world!" ), 0 );
```

Creates a push button that displays "Hello world!". If `tt1_la.qm` was found and contains a translation for "Hello world!", the translation appears; if not, the source text appears.

All classes that inherit QObject have a `tr()` function. Inside a member function of a QObject class, we simply write `tr("Hello world!")` instead of `QPushButton::tr("Hello world!")` or `QObject::tr("Hello world!")`.

Running the Application in English

Since we haven't made the translation file `tt1_la.qm`, the source text is shown when we run the application:

Tutorial 1 Screenshot, English version

Creating a Latin Message File

The first step is to create a project file, `tt1.pro`, that lists all the source files for the project. The project file can be a qmake project file, or even an ordinary makefile. Any file that contains

```
SOURCES         = main.cpp
TRANSLATIONS    = tt1_la.ts
```

will work. *TRANSLATIONS* specifies the message files we want to maintain. In this example, we just maintain one set of translations, namely Latin.

Note that the file extension is `.ts`, not `.qm`. The `.ts` translation source format is designed for use during the application's development. Programmers or release managers run the lupdate program to generate and update `.ts` files with the source text that is extracted from the source code. Translators read and update the `.ts` files using *Qt Linguist* adding and editing their translations.

The `.ts` format is human-readable XML that can be emailed directly and is easy to put under version control. If you edit this file manually, be aware that the default encoding for XML is UTF-8, not Latin-1 (ISO 8859-1). One way to type in a Latin-1 character such as 'ø' (Norwegian o with slash) is to use an XML entity: "ø". This will work for any Unicode character.

Once the translations are complete the lrelease program is used to convert the `.ts` files into the `.qm` Qt message file format. The `.qm` format is a compact binary format designed to deliver very fast lookup performance. Both lupdate and lrelease read all the project's source and header files (as specified in the HEADERS and SOURCES lines of the project file) and extract the strings that appear in `tr()` function calls.

lupdate is used to create and update the message files (`tt1_la.ts` in this case) to keep them in sync with the source code. It is safe to run lupdate at any time, as lupdate does not remove any information. For example, you can put it in the makefile, so the `.ts` files are updated whenever the source changes.

Try running lupdate right now, like this:

```
lupdate tt1.pro
```

You should now have a file `tt1_la.ts` in the current directory, containing this

```
QPushButton

    Hello world!
```

You don't need to understand the file format since it is read and updated using tools, e.g. lupdate and *Qt Linguist*.

Translating to Latin with Qt Linguist

We will use *Qt Linguist* to provide the translation, although you can use any XML or plain text editor to enter a translation into a `.ts` file.

To start *Qt Linguist*, type

```
linguist tt1_la.ts
```

You should now see the text "QPushButton" in the top left pane. Double-click it, then click on "Hello world!" and enter "Orbis, te saluto!" in the *Translation* pane (the middle right of the window). Don't forget the exclamation mark!

Click the *Done* checkbox and choose *File*|*Save* from the menu bar. The `.ts` file will no longer contain

but instead will have

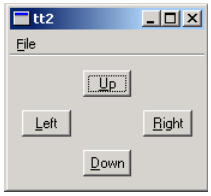
```
Orbis, te saluto!
```

Running the Application in Latin

To see the application running in Latin, we have to generate a `.qm` file from the `.ts` file. Generating a `.qm` file can be achieved either from within *Qt Linguist* (for a single `.ts` file), or by using the command line program lrelease which will produce one `.qm` file for each of the `.ts` files listed in the project file. Generate `tt1_la.qm` from `tt1_la.ts` by choosing *File*|*Release* from *Qt Linguist*'s menu bar and pressing *Save* in the file save dialog that pops up. Now run the *tt1* example program again. This time the button will be labelled "Orbis, te saluto!".

Tutorial 1 Screenshot, Latin version

Tutorial 2: Using Two or More Languages



Tutorial 2 Screenshot, English version

```
TEMPLATE        = app
CONFIG          += qt warn_on
HEADERS         = arrowpad.h \
                  mainwindow.h
SOURCES         = arrowpad.cpp \
                  main.cpp \
                  mainwindow.cpp
TRANSLATIONS    = tt2_fr.ts \
                  tt2_nl.ts
```

tt2.pro

This example is a slightly more involved and introduces a key *Qt Linguist* concept: "contexts".

- `arrowpad.h` contains the definition of `ArrowPad`, a custom widget;
- `arrowpad.cpp` contains the implementation of `ArrowPad`;
- `mainwindow.h` contains the definition of `MainWindow`, a subclass of QMainWindow
- `mainwindow.cpp` contains the implementation of `MainWindow`;
- `main.cpp` contains main().

We will use two translations, French and Dutch, although there is no effective limit on the number of possible translations that can be used with an application. The relevant lines of `tt2.pro` are

```
HEADERS         = arrowpad.h \
                  mainwindow.h
SOURCES         = arrowpad.cpp \
                  main.cpp \
                  mainwindow.cpp
TRANSLATIONS    = tt2_fr.ts \
                  tt2_nl.ts
```

Run lupdate; it should produce two identical message files `tt2_fr.ts` and `tt2_nl.ts`. These files will contain all the source texts marked for translation with `tr()` calls and their contexts.

Line by Line Walk-through

In `arrowpad.h` we define the `ArrowPad` subclass which is a subclass of QWidget. In the *Tutorial 2 Screenshot, English version*, above, the central widget with the four buttons is an `ArrowPad`.

```
class ArrowPad : public QGrid
```

When lupdate is run it not only extracts the source texts but it also groups them into contexts. A context is the name of the class in which the source text appears. Thus, in this example, "ArrowPad" is a context: it is the context of the texts in the `ArrowPad` class. The `Q_OBJECT` macro defines `tr(x)` in `ArrowPad` like this

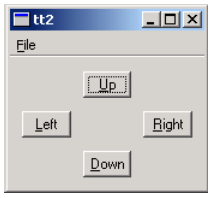
```
qApp->translate( "ArrowPad", x )
```

Knowing which class each source text appears in enables *Qt Linguist* to group texts that are logically related together, e.g. all the text in a dialog will have the context of the dialog's class name and will be shown together. This provides useful information for the translator since the context in which text appears may influence how it should be translated. For some translations keyboard accelerators may need to be changed and having all the source texts in a particular context (class) grouped together makes it easier for the translator to perform any accelerator changes without introducing conflicts.

In `arrowpad.cpp` we implement the `ArrowPad` class.

```
(void) new QPushButton( tr( "&Up" ), this );
```

We call `ArrowPad::tr()` for each button's label since the labels are user-visible text.



Tutorial 2 Screenshot, English version

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
```

In the *Tutorial 2 Screenshot, English version*, above, the whole window is a `MainWindow`. This is defined in the `mainwindow.h` header file. Here too, we use `Q_OBJECT`, so that `MainWindow` will become a context in *Qt Linguist*.

In the implementation of `MainWindow`, `mainwindow.cpp`, we create an instance of our `ArrowPad` class

```
ArrowPad *ap = new ArrowPad( this, "arrow pad" );
```

We also call `MainWindow::tr()` twice, once for the menu item and once for the accelerator.

```
file->insertItem( tr( "E&xit" ), qApp, SLOT(quit()),
                  QAccel::stringToKey( tr( "Ctrl+Q", "Quit" ) ) );
```

Note the use of QAccel::stringToKey() to support different keys in other languages. "Ctrl+Q" is a good choice for Quit in English, but a Dutch translator might want to use "Ctrl+A" (for Afsluiten) and a German translator "Strg+E" (for Beenden). When using `tr()` for Ctrl key accelerators the two argument form should be used with the second argument describing the function that the accelerator performs.

Our `main()` function is defined in `main.cpp` as usual.

```
QTranslator translator( 0 );
translator.load( QString( "tt2_" ) + QTextCodec::locale(), "." );
app.installTranslator( &translator );
```

We choose which translation to use according to the current locale. QTextCodec::locale() can be influenced by setting the `LANG` environment variable, for example. Notice that the use of a naming convention that incorporates the locale for `.qm` message files, (and `.ts` files), makes it easy to implement choosing the translation file according to locale.

If there is no `.qm` message file for the locale chosen the original source text will be used and no error raised.

Translating to French and Dutch

We'll begin by translating the example application into French. Start *Qt Linguist* with `tt2_fr.ts`. You should get the seven source texts ("&Up", "&Left", etc.) grouped in two contexts ("ArrowPad" and "MainWindow").

Now, enter the following translations:

- `ArrowPad`
  - &Up - &Haut
  - &Left - &Gauche
  - &Right - &Droite
  - &Down - &Bas
- `MainWindow`
  - E&xit - &Quitter
  - Ctrl+Q - Ctrl+Q
  - &File - &Fichier

It's quickest to press **Alt+D** (which clicks the *Done & Next* button) after typing each translation, since this marks the translation as done and moves on to the next source text.

Save the file and do the same for Dutch working with `tt2_nl.ts`:

- `ArrowPad`
  - &Up - &Boven
  - &Left - &Links
  - &Right - &Rechts
  - &Down - &Onder
- `MainWindow`
  - E&xit - &Afsluiten
  - Ctrl+Q - Ctrl+A
  - File - &Bestand

We have to convert the `tt1_fr.ts` and `tt1_nl.ts` translation source files into `.qm` files. We could use *Qt Linguist* as we've done before; however using the command line tool lrelease ensures that *all* the `.qm` files for the application are created without us having to remember to load and *File|Release* each one individually from *Qt Linguist*.

In practice we would include calls to lupdate and lrelease in the application's makefile to ensure that the latest translations are used.

Type

```
lrelease tt2.pro
```

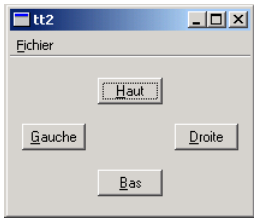
This should create both `tt2_fr.qm` and `tt2_nl.qm`. Set the `LANG` environment variable to `fr`. In Unix, one of the two following commands should work

```
export LANG=fr
setenv LANG fr
```

In Windows, either modify `autoexec.bat` or run

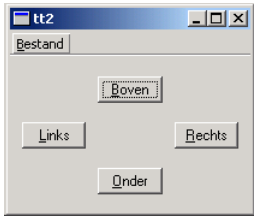
```
set LANG=fr
```

When you run the program, you should now see the French version:



Tutorial 2 Screenshot, French version

Try the same with Dutch, by setting `LANG=nl`. Now the Dutch version should appear:



Tutorial 2 Screenshot, Dutch version

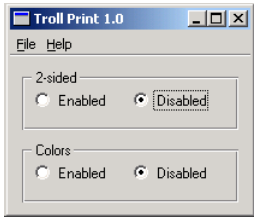
Exercises

Mark one of the translations in *Qt Linguist* as not done, i.e. by unchecking the "done" checkbox; run lupdate, then lrelease, then the example. What effect did this change have?

Set `LANG=fr_CA` (French Canada) and run the example program again. Explain why the result is the same as with `LANG=fr`.

Change one of the accelerators in the Dutch translation to eliminate the conflict between &Bestand and &Boven.

Tutorial 3: Disambiguating Identical Strings



Tutorial 3 Screenshot, "Troll Print 1.0", English version

```
TEMPLATE        = app
CONFIG          += qt warn_on
HEADERS         = mainwindow.h \
                  printpanel.h
SOURCES         = main.cpp \
                  mainwindow.cpp \
                  printpanel.cpp
TRANSLATIONS    = tt3_pt.ts
```

`tt3.pro`

We've included a translation file, `tt3_pt.ts`, which contains some Portuguese translations for this example.

We will consider two releases of the same application: Troll Print 1.0 and 1.1. We will learn to reuse the translations

created for one release in a subsequent release. (In this tutorial, you have to edit some source files. It's probably best to copy all the files to a new temporary directory and work from there.)

Troll Print is a toy example application that lets the user choose printer settings. It comes in two versions: English and Portuguese.

Version 1.0 consists of these files:

- `printpanel.h` contains the definition of PrintPanel;
- `printpanel.cpp` contains the implementation of PrintPanel;
- `mainwindow.h` contains the definition of `MainWindow`;
- `mainwindow.cpp` contains the implementation of `MainWindow`;
- `main.cpp` contains main();
- `tt3.pro` is the *qmake* project file.
- `tt3_pt.ts` is the Portuguese message file.

Line by Line Walk-through

The PrintPanel is defined in `printpanel.h`.

```
class PrintPanel : public QVBox
{
    Q_OBJECT
```

PrintPanel is a QWidget. It needs the `Q_OBJECT` macro for `tr()` to work properly.

The implementation file is `printpanel.cpp`.

```
/*
    QLabel *lab = new QLabel( tr( "TROLL PRINT" ), this );
    lab->setAlignment( AlignCenter );
*/
```

Some of the code is commented out in Troll Print 1.0; you will uncomment it later, for Troll Print 1.1.

```
    QHButtonGroup *twoSided = new QHButtonGroup( this );
    twoSided->setTitle( tr( "2-sided" ) );
    but = new QRadioButton( tr( "Enabled" ), twoSided );
    but = new QRadioButton( tr( "Disabled" ), twoSided );
    but->toggle();
    QHButtonGroup *colors = new QHButtonGroup( this );
    colors->setTitle( tr( "Colors" ) );
    but = new QRadioButton( tr( "Enabled" ), colors );
    but = new QRadioButton( tr( "Disabled" ), colors );
    but->toggle();
```

Notice the two occurrences of `tr("Enabled")` and of `tr("Disabled")` in PrintPanel. Since both "Enabled"s and "Disabled"s appear in the same context *Qt Linguist* will only display one occurrence of each and will use the same translations for the duplicates that it doesn't display. Whilst this is a useful timesaver, in some languages, such as Portuguese, the second occurrence requires a separate translation. We will see how *Qt Linguist* can be made to display all the occurrences for separate translation shortly.

The header file for `MainWindow`, `mainwindow.h`, contains no surprises. In the implementation, `mainwindow.cpp`, we have some user-visible source texts that must be marked for translation.

```
    setCaption( tr( "Troll Print 1.0" ) );
```

We must translate the window's caption.

```
file->insertItem( tr( "E&xit" ), qApp, SLOT(quit()),
                  QAccel::stringToKey( tr("Ctrl+Q", "Quit") ) );
QPopupMenu *help = new QPopupMenu( this );
help->insertItem( tr( "&About..." ), this, SLOT(about()), Key_F1 );
help->insertItem( tr( "About &Qt..." ), this, SLOT(aboutQt()) );

menuBar()->insertItem( tr( "&File" ), file );
menuBar()->insertSeparator();
menuBar()->insertItem( tr( "&Help" ), help );
```

We also need to translate the menu items. Note that the two argument form of `tr()` is used for the keyboard accelerator, "Ctrl+Q", since the second argument is the only clue the translator has to indicate what function that accelerator will perform.

```
QTranslator translator( 0 );
translator.load( QString( "tt3_" ) + QTextCodec::locale(), "." );
app.installTranslator( &translator );
```

The `main()` function in `main.cpp` is the same as the one in Tutorial 2. In particular it chooses a translation file based on the current locale.

Running Troll Print 1.0 in English and in Portuguese

We will use the translations in the `tt3_pt.ts` file that is provided.

Set the `LANG` environment variable to `pt`, and then run `tt3`. You should still see the English version, as shown in the *Tutorial 3 Screenshot, "Troll Print 1.0", English version*, above. Now run lrelease, e.g. `lrelease tt3.pro`, and then run the example again. Now you should see the Portuguese edition (Troll Imprimir 1.0):

Tutorial 3 Screenshot, "Troll Imprimir 1.0", (Bad) Portuguese version

Whilst the translation has appeared correctly, it is in fact wrong. In good Portuguese, the second occurrence of "Enabled" should be "Ativadas", not "Ativado" and the ending for the second translation of "Disabled" must change similarly too.

If you open `tt3_pt.ts` using *Qt Linguist*, you will see that there is just one occurrence of "Enabled" and of "Disabled" in the translation source file, even though there are two of each in the source code. This is because *Qt Linguist* tries to minimize the translator's work by using the same translation for duplicate source texts. In cases such as this where an identical translation is wrong, the programmer must disambiguate the duplicate occurrences. This is easily achieved by using the two argument form of `tr()`.

We can easily determine which file must be changed because the translator's "context" is in fact the class name for the class where the texts that must be changed appears. In this case the file is `printpanel.cpp`, where the there are four lines to change. Add the second argument "two-sided" in the appropriate `tr()` calls to the first pair of radio buttons:

```
but = new QRadioButton( tr("Enabled", "two-sided"), twoSided );
but = new QRadioButton( tr("Disabled", "two-sided"), twoSided );
```

and add the second argument "colors" in the appropriate `tr()` calls for the second pair of radio buttons:

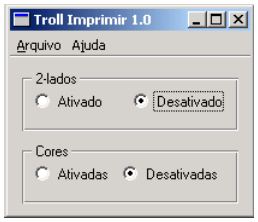
```
but = new QRadioButton( tr("Enabled", "colors"), colors );
but = new QRadioButton( tr("Disabled", "colors"), colors );
```

Now run lupdate and open `tt3_pt.ts` with *Qt Linguist*. You should now see two changes.

First, the translation source file now contains *three* "Enabled", "Disabled" pairs. The first pair is marked "(obs.)" signifying that they are obsolete. This is because these texts appeared in `tr()` calls that have been replaced by new calls with two arguments. The second pair has "two-sided" as their comment, and the third pair has "colors" as their comment. The comments are shown in the *Source text and comments* area in *Qt Linguist*.

Second, the translation text "Ativado" and "Desativado" have been automatically used as translations for the new "Enabled" and "Disabled" texts, again to minimize the translator's work. Of course in this case these are not correct for the second occurrence of each word, but they provide a good starting point.

Change the second "Ativado" into "Ativadas" and the second "Desativado" into "Desativadas", then save and quit. Run lrelease to obtain an up-to-date binary `tt3_pt.qm` file, and run Troll Print (or rather Troll Imprimir).



Tutorial 3 Screenshot, "Troll Imprimir 1.0", (Good) Portuguese version

The second argument to `tr()` calls, called "comments" in *Qt Linguist*, distinguish between identical source texts that occur in the same context (class). They are also useful in other cases to give clues to the translator, and in the case of Ctrl key accelerators are the only means of conveying the function performed by the accelerator to the translator.

An additional way of helping the translator is to provide information on how to navigate to the particular part of the application that contains the source texts they must translate. This helps them see the context in which the translation appears and also helps them to find and test the translations. This can be achieved by using a *TRANSLATOR* comment in the source code:

```
/* TRANSLATOR MainWindow

   In this application the whole application is a MainWindow.
   Choose Help|About from the menu bar to see some text
   belonging to MainWindow.
*/
```

Try adding these comments to some source files, particularly to dialog classes, describing the navigation necessary to reach the dialogs. You could also add them to the example files, e.g. `mainwindow.cpp` and `printpanel.cpp` are appropriate files. Run lupdate and then start *Qt Linguist* and load in `tt3_pt.ts`. You should see the comments in the *Source text and comments* area as you browse through the list of source texts.

Sometimes, particularly with large programs, it can be difficult for the translator to find their translations and check that they're correct. Comments that provide good navigation information can save them time:

```
/* TRANSLATOR ZClientErrorDialog

    Choose Client|Edit to reach the Client Edit dialog, then choose
    Client Specification from the drop down list at the top and pick
    client Bartel Leendert van der Waerden.  Now check the Profile
    checkbox and then click the Start Processing button.  You should
    now see a pop up window with the text "Error: Name too long!".
```

```
        This window is a ZClientErrorDialog.
    */
```

Troll Print 1.1

We'll now prepare release 1.1 of Troll Print. Start your favorite text editor and follow these steps:

- Uncomment the two lines that create a QLabel with the text "TROLL PRINT" in `printpanel.cpp`.
- Word-tidying: Replace "2-sided" by "Two-sided" in `printpanel.cpp`.
- Replace "1.0" with "1.1" everywhere it occurs in `mainwindow.cpp`.
- Update the copyright year to 1999-2000 in `mainwindow.cpp`.

(Of course the version number and copyright year would be consts or #defines in a real application.)

Once finished, run lupdate, then open `tt3_pt.ts` in *Qt Linguist*. The following items are of special interest:

- `MainWindow`
  - Troll Print 1.0 - marked "(obs.)", obsolete
  - About Troll Print 1.0 - marked "(obs.)", obsolete
  - Troll Print 1.0. Copyright 1999 Macroshaft, Inc. - marked "(obs.)", obsolete
  - Troll Print 1.1 - automatically translated as "Troll Imprimir 1.1"
  - About Troll Print 1.1 - automatically translated as "Troll Imprimir 1.1"
  - Troll Print 1.1. Copyright 1999-2000 Macroshaft, Inc. - automatically translated as "Troll Imprimir 1.1. Copyright 1999-2000 Macroshaft, Inc."

- `PrintPanel`
  - 2-sided - marked "(obs.)", obsolete
  - TROLL PRINT - unmarked, i.e. untranslated
  - Two-sided - unmarked, i.e. untranslated.

Notice that lupdate works hard behind the scenes to make revisions easier, and it's pretty smart with numbers.

Go over the translations in `MainWindow` and mark these as "done". Translate "TROLL PRINT" as "TROLL IMPRIMIR". When you're translating "Two-sided", press the *Guess Again* button to translate "Two-sided", but change the "2" into "Dois".

Save and quit, then run lrelease. The Portuguese version should look like this:

Tutorial 3 Screenshot, "Troll Imprimir 1.1", Portuguese version

Choose *Ajuda*|*Sobre*, (*Help*|*About*), to see the about box

Tutorial 3 Screenshot, About box, Portuguese version

If you choose *Ajuda|Sobre Qt*, (*Help|About Qt*), you'll get an English dialog... oops! Qt itself needs to be translated. See the document Internationalization with Qt for details.

Now set `LANG=en` to get the original English version:

Tutorial 3 Screenshot, "Troll Print 1.1", English version

Summary

These tutorials cover all that you need to know to prepare your Qt applications for translation.

At the beginning of a project add the translation source files to be used to the project file and add calls to lupdate and lrelease to the make file.

During the project all the programmer must do is wrap any user-visible text in `tr()` calls. They should also use the two argument form for Ctrl key accelerators, or when asked by the translator for the cases where the same text translates into two different forms in the same context. The programmer should also include *TRANSLATION* comments to help the translator navigate the application.

Index