Accessibility and Internationalization with Qt

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

QAccessible Class Reference

The QAccessible class provides enums and static functions relating to accessibility.

`#include <qaccessible.h>`

Inherited by QAccessibleInterface [p. 10].

Public Members

- enum **Event** { SoundPlayed = 0x0001, Alert = 0x0002, ForegroundChanged = 0x0003, MenuStart = 0x0004, MenuEnd = 0x0005, PopupMenuStart = 0x0006, PopupMenuEnd = 0x0007, ContextHelpStart = 0x000C, ContextHelpEnd = 0x000D, DragDropStart = 0x000E, DragDropEnd = 0x000F, DialogStart = 0x0010, DialogEnd = 0x0011, ScrollingStart = 0x0012, ScrollingEnd = 0x0013, MenuCommand = 0x0018, ObjectCreated = 0x8000, ObjectDestroyed = 0x8001, ObjectShow = 0x8002, ObjectHide = 0x8003, ObjectReorder = 0x8004, Focus = 0x8005, Selection = 0x8006, SelectionAdd = 0x8007, SelectionRemove = 0x8008, SelectionWithin = 0x8009, StateChanged = 0x800A, LocationChanged = 0x800B, NameChanged = 0x800C, DescriptionChanged = 0x800D, ValueChanged = 0x800E, ParentChanged = 0x800F, HelpChanged = 0x80A0, DefaultActionChanged = 0x80B0, AcceleratorChanged = 0x80C0 }
- enum **State** { Normal = 0x00000000, Unavailable = 0x00000001, Selected = 0x00000002, Focused = 0x00000004, Pressed = 0x00000008, Checked = 0x00000010, Mixed = 0x00000020, ReadOnly = 0x00000040, HotTracked = 0x00000080, Default = 0x00000100, Expanded = 0x00000200, Collapsed = 0x00000400, Busy = 0x00000800, Floating = 0x00001000, Marqueed = 0x00002000, Animated = 0x00004000, Invisible = 0x00008000, Offscreen = 0x00010000, Sizeable = 0x00020000, Moveable = 0x00040000, SelfVoicing = 0x00080000, Focusable = 0x00100000, Selectable = 0x00200000, Linked = 0x00400000, Traversed = 0x00800000, MultiSelectable = 0x01000000, ExtSelectable = 0x02000000, AlertLow = 0x04000000, AlertMedium = 0x08000000, AlertHigh = 0x10000000, Protected = 0x20000000, Valid = 0x3fffffff }
- enum **Role** { NoRole = 0x00000000, TitleBar = 0x00000001, MenuBar = 0x00000002, ScrollBar = 0x00000003, Grip = 0x00000004, Sound = 0x00000005, Cursor = 0x00000006, Caret = 0x00000007, AlertMessage = 0x00000008, Window = 0x00000009, Client = 0x0000000A, PopupMenu = 0x0000000B, MenuItem = 0x0000000C, ToolTip = 0x0000000D, Application = 0x0000000E, Document = 0x0000000F, Pane = 0x00000010, Chart = 0x00000011, Dialog = 0x00000012, Border = 0x00000013, Grouping = 0x00000014, Separator = 0x00000015, ToolBar = 0x00000016, StatusBar = 0x00000017, Table = 0x00000018, ColumnHeader = 0x00000019, RowHeader = 0x0000001A, Column = 0x0000001B, Row = 0x0000001C, Cell = 0x0000001D, Link = 0x0000001E, HelpBalloon = 0x0000001F, Character = 0x00000020, List = 0x00000021, ListItem = 0x00000022, Outline = 0x00000023, OutlineItem = 0x00000024, PageTab = 0x00000025, PropertyPage = 0x00000026, Indicator = 0x00000027, Graphic = 0x00000028, StaticText = 0x00000029, EditableText = 0x0000002A, PushButton = 0x0000002B, CheckBox = 0x0000002C, RadioButton = 0x0000002D, ComboBox = 0x0000002E, DropLest = 0x0000002F, ProgressBar = 0x00000030, Dial = 0x00000031, HotkeyField = 0x00000032, Slider = 0x00000033, SpinBox = 0x00000034, Diagram = 0x00000035, Animation = 0x00000036, Equation = 0x00000037, ButtonDropDown = 0x00000038,

ButtonMenu = 0x00000039, ButtonDropGrid = 0x0000003A, Whitespace = 0x0000003B, PageTabList = 0x0000003C, Clock = 0x0000003D }

- enum **NavDirection** { NavUp = 0x00000001, NavDown = 0x00000002, NavLeft = 0x00000003, NavRight = 0x00000004, NavNext = 0x00000005, NavPrevious = 0x00000006, NavFirstChild = 0x00000007, NavLastChild = 0x00000008, NavFocusChild = 0x00000009 }
- enum **Text** { Name = 0, Description, Value, Help, Accelerator, DefaultAction }

Static Public Members

- QRESULT **queryAccessibleInterface** (QObject * object, QAccessibleInterface ** iface)
- void **updateAccessibility** (QObject * object, int control, Event reason)

Detailed Description

The QAccessible class provides enums and static functions relating to accessibility.

Accessibility clients use implementations of the QAccessibleInterface to read information an accessible object exposes, or to call methods to manipulate the accessible object.

See the plugin documentation [Plugins with Qt] for more details about how to redistribute Qt plugins.

See also Miscellaneous Classes.

Member Type Documentation

QAccessible::Event

This enum type defines event types when the state of the accessible object has changed. Event types are

- `QAccessible::SoundPlayed`
- `QAccessible::Alert`
- `QAccessible::ForegroundChanged`
- `QAccessible::MenuStart`
- `QAccessible::MenuEnd`
- `QAccessible::PopupMenuStart`
- `QAccessible::PopupMenuEnd`
- `QAccessible::ContextHelpStart`
- `QAccessible::ContextHelpEnd`
- `QAccessible::DragDropStart`
- `QAccessible::DragDropEnd`
- `QAccessible::DialogStart`
- `QAccessible::DialogEnd`
- `QAccessible::ScrollingStart`
- `QAccessible::ScrollingEnd`

- `QAccessible::ObjectCreated`
- `QAccessible::ObjectDestroyed`
- `QAccessible::ObjectShow`
- `QAccessible::ObjectHide`
- `QAccessible::ObjectReorder`
- `QAccessible::Focus`
- `QAccessible::Selection`
- `QAccessible::SelectionAdd`
- `QAccessible::SelectionRemove`
- `QAccessible::SelectionWithin`
- `QAccessible::StateChanged`
- `QAccessible::LocationChanged`
- `QAccessible::NameChanged`
- `QAccessible::DescriptionChanged`
- `QAccessible::ValueChanged`
- `QAccessible::ParentChanged`
- `QAccessible::HelpChanged`
- `QAccessible::DefaultActionChanged`
- `QAccessible::AcceleratorChanged`
- `QAccessible::MenuCommand`

QAccessible::NavDirection

This enum specifies to which item to move when navigating.

- `QAccessible::NavUp` - sibling above
- `QAccessible::NavDown` - sibling below
- `QAccessible::NavLeft` - left sibling
- `QAccessible::NavRight` - right sibling
- `QAccessible::NavNext` - next sibling
- `QAccessible::NavPrevious` - previous sibling
- `QAccessible::NavFirstChild` - first child
- `QAccessible::NavLastChild` - last child
- `QAccessible::NavFocusChild` - child with focus

QAccessible::Role

This enum defines a number of roles an accessible object can have. Roles are

- `QAccessible::NoRole`
- `QAccessible::TitleBar`

- QAccessible::MenuBar
- QAccessible::ScrollBar
- QAccessible::Grip
- QAccessible::Sound
- QAccessible::Cursor
- QAccessible::Caret
- QAccessible::AlertMessage
- QAccessible::Window
- QAccessible::Client
- QAccessible::PopupMenu
- QAccessible::MenuItem
- QAccessible::ToolTip
- QAccessible::Application
- QAccessible::Document
- QAccessible::Pane
- QAccessible::Chart
- QAccessible::Dialog
- QAccessible::Border
- QAccessible::Grouping
- QAccessible::Separator
- QAccessible::ToolBar
- QAccessible::StatusBar
- QAccessible::Table
- QAccessible::ColumnHeader
- QAccessible::RowHeader
- QAccessible::Column
- QAccessible::Row
- QAccessible::Cell
- QAccessible::Link
- QAccessible::HelpBalloon
- QAccessible::Character
- QAccessible::List
- QAccessible::ListItem
- QAccessible::Outline
- QAccessible::OutlineItem
- QAccessible::PageTab
- QAccessible::PropertyPage
- QAccessible::Indicator
- QAccessible::Graphic
- QAccessible::StaticText

- `QAccessible::EditableText`
- `QAccessible::PushButton`
- `QAccessible::CheckBox`
- `QAccessible::RadioButton`
- `QAccessible::ComboBox`
- `QAccessible::DropLest`
- `QAccessible::ProgressBar`
- `QAccessible::Dial`
- `QAccessible::HotkeyField`
- `QAccessible::Slider`
- `QAccessible::SpinBox`
- `QAccessible::Diagram`
- `QAccessible::Animation`
- `QAccessible::Equation`
- `QAccessible::ButtonDropDown`
- `QAccessible::ButtonMenu`
- `QAccessible::ButtonDropGrid`
- `QAccessible::Whitespace`
- `QAccessible::PageTabList`
- `QAccessible::Clock`

QAccessible::State

This enum type defines bitflags that can be combined to indicate the state of the accessible object. Defined values are

- `QAccessible::Normal`
- `QAccessible::Unavailable`
- `QAccessible::Selected`
- `QAccessible::Focused`
- `QAccessible::Pressed`
- `QAccessible::Checked`
- `QAccessible::Mixed`
- `QAccessible::ReadOnly`
- `QAccessible::HotTracked`
- `QAccessible::Default`
- `QAccessible::Expanded`
- `QAccessible::Collapsed`
- `QAccessible::Busy`
- `QAccessible::Floating`
- `QAccessible::Marqueed`
- `QAccessible::Animated`

- `QAccessible::Invisible`
- `QAccessible::Offscreen`
- `QAccessible::Sizeable`
- `QAccessible::Moveable`
- `QAccessible::SelfVoicing`
- `QAccessible::Focusable`
- `QAccessible::Selectable`
- `QAccessible::Linked`
- `QAccessible::Traversed`
- `QAccessible::MultiSelectable`
- `QAccessible::ExtSelectable`
- `QAccessible::AlertLow`
- `QAccessible::AlertMedium`
- `QAccessible::AlertHigh`
- `QAccessible::Protected`
- `QAccessible::Valid`

QAccessible::Text

This enum specifies string information that an accessible object returns.

- `QAccessible::Name` - The name of the object
- `QAccessible::Description` - A short text describing the object
- `QAccessible::Value` - The value of the object
- `QAccessible::Help` - A longer text giving information about how to use the object
- `QAccessible::DefaultAction` - The default method to interact with the object
- `QAccessible::Accelerator` - The keyboard shortcut that executes the default action

Member Function Documentation

QRESULT QAccessible::queryAccessibleInterface (QObject * object, QAccessibleInterface ** iface) [static]

Sets *iface* to point to the implementation of the QAccessibleInterface for *object*, and returns QS_OK if successfull, or sets *iface* to 0 and returns QE_NOCOMPONENT if no accessibility implementation for *object* exists.

The function uses the classname of *object* to find a suitable implementation. If no implementation for the object's class is available the function tries to find an implementation for the object's parent class.

This function is called to answer an accessibility client's request for object information. You should never need to call this function yourself.

void QAccessible::updateAccessibility (QObject * object, int control, Event reason) [static]

Notifies accessibility clients about a change of the accessibility information of *object*.

reason designates the cause of this change, e.g. ValueChange when the position of e.g. a slider has been changed. *control* is the ID of the child element that has changed. When *control* is null, the object itself has changed.

Call this function whenever the state of your accessible object or one of it's subelements has been changed either programmatically (e.g. by calling QLabel::setText()) or by user interaction.

If there are no accessibility tools listening to this event, the performance penalty for calling this function is minor.

QAccessibleInterface Class Reference

The QAccessibleInterface class defines an interface that exposes information about accessible objects.

`#include <qaccessible.h>`

Inherits QAccessible [p. 3].

Inherited by QAccessibleObject [p. 14].

Public Members

- virtual bool **isValid** () const
- virtual int **childCount** () const
- virtual QRESULT **queryChild** (int control, QAccessibleInterface ** iface) const
- virtual QRESULT **queryParent** (QAccessibleInterface ** iface) const
- virtual int **controlAt** (int x, int y) const
- virtual QRect **rect** (int control) const
- virtual int **navigate** (NavDirection direction, int startControl) const
- virtual QString **text** (Text t, int control) const
- virtual void **setText** (Text t, int control, const QString & text)
- virtual Role **role** (int control) const
- virtual State **state** (int control) const
- virtual QMemArray<int> **selection** () const
- virtual bool **doDefaultAction** (int control)
- virtual bool **setFocus** (int control)
- virtual bool **setSelected** (int control, bool on, bool extend)
- virtual void **clearSelection** ()

Detailed Description

The QAccessibleInterface class defines an interface that exposes information about accessible objects.

See also Miscellaneous Classes.

Member Function Documentation

int QAccessibleInterface::childCount () const [virtual]

Returns the number of children that belong to this object. A child can provide accessibility information on it's own (e.g. a child widget), or be a subelement of this accessible object.

All objects provide this information.

See also queryChild() [p. 12].

void QAccessibleInterface::clearSelection () [virtual]

Removes any selection from the object.

See also setSelected() [p. 12].

int QAccessibleInterface::controlAt (int x, int y) const [virtual]

Returns the ID of the child that contains the screen coordinates (*x*, *y*). This function returns 0 if the point is positioned on the object itself. If the tested point is outside the boundaries of the object this function returns -1.

All visual objects provide this information.

bool QAccessibleInterface::doDefaultAction (int control) [virtual]

Calling this function performs the default action of the child object specified by *control*, or the default action of the object itself if *control* is 0.

bool QAccessibleInterface::isValid () const [virtual]

Returns TRUE if all data necessary to use this interface implementation is valid (e.g. all pointers are non-null), otherwise returns FALSE.

int QAccessibleInterface::navigate (NavDirection direction, int startControl) const [virtual]

This function traverses to another object, or to a subelement of the current object. *direction* specifies in which direction to navigate, and *startControl* specifies the start point of the navigation, which is either 0 if the navigation starts at the object itself, or an ID of one of the object's subelements.

The function returns the ID of the subelement located in the *direction* specified. If there is nothing at the navigated *direction*, this function returns -1.

All objects support navigation.

QRESULT QAccessibleInterface::queryChild (int control, QAccessibleInterface ** iface) const [virtual]

Sets *iface* to point to the implementation of the QAccessibleInterface for the child specified with *control*. If the child doesn't provide accessibility information on it's own, the value of *iface* is set to null. For those elements, this object is responsible for exposing the child's properties.

All objects provide this information.

See also childCount() [p. 11] and queryParent() [p. 12].

QRESULT QAccessibleInterface::queryParent (QAccessibleInterface ** iface) const [virtual]

Sets *iface* to point to the implementation of the QAccessibleInterface for the parent object, or to null if there is no such implementation or object.

All objects provide this information.

See also queryChild() [p. 12].

QRect QAccessibleInterface::rect (int control) const [virtual]

Returns the location of the child specified with *control* in screen coordinates. This function returns the location of the object itself if *control* is 0.

All visual objects provide this information.

Role QAccessibleInterface::role (int control) const [virtual]

Returns the role of the object if *control* is 0, or the role of the object's subelement with ID *control*. The role of an object is usually static. All accessible objects have a role.

See also text() [p. 13], state() [p. 13] and selection() [p. 12].

QMemArray<int> QAccessibleInterface::selection () const [virtual]

Returns the list of all element IDs that are selected.

See also text() [p. 13], role() [p. 12] and state() [p. 13].

bool QAccessibleInterface::setFocus (int control) [virtual]

Gives the focus to the child object specified by *control*, or to the object itself if *control* is 0.

Returns TRUE if the focus could be set, otherwise returns FALSE.

bool QAccessibleInterface::setSelected (int control, bool on, bool extend) [virtual]

Sets the selection of the child object with ID *control* to *on*. If *extend* is TRUE, all child elements between the focused item and the specified child object set the selection to *on*.

Returns TRUE if the selection could be set, otherwise returns FALSE.

See also setFocus() [p. 12] and clearSelection() [p. 11].

void QAccessibleInterface::setText (Text t, int control, const QString & text) [virtual]

Sets the text property *t* of the child object *control* to *text*. If *control* is 0, the text property of the object itself is set.

State QAccessibleInterface::state (int control) const [virtual]

Returns the current state of the object if *control* is 0, or the state of the object's subelement element with ID *control*. All objects have a state.

See also text() [p. 13], role() [p. 12] and selection() [p. 12].

QString QAccessibleInterface::text (Text t, int control) const [virtual]

Returns a string property *t* of the child object specified by *control*, or the string property of the object itself if *control* is 0.

The *Name* is a string used by clients to identify, find or announce an accessible object for the user. All objects must have a name that is unique within their container.

An accessible object's *Description* provides textual information about an object's visual appearance. The description is primarily used to provide greater context for low-vision or blind users, but is also used for context searching or other applications. Not all objects have a description. An "OK" button would not need a description, but a toolbutton that shows a picture of a smiley would.

The *Value* of an accessible object represents visual information contained by the object, e.g. the text in a line edit. Usually, the value can be modified by the user. Not all objects have a value, e.g. static text labels don't, and some objects have a state that already is the value, e.g. toggle buttons.

The *Help* text provides information about the function and useage of an accessible object. Not all objects provide this information.

An accessible object's *DefaultAction* describes the object's primary method of manipulation, and should be a verb or a short phrase, e.g. "Press" for a button.

The accelerator is a keyboard shortcut that activates the default action of the object. A keyboard shortcut is the underlined character in the text of a menu, menu item or control, and is either the character itself, or a combination of this character and a modifier key like ALT, CTRL or SHIFT. Command controls like tool buttons also have shortcut keys and usually display them in their tooltip.

See also role() [p. 12], state() [p. 13] and selection() [p. 12].

QAccessibleObject Class Reference

The QAccessibleObject class implements parts of the QAccessibleInterface for QObjects.

`#include <qaccessible.h>`

Inherits QObject [Additional Functionality with Qt] and QAccessibleInterface [p. 10].

Public Members

- **QAccessibleObject** (QObject * object)
- virtual **~QAccessibleObject** ()

Protected Members

- QObject * **object** () const

Detailed Description

The QAccessibleObject class implements parts of the QAccessibleInterface for QObjects.

This class is mainly provided for convenience. All further implementations of the QAccessibleInterface should use this class as the base class.

See also Miscellaneous Classes.

Member Function Documentation

QAccessibleObject::QAccessibleObject (QObject * object)

Creates a QAccessibleObject for *object*.

QAccessibleObject::~QAccessibleObject () [virtual]

Destroys the QAccessibleObject.

This will only happen if a call to release() decrements the internal reference counter to zero.

QObject * QAccessibleObject::object () const [protected]

Returns the QObject for which this QAccessibleInterface implementation provides information. Use isValid() to make sure the object pointer is safe to use.

Internationalization with Qt

The internationalization of an application is the process of making the application usable by people in countries other than one's own.

In some cases internationalization is simple, for example, making a US application accessible to Australian or British users may require little more than a few spelling corrections. But to make a US application usable by Japanese users, or a Korean application usable by German users, will require that the software operate not only in different languages, but use different input techniques, character encodings and presentation conventions.

See also the Qt Linguist manual.

Step by Step

Writing cross-platform international software with Qt is a gentle, incremental process. Your software can become internationalized in the following stages:

Use QString for all User-visible Text

Since QString uses the Unicode encoding internally, every language in the world can be processed transparently using familiar text processing operations. Also, since all Qt functions that present text to the user take a QString as a parameter, there is no char* to QString conversion time overhead.

Strings that are in "programmer space" (such as QObject names and file format texts) need not use QString; the traditional char* or the QCString class will suffice.

You're unlikely to notice that you are using Unicode; QString, and QChar are just like easier versions of the crude const char* and char from traditional C.

Use tr() for all Literal Text

Wherever your program uses `"quoted text"` for text that will be presented to the user, ensure that it is processed by the QApplication::translate() function. Essentially all that is necessary to achieve this is to use QObject::tr(). For example, assuming `LoginWidget` is a subclass of QWidget:

```
LoginWidget::LoginWidget()
{
    QLabel *label = new QLabel( tr("Password:"), this );
    ...
}
```

This accounts for 99% of the user-visible strings you're likely to write.

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 );
}
```

If you need to have translatable text completely outside a function, there are two macros to help: QT_TR_NOOP() and QT_TRANSLATE_NOOP(). They merely mark the text for extraction by the *lupdate* utility described below. 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] );
}
```

If you disable the const char* to QString automatic conversion by compiling your software with the macro QT_NO_CAST_ASCII defined, you'll be very likely to catch any strings you are missing. See QString::fromLatin1() for more information. Disabling the conversion makes programming cumbersome.

If your source language uses characters outside Latin-1, you might find QObject::trUtf8() more convenient than QObject::tr(), as tr() depends on the QApplication::defaultCodec(), which makes it more fragile than QObject::trUtf8().

Use QKeySequence() for Accelerator Values

Accelerator values such as Ctrl+Q or Alt+F need to be translated too. If you hardcode `CTRL+Key_Q` for "Quit" in your application, translators won't be able to override it. The correct idiom is

```
QPopupMenu *file = new QPopupMenu( this );
file->insertItem( tr("&Quit"), this, SLOT(quit()),
                  QKeySequence(tr("Ctrl+Q", "File|Quit")) );
```

Use QString::arg() for Simple Arguments

The printf() style of inserting arguments in strings is often a poor choice for internationalized text, as it is sometimes necessary to change the order of arguments when translating. Nonetheless, the QString::arg() functions offer a simple means for substituting arguments:

```
void FileCopier::showProgress( int done, int total,
                               const QString& current_file )
{
    label.setText( tr("%1 of %2 files copied.\nCopying: %3")
                   .arg(done)
                   .arg(total)
                   .arg(current_file) );
}
```

Produce Translations

Once you are using tr() throughout an application, you can start producing translations of the user-visible text in your program.

Qt Linguist's manual provides further information about Qt's translation tools, *Qt Linguist*, *lupdate* and *lrelease*.

Translation of a Qt application is a three-step process:

1. Run *lupdate* to extract translatable text from the C++ source code of the Qt application, resulting in a message file for translators (a .ts file). The utility recognizes the tr() construct and the QT_*_NOOP macros described above and produces .ts files (usually one per language).
2. Provide translations for the source texts in the .ts file, using Qt Linguist. Since .ts files are in XML format, you can also edit them by hand.
3. Run *lrelease* to obtain a light-weight message file (a .qm file) from the .ts file, suitable only for end use. You can see the .ts files as "source files", and .qm as "object files". The translator edits the .ts files, but the users of your application only need the .qm files. Both kinds of files are platform and locale independent.

Typically, you will repeat these steps for every release of your application. The *lupdate* utility does its best to reuse the translations from previous releases.

Before you run *lupdate*, you should prepare a project file. Here's an example project file (.pro file):

```
HEADERS         = funnydialog.h \
                  wackywidget.h
SOURCES         = funnydialog.cpp \
                  main.cpp \
                  wackywidget.cpp
FORMS           = fancybox.ui
TRANSLATIONS    = superapp_dk.ts \
                  superapp_fi.ts \
                  superapp_no.ts \
                  superapp_se.ts
```

When you run *lupdate* or *lrelease*, you must give the name of the project file as a command-line argument.

In this example, four exotic languages are supported: Danish, Finnish, Norwegian and Swedish. If you use qmake (or tmake), you usually don't need an extra project file for *lupdate*; your qmake project file will work fine once you add the TRANSLATIONS entry.

In your application, you must QTranslator::load() the translation files appropriate for the user's language, and install them using QApplication::installTranslator().

If you have been using the old Qt tools (findtr, msg2qm and mergetr), you can use *qm2ts* to convert your old .qm files.

linguist, *lupdate* and *lrelease* are installed in `$QTDIR/bin`. Click Help|Manual in Qt Linguist to access the user's manual; it contains a tutorial to get you started.

While these utilities offer a convenient way to create .qm files, any system that writes .qm files is sufficient. You could make an application that adds translations to a QTranslator with QTranslator::insert() and then writes a .qm file with QTranslator::save(). This way the translations can come from any source you choose.

Qt itself contains about 400 strings that will also need to be translated into the languages that you are targeting. You will find translation files for French and German in `$QTDIR/translations` as well as a template for translating to other languages.

Typically, your application's main() function will look like this:

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

    // translation file for Qt
    QTranslator qt( 0 );
    qt.load( QString( "qt_" ) + QTextCodec::locale(), "." );
    app.installTranslator( &qt );

    // translation file for application strings
    QTranslator myapp( 0 );
    myapp.load( QString( "myapp_" ) + QTextCodec::locale(), "." );
    app.installTranslator( &myapp );

    ...

    return app.exec();
}
```

Support for Encodings

The QTextCodec class and the facilities in QTextStream make it easy to support many input and output encodings for your users' data. When an application starts, the locale of the machine will determine the 8-bit encoding used when dealing with 8-bit data - such as for font selection, text display, 8-bit text I/O and character input.

The application may occasionally require encodings other than the default local 8-bit encoding. For example, an application in a Cyrillic KOI8-R locale (the de-facto standard locale in Russia) might need to output Cyrillic in the ISO 8859-5 encoding. Code for this would be:

```
QString string = ...; // some Unicode text

QTextCodec* codec = QTextCodec::codecForName( "ISO 8859-5" );
QCString encoded_string = codec->fromUnicode( string );

...; // use encoded_string in 8-bit operations
```

For converting Unicode to local 8-bit encodings, a shortcut is available: the local8Bit() method of QString returns such 8-bit data. Another useful shortcut is the utf8() method, which returns text in the 8-bit UTF-8 encoding - interesting in that it perfectly preserves Unicode information while looking like plain US-ASCII if the Unicode is wholly US-ASCII.

For converting the other way, there are the QString::fromUtf8() and QString::fromLocal8Bit() convenience functions, or the general code, demonstrated by this conversion from ISO 8859-5 Cyrillic to Unicode conversion:

```
QCString encoded_string = ...; // Some ISO 8859-5 encoded text.

QTextCodec* codec = QTextCodec::codecForName("ISO 8859-5");
QString string = codec->toUnicode(encoded_string);

...; // Use string in all of Qt's QString operations.
```

Ideally Unicode I/O should be used as this maximizes the portability of documents between users around the world, but in reality it is useful to support all the appropriate encodings that your users will need to process existing documents. In general, Unicode (UTF16 or UTF8) is best for information transferred between arbitrary people, while within a language or national group, a local standard is often more appropriate. The most important encoding to support is the one returned by QTextCodec::codecForLocale(), as this is the one the user is most likely to need for communicating with other people and applications (this is the codec used by local8Bit()).

Since most Unix systems do not have built-in support for converting between local 8-bit encodings and Unicode, it may be necessary to write your own QTextCodec subclass. Depending on the urgency, it may be useful to contact Trolltech technical support or ask on the qt-interest mailing list to see if someone else is already working on supporting the encoding. A useful interim measure can be to use the QTextCodec::loadCharmapFile() function to build a data-driven codec, although this approach has a memory and speed penalty, especially with dynamically loaded libraries. For details of writing your own QTextCodec, see the main QTextCodec class documentation.

Localize

Localization is the process of adapting to local conventions such as date and time presentations. Such localizations can be accomplished using appropriate tr() strings, even "magic" words, as this somewhat contrived example shows:

```
void Clock::setTime(const QTime& t)
{
```

```
        if ( tr("AMPM") == "AMPM" ) {
            // 12-hour clock
        } else {
            // 24-hour clock
        }
    }
```

Localizing images is not recommended. Choose clear icons that are appropriate for all localities, rather than relying on local puns or stretched metaphors.

System Support

Operating systems and window systems supporting Unicode are still in the early stages of development. The level of support available in the underlying system influences the support Qt provides on that platform, but applications written with Qt need not generally be too concerned with the actual limitations.

Unix/X11

- Locale-oriented fonts and input methods. Qt hides these and provides Unicode input and output.
- Filesystem conventions such as UTF-8 are under development in some Unix variants. All Qt file functions allow Unicode, but convert all filenames to the local 8-bit encoding, as this is the Unix convention (see QFile::setEncodingFunction() to explore alternative encodings).
- File I/O defaults to the local 8-bit encoding, with Unicode options in QTextStream.

Windows 95/98/NT

- Qt provides full Unicode support, including input methods, fonts, clipboard, drag-and-drop and file names.
- File I/O defaults to Latin-1, with Unicode options in QTextStream. Note that some Windows programs do not understand big-endian Unicode text files even though that is the order prescribed by the Unicode Standard in the absence of higher-level protocols.
- Unlike programs written with MFC or plain winlib, Qt programs are portable between Windows 95/98 and Windows NT. *You do not need different binaries to support Unicode.*

Supporting More Input Methods

While Trolltech doesn't have the resources or expertise in all the languages of the world to immediately include support in Qt, we are very keen to work with people who do have the expertise. Over the next few minor version numbers, we hope to add support for your language of choice, until everyone can use Qt and all the programs developed with Qt, regardless of their language.

Languages with single-byte encodings (European Latin-1 and KOI8-R, etc.) and multi-byte encodings (East Asian EUC-JP, etc.) are supported. Support for the "complex" encodings - those requiring right-to-left input or complex character composition (eg. Arabic, Hebrew, and Thai script) is implemented, but the range of Indic scripts (Hindi, Devanagari, Bengali, etc.) is still under development. The current state of activity is:

Encodings	Status
All encodings on Windows	The local encoding is always supported.
ISO standard encodings ISO 8859-1, ISO 8859-2, ISO 8859-3, ISO 8859-4, ISO 8859-5, ISO 8859-7, ISO 8859-9, and ISO 8859-15	Fully supported.
KOI8-R	Fully supported.
eucJP, JIS, and ShiftJIS	Fully supported. Uses eucJP with the XIM protocol on X11, and the IME Windows NT in Japanese Windows NT. Serika Kurusugawa and others are assisting with this effort. kinput2 is the tested input method for X11.
eucKR	Supported. Mizi Research are assisting with this effort. hanIM is the tested input method.
Big5	Qt contains a Big5 codec developed by Ming Che-Chuang. Testing is underway with the xcin (2.5.x) XIM server.
eucTW	Under external development.

More information on the support of different writing systems in Qt can be found in the documentation about writing systems.

If you are interested in contributing to existing efforts, or supporting new encodings beyond those mentioned above, your work can be considered for inclusion in the official Qt distribution, or just included with your application.

Eventually, we hope to help Unix become as Unicode-oriented as Windows is becoming. This means better font support in the font servers, with new developments like the True Type font servers xfsft, xfstt, and x-tt, as well as UTF-8 (a Unicode encoding) filenames such as with the Unicode support in Solaris 7.

Note about Locales on X11

Many Unix distributions contain only partial support for some locales. For example, if you have a `/usr/share/locale/ja_JP.EUC` directory, this does not necessarily mean you can display Japanese text; you also need JIS encoded fonts (or Unicode fonts), and that `/usr/share/locale/ja_JP.EUC` directory needs to be complete. For best results, use complete locales from your system vendor.

Relevant Qt Classes

These classes are relevant to internationalizing Qt applications.

QEucJpCodec	Conversion to and from EUC-JP character sets
QEucKrCodec	Conversion to and from EUC-KR character sets
QGbkCodec	Conversion to and from the Chinese GBK encoding
QHebrewCodec	Conversion to and from visually ordered Hebrew
QJisCodec	Conversion to and from JIS character sets
QSjisCodec	Conversion to and from Shift-JIS
QTextCodec	Conversion between text encodings
QTextDecoder	State-based decoder
QTextEncoder	State-based encoder
QTranslator	Internationalization support for text output
QTranslatorMessage	Translator message and its properties
QTsciiCodec	Conversion to and from the Tamil TSCII encoding

Qt's Text Engine

Qt 3 comes with a completely redesigned text processing and layout engine that is used throughout the whole library.

It has support for most writing systems that are used throughout the world, including

- Arabic
- Chinese
- Cyrillic (Russian)
- Greek
- Hebrew
- Japanese
- Korean
- Latin languages (e.g. English and many other European languages)
- Thai
- Vietnamese

Many of these writing systems exhibit special features:

- Special line breaking behaviour. Some of the Asian languages are written without spaces between words. Line breaking can occur either after every character (with exceptions) as in Chinese, Japanese and Korean, or after logical word boundaries as in Thai.
- Bidirectional writing. Arabic and Hebrew are written from right to left, except for numbers and embedded English text which is written left to right. The exact behaviour is defined in the Unicode Technical Report #9.
- Non spacing or diacritical marks (accents or umlauts in European languages). Some languages such as Vietnamese make extensive use of these marks and some characters can have a few marks at the same time to clarify pronunciation.
- Ligatures. In special contexts, some characters following each other directly get replaced by a combined glyph forming a ligature. Common examples are the ff and fi ligatures used in typesetting US and European books.

Except for ligatures which are currently only supported for the special case of Arabic, Qt tries to take care of all the special features listed above. You will usually never have to worry about these features as long as you use Qt's input (e.g. QLineEdit, QTextView or derived classes) and displaying controls (e.g. QLabel).

Support for these writing systems is transparent to the programmer and completely encapsulated in Qt's text engine. This implies that you don't need to have any knowledge about the writing system used in a particular language, except for a couple of small things listed below.

- QPainter::drawText(int x, int y, const QString &str) will always draw the string with it's left edge at the position specified with the x, y parameters. This will usually give you left aligned strings. Arabic and Hebrew application strings are usually right aligned, so for these languages use the version of drawText() that takes a QRect since this will align in accordance with the language.
- When you write your own text input controls, use QFontMetrics::charWidth() to determine the width of a character in a string. In some langauges (mainly Arabic), the width and shape of a glyph changes depending on the surrounding characters. Writing input controls usually requires a certain knowledge of the scripts it is going to be used in. Usually the easiest way is to subclass QLineEdit or QTextView.

About Unicode

Unicode is a 16-bit character set, portable across all major computing platforms and with decent coverage over most of the world. It is also single-locale; it includes no code pages or other complexities that make software harder to write and test. There is no competing character set that's reasonably cross-platform. For these reasons, Trolltech has chosen to make Unicode the native character set of Qt starting with version 2.0.

Information about Unicode on the web.

The Unicode Consortium has a number of documents available, including

- A technical introduction to Unicode
- The home page for the standard

The Standard

The current version of the standard is 3.0.1.

- The Unicode Standard, version 3.0. See also its home page.
- The Unicode Standard, version 2.0. See also the 2.1 update and 2.1.9 the 2.1.9 data files at www.unicode.org.

Unicode in Qt

In Qt, and in most applications that use Qt, most or all user-visible strings are stored in Unicode. Qt provides:

- Translation to/from legacy encodings for file I/O - see QTextCodec and QTextStream.
- Translation from Input Methods and 8-bit keyboard input.
- Translation to legacy character sets for on-screen display.
- A string class, QString, that stores Unicode characters, with support for migrating from C strings including fast (cached) translation to and from US-ASCII, and all the usual string operations.
- Unicode-aware widgets where appropriate.
- Unicode support detection on Windows 95/98/NT/2000, so that Qt provides Unicode even on Windows platforms that do not support it.

To obtain the benefits of Unicode, we recommend using QString for storing all user-visible strings and performing all text file I/O using QTextStream. Use QKeyEvent::text() for keyboard input in any custom widgets you write; it does not make much difference for slow typists in West Europe or North America, but for fast typists or people using special input methods using text() is beneficial.

All the function arguments in Qt that may be user-visible strings, QLabel::setText() and a many others, take `const QString &` as type. QString provides implicit casting from `const char *` such that things like

```
myLabel->setText( "Hello, Dolly!" );
```

will work. There is also a function, QObject::tr(), that provides translation support, like this:

```
myLabel->setText( tr("Hello, Dolly!") );
```

tr() (simplifying somewhat) maps from `const char *` to a Unicode string, and uses installable QTranslator objects to do the mapping.

Programs that need to talk to other programs or read/write files in legacy file formats, Qt provides a number of built-in QTextCodec classes, that is, classes that know how to translate between Unicode and legacy encodings.

By default, conversion to/from `const char *` uses a locale-dependent codec. However, the program can easily find codecs for other locales, and set any open file or network connection to use a special codec. It is also possible to install new codecs, for encodings that the built-in ones do not support. (At the time of writing, Vietnamese/VISCII is one such example.)

Since US-ASCII and ISO-8859-1 are so common, there are also especially fast functions for mapping to and from them. For example, to open an application's icon one might do this:

```
QFile f( QString::fromLatin1("appicon.png") );
```

Regarding output, Qt will do a best-effort conversion from Unicode to whatever encoding the system and fonts provide. Depending on operating system, locale, font availability and Qt's support for the characters used, this conversion may be good or bad. We will extend this in upcoming versions, with emphasis on the most common locales first.

QIMEvent Class Reference

The QIMEvent class provides parameters for input method events.

`#include <qevent.h>`

Inherits QEvent [Events, Actions, Layouts and Styles with Qt].

Public Members

- **QIMEvent** (Type type, const QString & text, int cursorPosition)
- const QString & **text** () const
- int **cursorPos** () const
- bool **isAccepted** () const
- void **accept** ()
- void **ignore** ()

Detailed Description

The QIMEvent class provides parameters for input method events.

Input method events are send to widgets, when an input method is used to enter text into a widget. Input methods are widely used to enter text in Asian languages.

The events are of interest to widgets that accept keyboard input and want to be able to correctly handle Asian languages. Text input in Asian languages is usually a three step process. When the user presses the first key on a keyboard an input context is created. This input context will contain a string with the typed characters. With every new key pressed, the input method will try to create a matching string for the text typed so far.

While the input context is active, the user can move the cursor only inside the string belonging to this input context. At some point, when the user presses the Spacebar, they get to the second stage, where they can choose from a number of strings that match the text they have typed so far. The user can press Enter to confirm their choice or Escape to cancel the input; in either case the input context will be closed. Note that the particular key presses used for a given input context may differ from those we've mentioned here, i.e. they may not be Spacebar, Enter and Escape.

These three stages are represented by three different types of events. The IMStartEvent, IMComposeEvent and IMEndEvent. When a new input context is created, an IMStartEvent will be sent to the widget and delivered to the QWidget::imStartEvent() function. The widget can then update internal data structures to reflect this.

After this, an IMComposeEvent will be send to the widget with every key the user presses. It will contain the current composition string the widget has to show and the current cursor position within the composition string. This string is temporary and can change with every key the user types, so the widget will need to store the state before the

composition started (the state it had when it received the IMStartEvent). IMComposeEvents will be delivered to the QWidget::imComposeEvent() function.

Usually, widgets try to mark the part of the text that is part of the current composition in a way that is visible to the user. Mostly this is achieved by using e.g. dotted underline.

After the user has selected the final string, and IMEndEvent will be sent to the widget. The event contains the final string the user selected. This string has to be accepted as the final text the user entered, and the intermediate composition string should be cleared. These events are delivered to QWidget::imEndEvent().

If the user clicks another widget, taking the focus out of the widget where the compose is taking place the IMEndEvent will be sent and the string it holds will be the result of the composition up to that point (which could be an empty string).

See also Event Classes.

Member Function Documentation

QIMEvent::QIMEvent (Type type, const QString & text, int cursorPosition)

Constructs a new QIMEvent with accept flag set to FALSE. *type* can be one of QEvent::IMStartEvent, QEvent::IMComposeEvents and QEvent::IMEndEvent. *text* contains the current compostion string and *cursorPosition* the current position of the cursor inside *text*.

void QIMEvent::accept ()

Sets the accept flag of the input method event object.

Setting the accept parameter indicates that the receiver of the event processed the input method event.

The accept flag is not set by default.

See also ignore() [p. 28].

int QIMEvent::cursorPos () const

Returns the current cursor position inside the composition string. Will return 0 for IMStartEvent and IMEndEvent.

void QIMEvent::ignore ()

Clears the accept flag parameter of the input method event object.

Clearing the accept parameter indicates that the event receiver does not want the input method event.

The accept flag is cleared by default.

See also accept() [p. 28].

bool QIMEvent::isAccepted () const

Returns TRUE if the receiver of the event processed the event; otherwise returns FALSE.

const QString & QIMEvent::text () const

Returns the composition text. This is a null string for an IMStartEvent, and contains the final accepted string in the IMEndEvent.

QTranslator Class Reference

The QTranslator class provides internationalization support for text output.

`#include <qtranslator.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QTranslator** (QObject * parent, const char * name = 0)
- **~QTranslator** ()
- QString find (const char * context, const char * sourceText, const char * comment = 0) const *(obsolete)*
- virtual QTranslatorMessage **findMessage** (const char * context, const char * sourceText, const char * comment) const
- bool **load** (const QString & filename, const QString & directory = QString::null, const QString & search_delimiters = QString::null, const QString & suffix = QString::null)
- void **clear** ()
- enum **SaveMode** { Everything, Stripped }
- bool **save** (const QString & filename, SaveMode mode = Everything)
- void **insert** (const QTranslatorMessage & message)
- void insert (const char * context, const char * sourceText, const QString & translation) *(obsolete)*
- void **remove** (const QTranslatorMessage & message)
- void remove (const char * context, const char * sourceText) *(obsolete)*
- bool **contains** (const char * context, const char * sourceText, const char * comment = 0) const
- void **squeeze** (SaveMode mode = Everything)
- void **unsqueeze** ()
- QValueList<QTranslatorMessage> **messages** () const

Detailed Description

The QTranslator class provides internationalization support for text output.

An object of this class contains a set of QTranslatorMessage objects, each of which specifies a translation from a source language to a target language. QTranslator provides functions to look up such translations, add new ones, remove them, load and save them, etc.

The most common use of QTranslator is expected to be loading a translator file made using Qt Linguist, installing it using QApplication::installTranslator(), and using it via QObject::tr(), like this:

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

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

    MyWidget m;
    app.setMainWidget( &m );
    m.show();

    return app.exec();
}
```

Most applications will never need to do anything else with this class. However, applications that work on translator files need the other functions in this class.

It is possible to do lookup using findMessage() (as tr() and QApplication::translate() do) and contains(), insert a new translation message using insert(), and remove it using remove().

Because end-user programs and translation tools have rather different requirements, QTranslator can use stripped translator files in a way that uses a minimum of memory and provides very little functionality other than findMessage().

Thus, load() may not load enough information to make anything more than findMessage() work. save() has an argument indicating whether to save just this minimum of information or to save everything.

"Everything" means that for each translation item the following information is kept:

- The *translated text* - the return value from tr().
- The input key:
  - The *source text* - usually the argument to tr().
  - The *context* - usually the class name for the tr() caller.
  - The *comment* - a comment that helps disambiguate different uses of the same text in the same context.

The minimum for each item is just the information necessary for findMessage() to return the right text. This may include the source, context and comment, but usually it is just a hash value and the translated text.

For example, the "Cancel" in a dialog might have "Anuluj" when the program runs in Polish (in this case the source text would be "Cancel"). The context would (normally) be the dialog's class name; there would normally be no comment, and the translated text would be "Anuluj".

But it's not always so simple. The Spanish version of a printer dialog with settings for two-sided printing and binding would probably require both "Activado" and "Activada" as translations for "Enabled". In this case the source text would be "Enabled" in both cases and the context would be the dialog's class name, but the two items would have disambiguating comments such as "two-sided printing" for one and "binding" for the other. The comment enables the translator to choose the appropriate gender for the Spanish version, and enables Qt to distinguish between translations.

Note that when QTranslator loads a stripped file, most functions do not work. The functions that do work with stripped files are explicitly documented as such.

See also QTranslatorMessage [p. 35], QApplication::installTranslator() [Additional Functionality with Qt], QApplication::removeTranslator() [Additional Functionality with Qt], QObject::tr() [Additional Functionality with Qt], QApplication::translate() [Additional Functionality with Qt], Environment Classes and Internationalization with Qt [p. 16].

Member Type Documentation

QTranslator::SaveMode

This enum type defines how QTranslator can write translation files. There are two modes:

- `QTranslator::Everything` - files are saved with all contents
- `QTranslator::Stripped` - files are saved with just what's needed for end-users

Note that when QTranslator loads a stripped file, most functions do not work. The functions that do work with stripped files are explicitly documented as such.

Member Function Documentation

QTranslator::QTranslator (QObject * parent, const char * name = 0)

Constructs an empty message file object that is not connected to any file. The object has parent *parent* and name *name*.

QTranslator::~QTranslator ()

Destroys the object and frees any allocated resources.

void QTranslator::clear ()

Empties this translator of all contents.

This function works with stripped translator files.

bool QTranslator::contains (const char * context, const char * sourceText, const char * comment = 0) const

Returns TRUE if this message file contains a message with the key (*context*, *sourceText*, *comment*); otherwise returns FALSE.

This function works with stripped translator files.

(This is is a one-liner that calls find().)

QString QTranslator::find (const char * context, const char * sourceText, const char * comment = 0) const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Please use findMessage() instead.

Returns the translation for the key (*context*, *sourceText*, *comment*) or QString::null if there is none in this translator.

QTranslatorMessage QTranslator::findMessage (const char * context, const char * sourceText, const char * comment) const [virtual]

Returns the QTranslatorMessage for the key (*context*, *sourceText*, *comment*).

void QTranslator::insert (const QTranslatorMessage & message)

Inserts *message* into this message file.

This function does *not* work with stripped translator files. It may seem to, but that is not dependable.

See also remove() [p. 34].

void QTranslator::insert (const char * context, const char * sourceText, const QString & translation)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

bool QTranslator::load (const QString & filename, const QString & directory = QString::null, const QString & search_delimiters = QString::null, const QString & suffix = QString::null)

Loads *filename*, which may be an absolute file name or relative to *directory*. The previous contents of this translator object is discarded.

If the full file name does not exist, other file names are tried in the following order:

1. File name with *suffix* appended (".qm" if the suffix is QString::null).
2. File name with text after a character in *search_delimiters* stripped ("_." is the default for *search_delimiters* if it is QString::null).
3. File name stripped and *suffix* appended.
4. File name stripped further, etc.

For example, an application running in the fr_CA locale (French-speaking Canada) might call load("foo.fr_ca", "/opt/foolib"), which would then try to open these files:

1. /opt/foolib/foo.fr_ca
2. /opt/foolib/foo.fr_ca.qm
3. /opt/foolib/foo.fr
4. /opt/foolib/foo.fr.qm
5. /opt/foolib/foo
6. /opt/foolib/foo.qm

See also save() [p. 34].

Example: i18n/main.cpp.

QValueList<QTranslatorMessage> QTranslator::messages () const

Returns a list of the messages in the translator. This function is rather slow; because it is seldom called, it's optimized for simplicity and small size, not speed.

void QTranslator::remove (const QTranslatorMessage & message)

Removes *message* from this translator.

This function works with stripped translator files.

See also insert() [p. 33].

void QTranslator::remove (const char * context, const char * sourceText)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Removes the translation associated to the key (*context*, *sourceText*, "") from this translator.

bool QTranslator::save (const QString & filename, SaveMode mode = Everything)

Saves this message file to *filename*, overwriting the previous contents of *filename*. If *mode* is Everything (the default), all the information is preserved. If *mode* is Stripped, any information that is not necessary for findMessage() is stripped away.

See also load() [p. 33].

void QTranslator::squeeze (SaveMode mode = Everything)

Converts this message file to the compact format used to store message files on disk.

You should never need to call this directly; save() and other functions call it as necessary. *mode* is for internal use.

See also save() [p. 34] and unsqueeze() [p. 34].

void QTranslator::unsqueeze ()

Converts this message file into an easily modifiable data structure, less compact than the format used in the files.

You should never need to call this function; it is called by insert() and friends as necessary.

See also squeeze() [p. 34].

QTranslatorMessage Class Reference

The API for this class is under development and is subject to change.
We do not recommend the use of this class for production work at this time.

The QTranslatorMessage class contains a translator message and its properties.

`#include <qtranslator.h>`

Public Members

- **QTranslatorMessage** ()
- **QTranslatorMessage** (const char * context, const char * sourceText, const char * comment, const QString & translation = QString::null)
- **QTranslatorMessage** (QDataStream & stream)
- **QTranslatorMessage** (const QTranslatorMessage & m)
- QTranslatorMessage & **operator=** (const QTranslatorMessage & m)
- uint **hash** () const
- const char * **context** () const
- const char * **sourceText** () const
- const char * **comment** () const
- void **setTranslation** (const QString & translation)
- QString **translation** () const
- enum **Prefix** { NoPrefix, Hash, HashContext, HashContextSourceText, HashContextSourceTextComment }
- void **write** (QDataStream & stream, bool strip = FALSE, Prefix prefix = HashContextSourceTextComment) const
- Prefix **commonPrefix** (const QTranslatorMessage & m) const
- bool **operator==** (const QTranslatorMessage & m) const
- bool **operator!=** (const QTranslatorMessage & m) const
- bool **operator<** (const QTranslatorMessage & m) const
- bool **operator<=** (const QTranslatorMessage & m) const
- bool **operator>** (const QTranslatorMessage & m) const
- bool **operator>=** (const QTranslatorMessage & m) const

Detailed Description

The QTranslatorMessage class contains a translator message and its properties.

This class is of no interest to most applications, just for translation tools such as Qt Linguist. It is provided simply to make the API complete and regular.

For a QTranslator object, a lookup key is a triple (*context*, *source text*, *comment*) that uniquely identifies a message. An extended key is a quadruple (*hash*, *context*, *source text*, *comment*), where *hash* is computed from the source text and the comment. Unless you plan to read and write messages yourself, you need not worry about the hash value.

QTranslatorMessage stores this triple or quadruple and the relevant translation if there is any.

See also QTranslator [p. 30], Environment Classes and Internationalization with Qt [p. 16].

Member Type Documentation

QTranslatorMessage::Prefix

Let (*h*, *c*, *s*, *m*) be the extended key. The possible prefixes are

- `QTranslatorMessage::NoPrefix` - no prefix
- `QTranslatorMessage::Hash` - only (*h*)
- `QTranslatorMessage::HashContext` - only (*h*, *c*)
- `QTranslatorMessage::HashContextSourceText` - only (*h*, *c*, *s*)
- `QTranslatorMessage::HashContextSourceTextComment` - the whole extended key, (*h*, *c*, *s*, *m*)

See also write() [p. 38] and commonPrefix() [p. 37].

Member Function Documentation

QTranslatorMessage::QTranslatorMessage ()

Constructs a translator message with the extended key (0, 0, 0, 0) and QString::null as translation.

QTranslatorMessage::QTranslatorMessage (const char * context, const char * sourceText, const char * comment, const QString & translation = QString::null)

Constructs an translator message with the extended key (*h*, *context*, *sourceText*, *comment*), where *h* is computed from *sourceText* and *comment*, and possibly with a *translation*.

QTranslatorMessage::QTranslatorMessage (QDataStream & stream)

Constructs a translator message read from a *stream*. The resulting message may have any combination of content.

See also QTranslator::save() [p. 34].

QTranslatorMessage::QTranslatorMessage (const QTranslatorMessage & m)

Constructs a copy of translator message *m*.

const char * QTranslatorMessage::comment () const

Returns the comment for this message (e.g. "File|Save").

Prefix QTranslatorMessage::commonPrefix (const QTranslatorMessage & m) const

Returns the widest lookup prefix that is common to this translator message and message *m*.

For example, if the extended key is for this message is (42, "PrintDialog", "Yes", "Print?") and that for *m* is (42, "PrintDialog", "No", "Print?"), this function returns HashContext.

See also write() [p. 38].

const char * QTranslatorMessage::context () const

Returns the context for this message (e.g. "MyDialog").

uint QTranslatorMessage::hash () const

Returns the hash value used internally to represent the lookup key. This value is zero only if this translator message was constructed from a stream containing invalid data.

The hashing function is unspecified, but it will remain unchanged in future versions of Qt.

bool QTranslatorMessage::operator!= (const QTranslatorMessage & m) const

Returns TRUE if the extended key of this object is different from that of *m*; otherwise returns FALSE.

bool QTranslatorMessage::operator< (const QTranslatorMessage & m) const

Returns TRUE if the extended key of this object is lexicographically before than that of *m*; otherwise returns FALSE.

bool QTranslatorMessage::operator<= (const QTranslatorMessage & m) const

Returns TRUE if the extended key of this object is lexicographically before that of *m* or if they are equal; otherwise returns FALSE.

QTranslatorMessage & QTranslatorMessage::operator= (const QTranslatorMessage & m)

Assigns message *m* to this translator message and returns a reference to this translator message.

bool QTranslatorMessage::operator== (const QTranslatorMessage & m) const

Returns TRUE if the extended key of this object is equal to that of *m*; otherwise returns FALSE.

bool QTranslatorMessage::operator> (const QTranslatorMessage & m) const

Returns TRUE if the extended key of this object is lexicographically after that of *m*; otherwise returns FALSE.

bool QTranslatorMessage::operator>= (const QTranslatorMessage & m) const

Returns TRUE if the extended key of this object is lexicographically after that of *m* or if they are equal; otherwise returns FALSE.

void QTranslatorMessage::setTranslation (const QString & translation)

Sets the translation of the source text to *translation*.

See also translation() [p. 38].

const char * QTranslatorMessage::sourceText () const

Returns the source text of this message (e.g. "&Save").

QString QTranslatorMessage::translation () const

Returns the translation of the source text (e.g., "&Sauvegarder").

See also setTranslation() [p. 38].

void QTranslatorMessage::write (QDataStream & stream, bool strip = FALSE, Prefix prefix = HashContextSourceTextComment) const

Writes this translator message to the *stream*. If *strip* is FALSE (the default), all the information in the message is written. If *strip* is TRUE, only the part of the extended key specified by *prefix* is written with the translation (HashContextSourceTextComment by default).

See also commonPrefix() [p. 37].

QTextCodec Class Reference

The QTextCodec class provides conversion between text encodings.

`#include <qtextcodec.h>`

Inherited by QEucJpCodec [p. 51], QEucKrCodec [p. 53], QGbkCodec [p. 55], QJisCodec [p. 59], QHebrewCodec [p. 57], QSjisCodec [p. 61] and QTsciiCodec [p. 63].

Public Members

- virtual **~QTextCodec** ()
- virtual const char * **name** () const
- virtual const char * **mimeName** () const
- virtual int **mibEnum** () const
- virtual QTextDecoder * **makeDecoder** () const
- virtual QTextEncoder * **makeEncoder** () const
- virtual QString **toUnicode** (const char * chars, int len) const
- virtual QCString **fromUnicode** (const QString & uc, int & lenInOut) const
- QCString **fromUnicode** (const QString & uc) const
- QString **toUnicode** (const QByteArray & a, int len) const
- QString **toUnicode** (const QByteArray & a) const
- QString **toUnicode** (const QCString & a, int len) const
- QString **toUnicode** (const QCString & a) const
- QString **toUnicode** (const char * chars) const
- virtual bool **canEncode** (QChar ch) const
- virtual bool **canEncode** (const QString & s) const
- virtual int **heuristicContentMatch** (const char * chars, int len) const
- virtual int **heuristicNameMatch** (const char * hint) const

Static Public Members

- QTextCodec * **loadCharmap** (QIODevice * iod)
- QTextCodec * **loadCharmapFile** (QString filename)
- QTextCodec * **codecForMib** (int mib)
- QTextCodec * **codecForName** (const char * name, int accuracy = 0)
- QTextCodec * **codecForContent** (const char * chars, int len)

- QTextCodec * **codecForIndex** (int i)
- QTextCodec * **codecForLocale** ()
- void **setCodecForLocale** (QTextCodec * c)
- void **deleteAllCodecs** ()
- const char * **locale** ()

Protected Members

- **QTextCodec** ()

Static Protected Members

- int **simpleHeuristicNameMatch** (const char * name, const char * hint)

Detailed Description

The QTextCodec class provides conversion between text encodings.

Qt uses Unicode to store, draw and manipulate strings. In many situations you may wish to deal with data that uses a different encoding. For example, most japanese documents are still stored in Shift-JIS or ISO2022, while Russian users often have their documents in koi8-r or CP1251.

Qt provides a set of QTextCodec classes to help with converting non-Unicode formats from and to Unicode. You can also create your own codec classes (see later).

The supported encodings are:

- Big5 (Chinese encoding)
- eucJP (one of the many Japanese encodings)
- eucKR (Korean)
- GBK (Chinese encoding)
- JIS7 (Japanese)
- Shift-JIS (Japanese)
- TSCII (Tamil)
- utf8 (Unicode, 8bit)
- utf16 (Unicode)
- KOI8-R (Russian)
- KOI8-U (Ukrainian)
- ISO8859-1 (Western)
- ISO8859-2 (Central Europe)
- ISO8859-3 (Central Europe)
- ISO8859-4 (Baltic)
- ISO8859-5 (Cyrillic)
- ISO8859-6 (Arabic)

- ISO8859-7 (Greek)
- ISO8859-8 (Hebrew, visually ordered)
- ISO8859-8-i (Hebrew, logically ordered)
- ISO8859-9 (Turkish)
- ISO8859-10
- ISO8859-13
- ISO8859-14
- ISO8859-15 (Western)
- CP 874
- CP 1250 (Central Europe)
- CP 1251 (Cyrillic)
- CP 1252 (Western)
- CP 1253 (Greek)
- CP 1254 (Turkish)
- CP 1255 (Hebrew)
- CP 1256 (Arabic)
- CP 1257 (Baltic)
- CP 1258
- Apple Roman
- TIS-620 (Thai)

QTextCodecs can be used as follows to convert some locally encoded string to Unicode. Suppose you have some string encoded in russian KOI8-R encoding, and want to convert it to Unicode. The simple way to do this is:

```
QCString locallyEncoded = "..."; // text to convert
QTextCodec *codec = QTextCodec::codecForName("KOI8-R"); // get the codec for KOI8-R
QString unicodeString = codec->toUnicode( locallyEncoded );
```

After this, `unicodeString` holds the text converted to Unicode. Converting a string from Unicode to the local encoding is as simple:

```
QString unicodeString = ...;
QTextCodec *codec = QTextCodec::codecForName("KOI8-R"); // get the codec for KOI8-R
QCString locallyEncoded = codec->fromUnicode( unicodeString );
```

Some care has to be taken when trying to convert the data in chunks (for example when receiving it over a network). In this case the above approach is too simplistic, because some encodings use more than one byte per character. In this case a character could be split between two chunks of data that are to be converted to Unicode, and the above approach would, at best, lose one character and in some other cases fail completely.

The approach to use in these situations is to create a QTextDecoder object for the codec and use this QTextDecoder for the whole decoding process, as shown below:

```
QTextCodec *c = QTextCodec::codecForName( "Shift-JIS" );
QTextDecoder *decoder = c->makeDecoder();
```

```
QString unicodeString;
while( receiving_data ) {
    QByteArray chunk = new_data;
    unicodeString += decoder->toUnicode( chunk.data(), chunk.length() );
}
```

The QTextDecoder object maintains state between chunks and therefore works correctly even if a multi-byte character is split between chunks.

Creating your own Codec class

By making objects of subclasses of QTextCodec, support for new text encodings can be added to Qt.

You may find it more convenient to make your codec class available as a plugin; see the plugin documentation for more details.

The abstract virtual functions describe the encoder to the system and the coder is used as required in the different text file formats supported by QTextStream, and under X11, for the locale-specific character input and output.

More recently created QTextCodec objects take precedence over earlier ones.

To add support for another 8-bit encoding to Qt, make a subclass of QTextCodec and implement at least the following methods:

```
const char* name() const
```

Return the official name for the encoding.

```
int mibEnum() const
```

Return the MIB enum for the encoding if it is listed in the IANA character-sets encoding file.

If the encoding is multi-byte then it will have "state"; that is, the interpretation of some bytes will be dependent on some preceding bytes. For such an encoding, you will need to implement:

```
QTextDecoder* makeDecoder() const
```

Return a QTextDecoder that remembers incomplete multibyte sequence prefixes or other required state.

If the encoding does *not* require state, you should implement:

```
QString toUnicode(const char* chars, int len) const
```

Converts *len* characters from *chars* to Unicode.

The base QTextCodec class has default implementations of the above two functions, *but they are mutually recursive*, so you must re-implement at least one of them, or both for improved efficiency.

For conversion from Unicode to 8-bit encodings, it is rarely necessary to maintain state. However, two functions similar to the two above are used for encoding:

```
QTextEncoder* makeEncoder() const
```

Return a QTextDecoder.

```
QCString fromUnicode(const QString& uc, int& lenInOut ) const
```

Converts *lenInOut* characters (of type QChar) from the start of the string *uc*, returning a QCString result, and also returning the length of the result in lenInOut.

Again, these are mutually recursive so only one needs to be implemented, or both if better efficiency is possible.

Finally, you must implement:

```
int heuristicContentMatch(const char* chars, int len) const
```

Gives a value indicating how likely it is that *len* characters from *chars* are in the encoding.

A good model for this function is the QWindowsLocalCodec::heuristicContentMatch function found in the Qt sources.

A QTextCodec subclass might have improved performance if you also re-implement:

```
bool canEncode( QChar ) const
```

Test if a Unicode character can be encoded.

```
bool canEncode( const QString& ) const
```

Test if a string of Unicode characters can be encoded.

```
int heuristicNameMatch(const char* hint) const
```

Test if a possibly non-standard name is referring to the codec.

Codecs can also be created as plugins.

See also Internationalization with Qt [p. 16].

Member Function Documentation

QTextCodec::QTextCodec () [protected]

Constructs a QTextCodec, and gives it the highest precedence. The QTextCodec should always be constructed on the heap (i.e. with new()), and once constructed it becomes the responsibility of Qt to delete it (which is done at QApplication destruction).

QTextCodec::~QTextCodec () [virtual]

Destroys the QTextCodec. Note that you should not delete codecs yourself: once created they become Qt's responsibility.

bool QTextCodec::canEncode (QChar ch) const [virtual]

Returns TRUE if the unicode character *ch* can be fully encoded with this codec; otherwise returns FALSE. The default implementation tests if the result of toUnicode(fromUnicode(ch)) is the original *ch*. Subclasses may be able to improve the efficiency.

bool QTextCodec::canEncode (const QString & s) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

s contains the string being tested for encode-ability.

QTextCodec * QTextCodec::codecForContent (const char * chars, int len) [static]

Searches all installed QTextCodec objects, returning the one which most recognizes the given content. May return 0.

Note that this is often a poor choice, since character encodings often use most of the available character sequences, and so only by linguistic analysis could a true match be made.

chars contains the string to check, and *len* contains the number of characters in the string to use.

See also heuristicContentMatch() [p. 45].

Example: qwerty/qwerty.cpp.

QTextCodec * QTextCodec::codecForIndex (int i) [static]

Returns the QTextCodec *i* positions from the most recently inserted codec, or 0 if there is no such QTextCodec. Thus, codecForIndex(0) returns the most recently created QTextCodec.

Example: qwerty/qwerty.cpp.

QTextCodec * QTextCodec::codecForLocale () [static]

Returns a pointer to the codec most suitable for this locale.

Example: qwerty/qwerty.cpp.

QTextCodec * QTextCodec::codecForMib (int mib) [static]

Returns the QTextCodec which matches the MIBenum *mib*.

QTextCodec * QTextCodec::codecForName (const char * name, int accuracy = 0) [static]

Searches all installed QTextCodec objects and returns the one which best matches *name*. Returns a null pointer if no codec's heuristicNameMatch() reports a match better than *accuracy*, or if *name* is a null string.

See also heuristicNameMatch() [p. 45].

void QTextCodec::deleteAllCodecs () [static]

Deletes all the created codecs.

Warning: Do not call this function.

QApplication calls this just before exiting, to delete any QTextCodec objects that may be lying around. Since various other classes hold pointers to QTextCodec objects, it is not safe to call this function earlier.

If you are using the utility classes (like QString) but not using QApplication, calling this function at the very end of your application can be helpful to chase down memory leaks, as QTextCodec objects will not show up.

QCString QTextCodec::fromUnicode (const QString & uc, int & lenInOut) const [virtual]

Subclasses of QTextCodec must reimplement either this function or makeEncoder(). It converts the first *lenInOut* characters of *uc* from Unicode to the encoding of the subclass. If *lenInOut* is negative or too large, the length of *uc* is used instead.

The value returned is owned by the caller, which is responsible for deleting it with "delete []". The length of the resulting Unicode character sequence is returned in *lenInOut*.

The default implementation makes an encoder with makeEncoder() and converts the input with that. Note that the default makeEncoder() implementation makes an encoder that simply calls this function, hence subclasses *must* reimplement one function or the other to avoid infinite recursion.

Reimplemented in QHebrewCodec.

QCString QTextCodec::fromUnicode (const QString & uc) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

uc is the unicode source string.

int QTextCodec::heuristicContentMatch (const char * chars, int len) const [virtual]

Subclasses of QTextCodec must reimplement this function. It examines the first *len* bytes of *chars* and returns a value indicating how likely it is that the string is a prefix of text encoded in the encoding of the subclass. A negative return value indicates that the text is detectably not in the encoding (e.g. it contains characters undefined in the encoding). A return value of 0 indicates that the text should be decoded with this codec rather than as ASCII, but there is no particular evidence. The value should range up to *len*. Thus, most decoders will return -1, 0, or *-len*.

The characters are not null terminated.

See also codecForContent() [p. 44].

int QTextCodec::heuristicNameMatch (const char * hint) const [virtual]

Returns a value indicating how likely it is that this decoder is appropriate for decoding some format that has the given name. The name is compared with the *hint*.

A good match returns a positive number around the length of the string. A bad match is negative.

The default implementation calls simpleHeuristicNameMatch() with the name of the codec.

QTextCodec * QTextCodec::loadCharmap (QIODevice * iod) [static]

Reads a POSIX2 charmap definition from *iod*. The parser recognizes the following lines:

<code_set_name> *name* <escape_char> *character* % alias *alias* CHARMAP <*token*> /x*hexbyte* <U*unicode*> ... <*token*> /d*decbyte* <U*unicode*> ... <*token*> /*octbyte* <U*unicode*> ... <*token*> /*any/any*... <U*unicode*> ... END CHARMAP

The resulting QTextCodec is returned (and also added to the global list of codecs). The name() of the result is taken from the code_set_name.

Note that a codec constructed in this way uses much more memory and is slower than a hand-written QTextCodec subclass, since tables in code are in memory shared by all applications simultaneously using Qt.

See also loadCharmapFile() [p. 46].

Example: qwerty/qwerty.cpp.

QTextCodec * QTextCodec::loadCharmapFile (QString filename) [static]

A convenience function for loadCharmap() that loads the charmap definition from the file *filename*.

const char * QTextCodec::locale () [static]

Returns a string representing the current language.

QTextDecoder * QTextCodec::makeDecoder () const [virtual]

Creates a QTextDecoder which stores enough state to decode chunks of char* data to create chunks of Unicode data. The default implementation creates a stateless decoder, which is sufficient for only the simplest encodings where each byte corresponds to exactly one Unicode character.

The caller is responsible for deleting the returned object.

QTextEncoder * QTextCodec::makeEncoder () const [virtual]

Creates a QTextEncoder which stores enough state to encode chunks of Unicode data as char* data. The default implementation creates a stateless encoder, which is sufficient for only the simplest encodings where each Unicode character corresponds to exactly one character.

The caller is responsible for deleting the returned object.

int QTextCodec::mibEnum () const [virtual]

Subclasses of QTextCodec must reimplement this function. It returns the MIBenum (see the IANA character-sets encoding file for more information). It is important that each QTextCodec subclass return the correct unique value for this function.

Reimplemented in QEucJpCodec.

const char * QTextCodec::mimeName () const [virtual]

Returns the preferred mime name of the encoding as defined in the IANA character-sets encoding file.

Reimplemented in QEucJpCodec, QEucKrCodec, QGbkCodec, QJisCodec, QHebrewCodec and QSjisCodec.

const char * QTextCodec::name () const [virtual]

Subclasses of QTextCodec must reimplement this function. It returns the name of the encoding supported by the subclass. When choosing a name for an encoding, consider these points:

- On X11, heuristicNameMatch(const char * hint) is used to test if a the QTextCodec can convert between Unicode and the encoding of a font with encoding *hint*, such as "iso8859-1" for Latin-1 fonts, "koi8-r" for Russian KOI8 fonts. The default algorithm of heuristicNameMatch() uses name().
- Some applications may use this function to present encodings to the end user.

Example: qwerty/qwerty.cpp.

void QTextCodec::setCodecForLocale (QTextCodec * c) [static]

Set the codec to *c*; this will be returned by codecForLocale. This might be needed for some applications, that want to use their own mechanism for setting the locale.

See also codecForLocale() [p. 44].

int QTextCodec::simpleHeuristicNameMatch (const char * name, const char * hint) [static protected]

A simple utility function for heuristicNameMatch(): it does some very minor character-skipping so that almost-exact matches score high. *name* is the text we're matching and *hint* is used for the comparison.

QString QTextCodec::toUnicode (const char * chars, int len) const [virtual]

Subclasses of QTextCodec must reimplement this function or makeDecoder(). It converts the first *len* characters of *chars* to Unicode.

The default implementation makes a decoder with makeDecoder() and converts the input with that. Note that the default makeDecoder() implementation makes a decoder that simply calls this function, hence subclasses *must* reimplement one function or the other to avoid infinite recursion.

QString QTextCodec::toUnicode (const QByteArray & a, int len) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a contains the source characters; *len* contains the number of characters in *a* to use.

QString QTextCodec::toUnicode (const QByteArray & a) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. *a* contains the source characters.

QString QTextCodec::toUnicode (const QCString & a, int len) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. *a* contains the source characters; *len* contains the number of characters in *a* to use.

QString QTextCodec::toUnicode (const QCString & a) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. *a* contains the source characters.

QString QTextCodec::toUnicode (const char * chars) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function. *chars* contains the source characters.

QTextDecoder Class Reference

The QTextDecoder class provides a state-based decoder.

`#include <qtextcodec.h>`

Public Members

- virtual **~QTextDecoder** ()
- virtual QString **toUnicode** (const char * chars, int len)

Detailed Description

The QTextDecoder class provides a state-based decoder.

The decoder converts a text format into Unicode, remembering any state that is required between calls.

See also QTextCodec::makeEncoder() [p. 46] and Internationalization with Qt [p. 16].

Member Function Documentation

QTextDecoder::~QTextDecoder () [virtual]

Destroys the decoder.

QString QTextDecoder::toUnicode (const char * chars, int len) [virtual]

Converts the first *len* bytes in *chars* to Unicode, returning the result.

If not all characters are used (e.g. if only part of a multi-byte encoding is at the end of the characters), the decoder remembers enough state to continue with the next call to this function.

QTextEncoder Class Reference

The QTextEncoder class provides a state-based encoder.

`#include <qtextcodec.h>`

Public Members

- virtual **~QTextEncoder** ()
- virtual QCString **fromUnicode** (const QString & uc, int & lenInOut)

Detailed Description

The QTextEncoder class provides a state-based encoder.

The encoder converts Unicode into another format, remembering any state that is required between calls.

See also QTextCodec::makeEncoder() [p. 46] and Internationalization with Qt [p. 16].

Member Function Documentation

QTextEncoder::~QTextEncoder () [virtual]

Destroys the encoder.

QCString QTextEncoder::fromUnicode (const QString & uc, int & lenInOut) [virtual]

Converts *lenInOut* characters (not bytes) from *uc*, producing a QCString. *lenInOut* will be set to the length of the result (in bytes).

The encoder is free to record state to use when subsequent calls are made to this function (for example, it might change modes with escape sequences if needed during the encoding of one string, then assume that mode applies when a subsequent call begins).

QEucJpCodec Class Reference

The QEucJpCodec class provides conversion to and from EUC-JP character sets.

`#include <qeucjpcodec.h>`

Inherits QTextCodec [p. 39].

Public Members

- virtual int **mibEnum** () const
- virtual const char * **mimeName** () const
- **QEucJpCodec** ()
- **~QEucJpCodec** ()

Detailed Description

The QEucJpCodec class provides conversion to and from EUC-JP character sets.

More precisely, the QEucJpCodec class subclasses QTextCodec to provide support for EUC-JP, the main legacy encoding for Unix machines in Japan.

The environment variable UNICODEMAP_JP can be used to fine-tune QJisCodec, QSjisCodec and QEucJpCodec. The QJisCodec documentation describes how to use this variable.

Most of the code here was written by Serika Kurusugawa, a.k.a. Junji Takagi, and is included in Qt with the author's permission and the grateful thanks of the Trolltech team. Here is the copyright statement for that code:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

See also Internationalization with Qt [p. 16].

Member Function Documentation

QEucJpCodec::QEucJpCodec ()

Constructs a QEucJpCodec.

QEucJpCodec::~QEucJpCodec ()

Destroys the codec.

int QEucJpCodec::mibEnum () const [virtual]

Returns 18.

Reimplemented from QTextCodec [p. 46].

const char * QEucJpCodec::mimeName () const [virtual]

Returns the codec's mime name.

Reimplemented from QTextCodec [p. 47].

QEucKrCodec Class Reference

The QEucKrCodec class provides conversion to and from EUC-KR character sets.

`#include <qeuckrcodec.h>`

Inherits QTextCodec [p. 39].

Public Members

- virtual const char * **mimeName** () const

Detailed Description

The QEucKrCodec class provides conversion to and from EUC-KR character sets.

The QEucKrCodec class subclasses QTextCodec to provide support for EUC-KR, the main legacy encoding for UNIX machines in Korea.

It was largely written by Mizi Research Inc. Here is the copyright statement for the code as it was at the point of contribution (Trolltech's subsequent modifications are covered by the usual copyright for Qt.)

See also Internationalization with Qt [p. 16].

Member Function Documentation

const char * QEucKrCodec::mimeName () const [virtual]

Returns the codec's mime name.

Reimplemented from QTextCodec [p. 47].

QGbkCodec Class Reference

The QGbkCodec class provides conversion to and from the Chinese GBK encoding.

`#include <qgbkcodec.h>`

Inherits QTextCodec [p. 39].

Public Members

- virtual const char * **mimeName** () const

Detailed Description

The QGbkCodec class provides conversion to and from the Chinese GBK encoding.

GBK, formally the Chinese Internal Code Specification, is a commonly used extension of GB 2312-80. Microsoft Windows uses it under the name code page 936.

The GBK codec was contributed to Qt by Justin Yu <justiny@turbolinux.com.cn> and Sean Chen <seanc@turbolinux.com.cn>. The copyright notice for their code follows:

Copyright 2000 TurboLinux, Inc. Written by Justin Yu and Sean Chen.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS", AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

See also Internationalization with Qt [p. 16].

Member Function Documentation

const char * QGbkCodec::mimeName () const [virtual]

Returns the codec's mime name.

Reimplemented from QTextCodec [p. 47].

QHebrewCodec Class Reference

The QHebrewCodec class provides conversion to and from visually ordered Hebrew.

`#include <qrtlcodec.h>`

Inherits QTextCodec [p. 39].

Public Members

- virtual const char * **mimeName** () const
- virtual QCString **fromUnicode** (const QString & uc, int & len_in_out) const

Detailed Description

The QHebrewCodec class provides conversion to and from visually ordered Hebrew.

Hebrew as a semitic language is written from right to left. As older computer systems couldn't handle reordering a string so that the first letter appears on the right, many older documents were encoded in visual order, so that the first letter of a line is the rightmost one in the string.

Opposed to this, Unicode defines characters to be in logical order (the order you would read the string). This codec tries to convert visually ordered Hebrew (8859-8) to Unicode. This might not always be 100%, as reversing the bidi algorithm that transforms from logical to visual order is non trivial.

Transformation from Unicode to visual Hebrew (8859-8) is done using the BiDi algorithm in Qt, and will produce correct results, as long as you feed one paragraph of text to the codec at a time. Places where newlines are supposed to start can be indicated by a newline character ('\n'). Please be aware, that these newline characters change the reordering behaviour of the algorithm, as the BiDi reordering only takes place within one line of text, whereas linebreaks are determined in visual order.

Visually ordered Hebrew is still used quite often in some places, mainly in email communication (as most email programs still don't understand logically ordered Hebrew) and on web pages. The use on web pages is strongly decreasing however, as there are nowadays a few browsers available that correctly support logically ordered Hebrew.

This codec has the name "iso8859-8". If you don't want any bidi reordering to happen during conversion, use the "iso8859-8-i" codec, which assumes logical order for the 8bit string.

See also Internationalization with Qt [p. 16].

Member Function Documentation

QCString QHebrewCodec::fromUnicode (const QString & uc, int & len_in_out) const [virtual]

Transforms the logically ordered QString, *uc*, into a visually ordered string in the 8859-8 encoding. Qt's BiDi algorithm is used to perform this task. Please note, that newline characters affect the reordering, as reordering is done on a line by line basis.

You might get wrong results if you feed the string line by line to this method, as the algorithm is designed to operate on a whole paragraph of text at a time, and the contents of a previous line may affect the reordering of the next line.

To ensure you get correct results always call this method with an entire paragraph of text to reorder.

Some encodings (for example japanese or utf8) are multibye (so one input character is mapped to two output characters). The *len_in_out* argument specifies the number of QChars that should be converted and is set to the number of characters returned.

Reimplemented from QTextCodec [p. 45].

const char * QHebrewCodec::mimeName () const [virtual]

Returns the codec's mime name.

Reimplemented from QTextCodec [p. 47].

QJisCodec Class Reference

The QJisCodec class provides conversion to and from JIS character sets.

`#include <qjiscodec.h>`

Inherits QTextCodec [p. 39].

Public Members

- virtual const char * **mimeName** () const

Detailed Description

The QJisCodec class provides conversion to and from JIS character sets.

More precisely, the QJisCodec class subclasses QTextCodec to provide support for JIS X 0201 Latin, JIS X 0201 Kana, JIS X 0208 and JIS X 0212.

The environment variable UNICODEMAP_JP can be used to fine-tune QJisCodec, QSjisCodec and QEucJpCodec. The mapping names are as for the Japanese XML working group's XML Japanese Profile, because it names and explains all the widely used mappings. Here are brief descriptions, written by Serika Kurusugawa:

- "unicode-0.9" or "unicode-0201" for Unicode style. This assumes JISX0201 for 0x00-0x7f. (0.9 is a table version of jisx02xx mapping used for Uniocde spec version 1.1.)
- "unicode-ascii" This assumes US-ASCII for 0x00-0x7f; some chars (JISX0208 0x2140 and JISX0212 0x2237) are different from Unicode 1.1 to avoid conflict.
- "open-19970715-0201" ("open-0201" for convenience) or "jisx0221-1995" for JISX0221-JISX0201 style. JIS X 0221 is JIS version of Unicode, but a few chars (0x5c, 0x7e, 0x2140, 0x216f, 0x2131) are different from Unicode 1.1. This is used when 0x5c is treated as YEN SIGN.
- "open-19970715-ascii" ("open-ascii" for convenience) for JISX0221-ASCII style. This is used when 0x5c is treated as REVERSE SOLIDUS.
- "open-19970715-ms" ("open-ms" for convenience) or "cp932" for Microsoft Windows style. Windows Code Page 932. Some chars (0x2140, 0x2141, 0x2142, 0x215d, 0x2171, 0x2172) are different from Unicode 1.1.
- "jdk1.1.7" for Sun's JDK style. Same as Unicode 1.1, except that JIS 0x2140 is mapped to UFF3C. Either ASCII or JISX0201 can be used for 0x00-0x7f.

In addition, the extensions "nec-vdc", "ibm-vdc" and "udc" are supported.

For example, if you want to use Unicode style conversion but with NEC's extension, set `UNICODEMAP_JP` to `unicode-0.9, nec-vdc`. (You will probably need to quote that in the shell command.)

Most of the code here was written by Serika Kurusugawa, a.k.a. Junji Takagi, and is included in Qt with the author's permission and the grateful thanks of the Trolltech team. Here is the copyright statement for that code:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

See also Internationalization with Qt [p. 16].

Member Function Documentation

const char * QJisCodec::mimeName () const [virtual]

Returns the codec's mime name.

Reimplemented from QTextCodec [p. 47].

QSjisCodec Class Reference

The QSjisCodec class provides conversion to and from Shift-JIS.

`#include <qsjiscodec.h>`

Inherits QTextCodec [p. 39].

Public Members

- virtual const char * **mimeName** () const
- **QSjisCodec** ()
- **~QSjisCodec** ()

Detailed Description

The QSjisCodec class provides conversion to and from Shift-JIS.

More precisely, the QSjisCodec class subclasses QTextCodec to provide support for Shift-JIS, an encoding of JIS X 0201 Latin, JIS X 0201 Kana or JIS X 0208.

The environment variable UNICODEMAP_JP can be used to fine-tune QJisCodec, QSjisCodec and QEucJpCodec. The QJisCodec documentation describes how to use this variable.

Most of the code here was written by Serika Kurusugawa, a.k.a. Junji Takagi, and is included in Qt with the author's permission and the grateful thanks of the Trolltech team. Here is the copyright statement for that code:

See also Internationalization with Qt [p. 16].

Member Function Documentation

QSjisCodec::QSjisCodec ()

Creates a Shift-JIS codec. Note that this is done automatically by the QApplication, you do not need construct your own.

QSjisCodec::~QSjisCodec ()

Destroys the Shift-JIS codec.

const char * QSjisCodec::mimeName () const [virtual]

Returns the codec's mime name.

Reimplemented from QTextCodec [p. 47].

QTsciiCodec Class Reference

The QTsciiCodec class provides conversion to and from the Tamil TSCII encoding.

`#include <qtsciicodec.h>`

Inherits QTextCodec [p. 39].

Detailed Description

The QTsciiCodec class provides conversion to and from the Tamil TSCII encoding.

TSCII, formally the Tamil Standard Code Information Interchange specification, is a commonly used charset for Tamils. The official page for the standard is at http://www.tamil.net/tscii/

This codec uses the mapping table found at

http://www.geocities.com/Athens/5180/tsciiset.html. Unfortunately Tamil uses composed Unicode. This might cause some trouble if you are using Unicode fonts instead of TSCII fonts.

The TSCII codec was contributed to Qt by Hans Petter Bieker <bieker@kde.org>. The copyright notice for his code follows:

See also Internationalization with Qt [p. 16].

Index