Input/Output and Networking 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

Network Module 4
Format of the QDataStream Operators 11
QClipboard Class Reference 15
QDataStream Class Reference 19
QDir Class Reference 29
QDns Class Reference 45
QFile Class Reference 50
QFileInfo Class Reference 59
QFtp Class Reference 69
QHostAddress Class Reference 71
QHttp Class Reference 74
QIODevice Class Reference 76
QLocalFs Class Reference 86
QLock Class Reference 87
QMimeSource Class Reference 89
QMimeSourceFactory Class Reference 91
QNetworkOperation Class Reference 96
QNetworkProtocol Class Reference 99
QProcess Class Reference 108
QServerSocket Class Reference 118
QSessionManager Class Reference 121
QSocket Class Reference 126
QSocketDevice Class Reference 135
QSocketNotifier Class Reference 142
QTextIStream Class Reference 146
QTextOStream Class Reference 148
QTextStream Class Reference 150

QUrl Class Reference **163**
QUrlInfo Class Reference **172**
QUrlOperator Class Reference **179**
QWindowsMime Class Reference **188**
Index **191**

Network Module

This module is part of the Qt Enterprise Edition.

Introduction

The network module offers classes to make network programming easier and portable. Basically there are three sets of classes, first very basic classes like QSocket, QServerSocket, QDns, etc. which allow to work in a portable way with TCP/IP sockets. In addition, there are classes like QNetworkProtocol, QNetworkOperation in the Qt base library, which provide an abstract layer for implementing network protocols and QUrlOperator which operates on such network protocols. Finally the third set of network classes are the passive ones, namely QUrl and QUrlInfo which do URL parsing and similar stuff.

The first set of classes (QSocket, QServerSocket, QDns, QFtp, etc.) are included in the "network" module of Qt.

The QSocket classes are not directly related to the QNetwork classes, but QSocket should and will be used for implementing network protocols, which are directly related to the QNetwork classes. E.g. the QFtp class (implementation of the FTP protocol) uses QSockets. But QSockets don't need to be used for protocol implementations, e.g. QLocalFs (which is an implementation of the local filesystem as network protocol) uses QDir and no QSocket. Using QNetworkProtocols you can implement everything which fits into a hierarchical structure and can be accessed using URLs. This could be e.g. a protocol which can read pictures from a digital camera using a serial connection.

Working Network Protocol independent with QUrlOperator and QNetworkOperation

To just use existing network protocol implementations and operate on URLs using them is quite easy. E.g. downloading a file from an FTP server to the local filesystem can be done with following code:

```
QUrlOperator op;
op.copy( "ftp://ftp.trolltech.com/qt/source/qt-2.1.0.tar.gz", "file:/tmp", FALSE );
```

And that's all! Of course an implementation of the FTP protocol has to be available and registered for doing that. More information on that later.

You can also do stuff like creating directories, removing files, renaming, etc. E.g. to create a folder on a private FTP account do

```
QUrlOperator op( "ftp://username:password@host.domain.no/home/username" );
op.mkdir( "New Directory" );
```

That's it again. To see all available operations, look at the QUrlOperator class documentation.

Now as everything works asynchronous, the function call for an operation returns before the operation has been processed. So you don't get a return value which tells you something about failure or success. The return value always is a pointer to a QNetworkOperation.

In this QNetworkOperation all information about the operation is stored. There is e.g. a method of QNetworkOperation which returns the state of this operation. Using that you can find out all the time in which state the operation currently is. Also you get the arguments you passed to the QUrlOperator method, the type of the operation and some more stuff from this QNetworkOperation object. For more details see the class documentation of QNetworkOperation.

Now, later you get signals emitted by the QUrlOperator, which inform you about the process of the operations. As you can call many methods which operate on a URL of one QUrlOperator, it queues up all these operations. So you can't know which operation the QUrlOperator just processes. Because of this you get in each signal as the last argument a pointer to the QNetworkOperation object which is just processed and from which this signal comes.

Some of these operations send a `start()` signal at the beginning (depending if it makes sense or not), then some of them send some signals during processing the operation, and *all* operations send a `finished()` signal after they are done. Now, finished could mean that the operation has been successfully finished or that it failed. To find that out you can use the QNetworkOperation pointer you got with the `finished()` signal. If QNetworkOperation::state() equals QNetworkProtocol::StDone the operation finished successful, if it is QNetworkProtocol::StFailed the operation failed.

Now, a slot which you connected to the `QUrlOperator::finished( QNetworkOperation * )` signal could look like this

```
void MyClass::slotOperationFinished( QNetworkOperation *op )
{
    switch ( op->operation() ) {
    case QNetworkProtocol::OpMkDir: {
        if ( op->state() == QNetworkProtocol::StFailed )
            qDebug( "Couldn't create directory %s", op->arg( 0 ).latin1() );
        else
            qDebug( "Successfully created directory %s", op->arg( 0 ).latin1() );
    } break;
    // ... and so on
    }
}
```

As mentioned before, some operations send other signals too. Let's take the list children operation as an example (e.g. read the directory of a directory on a FTP server):

```
QUrlOperator op;

MyClass::MyClass() : QObject(), op( "ftp://ftp.trolltech.com" )
{
    connect( &op, SIGNAL( newChildren( const QValueList &, QNetworkOperation * ) ),
             this, SLOT( slotInsertEntries( const QValueList &, QNetworkOperation * ) ) );
    connect( &op, SIGNAL( start( QNetworkOperation * ) ),
             this, SLOT( slotStart( QNetworkOperation *) ) );
    connect( &op, SIGNAL( finished( QNetworkOperation * ) ),
             this, SLOT( slotFinished( QNetworkOperation *) ) );
}

void MyClass::slotInsertEntries( const QValueList &info, QNetworkOperation * )
```

```
{
    QValueList::ConstIterator it = info.begin();
    for ( ; it != info.end(); ++it ) {
        const QUrlInfo &inf = *it;
        qDebug( "Name: %s, Size: %d, Last Modified: %s",
            inf.name().latin1(), inf.size(), inf.lastModified().toString().latin1() );
    }
}

void MyClass::slotStart( QNetworkOperation * )
{
    qDebug( "Start reading '%s'", op.toString().latin1() );
}

void MyClass::slotFinished( QNetworkOperation *operation )
{
    if ( operation->operation() == QNetworkProtocol::OpListChildren ) {
        if ( operation->state() == QNetworkProtocol::StFailed )
            qDebug( "Couldn't read '%s'! Following error occurred: %s",
                op.toString().latin1(), operation->protocolDetail().latin1() );
        else
            qDebug( "Finished reading '%s'!", op.toString().latin1() );
    }
}
```

These examples explained now how to use the QUrlOperator and `QNetworkOperations`. The network extension will contain some good examples for this too.

Implementing your own Network Protocol

QNetworkProtocol provides a base class for implementations of network protocols and an architecture to a dynamic registration and unregistration of network protocols. If you use this architecture you also don't need to care about asynchronous programming, as the architecture hides this and does all the work for you.

Limitation: As it is quite hard to design a base class for network protocols which satisfies all network protocols, the architecture described here is designed to work with all kinds of hierarchical structures, like filesystems. So everything which can be interpreted as hierarchical structure and accessed via URLs, can be implemented as network protocol and easily used in Qt. This is not limited to filesystems only!

To implement a network protocol create a class derived from QNetworkProtocol.

Other classes will use this network protocol implementation to operate on it. So you should reimplement following protected members

```
    void QNetworkProtocol::operationListChildren( QNetworkOperation *op );
    void QNetworkProtocol::operationMkDir( QNetworkOperation *op );
    void QNetworkProtocol::operationRemove( QNetworkOperation *op );
    void QNetworkProtocol::operationRename( QNetworkOperation *op );
    void QNetworkProtocol::operationGet( QNetworkOperation *op );
    void QNetworkProtocol::operationPut( QNetworkOperation *op );
```

Some words about how to reimplement these methods: You always get a pointer to a QNetworkOperation as argument. This pointer holds all information about the operation in the current state. If you start processing such an operation, set

the state to QNetworkProtocol::StInProgress. If you finished processing the operation, set the state to QNetworkProtocol::StDone if it was successful or QNetworkProtocol::StFailed if an error occurred. If an error occurred you have to set an error code (see QNetworkOperation::setErrorCode()) and if you know some details (e.g. an error message) you can also set this message to the operation pointer (see QNetworkOperation::setProtocolDetail()). Also you get all information (type, arguments, etc.) of the operation from this QNetworkOperation pointer. For details about which arguments you can get and set look at the class documentation of QNetworkOperation.

If you reimplement such an operation method, it's also very important to emit the correct signals at the correct time: In general always emit at the end of an operation (when you either successfully finished processing the operation or and error occurred) the `finished()` signal with the network operation as argument. The whole network architecture relies on correctly emitted `finished()` signals! So be careful with that! Then there are some more special signals which are specific to operations:

- Emit in `operationListChildren`:
  - `start()` just before starting listing the children
  - `newChildren()` when new children are read
- Emit in `operationMkDir`:
  - `createdDirectory()` after the directory has been created
  - `newChild()` (or newChildren()) after the directory has been created (as a new directory is a new child)
- Emit in `operationRemove`:
  - `removed()` after the child has been removed
- Emit in `operationRename`:
  - `itemChanged()` after the child has been renamed
- Emit in `operationGet`:
  - `data()` each time new data has been read
  - `dataTransferProgress()` each time new data has been read to indicate how much of the data has been read now.
- Emit in `operationPut`:
  - `dataTransferProgress()` each time data has been written to indicate how much of the data has been written. Although you know the whole data when this operation is called, it's suggested not to write the whole data at once, but to do it step by step to avoid blocking the GUI and also this way the progress can be made visible to the user.

And remember, always emit the `finished()` signal at the end!

For more details about the arguments of these signals take a look at the QNetworkProtocol class documentation.

Now, as argument in such a method you get the QNetworkOperation which you process. Here is a list which arguments of the QNetworkOperation you can get and which you have to set in which method:

(To get the URL on which you should work, use the QNetworkProtocol::url() method which returns the pointer to the URL operator. Using that you can get the path, host, name filter and everything else of the URL)

- In `operationListChildren`:
  - Nothing.
- In `operationMkDir`:

  - `QNetworkOperation::arg( 0 )` contains the name of the directory which should be created

- In `operationRemove`:
  - `QNetworkOperation::arg( 0 )` contains the name of the file which should be removed. Normally this is a relative name. But it may be absolute too, so use QUrl(op->arg(0)).fileName() to get the filename.

- In `operationRename`:
  - `QNetworkOperation::arg( 0 )` contains the name of the file which should be renamed
  - `QNetworkOperation::arg( 1 )` contains the name to which it should be renamed.

- In `operationGet`:
  - `QNetworkOperation::arg( 0 )` contains the full URL of the file which should be retrieved.

- In `operationPut`:
  - `QNetworkOperation::arg( 0 )` contains the full URL of the file in which the data should be stored.
  - `QNetworkOperation::rawArg( 1 )` contains the data which should be stored in `QNetworkOperation::arg( 0 )`

So, to sum it up: If you reimplement such an operation method, you have to emit some special signals and *always* at the end a `finished()` signal, either on success or on failure. Also you have to change the state of the QNetworkOperation during processing it and can get and set arguments of the operation as well.

But it's unlikely that the network protocol you implement supports all these operations. So, just reimplement the operations, which are supported by the protocol. Additionally you have to specify which operations are supported then. This is done by reimplementing

```
int QNetworkProtocol::supportedOperations() const;
```

In your implementation of this method return an int value which is constructed by or'ing together the correct values (supported operations) of the following enum (of QNetworkProtocol):

```
enum Operation {
    OpListChildren = 1,
    OpMkDir = 2,
    OpRemove = 4,
    OpRename = 8,
    OpGet = 32,
    OpPut = 64
};
```

So, if your protocol e.g. supports listing children and renaming them, do in your implementation of `supportedOperations()`:

```
return OpListChildren | OpRename;
```

The last method you have to reimplement is

```
bool QNetworkProtocol::checkConnection( QNetworkOperation *op );
```

Here you have to return TRUE, if the connection is up and ok (this means operations on the protocol can be done). If the connection is not ok, return FALSE and start to try opening it. If you will not be able to open the connection at all (e.g. because the host is not found), emit a `finished()` signal and set an error code and the QNetworkProtocol::StFailed state to the QNetworkOperation pointer you get here.

Now, you never need to check before doing an operation yourself, if the connection is ok. The network architecture does this, this means using `checkConnection()` it looks if an operation could be done and if not, it tries it again and again for some time and only calls an operation method if the connection is ok.

Using this knowledge it should be possible to implement network protocols. Finally to be able to use it with a QUrlOperator (and so e.g. in the QFileDialog), you have to register the network protocol implementation. This can be done like this:

```
QNetworkProtocol::registerNetworkProtocol( "myprot", new QNetworkProtocolFactory );
```

In this case `MyProtocol` would be a class you implemented like described here (derived from QNetworkProtocol) and the name of the protocol would be myprot. So if you want to use it, you would do something like

```
QUrlOperator op( "myprot://host/path" );
op.listChildren();
```

Finally as example for a network protocol implementation you could look at the implementation of QLocalFs. The network extension will also contain an example implementation of a network protocol

Error Handling

Error handling is important for both, implementing new network protocols and using them (through QUrlOperator). So first some words about error handling when using the network protocols:

As already mentioned quite some times after processing an operation has been finished the network operation and so the QUrlOperator emits the `finished()` signal. This has as argument the pointer to the processed QNetworkOperation. If the state of this operation is QNetworkProtocol::StFailed, the operation contains some more information about this error. Following error codes are defined in QNetworkProtocol:

- QNetworkProtocol::NoError - No error occurred
- QNetworkProtocol::ErrValid - The URL you are operating on is not valid
- QNetworkProtocol::ErrUnknownProtocol - There is no protocol implementation available for the protocol of the URL you are operating on (e.g. if the protocol is http and no http implementation has been registered)
- QNetworkProtocol::ErrUnsupported - The operation is not supported by the protocol
- QNetworkProtocol::ErrParse - Parse error of the URL
- QNetworkProtocol::ErrLoginIncorrect - You needed to login but the username and or password are wrong
- QNetworkProtocol::ErrHostNotFound - The specified host (in the URL) couldn't be found
- QNetworkProtocol::ErrListChildren - An error occurred while listing the children
- QNetworkProtocol::ErrMkDir - An error occurred when creating a directory
- QNetworkProtocol::ErrRemove -An error occurred while removing a child
- QNetworkProtocol::ErrRename - An error occurred while renaming a child
- QNetworkProtocol::ErrGet - An error occurred while getting (retrieving) data
- QNetworkProtocol::ErrPut - An error occurred while putting (uploading) data

- QNetworkProtocol::ErrFileNotExisting - A file which is needed by the operation doesn't exist
- QNetworkProtocol::ErrPermissionDenied - The permission for doing the operation has been denied

QNetworkOperation::errorCode() returns then one of these codes or maybe a different one if you use an own network protocol implementation which defines additional error codes.

`QNetworkOperation::protocolDetails()` may also return a string which contains an error message then which could e.g. be displayed for the user.

According to this information it should be possible to react on errors.

Now, if you implement your own network protocol, you will need to tell about errors which occurred. First you always need to be able to access the QNetworkOperation which is processed at the moment. This can be done using `QNetworkOperation::operationInProgress()`, which returns a pointer to the current network operation or 0 if no operation is processed at the moment.

Now if and error occurred and you need to handle it, do

```
if ( operationInProgress() ) {
    operationInProgress()->setErrorCode( error_code_of_your_error );
    operationInProgress()->setProtocolDetails( detail ); // optional!
    emit finished( operationInProgress() );
    return;
}
```

That's all. The connection to the QUrlOperator and so on is done automatically. Additionally, if the error was really bad so that no more operations can be done in the current state (e.g. if the host couldn't be found), call `QNetworkProtocol::clearOperationStack()` *before* emitting `finished()`.

Now, as error code you should use, if possible, one of the predefined error codes of QNetworkProtocol. If this is not possible, you can add own error codes - they are just normal `integers`. Just be careful that the value of the error code doesn't conflict with an existing one.

Documentation about the low-level classes like QSocket, QDns, etc. will be included in the seperate network extension.

For internal use only.

Format of the QDataStream Operators

The QDataStream allows you to serialize some of the Qt data types. The table below lists the data types that QDataStream can serialize and how they are represented.

- Q_INT8
  - signed byte
- Q_INT16
  - signed 16 bit integer
- Q_INT32
  - signed 32 bit integer
- Q_UINT8
  - unsigned byte
- Q_UINT16
  - unsigned 16 bit integer
- Q_UINT32
  - unsigned 32 bit integer
- float
  - 32-bit floating point number using the standard IEEE-754 format
- double
  - 64-bit floating point number using the standard IEEE-754 format
- char *
  - The size of the string including the terminating 0 (Q_UINT32)
  - The string bytes including the terminating 0

  The null string is represented as `(Q_UINT32) 0`.
- QBitArray
  - The array size (Q_UINT32)
  - The array bits, i.e. (size + 7)/8 bytes
- QBrush

  - The brush style (Q_UINT8)
  - The brush color (QColor)
  - If style is CustomPattern, the brush pixmap (QPixmap)

- QByteArray
  - The array size (Q_UINT32)
  - The array bytes, i.e. size bytes

- QCString
  - The size of the string including the terminating 0 (Q_UINT32)
  - The string bytes including the terminating 0

  The null string is represented as `(Q_UINT32) 0`.

- QColor
  - RGB value serialized as a Q_UINT32

- QColorGroup
  - foreground (QBrush)
  - button (QBrush)
  - light (QBrush)
  - midLight (QBrush)
  - dark (QBrush)
  - mid (QBrush)
  - text (QBrush)
  - brightText (QBrush)
  - ButtonText (QBrush)
  - base (QBrush)
  - background (QBrush)
  - shadow (QBrush)
  - highlight (QBrush)
  - highlightedText (QBrush)

- QCursor
  - Shape id (Q_INT16)
  - If shape is BitmapCursor: The bitmap (QPixmap), mask (QPixmap) and hot spot (QPoint)

- QDate
  - Julian day (Q_UINT32)

- QDateTime
  - Date (QDate)
  - Time (QTime)

- QFont
  - The point size (Q_INT16)
  - The style hint (Q_UINT8)
  - The char set (Q_UINT8)

  - The weight (Q_UINT8)
  - The font bits (Q_UINT8)
- QImage
  - Save it as a PNG image.
- QMap
  - The number of items (Q_UINT32)
  - For all items, the key and value
- QPalette
  - active (QColorGroup)
  - disabled (QColorGroup)
  - inactive (QColorGroup)
- QPen
  - The pen styles (Q_UINT8)
  - The pen width (Q_UINT8)
  - The pen color (QColor)
- QPicture
  - The size of the picture data (Q_UINT32)
  - The raw bytes of picture data (char)
- QPixmap
  - Save it as a PNG image.
- QPoint
  - The x coordinate (Q_INT32)
  - The y coordinate (Q_INT32)
- QPointArray
  - The array size (Q_UINT32)
  - The array points (QPoint)
- QRect
  - left (Q_INT32)
  - top (Q_INT32)
  - right (Q_INT32)
  - bottom (Q_INT32)
- QRegion
  - The size of the data, i.e. 8 + 16 * (number of rectangles) (Q_UINT32)
  - QRGN_RECTS (Q_INT32)
  - The number of rectangles (Q_UINT32)
  - The rectangles in sequential order (QRect)
- QSize

  - width (Q_INT32)
  - height (Q_INT32)

- QString
  - If the string is null: 0xffffffff (Q_UINT32)
  - Otherwise: The string length (Q_UINT32) followed by the data in UTF-16
- QTime
  - Milliseconds since midnight (Q_UINT32)
- QValueList
  - The number of list elements (Q_UINT32)
  - All the elements in sequential order
- QVariant
  - The type of the data (Q_UINT32)
  - The data of the specified type
- QWMatrix
  - m11 (double)
  - m12 (double)
  - m21 (double)
  - m22 (double)
  - dx (double)
  - dy (double)

QClipboard Class Reference

The QClipboard class provides access to the window system clipboard.

`#include <qclipboard.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- void **clear** ()
- bool **supportsSelection** () const
- bool **ownsSelection** () const
- bool **ownsClipboard** () const
- void **setSelectionMode** (bool enable)
- bool **selectionModeEnabled** () const
- QMimeSource * **data** () const
- void **setData** (QMimeSource * src)
- QString **text** () const
- QString **text** (QCString & subtype) const
- void **setText** (const QString & text)
- QImage **image** () const
- QPixmap **pixmap** () const
- void **setImage** (const QImage & image)
- void **setPixmap** (const QPixmap & pixmap)

Signals

- void **selectionChanged** ()
- void **dataChanged** ()

Detailed Description

The QClipboard class provides access to the window system clipboard.

The clipboard offers a simple mechanism to copy and paste data between applications.

QClipboard supports the same data types that QDragObject does, and uses similar mechanisms. For advanced clipboard usage, you should read the drag-and-drop documentation.

There is a single QClipboard object in an application, and you can access it using QApplication::clipboard().

Example:

```
QClipboard *cb = QApplication::clipboard();
QString text;

// Copy text from the clipboard (paste)
text = cb->text();
if ( text )
    qDebug( "The clipboard contains: %s", text );

// Copy text into the clipboard
cb->setText( "This text can be pasted by other programs" );
```

QClipboard features some convenience functions to access common data types: setText() allows the exchange of Unicode text and setPixmap() and setImage() allows the exchange of QPixmaps and QImages between applications. The setData() function is the ultimate in flexibility: it allows you to add any QMimeSource into the clipboard. (There are corresponding getters for each of these, e.g. text().)

You can clear the clipboard by calling clear().

The underlying clipboards of the X Window system and MS Windows differ. The X Window system has a concept of selection — when text is selected it is immediately available in the selection buffer; MS Windows only adds text to the clipboard when an explicit copy or cut is made. The X Window system also has a concept of ownership; if you change the selection within a window X11 will only notify the owner and the previous owner of the change; in MS Windows the clipboard is a fully global resource so all applications are notified of changes. See the multiclip example in the *Qt Designer* examples directory for an example of a cross-platform clipboard application that also demonstrates selection handling.

See also Input/Output and Networking.

Member Function Documentation

void QClipboard::clear ()

Clears the clipboard contents.

QMimeSource * QClipboard::data () const

Returns a reference to a QMimeSource representation of the current clipboard data.

void QClipboard::dataChanged () [signal]

This signal is emitted when the clipboard data is changed.

QImage QClipboard::image () const

Returns the clipboard image, or returns a null image if the clipboard does not contain an image or if it contains an image in an unsupported image format.

See also setImage() [p. 17], pixmap() [p. 17], data() [p. 16] and QImage::isNull() [Graphics with Qt].

bool QClipboard::ownsClipboard () const

Returns TRUE is this clipboard object owns the clipboard data, FALSE otherwise.

bool QClipboard::ownsSelection () const

Returns TRUE if this clipboard object owns the mouse selection data, FALSE otherwise.

QPixmap QClipboard::pixmap () const

Returns the clipboard pixmap, or null if the clipboard does not contain a pixmap. Note that this can lose information. For example, if the image is 24-bit and the display is 8-bit, the result is converted to 8 bits, and if the image has an alpha channel the result just has a mask.

See also setPixmap() [p. 18], image() [p. 17], data() [p. 16] and QPixmap::convertFromImage() [Graphics with Qt].

void QClipboard::selectionChanged () [signal]

This signal is emitted when the selection is changed. This only applies to windowing systems that support selections, e.g. X11. Windows doesn't support selections.

bool QClipboard::selectionModeEnabled () const

Returns the value set by setSelectionMode().

See also setSelectionMode() [p. 18] and supportsSelection() [p. 18].

void QClipboard::setData (QMimeSource * src)

Sets the clipboard data to *src*. Ownership of the data is transferred to the clipboard. If you want to remove the data either call clear() or call setData() again with new data.

The QDragObject subclasses are reasonable objects to put into the clipboard (but do not try to call QDragObject::drag() on the same object). Any QDragObject placed in the clipboard should have a parent of 0. Do not put QDragMoveEvent or QDropEvent subclasses in the clipboard, as they do not belong to the event handler which receives them.

The setText() and setPixmap() functions are simpler wrappers for setting text and image data respectively.

void QClipboard::setImage (const QImage & image)

Copies *image* into the clipboard.

This is shorthand for:

```
setData(new QImageDrag(image))
```

See also image() [p. 17], setPixmap() [p. 18] and setData() [p. 17].

void QClipboard::setPixmap (const QPixmap & pixmap)

Copies *pixmap* into the clipboard. Note that this is slower than setImage() - it needs to convert the QPixmap to a QImage first.

See also pixmap() [p. 17], setImage() [p. 17] and setData() [p. 17].

void QClipboard::setSelectionMode (bool enable)

Sets the clipboard selection mode. If *enable* is TRUE, then subsequent calls to QClipboard::setData() and other functions which put data into the clipboard will put the data into the mouse selection, otherwise the data will be put into the clipboard.

See also supportsSelection() [p. 18] and selectionModeEnabled() [p. 17].

void QClipboard::setText (const QString & text)

Copies *text* into the clipboard as plain text.

See also text() [p. 18] and setData() [p. 17].

bool QClipboard::supportsSelection () const

Returns TRUE if the clipboard supports mouse selection, FALSE otherwise.

QString QClipboard::text (QCString & subtype) const

Returns the clipboard text in subtype *subtype*, or a null string if the clipboard does not contain any text. If *subtype* is null, any subtype is acceptable, and *subtype* is set to the chosen subtype.

Common values for *subtype* are "plain" and "html".

See also setText() [p. 18], data() [p. 16] and QString::operator!() [Datastructures and String Handling with Qt].

QString QClipboard::text () const

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

Returns the clipboard text as plain text, or a null string if the clipboard does not contain any text.

See also setText() [p. 18], data() [p. 16] and QString::operator!() [Datastructures and String Handling with Qt].

QDataStream Class Reference

The QDataStream class provides serialization of binary data to a QIODevice.

`#include <qdatastream.h>`

Public Members

- **QDataStream** ()
- **QDataStream** (QIODevice * d)
- **QDataStream** (QByteArray a, int mode)
- virtual **~QDataStream** ()
- QIODevice * **device** () const
- void **setDevice** (QIODevice * d)
- void **unsetDevice** ()
- bool **atEnd** () const
- bool eof () const *(obsolete)*
- enum **ByteOrder** { BigEndian, LittleEndian }
- int **byteOrder** () const
- void **setByteOrder** (int bo)
- bool **isPrintableData** () const
- void **setPrintableData** (bool enable)
- int **version** () const
- void **setVersion** (int v)
- QDataStream & **operator>>** (Q_INT8 & i)
- QDataStream & **operator>>** (Q_UINT8 & i)
- QDataStream & **operator>>** (Q_INT16 & i)
- QDataStream & **operator>>** (Q_UINT16 & i)
- QDataStream & **operator>>** (Q_INT32 & i)
- QDataStream & **operator>>** (Q_UINT32 & i)
- QDataStream & **operator>>** (Q_LONG & i)
- QDataStream & **operator>>** (Q_ULONG & i)
- QDataStream & **operator>>** (float & f)
- QDataStream & **operator>>** (double & f)
- QDataStream & **operator>>** (char *& s)
- QDataStream & **operator<<** (Q_INT8 i)
- QDataStream & **operator<<** (Q_UINT8 i)

- QDataStream & **operator<<** (Q_INT16 i)
- QDataStream & **operator<<** (Q_UINT16 i)
- QDataStream & **operator<<** (Q_INT32 i)
- QDataStream & **operator<<** (Q_UINT32 i)
- QDataStream & **operator<<** (Q_LONG i)
- QDataStream & **operator<<** (Q_ULONG i)
- QDataStream & **operator<<** (float f)
- QDataStream & **operator<<** (double f)
- QDataStream & **operator<<** (const char * s)
- QDataStream & **readBytes** (char *& s, uint & l)
- QDataStream & **readRawBytes** (char * s, uint len)
- QDataStream & **writeBytes** (const char * s, uint len)
- QDataStream & **writeRawBytes** (const char * s, uint len)

Detailed Description

The QDataStream class provides serialization of binary data to a QIODevice.

A data stream is a binary stream of encoded information which is 100% independent of the host computer's operating system, CPU or byte order. For example a data stream that is written by a PC under Windows can be read by a Sun SPARC running Solaris.

You can also use a data stream to read/write raw unencoded binary data. If you want a "parsing" input stream, see QTextStream.

The QDataStream class implements serialization of primitive types, like `char`, `short`, `int`, `char*` etc. Serialization of more complex data is accomplished by breaking up the data into primitive units.

A data stream cooperates closely with a QIODevice. A QIODevice represents an input/output medium one can read data from and write data to. The QFile class is an example of an IO device.

Example (write binary data to a stream):

```
QFile f( "file.dta" );
f.open( IO_WriteOnly );
QDataStream s( &f );    // we will serialize the data into file f
s << "the answer is";   // serialize a string
s << (Q_INT32)42;       // serialize an integer
```

Example (read binary data from a stream):

```
QFile f( "file.dta" );
f.open( IO_ReadOnly );
QDataStream s( &f );    // read the data serialized from file f
QString str;
Q_INT32 a;
s >> str >> a;          // extract "the answer is" and 42
```

Each item written to the stream is written in a predefined binary format that varies depending on the item's type. Supported Qt types include QBrush, QColor, QDateTime, QFont, QPixmap, QString, QVariant and many others. For the complete list of all Qt types supporting data streaming see Format of the QDataStream operators .

To take one example, a `char*` string is written as a 32-bit integer equal to the length of the string including the NUL byte, followed by all the characters of the string including the NUL byte. When reading a `char*` string, 4 bytes are read to create the 32-bit length value, then that many characters for the `char*` string including the NUL are read.

The initial IODevice is usually set in the constructor, but can be changed with setDevice(). If you've reached the end of the data (or if there is no IODevice set) atEnd() will return TRUE.

If you want the data to be compatible with an earlier version of Qt use setVersion().

If you want the data to be human-readable, e.g. for debugging, you can set the data stream into printable data mode with setPrintableData(). The data is then written slower, in a bloated but human readable format.

If you are producing a new binary data format, such as a file format for documents created by your application, you could use a QDataStream to write the data in a portable format. Typically, you would write a brief header containing a magic string and a version number to give yourself room for future expansion. For example:

```
QFile f( "file.xxx" );
f.open( IO_WriteOnly );
QDataStream s( &f );

// Write a header with a "magic number" and a version
s << (Q_UINT32)0xa0b0c0d0;
s << (Q_INT32)123;

// Write the data
s << [lots of interesting data]
```

Then read it in with:

```
QFile f( "file.xxx" );
f.open( IO_ReadOnly );
QDataStream s( &f );

// Read and check the header
Q_UINT32 magic;
s >> magic;
if ( magic != 0xa0b0c0d0 )
    return XXX_BAD_FILE_FORMAT;

// Read the version
Q_INT32 version;
s >> version;
if ( version  123 )
    return XXX_BAD_FILE_TOO_NEW;
if ( version <= 110 )
    s.setVersion(1);

// Read the data
s >> [lots of interesting data];
if ( version > 120 )
    s >> [data new in XXX version 1.2];
s >> [other interesting data];
```

You can select which byte order to use when serializing data. The default setting is big endian (MSB first). Changing

it to little endian breaks the portability (unless the reader also changes to little endian). We recommend keeping this setting unless you have special requirements.

Reading and writing raw binary data

You may wish to read/write your own raw binary data to/from the data stream directly. Data may be read from the stream into a preallocated char* using readRawBytes(). Similarly data can be written to the stream using writeRawBytes(). Notice that any encoding/decoding of the data must be done by you.

A similar pair of functions is readBytes() and writeBytes(). These differ from their *raw* counterparts as follows: readBytes() reads a Q_UINT32 which is taken to be the length of the data to be read, then that number of bytes is read into the preallocated char*; writeBytes() writes a Q_UINT32 containing the length of the data, followed by the data. Notice that any encoding/decoding of the data (apart from the length Q_UINT32) must be done by you.

See also QTextStream [p. 150], QVariant [Datastructures and String Handling with Qt] and Input/Output and Networking.

Member Type Documentation

QDataStream::ByteOrder

The byte order used for reading/writing the data.

- `QDataStream::BigEndian` - the default
- `QDataStream::LittleEndian`

Member Function Documentation

QDataStream::QDataStream ()

Constructs a data stream that has no IO device.

See also setDevice() [p. 27].

QDataStream::QDataStream (QIODevice * d)

Constructs a data stream that uses the IO device *d*.

See also setDevice() [p. 27] and device() [p. 23].

QDataStream::QDataStream (QByteArray a, int mode)

Constructs a data stream that operates on a byte array, *a*, through an internal QBuffer device. The *mode* is a QIODevice::mode(), usually either IO_ReadOnly or IO_WriteOnly.

Example:

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray a;
```

```
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
QDataStream s( a, IO_ReadOnly );            // open on a's data
s >> [something];                           // read raw bindata
a.resetRawData( bindata, sizeof(bindata) ); // finished
```

The QByteArray::setRawData() function is not for the inexperienced.

QDataStream::~QDataStream () [virtual]

Destroys the data stream.

The destructor will not affect the current IO device, unless it is an internal IO device processing a QByteArray passed in the *constructor*, in which case the internal IO device is destroyed.

bool QDataStream::atEnd () const

Returns TRUE if the IO device has reached the end position (end of the stream or file) or if there is no IO device set; otherwise returns FALSE, i.e. if the current position of the IO device is before the end position.

See also QIODevice::atEnd() [p. 79].

int QDataStream::byteOrder () const

Returns the current byte order setting — either BigEndian or LittleEndian.

See also setByteOrder() [p. 27].

QIODevice * QDataStream::device () const

Returns the IO device currently set.

See also setDevice() [p. 27] and unsetDevice() [p. 28].

bool QDataStream::eof () const

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

Returns TRUE if the IO device has reached the end position (end of stream or file) or if there is no IO device set.

Returns FALSE if the current position of the read/write head of the IO device is somewhere before the end position.

See also QIODevice::atEnd() [p. 79].

bool QDataStream::isPrintableData () const

Returns TRUE if the printable data flag has been set.

See also setPrintableData() [p. 27].

QDataStream & QDataStream::operator<< (Q_INT8 i)

Writes a signed byte, *i*, to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_UINT8 i)

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

Writes an unsigned byte, *i*, to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_INT16 i)

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

Writes a signed 16-bit integer, *i*, to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_UINT16 i)

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

Writes an unsigned 16-bit integer, *i*, to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_INT32 i)

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

Writes a signed 32-bit integer, *i*, to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_UINT32 i)

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

Writes an unsigned integer, *i*, to the stream as a 32-bit unsigned integer (Q_UINT32). Returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_LONG i)

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

Writes a signed integer, *i*, of the system's word length to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (Q_ULONG i)

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

Writes an unsigned integer, *i*, of the system's word length to the stream and returns a reference to the stream.

QDataStream & QDataStream::operator<< (float f)

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

Writes a 32-bit floating point number, *f*, to the stream using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator<< (double f)

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

Writes a 64-bit floating point number, *f*, to the stream using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator<< (const char * s)

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

Writes the '\0'-terminated string *s* to the stream and returns a reference to the stream.

The string is serialized using writeBytes().

QDataStream & QDataStream::operator>> (Q_INT8 & i)

Reads a signed byte from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_UINT8 & i)

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

Reads an unsigned byte from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_INT16 & i)

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

Reads a signed 16-bit integer from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_UINT16 & i)

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

Reads an unsigned 16-bit integer from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_INT32 & i)

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

Reads a signed 32-bit integer from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_UINT32 & i)

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

Reads an unsigned 32-bit integer from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_LONG & i)

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

Reads a signed integer of the system's word length from the stream into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (Q_ULONG & i)

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

Reads an unsigned integer of the system's word length from the stream, into *i*, and returns a reference to the stream.

QDataStream & QDataStream::operator>> (float & f)

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

Reads a 32-bit floating point number from the stream into *f*, using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator>> (double & f)

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

Reads a 64-bit floating point number from the stream into *f*, using the standard IEEE754 format. Returns a reference to the stream.

QDataStream & QDataStream::operator>> (char *& s)

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

Reads the '\0'-terminated string *s* from the stream and returns a reference to the stream.

Space for the string is allocated using `new` — the caller must destroy it with delete[].

QDataStream & QDataStream::readBytes (char *& s, uint & l)

Reads the buffer *s* from the stream and returns a reference to the stream.

The buffer *s* is allocated using `new`. Destroy it with the `delete[]` operator. If the length is zero or *s* cannot be allocated, *s* is set to 0.

The *l* parameter will be set to the length of the buffer.

The serialization format is a Q_UINT32 length specifier first, then *l* bytes of data. Note that the data is *not* encoded.

See also readRawBytes() [p. 27] and writeBytes() [p. 28].

QDataStream & QDataStream::readRawBytes (char * s, uint len)

Reads *len* bytes from the stream into *s* and returns a reference to the stream.

The buffer *s* must be preallocated. The data is *not* encoded.

See also readBytes() [p. 26], QIODevice::readBlock() [p. 83] and writeRawBytes() [p. 28].

void QDataStream::setByteOrder (int bo)

Sets the serialization byte order to *bo*.

The *bo* parameter can be QDataStream::BigEndian or QDataStream::LittleEndian.

The default setting is big endian. We recommend leaving this setting unless you have special requirements.

See also byteOrder() [p. 23].

void QDataStream::setDevice (QIODevice * d)

void QDataStream::setDevice(QIODevice *d) Sets the IO device to *d*.

See also device() [p. 23] and unsetDevice() [p. 28].

void QDataStream::setPrintableData (bool enable)

Sets (if *enable* is TRUE) or clears the printable data flag.

If this flag is set, the write functions will generate output that consists of printable characters (7 bit ASCII).

We recommend enabling printable data only for debugging purposes (it is slower and creates larger output).

void QDataStream::setVersion (int v)

Sets the version number of the data serialization format.

You don't need to set a version if you are using the current version of Qt.

In order to accommodate new functionality, the datastream serialization format of some Qt classes has changed in some versions of Qt. If you want to read data that was created by an earlier version of Qt, or write data that can be read by a program that was compiled with an earlier version of Qt, use this function to modify the serialization format of QDataStream.

- For Qt 3.0 compatibility, use $v == 4$.
- For Qt 2.1.x and Qt 2.2.x compatibility, use $v == 3$.
- For Qt 2.0.x compatibility, use $v == 2$.
- For Qt 1.x compatibility, use $v == 1$.

See also version() [p. 28].

void QDataStream::unsetDevice ()

Unsets the IO device. This is the same as calling setDevice(0).

See also device() [p. 23] and setDevice() [p. 27].

int QDataStream::version () const

Returns the version number of the data serialization format. In Qt 3.0, this number is 4.

See also setVersion() [p. 27].

QDataStream & QDataStream::writeBytes (const char * s, uint len)

Writes the length specifier *len* and the buffer *s* to the stream and returns a reference to the stream.

The *len* is serialized as a Q_UINT32, followed by *len* bytes from *s*. Note that the data is *not* encoded.

See also writeRawBytes() [p. 28] and readBytes() [p. 26].

QDataStream & QDataStream::writeRawBytes (const char * s, uint len)

Writes *len* bytes from *s* to the stream and returns a reference to the stream. The data is *not* encoded.

See also writeBytes() [p. 28], QIODevice::writeBlock() [p. 85] and readRawBytes() [p. 27].

QDir Class Reference

The QDir class provides access to directory structures and their contents in a platform-independent way.

`#include <qdir.h>`

Public Members

- enum **FilterSpec** { Dirs = 0x001, Files = 0x002, Drives = 0x004, NoSymLinks = 0x008, All = 0x007, TypeMask = 0x00F, Readable = 0x010, Writable = 0x020, Executable = 0x040, RWEMask = 0x070, Modified = 0x080, Hidden = 0x100, System = 0x200, AccessMask = 0x3F0, DefaultFilter = -1 }
- enum **SortSpec** { Name = 0x00, Time = 0x01, Size = 0x02, Unsorted = 0x03, SortByMask = 0x03, DirsFirst = 0x04, Reversed = 0x08, IgnoreCase = 0x10, DefaultSort = -1 }
- **QDir** ()
- **QDir** (const QString & path, const QString & nameFilter = QString::null, int sortSpec = Name | IgnoreCase, int filterSpec = All)
- **QDir** (const QDir & d)
- virtual **~QDir** ()
- QDir & **operator=** (const QDir & d)
- QDir & **operator=** (const QString & path)
- virtual void **setPath** (const QString & path)
- virtual QString **path** () const
- virtual QString **absPath** () const
- virtual QString **canonicalPath** () const
- virtual QString **dirName** () const
- virtual QString **filePath** (const QString & fileName, bool acceptAbsPath = TRUE) const
- virtual QString **absFilePath** (const QString & fileName, bool acceptAbsPath = TRUE) const
- virtual bool **cd** (const QString & dirName, bool acceptAbsPath = TRUE)
- virtual bool **cdUp** ()
- QString **nameFilter** () const
- virtual void **setNameFilter** (const QString & nameFilter)
- FilterSpec **filter** () const
- virtual void **setFilter** (int filterSpec)
- SortSpec **sorting** () const
- virtual void **setSorting** (int sortSpec)
- bool **matchAllDirs** () const
- virtual void **setMatchAllDirs** (bool enable)
- uint **count** () const

- QString **operator[]** (int index) const
- virtual QStrList encodedEntryList (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const *(obsolete)*
- virtual QStrList encodedEntryList (const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const *(obsolete)*
- virtual QStringList **entryList** (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const
- virtual QStringList **entryList** (const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const
- virtual const QFileInfoList * **entryInfoList** (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const
- virtual const QFileInfoList * **entryInfoList** (const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const
- virtual bool **mkdir** (const QString & dirName, bool acceptAbsPath = TRUE) const
- virtual bool **rmdir** (const QString & dirName, bool acceptAbsPath = TRUE) const
- virtual bool **isReadable** () const
- virtual bool **exists** () const
- virtual bool **isRoot** () const
- virtual bool **isRelative** () const
- virtual void **convertToAbs** ()
- virtual bool **operator==** (const QDir & d) const
- virtual bool **operator!=** (const QDir & d) const
- virtual bool **remove** (const QString & fileName, bool acceptAbsPath = TRUE)
- virtual bool **rename** (const QString & oldName, const QString & newName, bool acceptAbsPaths = TRUE)
- virtual bool **exists** (const QString & name, bool acceptAbsPath = TRUE)

Static Public Members

- QString **convertSeparators** (const QString & pathName)
- const QFileInfoList * **drives** ()
- char **separator** ()
- bool **setCurrent** (const QString & path)
- QDir **current** ()
- QDir **home** ()
- QDir **root** ()
- QString **currentDirPath** ()
- QString **homeDirPath** ()
- QString **rootDirPath** ()
- bool **match** (const QStringList & filters, const QString & fileName)
- bool **match** (const QString & filter, const QString & fileName)
- QString **cleanDirPath** (const QString & filePath)
- bool **isRelativePath** (const QString & path)

Detailed Description

The QDir class provides access to directory structures and their contents in a platform-independent way.

A QDir is used to manipulate path names, access information regarding paths and files, and manipulate the underlying file system.

A QDir can point to a file using either a relative or an absolute file path. Absolute file paths begin with the directory separator "/" or with a drive specification (except under Unix). If you always use "/" as a directory separator, Qt will translate your paths to conform to the underlying operating system. Relative file names begin with a directory name or a file name and specify a path relative to the current directory.

The "current" path refers to the application's working directory. A QDir's own path is set and retrieved with setPath() and path().

An example of an absolute path is the string "/tmp/quartz", a relative path might look like "src/fatlib". You can use the function isRelative() to check if a QDir is using a relative or an absolute file path. Call convertToAbs() to convert a relative QDir to an absolute one. For a simplified path use cleanDirPath(). To obtain a path which has no symbolic links or redundant ".." elements use canonicalPath(). The path can be set with setPath(), or changed with cd() and cdUp().

QDir provides several static functions, for example, setCurrent() to set the application's working directory and currentDirPath() to retrieve the application's working directory. Access to some common paths is provided with the static functions, current(), home() and root() which return QDir objects or currentDirPath(), homeDirPath() and rootDirPath() which return the path as a string.

The number of entries in a directory is returned by count(). Obtain a string list of the names of all the files and directories in a directory with entryList(). If you prefer a list of QFileInfo pointers use entryInfoList(). Both these functions can apply a name filter, an attributes filter (e.g. read-only, files not directories, etc.), and a sort order. The filters and sort may be set with calls to setNameFilter(), setFilter() and setSorting(). They may also be specified in the entryList() and entryInfoList()'s arguments.

Create a new directory with mkdir(), rename a directory with rename() and remove an existing directory with rmdir(). Remove a file with remove(). You can interrogate a directory with exists(), isReadable() and isRoot().

To get a path with a filename use filePath(), and to get a directory name use dirName(); neither of these functions checks for the existence of the file or directory.

The list of root directories is provided by drives(); on Unix systems this returns a list containing one root directory, "/"; on Windows the list will usually contain "C:/", and possibly "D:/", etc.

If you need the path in a form suitable for the underlying operating system use convertSeparators().

Examples:

See if a directory exists.

```
QDir d( "example" );                        // "./example"
if ( !d.exists() )
    qWarning( "Cannot find the example directory" );
```

Traversing directories and reading a file.

```
QDir d = QDir::root();                      // "/"
if ( !d.cd("tmp") ) {                       // "/tmp"
    qWarning( "Cannot find the \"/tmp\" directory" );
} else {
    QFile f( d.filePath("ex1.txt") );       // "/tmp/ex1.txt"
```

```
        if ( !f.open(IO_ReadWrite) )
            qWarning( "Cannot create the file %s", f.name() );
    }
```

A program that lists all the files in the current directory (excluding symbolic links), sorted by size, smallest first.

```
    #include
    #include <qdir.h>

    int main( int argc, char **argv )
    {
        QDir d;
        d.setFilter( QDir::Files | QDir::Hidden | QDir::NoSymLinks );
        d.setSorting( QDir::Size | QDir::Reversed );

        const QFileInfoList *list = d.entryInfoList();
        QFileInfoListIterator it( *list );
        QFileInfo *fi;

        printf( "     Bytes Filename\n" );
        while ( (fi = it.current()) != 0 ) {
            printf( "%10li %s\n", fi->size(), fi->fileName().latin1() );
            ++it;
        }
        return 0;
    }
```

See also Input/Output and Networking.

Member Type Documentation

QDir::FilterSpec

This enum describes how QDir is to select which entries in a directory to return. The filter value is specified by OR-ing together values from the following list:

- `QDir::Dirs` - List directories only.
- `QDir::Files` - List files only.
- `QDir::Drives` - List disk drives (ignored under Unix).
- `QDir::NoSymLinks` - Do not list symbolic links (ignored by operating systems that don't support symbolic links).
- `QDir::All` - List directories, files, drives and symlinks (this does not list borken symlinks unless you specify System).
- `QDir::TypeMask` - A mask for the the Dirs, Files, Drives and NoSymLinks flags.
- `QDir::Readable` - List files for which the application has read access.
- `QDir::Writable` - List files for which the application has write access.
- `QDir::Executable` - List files for which the application has execute access.
- `QDir::RWEMask` - A mask for the Readable, Writable and Executable flags.

- `QDir::Modified` - Only list files that have been modified (ignored under Unix).
- `QDir::Hidden` - List hidden files (on Unix, files starting with a .).
- `QDir::System` - List system files (on Unix, FIFOs, sockets and device files)
- `QDir::AccessMask` - A mask for the Readable, Writable, Executable Modified, Hidden and System flags
- `QDir::DefaultFilter` - Internal flag.

If you do not set any of Readable, Writable or Executable, QDir will set all three of them. This makes the default easy to write and at the same time useful.

Examples: `Readable|Writable` means list all files for which the application has read access, write access or both. `Dirs|Drives` means list drives, directories, all files that the application can read, write or execute, and also symlinks to such files/directories.

QDir::SortSpec

This enum describes how QDir is to sort the list of entries returned by entryList() or entryInfoList(). The sort value is specified by OR-ing together values from the following list:

- `QDir::Name` - Sort by name.
- `QDir::Time` - Sort by time (modification time).
- `QDir::Size` - Sort by file size.
- `QDir::Unsorted` - Do not sort.
- `QDir::SortByMask` - A mask for Name, Time and Size.
- `QDir::DirsFirst` - Put the directories first, then the files.
- `QDir::Reversed` - Reverse the sort order.
- `QDir::IgnoreCase` - Sort case-insensitively.
- `QDir::DefaultSort` - Internal flag.

You can only specify one of the first four.

If you specify both DirsFirst and Reversed, directories are still put first, but in reverse order; the files will be listed after the directories, again in reverse order.

Member Function Documentation

QDir::QDir ()

Constructs a QDir pointing to the current directory.

See also currentDirPath() [p. 36].

QDir::QDir (const QString & path, const QString & nameFilter = QString::null, int sortSpec = Name | IgnoreCase, int filterSpec = All)

Constructs a QDir for with path *path* that filters its entries by name with *nameFilter* and by attributes with *filterSpec*. It also sorts the names using *sortSpec*.

The default *nameFilter* is an empty string, which excludes nothing; the default *filterSpec* is All, which also means exclude nothing. The default *sortSpec* is `Name|IgnoreCase`, i.e. sort by name case-insensitively.

Example that lists all the files in "/tmp":

```
QDir d( "/tmp" );
for ( int i = 0; i < d.count(); i++ )
    printf( "%s\n", d[i] );
```

If *path* is "" or null, QDir uses "." (the current directory). If *nameFilter* is "" or null, QDir uses the name filter "*" (all files).

Note that *path* need not exist.

See also exists() [p. 38], setPath() [p. 43], setNameFilter() [p. 43], setFilter() [p. 43] and setSorting() [p. 44].

QDir::QDir (const QDir & d)

Constructs a QDir that is a copy of the directory *d*.

See also operator=() [p. 41].

QDir::~QDir () [virtual]

Destroys the QDir frees up its resources.

QString QDir::absFilePath (const QString & fileName, bool acceptAbsPath = TRUE) const [virtual]

Returns the absolute path name of a file in the directory. Does *not* check if the file actually exists in the directory. Redundant multiple separators or "." and ".." directories in *fileName* will not be removed (see cleanDirPath()).

If *acceptAbsPath* is TRUE a *fileName* starting with a separator "/" will be returned without change. If *acceptAbsPath* is FALSE an absolute path will be prepended to the fileName and the resultant string returned.

See also filePath() [p. 38].

QString QDir::absPath () const [virtual]

Returns the absolute path (a path that starts with "/" or a drive specification), which may contain symbolic links, but never contains redundant ".", ".." or multiple separators.

See also setPath() [p. 43], canonicalPath() [p. 34], exists() [p. 38], cleanDirPath() [p. 35], dirName() [p. 36] and absFilePath() [p. 34].

Example: fileiconview/qfileiconview.cpp.

QString QDir::canonicalPath () const [virtual]

Returns the canonical path, i.e. a path without symbolic links or redundant "." or ".." elements.

On systems that do not have symbolic links this function will always return the same string that absPath returns. If the canonical path does not exist (normally due to dangling symbolic links) canonicalPath() returns a null string.

See also path() [p. 42], absPath() [p. 34], exists() [p. 38], cleanDirPath() [p. 35], dirName() [p. 36], absFilePath() [p. 34] and QString::isNull() [Datastructures and String Handling with Qt].

bool QDir::cd (const QString & dirName, bool acceptAbsPath = TRUE) [virtual]

Changes the QDir's directory to directory *dirName*.

If *acceptAbsPath* is TRUE a path starting with separator "/" will cause the function to change to the absolute directory. If *acceptAbsPath* is FALSE any number of separators at the beginning of *dirName* will be removed and the function will decend into *dirName*.

Returns TRUE if the new directory exists and is readable. Note that the logical cd() operation is not performed if the new directory does not exist.

Calling cd("..") is equivalent to calling cdUp().

See also cdUp() [p. 35], isReadable() [p. 39], exists() [p. 38] and path() [p. 42].

Example: fileiconview/mainwindow.cpp.

bool QDir::cdUp () [virtual]

Changes directory by moving one directory up from the QDir's current directory.

Returns TRUE if the new directory exists and is readable. Note that the logical cdUp() operation is not performed if the new directory does not exist.

See also cd() [p. 35], isReadable() [p. 39], exists() [p. 38] and path() [p. 42].

QString QDir::cleanDirPath (const QString & filePath) [static]

Removes all multiple directory separators "/" and resolves any "."s or ".."s found in the path, *filePath*.

Symbolic links are kept. This function does not return the canonical path, but rather the most simplified version of the input. "./local" becomes "local", "local/../bin" becomes "bin" and "/local/usr/../bin" becomes "/local/bin".

See also absPath() [p. 34] and canonicalPath() [p. 34].

QString QDir::convertSeparators (const QString & pathName) [static]

Converts the '/' separators in *pathName* to the separators appropriate for the underlying operating system. Returns the translated string.

On Windows, convertSeparators("c:/winnt/system32") returns "c:\winnt\system32".

The returned string may be the same as the argument on some operating systems, for example on Unix.

void QDir::convertToAbs () [virtual]

Converts the directory path to an absolute path. If it is already absolute nothing is done.

See also isRelative() [p. 39].

uint QDir::count () const

Returns the total number of directories and files that were found.

Equivalent to entryList().count().

See also operator[]() [p. 41] and entryList() [p. 38].

QDir QDir::current () [static]

Returns the application's current directory.

Use path() to access a QDir object's path.

See also currentDirPath() [p. 36] and QDir::QDir() [p. 33].

QString QDir::currentDirPath () [static]

Returns the absolute path of the application's current directory.

See also current() [p. 36].

Examples: helpviewer/helpwindow.cpp and qdir/qdir.cpp.

QString QDir::dirName () const [virtual]

Returns the name of the directory; this is *not* the same as the path, e.g. a directory with the name "mail", might have the path "/var/spool/mail". If the directory has no name (e.g. it is the root directory) a null string is returned.

No check is made to ensure that a directory with this name actually exists.

See also path() [p. 42], absPath() [p. 34], absFilePath() [p. 34], exists() [p. 38] and QString::isNull() [Datastructures and String Handling with Qt].

const QFileInfoList * QDir::drives () [static]

Returns a list of the root directories on this system. On win32, this returns a number of QFileInfo objects containing "C:/", "D:/" etc. On other operating systems, it returns a list containing just one root directory (e.g. "/").

The returned pointer is owned by Qt. Callers should *not* delete or modify it.

Example: dirview/main.cpp.

QStrList QDir::encodedEntryList (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const [virtual]

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

This function is included to easy porting from Qt 1.x to Qt 2.0, it is the same as entryList(), but encodes the filenames as 8-bit strings using QFile::encodedName().

It is more efficient to use entryList().

QStrList QDir::encodedEntryList (const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const [virtual]

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

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

This function is included to easy porting from Qt 1.x to Qt 2.0, it is the same as entryList(), but encodes the filenames as 8-bit strings using QFile::encodedName().

It is more efficient to use entryList().

const QFileInfoList * QDir::entryInfoList (const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const [virtual]

Returns a list of QFileInfo objects for all files and directories in the directory, ordered in accordance with setSorting() and filtered in accordance with setFilter() and setNameFilter().

The filter and sorting specifications can be overridden using the *nameFilter*, *filterSpec* and *sortSpec* arguments.

Returns 0 if the directory is unreadable or does not exist.

The returned pointer is a const pointer to a QFileInfoList. The list is owned by the QDir object and will be reused on the next call to entryInfoList() for the same QDir instance. If you want to keep the entries of the list after a subsequent call to this function you will need to copy them.

See also entryList() [p. 38], setNameFilter() [p. 43], setSorting() [p. 44] and setFilter() [p. 43].

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

const QFileInfoList * QDir::entryInfoList (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const [virtual]

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

Returns a list of QFileInfo objects for all files and directories in the directory, ordered in accordance with setSorting() and filtered in accordance with setFilter() and setNameFilter().

The filter and sorting specifications can be overridden using the *filterSpec* and *sortSpec* arguments.

Returns 0 if the directory is unreadable or does not exist.

The returned pointer is a const pointer to a QFileInfoList. The list is owned by the QDir object and will be reused on the next call to entryInfoList() for the same QDir instance. If you want to keep the entries of the list after a subsequent call to this function you will need to copy them.

See also entryList() [p. 38], setNameFilter() [p. 43], setSorting() [p. 44] and setFilter() [p. 43].

QStringList QDir::entryList (const QString & nameFilter, int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const [virtual]

Returns a list of the names of all files and directories in the directory, ordered in accordance with setSorting() and filtered in accordance with setFilter() and setNameFilter().

The filter and sorting specifications can be overridden using the *nameFilter*, *filterSpec* and *sortSpec* arguments.

Returns an empty list if the directory is unreadable or does not exist.

See also entryInfoList() [p. 37], setNameFilter() [p. 43], setSorting() [p. 44] and setFilter() [p. 43].

QStringList QDir::entryList (int filterSpec = DefaultFilter, int sortSpec = DefaultSort) const [virtual]

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

Returns a list of the names of all files and directories in the directory, ordered in accordance with setSorting() and filtered in accordance with setFilter() and setNameFilter().

The filter and sorting specifications can be overridden using the *filterSpec* and *sortSpec* arguments.

Returns an empty list if the directory is unreadable or does not exist.

See also entryInfoList() [p. 37], setNameFilter() [p. 43], setSorting() [p. 44] and setFilter() [p. 43].

bool QDir::exists (const QString & name, bool acceptAbsPath = TRUE) [virtual]

Checks for existence of the file *name*.

If *acceptAbsPath* is TRUE a path starting with separator "/" will check the file with the absolute path. If *acceptAbsPath* is FALSE any number of separators at the beginning of *name* will be removed and the resultant file name will be checked.

Returns TRUE if the file exists; otherwise returns FALSE.

See also QFileInfo::exists() [p. 63] and QFile::exists() [p. 54].

bool QDir::exists () const [virtual]

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

Returns TRUE if the *directory* exists. (If a file with the same name is found this function will return FALSE).

See also QFileInfo::exists() [p. 63] and QFile::exists() [p. 54].

QString QDir::filePath (const QString & fileName, bool acceptAbsPath = TRUE) const [virtual]

Returns the path name of a file in the directory. Does *not* check if the file actually exists in the directory. If the QDir is relative the returned path name will also be relative. Redundant multiple separators or "." and ".." directories in *fileName* will not be removed (see cleanDirPath()).

If *acceptAbsPath* is TRUE a *fileName* starting with a separator "/" will be returned without change. If *acceptAbsPath* is FALSE an absolute path will be prepended to the fileName and the resultant string returned.

See also absFilePath() [p. 34], isRelative() [p. 39] and canonicalPath() [p. 34].

FilterSpec QDir::filter () const

Returns the value set by setFilter()

QDir QDir::home () [static]

Returns the home directory.

Under Windows NT/2000 the function forms the path by concatenating the `HOMEDRIVE` and `HOMEPATH` environment variables.

Under Windows 9x and non-Windows operating systems the `HOME` environment variable is used.

If the environment variables aren't set, rootDirPath() is used instead.

See also homeDirPath() [p. 39].

QString QDir::homeDirPath () [static]

Returns the absolute path of the user's home directory,

See also home() [p. 39].

bool QDir::isReadable () const [virtual]

Returns TRUE if the directory is readable AND we can open files by name. This function will return FALSE if only one of these is present. **Warning:** A FALSE value from this function is not a guarantee that files in the directory are not accessible.

See also QFileInfo::isReadable() [p. 65].

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

bool QDir::isRelative () const [virtual]

Returns TRUE if the directory path is relative to the current directory and returns FALSE if the path is absolute (e.g. under UNIX a path is relative if it does not start with a "/").

See also convertToAbs() [p. 35].

bool QDir::isRelativePath (const QString & path) [static]

Returns TRUE if *path* is relative; returns FALSE if it is absolute.

See also isRelative() [p. 39].

bool QDir::isRoot () const [virtual]

Returns TRUE if the directory is the root directory, otherwise FALSE.

Note: If the directory is a symbolic link to the root directory this function returns FALSE. If you want to test for this you can use canonicalPath():

Example:

```
QDir d( "/tmp/root_link" );
d = d.canonicalPath();
if ( d.isRoot() )
    qWarning( "It IS a root link!" );
```

See also root() [p. 42] and rootDirPath() [p. 43].

bool QDir::match (const QString & filter, const QString & fileName) [static]

Returns TRUE if the *fileName* matches the wildcard (glob) pattern *filter*. The *filter* may also contain multiple patterns separated by spaces or semicolons.

(See QRegExp wildcard matching.)

See also QRegExp::match() [Additional Functionality with Qt].

bool QDir::match (const QStringList & filters, const QString & fileName) [static]

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

Returns TRUE if the *fileName* matches any the wildcard (glob) patterns in the list of *filters*.

(See QRegExp wildcard matching.)

See also QRegExp::match() [Additional Functionality with Qt].

bool QDir::matchAllDirs () const

Returns the value set by setMatchAllDirs()

See also setMatchAllDirs() [p. 43].

bool QDir::mkdir (const QString & dirName, bool acceptAbsPath = TRUE) const [virtual]

Creates a directory.

If *acceptAbsPath* is TRUE a path starting with a separator ('/') will create the absolute directory, if *acceptAbsPath* is FALSE any number of separators at the beginning of *dirName* will be removed.

Returns TRUE if successful, otherwise FALSE.

See also rmdir() [p. 42].

QString QDir::nameFilter () const

Returns the string set by setNameFilter()

bool QDir::operator!= (const QDir & d) const [virtual]

Returns TRUE if directory *d* and this directory have different paths or different sort or filter settings; otherwise returns FALSE.

Example:

```
// The current directory is "/usr/local"
QDir d1( "/usr/local/bin" );
QDir d2( "bin" );
if ( d1 != d2 ) qDebug( "They differ\n" ); // This is printed
```

QDir & QDir::operator= (const QDir & d)

Makes a copy of QDir *d* and assigns it to this QDir.

QDir & QDir::operator= (const QString & path)

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

Sets the directory path to be the given *path*.

bool QDir::operator== (const QDir & d) const [virtual]

Returns TRUE if directory *d* and this directory have the same path and their sort and filter settings are the same; otherwise returns FALSE.

Example:

```
// The current directory is "/usr/local"
QDir d1( "/usr/local/bin" );
QDir d2( "bin" );
d2.convertToAbs();
if ( d1 == d2 ) qDebug( "They're the same\n" ); // This is printed
```

QString QDir::operator[] (int index) const

Returns the file name at position *index* in the list of file names. Equivalent to entryList().at(index).

Returns a null string if the *index* is out of range or if the entryList() function failed.

See also count() [p. 36] and entryList() [p. 38].

QString QDir::path () const [virtual]

Returns the path, this may contain symbolic links, but never contains redundant ".", ".." or multiple separators.

The returned path can be either absolute or relative (see setPath()).

See also setPath() [p. 43], absPath() [p. 34], exists() [p. 38], cleanDirPath() [p. 35], dirName() [p. 36], absFilePath() [p. 34] and convertSeparators() [p. 35].

bool QDir::remove (const QString & fileName, bool acceptAbsPath = TRUE) [virtual]

Removes a file.

If *acceptAbsPath* is TRUE a path starting with separator "/" will remove the file with the absolute path. If *acceptAbsPath* is FALSE any number of separators at the beginning of *fileName* will be removed and the resultant file name will be removed.

Returns TRUE if the file is removed successfully; otherwise returns FALSE.

bool QDir::rename (const QString & oldName, const QString & newName, bool acceptAbsPaths = TRUE) [virtual]

Renames a file or directory.

If *acceptAbsPaths* is TRUE a path starting with a separator ('/') will rename the file with the absolute path, if *acceptAbsPaths* is FALSE any number of separators at the beginning of the names will be removed.

Returns TRUE if successful; otherwise returns FALSE.

On most file systems, rename() fails only if *oldName* does not exist or if *newName* and *oldName* are not on the same partition. On Windows, rename() will fail if *newName* already exists. However, there are also other reasons why rename() can fail. For example, on at least one file system rename() fails if newName points to an open file.

Example: fileiconview/qfileiconview.cpp.

bool QDir::rmdir (const QString & dirName, bool acceptAbsPath = TRUE) const [virtual]

Removes a directory.

If *acceptAbsPath* is TRUE a path starting with a separator ('/') will remove the absolute directory, if *acceptAbsPath* is FALSE any number of separators at the beginning of *dirName* will be removed.

The directory must be empty for rmdir() to succeed.

Returns TRUE if successful, otherwise FALSE.

See also mkdir() [p. 40].

QDir QDir::root () [static]

Returns the root directory.

See also rootDirPath() [p. 43] and drives() [p. 36].

QString QDir::rootDirPath () [static]

Returns the absolute path for the root directory.

For UNIX operating systems this returns "/". For Windows file systems this returns "c:/".

See also root() [p. 42] and drives() [p. 36].

char QDir::separator () [static]

Returns the native directory separator; "/" under UNIX and "\" under MS-DOS, Windows NT and OS/2.

You do not need to use this function to build file paths. If you always use "/", Qt will translate your paths to conform to the underlying operating system.

bool QDir::setCurrent (const QString & path) [static]

Sets the application's current working directory to *path*. Returns TRUE if the directory was successfully changed; otherwise returns FALSE.

void QDir::setFilter (int filterSpec) [virtual]

Sets the filter used by entryList() and entryInfoList() to *filterSpec*. The filter is used to specify the kind of files that should be returned by entryList() and entryInfoList(). See QDir::FilterSpec.

See also filter() [p. 39] and setNameFilter() [p. 43].

void QDir::setMatchAllDirs (bool enable) [virtual]

If *enable* is TRUE then all directories are included (e.g. in entryList()), and the nameFilter() is only applied to the files. If *enable* is FALSE then the nameFilter() is applied to both directories and files.

See also matchAllDirs() [p. 40].

void QDir::setNameFilter (const QString & nameFilter) [virtual]

Sets the name filter used by entryList() and entryInfoList() to *nameFilter*.

The *nameFilter* is a wildcard (globbing) filter that understands "*" and "?" wildcards. (See QRegExp wildcard matching.) You may specify several filter entries all separated by a single space " " or by a semi-colon ";".

For example, if you want entryList() and entryInfoList() to list all files ending with ".cpp" and all files ending with ".h", you would use either dir.setNameFilter("*.cpp *.h") or dir.setNameFilter("*.cpp;*.h").

See also nameFilter() [p. 41] and setFilter() [p. 43].

void QDir::setPath (const QString & path) [virtual]

Sets the path of the directory to *path*. The path is cleaned of redundant ".", ".." and of multiple separators. No check is made to ensure that a directory with this path exists.

The path can be either absolute or relative. Absolute paths begin with the directory separator "/" or a drive specification (except under Unix). Relative file names begin with a directory name or a file name and specify a path relative to the current directory. An example of an absolute path is the string "/tmp/quartz", a relative path might look like "src/fatlib".

See also path() [p. 42], absPath() [p. 34], exists() [p. 38], cleanDirPath() [p. 35], dirName() [p. 36], absFilePath() [p. 34], isRelative() [p. 39] and convertToAbs() [p. 35].

void QDir::setSorting (int sortSpec) [virtual]

Sets the sort order used by entryList() and entryInfoList().

The *sortSpec* is specified by OR-ing values from the enum QDir::SortSpec.

See also sorting() [p. 44] and SortSpec [p. 33].

SortSpec QDir::sorting () const

Returns the value set by setSorting()

See also setSorting() [p. 44] and SortSpec [p. 33].

QDns Class Reference

The QDns class provides asynchronous DNS lookups.

This class is part of the **network** module.

`#include <qdns.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- enum **RecordType** { None, A, Aaaa, Mx, Srv, Cname, Ptr, Txt }
- **QDns** ()
- **QDns** (const QString & label, RecordType rr = A)
- **QDns** (const QHostAddress & address, RecordType rr = Ptr)
- virtual **~QDns** ()
- virtual void **setLabel** (const QString & label)
- virtual void **setLabel** (const QHostAddress & address)
- QString **label** () const
- virtual void **setRecordType** (RecordType rr = A)
- RecordType **recordType** () const
- bool **isWorking** () const
- QValueList<QHostAddress> **addresses** () const
- QValueList<MailServer> **mailServers** () const
- QValueList<Server> **servers** () const
- QStringList **hostNames** () const
- QStringList **texts** () const
- QString **canonicalName** () const
- QStringList **qualifiedNames** () const

Signals

- void **resultsReady** ()

Detailed Description

The QDns class provides asynchronous DNS lookups.

Both Windows and Unix provide synchronous DNS lookups; Windows provides some asynchronous support too. At the time of writing neither operating system provides asynchronous support for anything other than hostname-to-address mapping.

QDns rectifies this shortcoming, by providing asynchronous caching lookups for the record types that we expect modern GUI applications to need in the near future.

The class is *not* straightforward to use (although it is much simpler than the native APIs); QSocket provides much easier to use TCP connection facilities. The aim of QDns is to provide a correct and small API to the DNS and nothing more. (We use "correctness" to mean that the DNS information is correctly cached, and correctly timed out.)

The API comprises a constructor, functions to set the DNS node (the domain in DNS terminology) and record type (setLabel() and setRecordType()), the corresponding get functions, an isWorking() function to determine whether QDns is working or reading, a resultsReady() signal and query functions for the result.

There is one query function for each RecordType, namely addresses(), mailServers(), servers(), hostNames() and texts(). There are also two generic query functions: canonicalName() returns the name you'll presumably end up using (the exact meaning of this depends on the record type) and qualifiedNames() returns a list of the fully qualified names label() maps to.

See also QSocket [p. 126] and Input/Output and Networking.

Member Type Documentation

QDns::RecordType

This enum type defines the record types QDns can handle. The DNS provides many more; these are the ones we've judged to be in current use, useful for GUI programs and important enough to support right away:

- `QDns::None` - No information. This exists only so that QDns can have a default.
- `QDns::A` - IPv4 addresses. By far the most common type.
- `QDns::Aaaa` - IPv6 addresses. So far mostly unused.
- `QDns::Mx` - Mail eXchanger names. Used for mail delivery.
- `QDns::Srv` - SeRVer names. Generic record type for finding servers. So far mostly unused.
- `QDns::Cname` - Canonical names. Maps from nicknames to the true name (the canonical name) for a host.
- `QDns::Ptr` - name PoinTeRs. Maps from IPv4 or IPv6 addresses to hostnames.
- `QDns::Txt` - arbitrary TeXT for domains.

We expect that some support for the RFC-2535 extensions will be added in future versions.

Member Function Documentation

QDns::QDns ()

Constructs a DNS query object with invalid settings for both the label and the search type.

QDns::QDns (const QString & label, RecordType rr = A)

Constructs a DNS query object that will return record type *rr* information about *label*.

The DNS lookup is started the next time the application enters the event loop. When the result is found the signal resultsReady() is emitted.

rr defaults to A, IPv4 addresses.

QDns::QDns (const QHostAddress & address, RecordType rr = Ptr)

Constructs a DNS query object that will return record type *rr* information about host address *address*. The label is set to the IN-ADDR.ARPA domain name. This is useful in combination with the Ptr record type (e.g. if you want to look up a hostname for a given address).

The DNS lookup is started the next time the application enters the event loop. When the result is found the signal resultsReady() is emitted.

rr defaults to Ptr, that maps addresses to hostnames.

QDns::~QDns () [virtual]

Destroys the DNS query object and frees its allocated resources.

QValueList<QHostAddress> QDns::addresses () const

Returns a list of the addresses for this name if this QDns object has a recordType() of QDns::A or QDns::Aaaa and the answer is available; otherwise returns an empty list.

As a special case, if label() is a valid numeric IP address, this function returns that address.

QString QDns::canonicalName () const

Returns the canonical name for this DNS node. (This works regardless of what recordType() is set to.)

If the canonical name isn't known, this function returns a null string.

The canonical name of a DNS node is its full name, or the full name of the target of its CNAME. For example, if l.trolltech.com is a CNAME to lupinella.troll.no, and the search path for QDns is "trolltech.com", then the canonical name for all of "lupinella", "l", "lupinella.troll.no." and "l.trolltech.com" is "lupinella.troll.no.".

QStringList QDns::hostNames () const

Returns a list of host names if the record type is Ptr.

bool QDns::isWorking () const

Returns TRUE if QDns is doing a lookup for this object, and FALSE if this object already has the information it wants.

QDns emits the resultsReady() signal when the status changes to FALSE.

Example: network/mail/smtp.cpp.

QString QDns::label () const

Returns the domain name for which this object returns information.

See also setLabel() [p. 49].

QValueList<MailServer> QDns::mailServers () const

Returns a list of mail servers if the record type is Mx. The class `QDns::MailServer` contains the following public variables:

- QString QDns::MailServer::name
- Q_UINT16 QDns::MailServer::priority

Example: network/mail/smtp.cpp.

QStringList QDns::qualifiedNames () const

Returns a list of the fully qualified names label() maps to.

RecordType QDns::recordType () const

Returns the record type of this DNS query object.

See also setRecordType() [p. 49] and RecordType [p. 46].

void QDns::resultsReady () [signal]

This signal is emitted when results are available for one of the qualifiedNames().

Example: network/mail/smtp.cpp.

QValueList<Server> QDns::servers () const

Returns a list of servers if the record type is Srv. The class `QDns::Server` contains the following public variables:

- QString QDns::Server::name
- Q_UINT16 QDns::Server::priority
- Q_UINT16 QDns::Server::weight
- Q_UINT16 QDns::Server::port

void QDns::setLabel (const QString & label) [virtual]

Sets this DNS query object to query for information about *label*.

This does not change the recordType(), but its isWorking() status will probably change as a result.

The DNS lookup is started the next time the application enters the event loop. When the result is found the signal resultsReady() is emitted.

void QDns::setLabel (const QHostAddress & address) [virtual]

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

Sets this DNS query object to query for information about the host address *address*. The label is set to the IN-ADDR.ARPA domain name. This is useful in combination with the Ptr record type (e.g. if you want to look up a hostname for a given address).

void QDns::setRecordType (RecordType rr = A) [virtual]

Sets this object to query for record type *rr* records.

The DNS lookup is started the next time the application enters the event loop. When the result is found the signal resultsReady() is emitted.

See also RecordType [p. 46].

QStringList QDns::texts () const

Returns a list of texts if the record type is Txt.

QFile Class Reference

The QFile class is an I/O device that operates on files.

`#include <qfile.h>`

Inherits QIODevice [p. 76].

Public Members

- **QFile** ()
- **QFile** (const QString & name)
- **~QFile** ()
- QString **name** () const
- void **setName** (const QString & name)
- typedef QCString (* **EncoderFn**) (const QString & fileName)
- typedef QString (* **DecoderFn**) (const QCString & localfileName)
- bool **exists** () const
- bool **remove** ()
- virtual bool **open** (int m)
- bool **open** (int m, FILE * f)
- bool **open** (int m, int f)
- virtual void **close** ()
- virtual void **flush** ()
- virtual Offset **size** () const
- virtual Offset **at** () const
- virtual bool **at** (Offset pos)
- virtual bool **atEnd** () const
- virtual Q_LONG **readBlock** (char * p, Q_ULONG len)
- virtual Q_LONG **readLine** (char * p, Q_ULONG maxlen)
- Q_LONG **readLine** (QString & s, Q_ULONG maxlen)
- virtual int **getch** ()
- virtual int **putch** (int ch)
- virtual int **ungetch** (int ch)
- int **handle** () const

Static Public Members

- QCString **encodeName** (const QString & fileName)
- QString **decodeName** (const QCString & localFileName)
- void **setEncodingFunction** (EncoderFn f)
- void **setDecodingFunction** (DecoderFn f)
- bool **exists** (const QString & fileName)
- bool **remove** (const QString & fileName)

Important Inherited Members

- virtual QByteArray **readAll** ()

Detailed Description

The QFile class is an I/O device that operates on files.

QFile is an I/O device for reading and writing binary and text files. A QFile may be used by itself or more conveniently with a QDataStream or QTextStream.

The file name is usually passed in the constructor but can be changed with setName(). You can check for a file's existence with exists() and remove a file with remove().

The file is opened with open(), closed with close() and flushed with flush(). Data is usually read and written using QDataStream or QTextStream, but you can read with readBlock() and readLine() and write with writeBlock(). QFile also supports getch(), ungetch() and putch().

The size of the file is returned by size(). You can get the current file position or move to a new file position using the at() functions. If you've reached the end of the file, atEnd() returns TRUE. The file handle is returned by handle().

Here is a code fragment that uses QTextStream to read a text file line by line. It prints each line with a line number.

```
QFile f("file.txt");
if ( f.open(IO_ReadOnly) ) {    // file opened successfully
    QTextStream t( &f );        // use a text stream
    QString s;
    int n = 1;
    while ( !t.eof() ) {        // until end of file...
        s = t.readLine();       // line of text excluding '\n'
        printf( "%3d: %s\n", n++, s.latin1() );
    }
    f.close();
}
```

The QFileInfo class holds detailed information about a file, such as access permissions, file dates and file types.

The QDir class manages directories and lists of file names.

Qt uses Unicode file names. If you want to do your own I/O on Unix systems you may want to use encodeName() (and decodeName()) to convert the file name into the local encoding.

See also QDataStream [p. 19], QTextStream [p. 150] and Input/Output and Networking.

Member Type Documentation

QFile::DecoderFn

This is used by QFile::setDecodingFunction().

QFile::EncoderFn

This is used by QFile::setEncodingFunction().

Member Function Documentation

QFile::QFile ()

Constructs a QFile with no name.

QFile::QFile (const QString & name)

Constructs a QFile with a file name *name*.

See also setName() [p. 58].

QFile::~QFile ()

Destroys a QFile. Calls close().

bool QFile::at (Offset pos) [virtual]

Sets the file index to *pos*. Returns TRUE if successful; otherwise returns FALSE.

Example:

```
QFile f( "data.bin" );
f.open( IO_ReadOnly );                  // index set to 0
f.at( 100 );                            // set index to 100
f.at( f.at()+50 );                      // set index to 150
f.at( f.size()-80 );                    // set index to 80 before EOF
f.close();
```

Use at() without arguments to retrieve the file offset.

Warning: The result is undefined if the file was open()'ed using the IO_Append specifier.

See also size() [p. 58] and open() [p. 55].

Reimplemented from QIODevice [p. 79].

Offset QFile::at () const [virtual]

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

Returns the position in the file.

See also size() [p. 58].

Reimplemented from QIODevice [p. 79].

bool QFile::atEnd () const [virtual]

Returns TRUE if the end of file has been reached; otherwise returns FALSE.

See also size() [p. 58].

Reimplemented from QIODevice [p. 79].

void QFile::close () [virtual]

Closes an open file.

The file is not closed if it was opened with an existing file handle. If the existing file handle is a `FILE*`, the file is flushed. If the existing file handle is an `int` file descriptor, nothing is done to the file.

Some "write-behind" filesystems may report an unspecified error on closing the file. These errors only indicate that something may have gone wrong since the previous open(). In such a case status() reports IO_UnspecifiedError after close(), otherwise IO_Ok.

See also open() [p. 55] and flush() [p. 54].

Examples: addressbook/centralwidget.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp, qdir/qdir.cpp, qwerty/qwerty.cpp and xml/outliner/outlinetree.cpp.

Reimplemented from QIODevice [p. 79].

QString QFile::decodeName (const QCString & localFileName) [static]

This does the reverse of QFile::encodeName() using *localFileName*.

See also setDecodingFunction() [p. 57].

QCString QFile::encodeName (const QString & fileName) [static]

When you use QFile, QFileInfo, and QDir to access the file system with Qt, you can use Unicode file names. On Unix, these file names are converted to an 8-bit encoding. If you want to do your own file I/O on Unix, you should convert the file name using this function. On Windows NT, Unicode file names are supported directly in the file system and this function should be avoided. On Windows 95, non-Latin1 locales are not supported at this time.

By default, this function converts *fileName* to the local 8-bit encoding determined by the user's locale. This is sufficient for file names that the user chooses. File names hard-coded into the application should only use 7-bit ASCII filename characters.

The conversion scheme can be changed using setEncodingFunction(). This might be useful if you wish to give the user an option to store file names in utf-8, etc., but be ware that such file names would probably then be unrecognizable

when seen by other programs.

See also decodeName() [p. 53].

bool QFile::exists (const QString & fileName) [static]

Returns TRUE if the file given by *fileName* exists; otherwise returns FALSE.

Examples: dirview/dirview.cpp and helpviewer/helpwindow.cpp.

bool QFile::exists () const

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

Returns TRUE if this file exists; otherwise returns FALSE.

See also name() [p. 54].

void QFile::flush () [virtual]

Flushes the file buffer to the disk.

close() also flushes the file buffer.

Reimplemented from QIODevice [p. 80].

int QFile::getch () [virtual]

Reads a single byte/character from the file.

Returns the byte/character read, or -1 if the end of the file has been reached.

See also putch() [p. 56] and ungetch() [p. 58].

Reimplemented from QIODevice [p. 80].

int QFile::handle () const

Returns the file handle of the file.

This is a small positive integer, suitable for use with C library functions such as fdopen() and fcntl(), as well as with QSocketNotifier.

If the file is not open or there is an error, handle() returns -1.

See also QSocketNotifier [p. 142].

QString QFile::name () const

Returns the name set by setName().

See also setName() [p. 58] and QFileInfo::fileName() [p. 64].

bool QFile::open (int m) [virtual]

Opens the file specified by the file name currently set, using the mode *m*. Returns TRUE if successful, otherwise FALSE.

The mode parameter *m* must be a combination of the following flags:

- IO_Raw specified raw (non-buffered) file access.
- IO_ReadOnly opens the file in read-only mode.
- IO_WriteOnly opens the file in write-only mode (and truncates).
- IO_ReadWrite opens the file in read/write mode, equivalent to `(IO_ReadOnly | IO_WriteOnly)`.
- IO_Append opens the file in append mode. This mode is very useful when you want to write something to a log file. The file index is set to the end of the file. Note that the result is undefined if you position the file index manually using at() in append mode.
- IO_Truncate truncates the file.
- IO_Translate enables carriage returns and linefeed translation for text files under MS-DOS, Windows and OS/2.

The raw access mode is best when I/O is block-operated using 4kB block size or greater. Buffered access works better when reading small portions of data at a time.

Important: When working with buffered files, data may not be written to the file at once. Call flush() to make sure the data is really written.

Warning: We have experienced problems with some C libraries when a buffered file is opened for both reading and writing. If a read operation takes place immediately after a write operation, the read buffer contains garbage data. Worse, the same garbage is written to the file. Calling flush() before readBlock() solved this problem.

If the file does not exist and IO_WriteOnly or IO_ReadWrite is specified, it is created.

Example:

```
QFile f1( "/tmp/data.bin" );
QFile f2( "readme.txt" );
f1.open( IO_Raw | IO_ReadWrite | IO_Append );
f2.open( IO_ReadOnly | IO_Translate );
```

See also name() [p. 54], close() [p. 53], isOpen() [p. 81] and flush() [p. 54].

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp, qdir/qdir.cpp, qwerty/qwerty.cpp and xml/outliner/outlinetree.cpp.

Reimplemented from QIODevice [p. 82].

bool QFile::open (int m, FILE * f)

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

Opens a file in the mode *m* using an existing file handle *f*. Returns TRUE if successful, otherwise FALSE.

Example:

```
#include

void printError( const char* msg )
{
```

```
    QFile f;
    f.open( IO_WriteOnly, stderr );
    f.writeBlock( msg, qstrlen(msg) );      // write to stderr
    f.close();
}
```

When a QFile is opened using this function, close() does not actually close the file, only flushes it.

Warning: If *f* is `stdin`, `stdout`, `stderr`, you may not be able to seek. See QIODevice::isSequentialAccess() for more information.

See also close() [p. 53].

bool QFile::open (int m, int f)

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

Opens a file in the mode *m* using an existing file descriptor *f*. Returns TRUE if successful, otherwise FALSE.

When a QFile is opened using this function, close() does not actually close the file.

The QFile that is opened using this function, is automatically set to be in raw mode; this means that the file input/output functions are slow. If you run into performance issues, you should try to use one of the other open functions.

Warning: If *f* is one of 0 (stdin), 1 (stdout) or 2 (stderr), you may not be able to seek. size() is set to `INT_MAX` (in limits.h).

See also close() [p. 53].

int QFile::putch (int ch) [virtual]

Writes the character *ch* to the file.

Returns *ch*, or -1 if some error occurred.

See also getch() [p. 54] and ungetch() [p. 58].

Reimplemented from QIODevice [p. 83].

QByteArray QIODevice::readAll () [virtual]

This convenience function returns all of the remaining data in the device.

Q_LONG QFile::readBlock (char * p, Q_ULONG len) [virtual]

Reads at most *len* bytes from the file into *p* and returns the number of bytes actually read.

Returns -1 if a serious error occurred.

Warning: We have experienced problems with some C libraries when a buffered file is opened for both reading and writing. If a read operation takes place immediately after a write operation, the read buffer contains garbage data. Worse, the same garbage is written to the file. Calling flush() before readBlock() solved this problem.

See also writeBlock() [p. 85].

Example: qwerty/qwerty.cpp.

Reimplemented from QIODevice [p. 83].

Q_LONG QFile::readLine (char * p, Q_ULONG maxlen) [virtual]

Reads a line of text.

Reads bytes from the file into the char* *p*, until end-of-line or *maxlen* bytes have been read, whichever occurs first. Returns the number of bytes read, or -1 if there was an error. The terminating newline is not stripped.

This function is efficient only for buffered files. Avoid readLine() for files that have been opened with the IO_Raw flag.

See also readBlock() [p. 56] and QTextStream::readLine() [p. 159].

Reimplemented from QIODevice [p. 83].

Q_LONG QFile::readLine (QString & s, Q_ULONG maxlen)

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

Reads a line of text.

Reads bytes from the file into string *s*, until end-of-line or *maxlen* bytes have been read, whichever occurs first. Returns the number of bytes read, or -1 if there was an error.g. end of file. The terminating newline is not stripped.

This function is efficient only for buffered files. Avoid readLine() for files that have been opened with the IO_Raw flag.

Note that the string is read as plain Latin1 bytes, not Unicode.

See also readBlock() [p. 56] and QTextStream::readLine() [p. 159].

bool QFile::remove ()

Removes the file specified by the file name currently set. Returns TRUE if successful; otherwise returns FALSE.

The file is closed before it is removed.

bool QFile::remove (const QString & fileName) [static]

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

Removes the file *fileName*. Returns TRUE if successful, otherwise FALSE.

void QFile::setDecodingFunction (DecoderFn f) [static]

Sets the function for decoding 8-bit file names to *f*. The default uses the locale-specific 8-bit encoding.

See also encodeName() [p. 53] and decodeName() [p. 53].

void QFile::setEncodingFunction (EncoderFn f) [static]

Sets the function for encoding Unicode file names to *f*. The default encodes in the locale-specific 8-bit encoding.

See also encodeName() [p. 53].

void QFile::setName (const QString & name)

Sets the name of the file to *name*. The name may have no path, a relative path or an absolute absolute directory path.

Do not call this function if the file has already been opened.

If the file name has no path or a relative path, the path used will be whatever the application's current directory path is *at the time of the open()* call.

Example:

```
QFile f;
QDir::setCurrent( "/tmp" );
f.setName( "readme.txt" );
QDir::setCurrent( "/home" );
f.open( IO_ReadOnly );        // opens "/home/readme.txt" under Unix
```

Note that the directory separator "/" works for all operating systems supported by Qt.

See also name() [p. 54], QFileInfo [p. 59] and QDir [p. 29].

Offset QFile::size () const [virtual]

Returns the file size.

See also at() [p. 52].

Reimplemented from QIODevice [p. 84].

int QFile::ungetch (int ch) [virtual]

Puts the character *ch* back into the file and decrements the index if it is not zero.

This function is normally called to "undo" a getch() operation.

Returns *ch*, or -1 if some error occurred.

See also getch() [p. 54] and putch() [p. 56].

Reimplemented from QIODevice [p. 84].

QFileInfo Class Reference

The QFileInfo class provides system-independent file information.

`#include <qfileinfo.h>`

Public Members

- enum **PermissionSpec** { ReadUser = 0400, WriteUser = 0200, ExeUser = 0100, ReadGroup = 0040, WriteGroup = 0020, ExeGroup = 0010, ReadOther = 0004, WriteOther = 0002, ExeOther = 0001 }
- **QFileInfo** ()
- **QFileInfo** (const QString & file)
- **QFileInfo** (const QFile & file)
- **QFileInfo** (const QDir & d, const QString & fileName)
- **QFileInfo** (const QFileInfo & fi)
- **~QFileInfo** ()
- QFileInfo & **operator=** (const QFileInfo & fi)
- void **setFile** (const QString & file)
- void **setFile** (const QFile & file)
- void **setFile** (const QDir & d, const QString & fileName)
- bool **exists** () const
- void **refresh** () const
- bool **caching** () const
- void **setCaching** (bool enable)
- QString **filePath** () const
- QString **fileName** () const
- QString **absFilePath** () const
- QString **baseName** (bool complete = FALSE) const
- QString **extension** (bool complete = TRUE) const
- QString **dirPath** (bool absPath = FALSE) const
- QDir **dir** (bool absPath = FALSE) const
- bool **isReadable** () const
- bool **isWritable** () const
- bool **isExecutable** () const
- bool **isRelative** () const
- bool **convertToAbs** ()
- bool **isFile** () const
- bool **isDir** () const

- bool **isSymLink** () const
- QString **readLink** () const
- QString **owner** () const
- uint **ownerId** () const
- QString **group** () const
- uint **groupId** () const
- bool **permission** (int permissionSpec) const
- uint **size** () const
- QDateTime **created** () const
- QDateTime **lastModified** () const
- QDateTime **lastRead** () const

Detailed Description

The QFileInfo class provides system-independent file information.

QFileInfo provides information about a file's name and position (path) in the file system, its access rights and whether it is a directory or symbolic link, etc. The file's size and last modified/read times are also available.

A QFileInfo can point to a file with either a relative or an absolute file path. Absolute file paths begin with the directory separator "/" or a drive specification (except on Unix). Relative file names begin with a directory name or a file name and specify a path relative to the current working directory. An example of an absolute path is the string "/tmp/quartz". A relative path might look like "src/fatlib". You can use the function isRelative() to check whether a QFileInfo is using a relative or an absolute file path. You can call the function convertToAbs() to convert a relative QFileInfo's path to an absolute path.

The file that the QFileInfo works on is set in the constructor or later with setFile(). Use exists() to see if the file exists and size() to get its size.

To speed up performance, QFileInfo caches information about the file. Because files can be changed by other users or programs, or even by other parts of the same program, there is a function that refreshes the file information: refresh(). If you want to switch off a QFileInfo's caching and force it to access the file system every time you request information from it call setCaching(FALSE).

The file's type is obtained with isFile(), isDir() and isSymLink(). The readLink() function provides the name of the file the symlink points to.

Elements of the file's name can be extracted with dirPath() and fileName(). The fileName()'s parts can be extracted with baseName() and extension().

The file's dates are returned by created(), lastModified() and lastRead(). Information about the file's access permissions is obtained with isReadable(), isWritable() and isExecutable(). The file's ownership is available from owner(), ownerId(), group() and groupId(). You can examine a file's permissions and ownership in a single statement using the permission() function.

If you need to read and traverse directories, see the QDir class.

See also Input/Output and Networking.

Member Type Documentation

QFileInfo::PermissionSpec

This enum is used by the permission() function to report the permissions and ownership of a file. The values may be OR-ed together to test multiple permissions and ownership values.

- `QFileInfo::ReadUser` - The file is readable by the user.
- `QFileInfo::WriteUser` - The file is writable by the user.
- `QFileInfo::ExeUser` - The file is executable by the user.
- `QFileInfo::ReadGroup` - The file is readable by the group.
- `QFileInfo::WriteGroup` - The file is writable by the group.
- `QFileInfo::ExeGroup` - The file is executable by the group.
- `QFileInfo::ReadOther` - The file is readable by anyone.
- `QFileInfo::WriteOther` - The file is writable by anyone.
- `QFileInfo::ExeOther` - The file is executable by anyone.

Member Function Documentation

QFileInfo::QFileInfo ()

Constructs a new empty QFileInfo.

QFileInfo::QFileInfo (const QString & file)

Constructs a new QFileInfo that gives information about the given file. The *file* can be an absolute or a relative file path.

See also setFile() [p. 67], isRelative() [p. 65], QDir::setCurrent() [p. 43] and QDir::isRelativePath() [p. 39].

QFileInfo::QFileInfo (const QFile & file)

Constructs a new QFileInfo that gives information about file *file*.

If the *file* has a relative path, the QFileInfo will also have a relative path.

See also isRelative() [p. 65].

QFileInfo::QFileInfo (const QDir & d, const QString & fileName)

Constructs a new QFileInfo that gives information about the file named *fileName* in the directory *d*.

If the directory has a relative path, the QFileInfo will also have a relative path.

See also isRelative() [p. 65].

QFileInfo::QFileInfo (const QFileInfo & fi)

Constructs a new QFileInfo that is a copy of *fi*.

QFileInfo::~QFileInfo ()

Destroys the QFileInfo and frees its resources.

QString QFileInfo::absFilePath () const

Returns the absolute path including the file name.

The absolute path name consists of the full path and the file name. On Unix this will always begin with the root, '/', directory. On Windows this will always begin 'D:/' where D is a drive letter, except for network shares that are not mapped to a drive letter, in which case the path will begin '//sharename/'.

This function returns the same as filePath(), unless isRelative() is TRUE.

This function can be time consuming under Unix (in the order of milliseconds).

See also isRelative() [p. 65] and filePath() [p. 64].

Examples: biff/biff.cpp and fileiconview/qfileiconview.cpp.

QString QFileInfo::baseName (bool complete = FALSE) const

Returns the base name of the file.

If *complete* is FALSE (the default) the base name consists of all characters in the file name up to (but not including) the *first* '.' character.

If *complete* is TRUE the base name consists of all characters in the file up to (but not including) the *last* '.' character.

The path is not included in either case.

Example:

```
QFileInfo fi( "/tmp/archive.tar.gz" );
QString base = fi.baseName();  // base = "archive"
base = fi.baseName( TRUE );    // base = "archive.tar"
```

See also fileName() [p. 64] and extension() [p. 63].

bool QFileInfo::caching () const

Returns TRUE if caching is enabled; otherwise returns FALSE.

See also setCaching() [p. 67] and refresh() [p. 67].

bool QFileInfo::convertToAbs ()

Converts the file path name to an absolute path.

If it is already absolute, nothing is done.

See also filePath() [p. 64] and isRelative() [p. 65].

QDateTime QFileInfo::created () const

Returns the date and time when the file was created.

On platforms where this information is not available, returns the same as lastModified().

See also lastModified() [p. 65] and lastRead() [p. 66].

QDir QFileInfo::dir (bool absPath = FALSE) const

Returns the directory path of the file.

If the QFileInfo is relative and *absPath* is FALSE, the QDir will be relative; otherwise it will be absolute.

See also dirPath() [p. 63], filePath() [p. 64], fileName() [p. 64] and isRelative() [p. 65].

Example: fileiconview/qfileiconview.cpp.

QString QFileInfo::dirPath (bool absPath = FALSE) const

Returns the directory path of the file.

If *absPath* is TRUE an absolute path is returned.

See also dir() [p. 63], filePath() [p. 64], fileName() [p. 64] and isRelative() [p. 65].

Example: fileiconview/qfileiconview.cpp.

bool QFileInfo::exists () const

Returns TRUE if the file exists; otherwise returns FALSE.

Examples: biff/biff.cpp and i18n/main.cpp.

QString QFileInfo::extension (bool complete = TRUE) const

Returns the file's extension name.

If *complete* is TRUE (the default), extension() returns the string of all characters in the file name after (but not including) the first '.' character.

If *complete* is FALSE, extension() returns the string of all characters in the file name after (but not including) the last '.' character.

Example:

```
QFileInfo fi( "/tmp/archive.tar.gz" );
QString ext = fi.extension();  // ext = "tar.gz"
ext = fi.extension( FALSE );   // ext = "gz"
```

See also fileName() [p. 64] and baseName() [p. 62].

Example: qdir/qdir.cpp.

QString QFileInfo::fileName () const

Returns the name of the file, the file path is not included.

Example:

```
QFileInfo fi( "/tmp/archive.tar.gz" );
QString name = fi.fileName();              // name = "archive.tar.gz"
```

See also isRelative() [p. 65], filePath() [p. 64], baseName() [p. 62] and extension() [p. 63].

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

QString QFileInfo::filePath () const

Returns the file name, including the path (which may be absolute or relative).

See also isRelative() [p. 65] and absFilePath() [p. 62].

Examples: dirview/main.cpp and fileiconview/qfileiconview.cpp.

QString QFileInfo::group () const

Returns the group of the file. On Windows, on systems where files do not have groups, or if an error occurs, a null string is returned.

This function can be time consuming under Unix (in the order of milliseconds).

See also groupId() [p. 64], owner() [p. 66] and ownerId() [p. 66].

uint QFileInfo::groupId () const

Returns the id of the group the file belongs to.

On Windows and on systems where files do not have groups this function always returns (uint) -2.

See also group() [p. 64], owner() [p. 66] and ownerId() [p. 66].

bool QFileInfo::isDir () const

Returns TRUE if this object points to a directory or to a symbolic link to a directory; otherwise returns FALSE.

See also isFile() [p. 65] and isSymLink() [p. 65].

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

bool QFileInfo::isExecutable () const

Returns TRUE if the file is executable; otherwise returns FALSE.

See also isReadable() [p. 65], isWritable() [p. 65] and permission() [p. 66].

bool QFileInfo::isFile () const

Returns TRUE if this object points to a file. Returns FALSE if the object points to something which isn't a file, e.g. a directory or a symlink.

See also isDir() [p. 64] and isSymLink() [p. 65].

Examples: dirview/dirview.cpp, fileiconview/qfileiconview.cpp and qdir/qdir.cpp.

bool QFileInfo::isReadable () const

Returns TRUE if the file is readable; otherwise returns FALSE.

See also isWritable() [p. 65], isExecutable() [p. 65] and permission() [p. 66].

bool QFileInfo::isRelative () const

Returns TRUE if the file path name is relative. Returns FALSE if the path is absolute (e.g. under Unix a path is absolute if it begins with a "/").

bool QFileInfo::isSymLink () const

Returns TRUE if this object points to a symbolic link (or to a shortcut on Windows); otherwise returns FALSE.

See also isFile() [p. 65], isDir() [p. 64] and readLink() [p. 66].

Examples: dirview/dirview.cpp and fileiconview/qfileiconview.cpp.

bool QFileInfo::isWritable () const

Returns TRUE if the file is writable; otherwise returns FALSE.

See also isReadable() [p. 65], isExecutable() [p. 65] and permission() [p. 66].

QDateTime QFileInfo::lastModified () const

Returns the date and time when the file was last modified.

See also created() [p. 63] and lastRead() [p. 66].

Example: biff/biff.cpp.

QDateTime QFileInfo::lastRead () const

Returns the date and time when the file was last read (accessed).

On platforms where this information is not available, returns the same as lastModified().

See also created() [p. 63] and lastModified() [p. 65].

Example: biff/biff.cpp.

QFileInfo & QFileInfo::operator= (const QFileInfo & fi)

Makes a copy of *fi* and assigns it to this QFileInfo.

QString QFileInfo::owner () const

Returns the owner of the file. On Windows, on systems where files do not have owners, or if an error occurs, a null string is returned.

This function can be time consuming under Unix (in the order of milliseconds).

See also ownerId() [p. 66], group() [p. 64] and groupId() [p. 64].

uint QFileInfo::ownerId () const

Returns the id of the owner of the file.

On Windows and on systems where files do not have owners this function returns ((uint) -2).

See also owner() [p. 66], group() [p. 64] and groupId() [p. 64].

bool QFileInfo::permission (int permissionSpec) const

Tests for file permissions. The *permissionSpec* argument can be several flags of type PermissionSpec OR-ed together to check for permission combinations.

On systems where files do not have permissions this function always returns TRUE.

Example:

```
QFileInfo fi( "/tmp/archive.tar.gz" );
if ( fi.permission( QFileInfo::WriteUser | QFileInfo::ReadGroup ) )
    qWarning( "I can change the file; my group can read the file.");
if ( fi.permission( QFileInfo::WriteGroup | QFileInfo::WriteOther ) )
    qWarning( "The group or others can change the file!" );
```

See also isReadable() [p. 65], isWritable() [p. 65] and isExecutable() [p. 65].

QString QFileInfo::readLink () const

Returns the name a symlink (or shortcut on Windows) points to, or a null QString if the object isn't a symbolic link.

This name may not represent an existing file; it is only a string. QFileInfo::exists() returns TRUE if the symlink points to an existing file.

See also exists() [p. 63], isSymLink() [p. 65], isDir() [p. 64] and isFile() [p. 65].

void QFileInfo::refresh () const

Refreshes the information about the file, i.e. reads in information from the file system the next time a cached property is fetched.

See also setCaching() [p. 67].

void QFileInfo::setCaching (bool enable)

If *enable* is TRUE, enables caching of file information. If *enable* is FALSE caching is disabled.

When caching is enabled, QFileInfo reads the file information from the file system the first time it's needed, but generally not later.

Caching is enabled by default.

See also refresh() [p. 67] and caching() [p. 62].

void QFileInfo::setFile (const QString & file)

Sets the file that the QFileInfo provides information about to *file*.

The string given can be an absolute or a relative file path. Absolute paths begin with the directory separator (e.g. "/" under Unix) or a drive specification (not applicable to Unix). Relative file names begin with a directory name or a file name and specify a path relative to the current directory.

Example:

```
QString absolute = "/local/bin";
QString relative = "local/bin";
QFileInfo absFile( absolute );
QFileInfo relFile( relative );

QDir::setCurrent( QDir::rootDirPath() );
// absFile and relFile now point to the same file

QDir::setCurrent( "/tmp" );
// absFile now points to "/local/bin",
// while relFile points to "/tmp/local/bin"
```

See also isRelative() [p. 65], QDir::setCurrent() [p. 43] and QDir::isRelativePath() [p. 39].

Example: biff/biff.cpp.

void QFileInfo::setFile (const QFile & file)

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

Sets the file that the QFileInfo provides information about to *file*.

If the file has a relative path, the QFileInfo will also have a relative path.

See also isRelative() [p. 65].

void QFileInfo::setFile (const QDir & d, const QString & fileName)

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

Sets the file that the QFileInfo provides information about to *fileName* in directory *d*.

If the file has a relative path, the QFileInfo will also have a relative path.

See also isRelative() [p. 65].

uint QFileInfo::size () const

Returns the file size in bytes, or 0 if the file does not exist or if the size is 0 or if the size cannot be fetched.

Example: qdir/qdir.cpp.

QFtp Class Reference

The QFtp class provides an implementation of the FTP protocol.

This class is part of the **network** module.

`#include <qftp.h>`

Inherits QNetworkProtocol [p. 99].

Public Members

- **QFtp** ()
- virtual **~QFtp** ()

Protected Members

- void **parseDir** (const QString & buffer, QUrlInfo & info)

Protected Slots

- void **readyRead** ()
- void **dataConnected** ()
- void **dataClosed** ()
- void **dataReadyRead** ()
- void **dataBytesWritten** (int nbytes)

Detailed Description

The QFtp class provides an implementation of the FTP protocol.

This class is derived from QNetworkProtocol. QFtp is not normally used directly, but rather through a QUrlOperator, for example:

```
QUrlOperator op( "ftp://ftp.trolltech.com" );
op.listChildren(); // Asks the server to provide a directory listing
```

This code will only work if the QFtp class is registered; to register the class, you must call qInitNetworkProtocols() before using a QUrlOperator with QFtp.

If you really need to use QFtp directly, don't forget to set its QUrlOperator with setUrl().

See also Qt Network Documentation [p. 4], QNetworkProtocol [p. 99], QUrlOperator [p. 179] and Input/Output and Networking.

Member Function Documentation

QFtp::QFtp ()

Constructs a QFtp object.

QFtp::~QFtp () [virtual]

Destructor

void QFtp::dataBytesWritten (int nbytes) [protected slot]

This function is called, when *nbytes* have been successfully written to the data socket.

void QFtp::dataClosed () [protected slot]

This function is called when the data connection has been closed.

void QFtp::dataConnected () [protected slot]

Some operations require a data connection to the server. If this connection could be opened, this function handles the data connection.

void QFtp::dataReadyRead () [protected slot]

This function is called when new data arrived on the data socket.

void QFtp::parseDir (const QString & buffer, QUrlInfo & info) [protected]

Parses the string, *buffer*, which is one line of a directory listing which came from the FTP server, and sets the values which have been parsed to the url info object, *info*.

void QFtp::readyRead () [protected slot]

If data has arrived on the command socket, this slot is called. The function loks at the data and passes it on to the function which can handle it

QHostAddress Class Reference

The QHostAddress class provides an IP address.

This class is part of the **network** module.

```
#include <qhostaddress.h>
```

Public Members

- **QHostAddress** ()
- **QHostAddress** (Q_UINT32 ip4Addr)
- **QHostAddress** (Q_UINT8 * ip6Addr)
- **QHostAddress** (const QHostAddress & address)
- virtual **~QHostAddress** ()
- QHostAddress & **operator=** (const QHostAddress & address)
- void **setAddress** (Q_UINT32 ip4Addr)
- void **setAddress** (Q_UINT8 * ip6Addr)
- bool **setAddress** (const QString & address)
- bool **isIp4Addr** () const
- Q_UINT32 **ip4Addr** () const
- QString **toString** () const
- bool **operator==** (const QHostAddress & other) const

Detailed Description

The QHostAddress class provides an IP address.

This class contains an IP address in a platform and protocol independent manner. It stores both IPv4 and IPv6 addresses in a way that you can easily access on any platform.

QHostAddress is normally used with the classes QSocket, QServerSocket and QSocketDevice to set up a server or to connect to a host.

Host addresses may be set with setAddress() and retrieved with ip4Addr() or toString().

See also QSocket [p. 126], QServerSocket [p. 118], QSocketDevice [p. 135] and Input/Output and Networking.

Member Function Documentation

QHostAddress::QHostAddress ()

Creates a host address object with the IP address 0.0.0.0.

QHostAddress::QHostAddress (Q_UINT32 ip4Addr)

Creates a host address object for the IPv4 address *ip4Addr*.

QHostAddress::QHostAddress (Q_UINT8 * ip6Addr)

Creates a host address object with the specified IPv6 address.

ip6Addr must be a 16 byte array in network byte order (high-order byte first)

QHostAddress::QHostAddress (const QHostAddress & address)

Creates a copy of *address*.

QHostAddress::~QHostAddress () [virtual]

Destroys the host address object.

Q_UINT32 QHostAddress::ip4Addr () const

Returns the IPv4 address as a number.

For example, if the address is 127.0.0.1, the returned value is 2130706433 (hex: 7f000001).

This value is only valid when isIp4Addr() returns TRUE.

See also toString() [p. 73].

bool QHostAddress::isIp4Addr () const

Returns TRUE if the host address represents a IPv4 address; otherwise returns FALSE.

QHostAddress & QHostAddress::operator= (const QHostAddress & address)

Assigns another host address object *address* to this object and returns a reference to this object.

bool QHostAddress::operator== (const QHostAddress & other) const

Returns TRUE if this host address is the same as *other*; otherwise returns FALSE.

void QHostAddress::setAddress (Q_UINT32 ip4Addr)

Set the IPv4 address specified by *ip4Addr*.

void QHostAddress::setAddress (Q_UINT8 * ip6Addr)

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

Set the IPv6 address specified by *ip6Addr*.

ip6Addr must be a 16 byte array in network byte order (high-order byte first)

bool QHostAddress::setAddress (const QString & address)

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

Sets the IPv4 or IPv6 address specified by the string representation *address* (e.g. "127.0.0.1"). Returns TRUE and sets the address if the address was successfully parsed; otherwise returns FALSE and leaves the address unchanged.

QString QHostAddress::toString () const

Returns the address as a string.

For example, if the address is the IPv4 address 127.0.0.1, the returned string is "127.0.0.1".

See also ip4Addr() [p. 72].

QHttp Class Reference

The QHttp class provides an implementation of the HTTP protocol.

This class is part of the **network** module.

`#include <qhttp.h>`

Inherits QNetworkProtocol [p. 99].

Public Members

- **QHttp** ()
- virtual **~QHttp** ()

Detailed Description

The QHttp class provides an implementation of the HTTP protocol.

This class is derived from QNetworkProtocol and can be used with QUrlOperator. In practice this class is used through a QUrlOperator rather than directly, for example:

```
QUrlOperator op( "http://www.trolltech.com" );
op.get( "index.html" );
```

Note: this code will only work if the QHttp class is registered; to register the class, you must call qInitNetworkProtocols() before using a QUrlOperator with HTTP.

QHttp only supports the operations operationGet() and operationPut(), i.e. QUrlOperator::get() and QUrlOperator::put(), if you use it with a QUrlOperator.

If you really need to use QHttp directly, don't forget to set the QUrlOperator on which it operates using setUrl().

See also Qt Network Documentation [p. 4], QNetworkProtocol [p. 99], QUrlOperator [p. 179] and Input/Output and Networking.

Member Function Documentation

QHttp::QHttp ()

Constructs a QHttp object. Usually there is no need to use QHttp directly, since it is more convenient to use it through a QUrlOperator. If you want to use it directly, you must set the QUrlOperator on which it operates using setUrl().

QHttp::~QHttp () [virtual]

Destroys the QHttp object.

QIODevice Class Reference

The QIODevice class is the base class of I/O devices.

`#include <qiodevice.h>`

Inherited by QBuffer [Graphics with Qt], QFile [p. 50], QSocket [p. 126] and QSocketDevice [p. 135].

Public Members

- typedef off_t **Offset**
- **QIODevice** ()
- virtual **~QIODevice** ()
- int **flags** () const
- int **mode** () const
- int **state** () const
- bool **isDirectAccess** () const
- bool **isSequentialAccess** () const
- bool **isCombinedAccess** () const
- bool **isBuffered** () const
- bool **isRaw** () const
- bool **isSynchronous** () const
- bool **isAsynchronous** () const
- bool **isTranslated** () const
- bool **isReadable** () const
- bool **isWritable** () const
- bool **isReadWrite** () const
- bool **isInactive** () const
- bool **isOpen** () const
- int **status** () const
- void **resetStatus** ()
- virtual bool **open** (int mode)
- virtual void **close** ()
- virtual void **flush** ()
- virtual Offset **size** () const
- virtual Offset **at** () const
- virtual bool **at** (Offset pos)
- virtual bool **atEnd** () const

- bool **reset** ()
- virtual Q_LONG **readBlock** (char * data, Q_ULONG maxlen)
- virtual Q_LONG **writeBlock** (const char * data, Q_ULONG len)
- virtual Q_LONG **readLine** (char * data, Q_ULONG maxlen)
- Q_LONG **writeBlock** (const QByteArray & data)
- virtual QByteArray **readAll** ()
- virtual int **getch** ()
- virtual int **putch** (int ch)
- virtual int **ungetch** (int ch)

Detailed Description

The QIODevice class is the base class of I/O devices.

An I/O device represents a medium that one can read bytes from and/or write bytes to. The QIODevice class is the abstract superclass of all such devices; classes such as QFile, QBuffer and QSocket inherit QIODevice and implement virtual functions such as write() appropriately.

Although applications sometimes use QIODevice directly, it is usually better to go through QTextStream and QDataStream, which provide stream operations on any QIODevice subclass. QTextStream provides text-oriented stream functionality (for human-readable ASCII files, for example), whereas QDataStream deals with binary data in a totally platform-independent manner.

The public member functions in QIODevice roughly fall into two groups: the action functions and the state access functions. The most important action functions are:

- open() opens a device for reading and/or writing, depending on the argument to open().
- close() closes the device and tidies up.
- readBlock() reads a block of data from the device.
- writeBlock() writes a block of data to the device.
- readLine() reads a line (of text, usually) from the device.
- flush() ensures that all buffered data are written to the real device.

There are also some other, less used, action functions:

- getch() reads a single character.
- ungetch() forgets the last call to getch(), if possible.
- putch() writes a single character.
- size() returns the size of the device, if there is one.
- at() returns the current read/write pointer's position, if there is one for this device, or it moves the pointer.
- atEnd() says whether there is more to read, if that is a meaningful question for this device.
- reset() moves the read/write pointer to the start of the device, if that is possible for this device.

The state access are all "get" functions. The QIODevice subclass calls setState() to update the state, and simple access functions tell the user of the device what the device's state is. Here are the settings, and their associated access functions:

- Access type. Some devices are direct access (it is possible to read/write anywhere), whereas others are sequential. QIODevice provides the access functions (isDirectAccess(), isSequentialAccess(), and isCombinedAccess()) to tell users what a given I/O device supports.
- Buffering. Some devices are accessed in raw mode, whereas others are buffered. Buffering usually provides greater efficiency, particularly for small read/write operations. isBuffered() tells the user whether a given device is buffered. (This can often be set by the application in the call to open().)
- Synchronicity. Synchronous devices work immediately (for example, files). When you read from a file, the file delivers its data straight away. Other kinds of device, such as a socket connected to a HTTP server, may not deliver the data until seconds after you ask to read it. isSynchronous() and isAsynchronous() tell the user how this device operates.
- CR/LF translation. For simplicity, applications often like to see just a single CR/LF style, and QIODevice subclasses can provide this. isTranslated() returns TRUE if this object translates CR/LF to just LF. (This can often be set by the application in the call to open().)
- Permissions. Some files cannot be written. For example, isReadable(), isWritable() and isReadWrite() tell the application whether it can read from and write to a given device. (This can often be set by the application in the call to open().)
- Finally, isOpen() returns TRUE if the device is open, i.e. after an open() call.

QIODevice provides numerous pure virtual functions that you need to implement when subclassing it. Here is a skeleton subclass with all the members you are certain to need and some that you probably will need:

```
class MyDevice : public QIODevice
{
public:
    MyDevice();
    ~MyDevice();

    bool open( int mode );
    void close();
    void flush();

    uint size() const;
    int  at() const;        // non-pure virtual
    bool at( int );         // non-pure virtual
    bool atEnd() const;     // non-pure virtual

    int readBlock( char *data, uint maxlen );
    int writeBlock( const char *data, uint len );
    int readLine( char *data, uint maxlen );

    int getch();
    int putch( int );
    int ungetch( int );
};
```

The three non-pure virtual functions need not be reimplemented for sequential devices.

See also QDataStream [p. 19], QTextStream [p. 150] and Input/Output and Networking.

Member Type Documentation

QIODevice::Offset

The offset within the device.

Member Function Documentation

QIODevice::QIODevice ()

Constructs an I/O device.

QIODevice::~QIODevice () [virtual]

Destroys the I/O device.

Offset QIODevice::at () const [virtual]

Virtual function that returns the current I/O device position.

This is the position of the data read/write head of the I/O device.

See also size() [p. 84].

Reimplemented in QFile and QSocket.

bool QIODevice::at (Offset pos) [virtual]

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

Virtual function that sets the I/O device position to *pos*.

See also size() [p. 84].

Reimplemented in QFile and QSocket.

bool QIODevice::atEnd () const [virtual]

Virtual function that returns TRUE if the I/O device position is at the end of the input; otherwise returns FALSE.

Reimplemented in QFile and QSocket.

void QIODevice::close () [virtual]

Closes the I/O device.

This virtual function must be reimplemented by all subclasses.

See also open() [p. 82].

Reimplemented in QFile and QSocket.

int QIODevice::flags () const

Returns the current I/O device flags setting.

Flags consists of mode flags and state flags.

See also mode() [p. 82] and state() [p. 84].

void QIODevice::flush () [virtual]

Flushes an open I/O device.

This virtual function must be reimplemented by all subclasses.

Reimplemented in QFile and QSocket.

int QIODevice::getch () [virtual]

Reads a single byte/character from the I/O device.

Returns the byte/character read, or -1 if the end of the I/O device has been reached.

This virtual function must be reimplemented by all subclasses.

See also putch() [p. 83] and ungetch() [p. 84].

Reimplemented in QFile and QSocket.

bool QIODevice::isAsynchronous () const

Returns TRUE if the device is an asynchronous device; otherwise returns FALSE, i.e. if the device is a synchronous device.

This mode is currently not in use.

See also isSynchronous() [p. 82].

bool QIODevice::isBuffered () const

Returns TRUE if the I/O device is a buffered device; otherwise returns FALSE, i.e. the device is a raw device.

See also isRaw() [p. 81].

bool QIODevice::isCombinedAccess () const

Returns TRUE if the I/O device is a combined access (both direct and sequential) device; otherwise returns FALSE.

This access method is currently not in use.

bool QIODevice::isDirectAccess () const

Returns TRUE if the I/O device is a direct access device; otherwise returns FALSE, i.e. if the device is a sequential access device.

See also isSequentialAccess() [p. 81].

bool QIODevice::isInactive () const

Returns TRUE if the I/O device state is 0, i.e. the device is not open; otherwise returns FALSE.

See also isOpen() [p. 81].

bool QIODevice::isOpen () const

Returns TRUE if the I/O device has been opened; otherwise returns FALSE.

See also isInactive() [p. 81].

Example: network/networkprotocol/nntp.cpp.

bool QIODevice::isRaw () const

Returns TRUE if the device is a raw device; otherwise returns FALSE, i.e. if the device is a buffered device.

See also isBuffered() [p. 80].

bool QIODevice::isReadWrite () const

Returns TRUE if the I/O device was opened using IO_ReadWrite mode; otherwise returns FALSE.

See also isReadable() [p. 81] and isWritable() [p. 82].

bool QIODevice::isReadable () const

Returns TRUE if the I/O device was opened using IO_ReadOnly or IO_ReadWrite mode; otherwise returns FALSE.

See also isWritable() [p. 82] and isReadWrite() [p. 81].

bool QIODevice::isSequentialAccess () const

Returns TRUE if the device is a sequential access device; otherwise returns FALSE, i.e. if the device is a direct access device.

Operations involving size() and at(int) are not valid on sequential devices.

See also isDirectAccess() [p. 81].

bool QIODevice::isSynchronous () const

Returns TRUE if the I/O device is a synchronous device; otherwise returns FALSE, i.e. the device is an asynchronous device.

See also isAsynchronous() [p. 80].

bool QIODevice::isTranslated () const

Returns TRUE if the I/O device translates carriage-return and linefeed characters; otherwise returns FALSE.

A QFile is translated if it is opened with the IO_Translate mode flag.

bool QIODevice::isWritable () const

Returns TRUE if the I/O device was opened using IO_WriteOnly or IO_ReadWrite mode; otherwise returns FALSE.

See also isReadable() [p. 81] and isReadWrite() [p. 81].

int QIODevice::mode () const

Returns bits OR'ed together that specify the current operation mode.

These are the flags that were given to the open() function.

The flags are IO_ReadOnly, IO_WriteOnly, IO_ReadWrite, IO_Append, IO_Truncate and IO_Translate.

bool QIODevice::open (int mode) [virtual]

Opens the I/O device using the specified *mode*. Returns TRUE if the device was successfully opened; otherwise returns FALSE.

The mode parameter *mode* must be an OR'ed combination of the following flags.

- IO_Raw specified raw (unbuffered) file access.
- IO_ReadOnly opens a file in read-only mode.
- IO_WriteOnly opens a file in write-only mode.
- IO_ReadWrite opens a file in read/write mode.
- IO_Append sets the file index to the end of the file.
- IO_Truncate truncates the file.
- IO_Translate enables carriage returns and linefeed translation for text files under MS-DOS, Windows and Macintosh. On Unix systems this flag has no effect. Use with caution as it will also transform every linefeed written to the file into a CRLF pair. This is likely to corrupt your file if you write write binary data. Cannot be combined with IO_Raw.

This virtual function must be reimplemented by all subclasses.

See also close() [p. 79].

Reimplemented in QFile and QSocket.

int QIODevice::putch (int ch) [virtual]

Writes the character *ch* to the I/O device.

Returns *ch*, or -1 if an error occurred.

This virtual function must be reimplemented by all subclasses.

See also getch() [p. 80] and ungetch() [p. 84].

Reimplemented in QFile and QSocket.

QByteArray QIODevice::readAll () [virtual]

This convenience function returns all of the remaining data in the device.

Q_LONG QIODevice::readBlock (char * data, Q_ULONG maxlen) [virtual]

Reads at most *maxlen* bytes from the I/O device into *data* and returns the number of bytes actually read.

This virtual function must be reimplemented by all subclasses.

See also writeBlock() [p. 85].

Reimplemented in QFile, QSocket and QSocketDevice.

Q_LONG QIODevice::readLine (char * data, Q_ULONG maxlen) [virtual]

Reads a line of text, (or up to *maxlen* bytes if a newline isn't encountered) plus a terminating \0 into *data*. If there is a newline at the end if the line, it is not stripped.

Returns the number of bytes read including the terminating \0, or -1 in case of error.

This virtual function can be reimplemented much more efficiently by the most subclasses.

See also readBlock() [p. 83] and QTextStream::readLine() [p. 159].

Reimplemented in QFile.

bool QIODevice::reset ()

Sets the device index position to 0.

See also at() [p. 79].

void QIODevice::resetStatus ()

Sets the I/O device status to IO_Ok.

See also status() [p. 84].

Offset QIODevice::size () const [virtual]

Virtual function that returns the size of the I/O device.

See also at() [p. 79].

Reimplemented in QFile and QSocket.

int QIODevice::state () const

Returns bits OR'ed together that specify the current state.

The flags are: `IO_Open`.

Subclasses may define additional flags.

int QIODevice::status () const

Returns the I/O device status.

The I/O device status returns an error code. If open() returns FALSE or readBlock() or writeBlock() return -1, this function can be called to find out the reason why the operation did not succeed.

The status codes are:

- IO_Ok - The operation was successful.
- IO_ReadError - Could not read from the device.
- IO_WriteError - Could not write to the device.
- IO_FatalError - A fatal unrecoverable error occurred.
- IO_OpenError - Could not open the device.
- IO_ConnectError - Could not connect to the device.
- IO_AbortError - The operation was unexpectedly aborted.
- IO_TimeOutError - The operation timed out.
- IO_UnspecifiedError - An unspecified error happened on close.

See also resetStatus() [p. 83].

int QIODevice::ungetch (int ch) [virtual]

Puts the character *ch* back into the I/O device and decrements the index position if it is not zero.

This function is normally called to "undo" a getch() operation.

Returns *ch*, or -1 if an error occurred.

This virtual function must be reimplemented by all subclasses.

See also getch() [p. 80] and putch() [p. 83].

Reimplemented in QFile and QSocket.

Q_LONG QIODevice::writeBlock (const char * data, Q_ULONG len) [virtual]

Writes *len* bytes from *data* to the I/O device and returns the number of bytes actually written.

This virtual function must be reimplemented by all subclasses.

See also readBlock() [p. 83].

Reimplemented in QBuffer, QSocket and QSocketDevice.

Q_LONG QIODevice::writeBlock (const QByteArray & data)

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

This convenience function is the same as calling writeBlock(data.data(), data.size()).

QLocalFs Class Reference

The QLocalFs class is an implementation of a QNetworkProtocol that works on the local file system.

This class is part of the **network** module.

`#include <qlocalfs.h>`

Inherits QNetworkProtocol [p. 99].

Public Members

- **QLocalFs** ()

Detailed Description

The QLocalFs class is an implementation of a QNetworkProtocol that works on the local file system.

This class is derived from QNetworkProtocol. QLocalFs is not normally used directly, but rather through a QUrlOperator, for example:

```
QUrlOperator op( "file:///tmp" );
op.listChildren(); // Asks the server to provide a directory listing
```

This code will only work if the QLocalFs class is registered; to register the class, you must call qInitNetworkProtocols() before using a QUrlOperator with QLocalFs.

If you really need to use QLocalFs directly, don't forget to set its QUrlOperator with setUrl().

See also Qt Network Documentation [p. 4], QNetworkProtocol [p. 99], QUrlOperator [p. 179] and Input/Output and Networking.

Member Function Documentation

QLocalFs::QLocalFs ()

Constructor.

QLock Class Reference

The QLock class is a wrapper round a System V shared semaphore.

`#include <qlock_qws.h>`

Public Members

- **QLock** (const QString & filename, char id, bool create = FALSE)
- **~QLock** ()
- enum **Type** { Read, Write }
- bool **isValid** () const
- void **lock** (Type t)
- void **unlock** ()
- bool **locked** () const

Detailed Description

The QLock class is a wrapper round a System V shared semaphore.

It is used by Qt/Embedded for synchronizing access to the graphics card and shared memory region between processes.

See also Input/Output and Networking and Qt/Embedded.

Member Type Documentation

QLock::Type

- `QLock::Read`
- `QLock::Write`

Member Function Documentation

QLock::QLock (const QString & filename, char id, bool create = FALSE)

Creates a lock. *filename* is the file path of the Unix-domain socket the Qt/Embedded client is using. *id* is the name of the particular lock to be created on that socket. If *create* is TRUE the lock is to be created (as the Qt/Embedded server does); if *create* is FALSE the lock should exist already (as the Qt/Embedded client expects).

QLock::~QLock ()

Destroys a lock

bool QLock::isValid () const

Returns TRUE if the lock constructor was succesful; returns FALSE if the lock could not be created or was not available to connect to.

void QLock::lock (Type t)

Locks the semaphore with a lock of type *t*. Locks can either be Read or Write. If a lock is Read, attempts to lock by other processes as Read will succeed, Write attempts will block until the lock is unlocked. If locked as Write, all attempts to lock by other processes will block until the lock is unlocked. Locks are stacked: i.e. a given QLock can be locked multiple times by the same process without blocking, and will only be unlocked after a corresponding number of unlock() calls.

bool QLock::locked () const

Returns TRUE if the lock is currently held by the current process; otherwise returns FALSE.

void QLock::unlock ()

Unlocks the semaphore. If other processes were blocking waiting to lock() the semaphore, one of them will wake up and succeed in lock()ing.

QMimeSource Class Reference

The QMimeSource class is an abstraction of objects which provide formatted data of a certain MIME type.

`#include <qmime.h>`

Inherited by QDragObject [Events, Actions, Layouts and Styles with Qt] and QDropEvent [Events, Actions, Layouts and Styles with Qt].

Public Members

- **QMimeSource** ()
- virtual **~QMimeSource** ()
- virtual const char * **format** (int i = 0) const
- virtual bool **provides** (const char * mimeType) const
- virtual QByteArray **encodedData** (const char *) const
- int **serialNumber** () const

Detailed Description

The QMimeSource class is an abstraction of objects which provide formatted data of a certain MIME type.

Drag-and-drop and clipboard use this abstraction.

See also IANA list of MIME media types, Drag And Drop Classes, Input/Output and Networking and Miscellaneous Classes.

Member Function Documentation

QMimeSource::QMimeSource ()

Constructs a mime source and assigns a globally unique serial number to it.

See also serialNumber() [p. 90].

QMimeSource::~QMimeSource () [virtual]

Provided to ensure that subclasses destroy themselves correctly.

QByteArray QMimeSource::encodedData (const char *) const [virtual]

Returns the encoded data of this object in the specified MIME format.

Subclasses must reimplement this function.

Reimplemented in QStoredDrag, QDropEvent and QIconDrag.

const char * QMimeSource::format (int i = 0) const [virtual]

Returns the *i*-th supported MIME format, or 0.

Example: fileiconview/qfileiconview.cpp.

Reimplemented in QDropEvent.

bool QMimeSource::provides (const char * mimeType) const [virtual]

Returns TRUE if the object can provide the data in format *mimeType*; otherwise returns FALSE.

If you inherit from QMimeSource for consistency reasons it is better to implement the more abstract canDecode() functions such as QTextDrag::canDecode() and QImageDrag::canDecode().

Reimplemented in QDropEvent.

int QMimeSource::serialNumber () const

Returns the globally unique serial number of the mime source

QMimeSourceFactory Class Reference

The QMimeSourceFactory class is an extensible provider of mime-typed data.

`#include <qmime.h>`

Public Members

- **QMimeSourceFactory** ()
- virtual **~QMimeSourceFactory** ()
- virtual const QMimeSource * **data** (const QString & abs_name) const
- virtual QString **makeAbsolute** (const QString & abs_or_rel_name, const QString & context) const
- const QMimeSource * **data** (const QString & abs_or_rel_name, const QString & context) const
- virtual void **setText** (const QString & abs_name, const QString & text)
- virtual void **setImage** (const QString & abs_name, const QImage & image)
- virtual void **setPixmap** (const QString & abs_name, const QPixmap & pixmap)
- virtual void **setData** (const QString & abs_name, QMimeSource * data)
- virtual void **setFilePath** (const QStringList & path)
- virtual QStringList **filePath** () const
- void **addFilePath** (const QString & p)
- virtual void **setExtensionType** (const QString & ext, const char * mimetype)

Static Public Members

- QMimeSourceFactory * **defaultFactory** ()
- void **setDefaultFactory** (QMimeSourceFactory * factory)
- QMimeSourceFactory * **takeDefaultFactory** ()
- void **addFactory** (QMimeSourceFactory * f)
- void **removeFactory** (QMimeSourceFactory * f)

Detailed Description

The QMimeSourceFactory class is an extensible provider of mime-typed data.

A QMimeSourceFactory provides an abstract interface to a collection of information. Each piece of information is represented by a QMimeSource object which can be examined and converted to concrete data types by functions such as QImageDrag::canDecode() and QImageDrag::decode().

The base QMimeSourceFactory can be used in two ways: as an abstraction of a collection of files or as specifically stored data. For it to access files, call setFilePath() before accessing data. For stored data, call setData() for each item (there are also convenience functions setText(), setImage(), and setPixmap() that simply call setData() with massaged parameters).

The rich text widgets QTextEdit and QTextBrowser use QMimeSourceFactory to resolve references such as images or links within rich text documents. They either access the default factory (see defaultFactory()) or their own (see QTextEdit::setMimeSourceFactory()). Other classes that are capable of displaying rich text (such as QLabel, QWhatsThis or QMessageBox) always use the default factory.

As mentioned earlier, a factory can also be used as container to store data associated with a name. This technique is useful whenever rich text contains images that are stored in the program itself, not loaded from the hard disk. Your program may, for example, define some image data as

```
static const char* myimage_data[]={
"...",
...
"..."};
```

To be able to use this image within some rich text, for example inside a QLabel, you have to create a QImage from the raw data and insert it into the factory with a unique name:

```
QMimeSourceFactory::defaultFactory()->setImage( "myimage", QImage(myimage_data) );
```

Now you can create a rich text QLabel with

```
QLabel* label = new QLabel(
    "Rich text with embedded image:"
    "Isn't that cute?" );
```

See also Environment Classes and Input/Output and Networking.

Member Function Documentation

QMimeSourceFactory::QMimeSourceFactory ()

Constructs a QMimeSourceFactory that has no file path and no stored content.

QMimeSourceFactory::~QMimeSourceFactory () [virtual]

Destroys the QMimeSourceFactory, deleting all stored content.

void QMimeSourceFactory::addFactory (QMimeSourceFactory * f) [static]

Adds the QMimeSourceFactory *f* to the list of available mimesource factories. If the defaultFactory() can't resolve a data() it iterates over the list of installed mimesource factories until the data can be resolved.

See also removeFactory() [p. 94].

void QMimeSourceFactory::addFilePath (const QString & p)

Adds another search path, *p* to the existing search paths.

See also setFilePath() [p. 94].

const QMimeSource * QMimeSourceFactory::data (const QString & abs_name) const [virtual]

Returns a reference to the data associated with *abs_name*. The return value remains valid only until the next data() or setData() call, so you should immediately decode the result.

If there is no data associated with *abs_name* in the factory's store, the factory tries to access the local filesystem. If *abs_name* isn't an absolute file name, the factory will search for it on all defined paths (see setFilePath()).

The factory understands all image formats supported by QImageIO. Any other mime types are determined by the file name extension. The default settings are

```
setExtensionType("html", "text/html;charset=iso8859-1");
setExtensionType("htm", "text/html;charset=iso8859-1");
setExtensionType("txt", "text/plain");
setExtensionType("xml", "text/xml;charset=UTF-8");
```

The effect of these is that file names ending in "html" or "htm" will be treated as text encoded in the iso8859-1 encoding, those ending in "txt" will be treated as text encoded in the local encoding; those ending in "xml" will be treated as text encoded in the UTF8 encoding. The text subtype ("html", "plain", or "xml") does not affect the factory, but users of the factory may behave differently. We recommend creating "xml" files where practical. These files can be viewed regardless of the runtime encoding and can encode any Unicode characters without resorting to encoding definitions inside the file.

Any file data that is not recognized will be retrieved as a QMimeSource providing the "application/octet-stream" mime type, meaning uninterpreted binary data.

You can add further extensions or change existing ones with subsequent calls to setExtensionType(). If the extension mechanism is not sufficient for your problem domain, you may inherit QMimeSourceFactory and reimplement this function to perform some more clever mime-type detection. The same applies if you want to use the mime source factory to access URL referenced data over a network.

const QMimeSource * QMimeSourceFactory::data (const QString & abs_or_rel_name, const QString & context) const

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

A convenience function. See data(const QString& abs_name). The file name is given in *abs_or_rel_name* and the path is in *context*.

QMimeSourceFactory * QMimeSourceFactory::defaultFactory () [static]

Returns the application-wide default mime source factory. This factory is used by rich text rendering classes such as QSimpleRichText, QWhatsThis and QMessageBox to resolve named references within rich text documents. It serves also as initial factory for the more complex render widgets QTextEdit and QTextBrowser.

See also setDefaultFactory() [p. 94].

Examples: action/application.cpp and application/application.cpp.

QStringList QMimeSourceFactory::filePath () const [virtual]

Returns the currently set search paths.

QString QMimeSourceFactory::makeAbsolute (const QString & abs_or_rel_name, const QString & context) const [virtual]

Converts the absolute or relative data item name *abs_or_rel_name* to an absolute name, interpreted within the context (path) of the data item named *context* (this must be an absolute name).

void QMimeSourceFactory::removeFactory (QMimeSourceFactory * f) [static]

Removes the mimesource factory *f* from the list of available mimesource factories.

See also addFactory() [p. 92].

void QMimeSourceFactory::setData (const QString & abs_name, QMimeSource * data) [virtual]

Sets *data* to be the data item associated with the absolute name *abs_name*. Note that the ownership of *data* is transferred to the factory: do not delete or access the pointer after passing it to this function.

void QMimeSourceFactory::setDefaultFactory (QMimeSourceFactory * factory) [static]

Sets the default *factory*, destroying any previously set mime source provider. The ownership of the factory is transferred.

See also defaultFactory() [p. 93].

void QMimeSourceFactory::setExtensionType (const QString & ext, const char * mimetype) [virtual]

Sets the mime-type to be associated with the file name extension, *ext* to *mimetype*. This determines the mime-type for files found via the paths set by setFilePath().

void QMimeSourceFactory::setFilePath (const QStringList & path) [virtual]

Sets the list of directories that will be searched when named data is requested to the those given in the string list *path*.

See also filePath() [p. 94].

void QMimeSourceFactory::setImage (const QString & abs_name, const QImage & image) [virtual]

Sets *image* to be the data item associated with the absolute name *abs_name*.

Equivalent to setData(abs_name, new QImageDrag(image)).

void QMimeSourceFactory::setPixmap (const QString & abs_name, const QPixmap & pixmap) [virtual]

Sets *pixmap* to be the data item associated with the absolute name *abs_name*.

void QMimeSourceFactory::setText (const QString & abs_name, const QString & text) [virtual]

Sets *text* to be the data item associated with the absolute name *abs_name*.

Equivalent to setData(abs_name, new QTextDrag(text)).

QMimeSourceFactory * QMimeSourceFactory::takeDefaultFactory () [static]

Sets the defaultFactory() to 0 and returns the previous one.

QNetworkOperation Class Reference

The QNetworkOperation class provides common operations for network protocols.

`#include <qnetworkprotocol.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QNetworkOperation** (QNetworkProtocol::Operation operation, const QString & arg0, const QString & arg1, const QString & arg2)
- **QNetworkOperation** (QNetworkProtocol::Operation operation, const QByteArray & arg0, const QByteArray & arg1, const QByteArray & arg2)
- **~QNetworkOperation** ()
- void **setState** (QNetworkProtocol::State state)
- void **setProtocolDetail** (const QString & detail)
- void **setErrorCode** (int ec)
- void **setArg** (int num, const QString & arg)
- void **setRawArg** (int num, const QByteArray & arg)
- QNetworkProtocol::Operation **operation** () const
- QNetworkProtocol::State **state** () const
- QString **arg** (int num) const
- QByteArray **rawArg** (int num) const
- QString **protocolDetail** () const
- int **errorCode** () const
- void **free** ()

Detailed Description

The QNetworkOperation class provides common operations for network protocols.

An object is created to describe the operation and the current state for each operation that a network protocol should process.

For a detailed description of the Qt Network Architecture how to implement and use network protocols in Qt, see the Qt Network Documentation.

See also QNetworkProtocol [p. 99] and Input/Output and Networking.

Member Function Documentation

QNetworkOperation::QNetworkOperation (QNetworkProtocol::Operation operation, const QString & arg0, const QString & arg1, const QString & arg2)

Constructs a network operation object. *operation* is the type of the operation, and *arg0*, *arg1* and *arg2* are the first three arguments of the operation. The state is initialized to QNetworkProtocol::StWaiting.

See also QNetworkProtocol::Operation [p. 101] and QNetworkProtocol::State [p. 102].

QNetworkOperation::QNetworkOperation (QNetworkProtocol::Operation operation, const QByteArray & arg0, const QByteArray & arg1, const QByteArray & arg2)

Constructs a network operation object. *operation* is the type of the operation, and *arg0*, *arg1* and *arg2* are the first three raw data arguments of the operation. The state is initialized to QNetworkProtocol::StWaiting.

See also QNetworkProtocol::Operation [p. 101] and QNetworkProtocol::State [p. 102].

QNetworkOperation::~QNetworkOperation ()

Destructor.

QString QNetworkOperation::arg (int num) const

Returns the argument *num* of the operation. If this argument was not already set, an empty string is returned.

Examples: network/ftpclient/ftpmainwindow.cpp and network/networkprotocol/nntp.cpp.

int QNetworkOperation::errorCode () const

Returns the error code for the last error that occurred.

Example: network/ftpclient/ftpmainwindow.cpp.

void QNetworkOperation::free ()

Sets this object to delete itself when it hasn't been used for one second.

Because QNetworkOperation pointers are passed around a lot the QNetworkProtocol generally does not have enough knowledge to delete these at the correct time. If a QNetworkProtocol doesn't need an operation any more it will call this function instead.

Note: you should never need to call the method yourself.

QNetworkProtocol::Operation QNetworkOperation::operation () const

Returns the type of the operation.

Example: network/ftpclient/ftpmainwindow.cpp.

QString QNetworkOperation::protocolDetail () const

Returns a detailed error message for the last error. This must have been set using setProtocolDetail().

Example: network/ftpclient/ftpmainwindow.cpp.

QByteArray QNetworkOperation::rawArg (int num) const

Returns the raw data argument *num* of the operation. If this argument was not already set, an empty bytearray is returned.

void QNetworkOperation::setArg (int num, const QString & arg)

Sets the argument *num* of the network operation to *arg*.

void QNetworkOperation::setErrorCode (int ec)

Sets the error code to *ec*.

If the operation failed, the protocol should set an error code to describe the error in more detail. If possible, one of the error codes defined in QNetworkProtocol should be used.

See also setProtocolDetail() [p. 98] and QNetworkProtocol::Error [p. 101].

void QNetworkOperation::setProtocolDetail (const QString & detail)

If the operation failed, the error message can be specified as *detail*.

void QNetworkOperation::setRawArg (int num, const QByteArray & arg)

Sets the raw data argument *num* of the network operation to *arg*.

void QNetworkOperation::setState (QNetworkProtocol::State state)

Sets the *state* of the operation object. This should be done by the network protocol during processing; at the end it should be set to QNetworkProtocol::StDone or QNetworkProtocol::StFailed, depending on success or failure.

See also QNetworkProtocol::State [p. 102].

QNetworkProtocol::State QNetworkOperation::state () const

Returns the state of the operation. You can determine whether an operation is still waiting to be processed, is being processed, has been processed successfully, or failed.

Example: network/ftpclient/ftpmainwindow.cpp.

QNetworkProtocol Class Reference

The QNetworkProtocol class provides a common API for network protocols.

`#include <qnetworkprotocol.h>`

Inherits QObject [Additional Functionality with Qt].

Inherited by QFtp [p. 69], QHttp [p. 74] and QLocalFs [p. 86].

Public Members

- enum **State** { StWaiting = 0, StInProgress, StDone, StFailed, StStopped }
- enum **Operation** { OpListChildren = 1, OpMkDir = 2, OpMkdir = OpMkDir, OpRemove = 4, OpRename = 8, OpGet = 32, OpPut = 64 }
- enum **ConnectionState** { ConHostFound, ConConnected, ConClosed }
- enum **Error** { NoError = 0, ErrValid, ErrUnknownProtocol, ErrUnsupported, ErrParse, ErrLoginIncorrect, ErrHostNotFound, ErrListChildren, ErrListChlidren = ErrListChildren, ErrMkDir, ErrMkdir = ErrMkDir, ErrRemove, ErrRename, ErrGet, ErrPut, ErrFileNotExisting, ErrPermissionDenied }
- **QNetworkProtocol** ()
- virtual **~QNetworkProtocol** ()
- virtual void **setUrl** (QUrlOperator * u)
- virtual void **setAutoDelete** (bool b, int i = 10000)
- bool **autoDelete** () const
- virtual int **supportedOperations** () const
- virtual void **addOperation** (QNetworkOperation * op)
- QUrlOperator * **url** () const
- QNetworkOperation * **operationInProgress** () const
- virtual void **clearOperationQueue** ()
- virtual void **stop** ()

Signals

- void **data** (const QByteArray & data, QNetworkOperation * op)
- void **connectionStateChanged** (int state, const QString & data)
- void **finished** (QNetworkOperation * op)
- void **start** (QNetworkOperation * op)
- void **newChildren** (const QValueList<QUrlInfo> & i, QNetworkOperation * op)

- void **newChild** (const QUrlInfo & i, QNetworkOperation * op)
- void **createdDirectory** (const QUrlInfo & i, QNetworkOperation * op)
- void **removed** (QNetworkOperation * op)
- void **itemChanged** (QNetworkOperation * op)
- void **dataTransferProgress** (int bytesDone, int bytesTotal, QNetworkOperation * op)

Static Public Members

- void **registerNetworkProtocol** (const QString & protocol, QNetworkProtocolFactoryBase * protocolFactory)
- QNetworkProtocol * **getNetworkProtocol** (const QString & protocol)
- bool **hasOnlyLocalFileSystem** ()

Protected Members

- virtual void **operationListChildren** (QNetworkOperation * op)
- virtual void **operationMkDir** (QNetworkOperation * op)
- virtual void **operationRemove** (QNetworkOperation * op)
- virtual void **operationRename** (QNetworkOperation * op)
- virtual void **operationGet** (QNetworkOperation * op)
- virtual void **operationPut** (QNetworkOperation * op)
- virtual bool **checkConnection** (QNetworkOperation * op)

Detailed Description

The QNetworkProtocol class provides a common API for network protocols.

This is a base class which should be used for implementations of network protocols that can then be used in Qt (e.g. in the file dialog) together with the QUrlOperator.

The easiest way to implement a new network protocol is to reimplement the operation...() methods, e.g. operationGet(). Only the supported operations should be reimplemented. To specify which operations are supported, also reimplement supportedOperations() and return an int that is OR'd together using the supported operations from the QNetworkProtocol::Operation enum.

When you implement a network protocol this way, be careful to always emit the correct signals. Also, always emit the finished() signal when an operation is done (on success and on failure). Qt relies on correctly emitted finished() signals.

For a detailed description of the Qt Network Architecture and how to implement and use network protocols in Qt, see the Qt Network Documentation.

See also Input/Output and Networking.

Member Type Documentation

QNetworkProtocol::ConnectionState

When the connection state of a network protocol changes it emits the signal connectionStateChanged(). The first argument is one of the following values:

- `QNetworkProtocol::ConHostFound` - Host has been found.
- `QNetworkProtocol::ConConnected` - Connection to the host has been established.
- `QNetworkProtocol::ConClosed` - Connection has been closed.

QNetworkProtocol::Error

When an operation fails (finishes unsuccessfully), the QNetworkOperation of the operation returns an error code which has one of the following values:

- `QNetworkProtocol::NoError` - No error occurred.
- `QNetworkProtocol::ErrValid` - The URL you are operating on is not valid.
- `QNetworkProtocol::ErrUnknownProtocol` - There is no protocol implementation available for the protocol of the URL you are operating on (e.g. if the protocol is http and no http implementation has been registered).
- `QNetworkProtocol::ErrUnsupported` - The operation is not supported by the protocol.
- `QNetworkProtocol::ErrParse` - Parse error of the URL.
- `QNetworkProtocol::ErrLoginIncorrect` - You needed to login but the username and/or password are wrong.
- `QNetworkProtocol::ErrHostNotFound` - The specified host (in the URL) couldn't be found.
- `QNetworkProtocol::ErrListChildren` - An error occurred while listing the children (files).
- `QNetworkProtocol::ErrMkDir` - An error occurred when creating a directory.
- `QNetworkProtocol::ErrRemove` - An error occurred when removing a child (file).
- `QNetworkProtocol::ErrRename` - An error occurred when renaming a child (file).
- `QNetworkProtocol::ErrGet` - An error occurred while getting (retrieving) data.
- `QNetworkProtocol::ErrPut` - An error occurred while putting (uploading) data.
- `QNetworkProtocol::ErrFileNotExisting` - A file which is needed by the operation doesn't exist.
- `QNetworkProtocol::ErrPermissionDenied` - Permission for doing the operation has been denied.

You should also use these error codes when implementing custom network protocols. If this is not possible, you can define your own error codes by using integer values that don't conflict with any of these values.

QNetworkProtocol::Operation

This enum lists the possible operations that a network protocol can support. supportedOperations() returns an int of these that is OR'd together. Also, the type() or a QNetworkOperation is always one of these values.

- `QNetworkProtocol::OpListChildren` - List the children of a URL, e.g. of a directory.
- `QNetworkProtocol::OpMkDir` - Create a directory.
- `QNetworkProtocol::OpRemove` - Remove a child (e.g. a file).

- `QNetworkProtocol::OpRename` - Rename a child (e.g. a file).
- `QNetworkProtocol::OpGet` - Get data from a location.
- `QNetworkProtocol::OpPut` - Put data to a location.

QNetworkProtocol::State

This enum contains the state that a QNetworkOperation can have:

- `QNetworkProtocol::StWaiting` - The operation is in the queue of the QNetworkProtocol and is waiting to be prcessed.
- `QNetworkProtocol::StInProgress` - The operation is being processed.
- `QNetworkProtocol::StDone` - The operation has been processed succesfully.
- `QNetworkProtocol::StFailed` - The operation has been processed but an error occurred.
- `QNetworkProtocol::StStopped` - The operation has been processed but has been stopped before it finished, and is waiting to be processed.

Member Function Documentation

QNetworkProtocol::QNetworkProtocol ()

Constructor of the network protocol base class. Does some initialization and connecting of signals and slots.

QNetworkProtocol::~QNetworkProtocol () [virtual]

Destructor.

void QNetworkProtocol::addOperation (QNetworkOperation * op) [virtual]

Adds the operation *op* to the operation queue. The operation will be processed as soon as possible. This method returns immediately.

bool QNetworkProtocol::autoDelete () const

Returns TRUE if auto-deleting is enabled; otherwise returns FALSE.

See also QNetworkProtocol::setAutoDelete() [p. 107].

bool QNetworkProtocol::checkConnection (QNetworkOperation * op) [virtual protected]

For processing operations the network protocol base class calls this method quite often. This should be reimplemented by new network protocols. It should return TRUE if the connection is OK (open); otherwise it should return FALSE. If the connection is not open the protocol should open it.

If the connection can't be opened (e.g. because you already tried but the host couldn't be found), set the state of *op* to QNetworkProtocol::StFailed and emit the finished() signal with this QNetworkOperation as argument.

op is the operation that needs an open connection.

Example: network/networkprotocol/nntp.cpp.

void QNetworkProtocol::clearOperationQueue () [virtual]

Clears the operation queue.

void QNetworkProtocol::connectionStateChanged (int state, const QString & data) [signal]

This signal is emitted whenever the state of the connection of the network protocol is changed. *state* describes the new state, which is one of, ConHostFound, ConConnected or ConClosed. *data* is a message text.

void QNetworkProtocol::createdDirectory (const QUrlInfo & i, QNetworkOperation * op) [signal]

This signal is emitted when mkdir() has been succesful and the directory has been created. *i* holds the information about the new directory. *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc. Using op->arg(0), you can get the file name of the new directory.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

void QNetworkProtocol::data (const QByteArray & data, QNetworkOperation * op) [signal]

This signal is emitted when new *data* has been received after calling get() or put(). *op* holds the name of the file from which data is retrieved or uploaded in its first argument, and the (raw) data in its second argument. You can get them with op->arg(0) and op->rawArg(1). *op* is the pointer to the operation object, which contains all the information about the operation, including the state, etc.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator (which is used by the network protocol) emit its corresponding signal.

void QNetworkProtocol::dataTransferProgress (int bytesDone, int bytesTotal, QNetworkOperation * op) [signal]

This signal is emitted during the transfer of data (using put() or get()). *bytesDone* is how many bytes of *bytesTotal* have been transferred. *bytesTotal* may be -1, which means that the total number of bytes is not known. *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

void QNetworkProtocol::finished (QNetworkOperation * op) [signal]

This signal is emitted when an operation finishes. This signal is always emitted, for both success and failure. *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc. Check the state and error code of the operation object to determine whether or not the operation was successful.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

QNetworkProtocol * QNetworkProtocol::getNetworkProtocol (const QString & protocol) [static]

Static method to get a new instance of the network protocol *protocol*. For example, if you need to do some FTP operations, do the following:

```
QFtp *ftp = QNetworkProtocol::getNetworkProtocol( "ftp" );
```

This returns a pointer to a new instance of an ftp implementation or null if no protocol for ftp was registered. The ownership of the pointer is transferred to you, so you must delete it if you don't need it anymore.

Normally you should not work directly with network protocols, so you will not need to call this method yourself. Instead, use QUrlOperator, which makes working with network protocols much more convenient.

See also QUrlOperator [p. 179].

bool QNetworkProtocol::hasOnlyLocalFileSystem () [static]

Returns TRUE if the only protocol registered is for working on the local filesystem; returns FALSE if other network protocols are also registered.

void QNetworkProtocol::itemChanged (QNetworkOperation * op) [signal]

This signal is emitted whenever a file which is a child of this URL has been changed, e.g. by successfully calling rename(). *op* holds the original and the new file names in the first and second arguments, accessible with op->arg(0) and op->arg(1) respectively. *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

void QNetworkProtocol::newChild (const QUrlInfo & i, QNetworkOperation * op) [signal]

This signal is emitted if a new child (file) has been read. QNetworkProtocol automatically connects it to a slot which creates a list of QUrlInfo objects (with just one QUrlInfo *i*) and emits the newChildren() signal with this list. *op* is the pointer to the operation object which contains all the information about the operation that has finished, including the state, etc.

This is just a convenience signal useful for implementing your own network protocol. In all other cases connect to the newChildren() signal with its list of QUrlInfo objects.

void QNetworkProtocol::newChildren (const QValueList<QUrlInfo> & i, QNetworkOperation * op) [signal]

This signal is emitted after listChildren() was called and new children (files) have been read from the list of files. *i* holds the information about the new children. *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

When implementing your own network protocol and reading children, you usually don't read one child at once, but rather a list of them. That's why this signal takes a list of QUrlInfo objects. If you prefer to read only one child at a time you can use the convenience signal newChild(), which takes a single QUrlInfo object.

void QNetworkProtocol::operationGet (QNetworkOperation * op) [virtual protected]

When implementing a new network protocol, this method should be reimplemented if the protocol supports getting data; this method should then process the QNetworkOperation.

When you reimplement this method it's very important that you emit the correct signals at the correct time (especially the finished() signal after processing an operation). Take a look at the Qt Network Documentation which describes in detail how to reimplement this method. You may also want to look at the example implementation in examples/network/networkprotocol/nntp.cpp.

op is the pointer to the operation object which contains all the information on the operation that has finished, including the state, etc.

Example: network/networkprotocol/nntp.cpp.

QNetworkOperation * QNetworkProtocol::operationInProgress () const

Returns the operation, which is being processed, or 0 of no operation is being processed at the moment.

void QNetworkProtocol::operationListChildren (QNetworkOperation * op) [virtual protected]

When implementing a new network protocol, this method should be reimplemented if the protocol supports listing children (files); this method should then process this QNetworkOperation.

When you reimplement this method it's very important that you emit the correct signals at the correct time (especially the finished() signal after processing an operation). Take a look at the Qt Network Documentation which describes in detail how to reimplement this method. You may also want to look at the example implementation in examples/network/networkprotocol/nntp.cpp.

op is the pointer to the operation object which contains all the information on the operation that has finished, including the state, etc.

Example: network/networkprotocol/nntp.cpp.

void QNetworkProtocol::operationMkDir (QNetworkOperation * op) [virtual protected]

When implementing a new network protocol, this method should be reimplemented if the protocol supports making directories; this method should then process this QNetworkOperation.

When you reimplement this method it's very important that you emit the correct signals at the correct time (especially the finished() signal after processing an operation). Take a look at the Qt Network Documentation which describes in detail how to reimplement this method. You may also want to look at the example implementation in examples/network/networkprotocol/nntp.cpp.

op is the pointer to the operation object which contains all the information on the operation that has finished, including the state, etc.

void QNetworkProtocol::operationPut (QNetworkOperation * op) [virtual protected]

When implementing a new network protocol, this method should be reimplemented if the protocol supports putting (uploading) data; this method should then process the QNetworkOperation.

When you reimplement this method it's very important that you emit the correct signals at the correct time (especially the finished() signal after processing an operation). Take a look at the Qt Network Documentation which describes in detail how to reimplement this method. You may also want to look at the example implementation in examples/network/networkprotocol/nntp.cpp.

op is the pointer to the operation object which contains all the information on the operation that has finished, including the state, etc.

void QNetworkProtocol::operationRemove (QNetworkOperation * op) [virtual protected]

When implementing a new network protocol, this method should be reimplemented if the protocol supports removing children (files); this method should then process this QNetworkOperation.

When you reimplement this method it's very important that you emit the correct signals at the correct time (especially the finished() signal after processing an operation). Take a look at the Qt Network Documentation which is describes in detail how to reimplement this method. You may also want to look at the example implementation in examples/network/networkprotocol/nntp.cpp.

op is the pointer to the operation object which contains all the information on the operation that has finished, including the state, etc.

void QNetworkProtocol::operationRename (QNetworkOperation * op) [virtual protected]

When implementing a new newtork protocol, this method should be reimplemented if the protocol supports renaming children (files); this method should then process this QNetworkOperation.

When you reimplement this method it's very important that you emit the correct signals at the correct time (especially the finished() signal after processing an operation). Take a look at the Qt Network Documentation which describes in detail how to reimplement this method. You may also want to look at the example implementation in examples/network/networkprotocol/nntp.cpp.

op is the pointer to the operation object which contains all the information on the operation that has finished, including the state, etc.

void QNetworkProtocol::registerNetworkProtocol (const QString & protocol, QNetworkProtocolFactoryBase * protocolFactory) [static]

Static method to register a network protocol for Qt. For example, if you have an implementation of NNTP (called Nntp) which is derived from QNetworkProtocol, call:

```
QNetworkProtocol::registerNetworkProtocol( "nntp", new QNetworkProtocolFactory );
```

after which your implementation is registered for future nntp operations.

The name of the protocol is given in *protocol* and a pointer to the protocol factory is given in *protocolFactory*.

void QNetworkProtocol::removed (QNetworkOperation * op) [signal]

This signal is emitted when remove() has been succesful and the file has been removed. *op* holds the file name of the removed file in the first argument, accessible with op->arg(0). *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

void QNetworkProtocol::setAutoDelete (bool b, int i = 10000) [virtual]

Because it's sometimes hard to take care of removing network protocol instances, QNetworkProtocol provides an auto-delete mechanism. If you set *b* to TRUE, the network protocol instance is removed after it has been inactive for *i* milliseconds (i.e. *i* milliseconds after the last operation has been processed). If you set *b* to FALSE the auto-delete mechanism is switched off.

If you switch on auto-delete, the QNetworkProtocol also deletes its QUrlOperator.

void QNetworkProtocol::setUrl (QUrlOperator * u) [virtual]

Sets the QUrlOperator, on which the protocol works to *u*.

See also QUrlOperator [p. 179].

void QNetworkProtocol::start (QNetworkOperation * op) [signal]

Some operations (such as listChildren()) emit this signal when they start processing the operation. *op* is the pointer to the operation object which contains all the information about the operation, including the state, etc.

When a protocol emits this signal, QNetworkProtocol is smart enough to let the QUrlOperator, which is used by the network protocol, emit its corresponding signal.

void QNetworkProtocol::stop () [virtual]

Stops the current operation that is being processed and clears all waiting operations.

int QNetworkProtocol::supportedOperations () const [virtual]

Returns an int that is OR'd together using the enum values of QNetworkProtocol::Operation, which describes which operations are supported by the network protocol. Should be reimplemented by new network protocols.

Example: network/networkprotocol/nntp.cpp.

QUrlOperator * QNetworkProtocol::url () const

Returns the QUrlOperator on which the protocol works.

QProcess Class Reference

The QProcess class is used to start external programs and to communicate with them.

`#include <qprocess.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QProcess** (QObject * parent = 0, const char * name = 0)
- **QProcess** (const QString & arg0, QObject * parent = 0, const char * name = 0)
- **QProcess** (const QStringList & args, QObject * parent = 0, const char * name = 0)
- **~QProcess** ()
- QStringList **arguments** () const
- void **clearArguments** ()
- virtual void **setArguments** (const QStringList & args)
- virtual void **addArgument** (const QString & arg)
- QDir **workingDirectory** () const
- virtual void **setWorkingDirectory** (const QDir & dir)
- enum **Communication** { Stdin = 0x01, Stdout = 0x02, Stderr = 0x04, DupStderr = 0x08 }
- void **setCommunication** (int commFlags)
- int **communication** () const
- virtual bool **start** (QStringList * env = 0)
- virtual bool **launch** (const QString & buf, QStringList * env = 0)
- virtual bool **launch** (const QByteArray & buf, QStringList * env = 0)
- bool **isRunning** () const
- bool **normalExit** () const
- int **exitStatus** () const
- virtual QByteArray **readStdout** ()
- virtual QByteArray **readStderr** ()
- bool **canReadLineStdout** () const
- bool **canReadLineStderr** () const
- virtual QString **readLineStdout** ()
- virtual QString **readLineStderr** ()
- PID **processIdentifier** ()

Public Slots

- void **tryTerminate** () const
- void **kill** () const
- virtual void **writeToStdin** (const QByteArray & buf)
- virtual void **writeToStdin** (const QString & buf)
- virtual void **closeStdin** ()

Signals

- void **readyReadStdout** ()
- void **readyReadStderr** ()
- void **processExited** ()
- void **wroteToStdin** ()
- void **launchFinished** ()

Detailed Description

The QProcess class is used to start external programs and to communicate with them.

You can write to the started program's standard input, and can read the program's standard output and standard error. You can pass command line arguments to the program either in the constructor or with setArguments() or addArgument(). The program's working directory can be set with setWorkingDirectory(). If you need to set up environment variables pass them to the start() or launch() function (see below). The processExited() signal is emitted if the program exits. The program's exit status is available from exitStatus(), although you could simply call normalExit() to see if the program terminated normally.

There are two different ways to start a process. If you just want to run a program, optionally passing data to its standard input at the beginning, use one of the launch() functions. If you want full control of the program's standard input, standard output and standard error, use the start() function.

If you use start() you can write to the program's standard input using writeToStdin() and you can close the standard input with closeStdin(). The wroteToStdin() signal is emitted if the data sent to standard input has been written. You can read from the program's standard output using readStdout() or readLineStdout(). These functions return an empty QByteArray if there is no data to read. The readyReadStdout() signal is emitted when there is data available to be read from standard output. Standard error has a set of functions that correspond to the standard output functions, i.e. readStderr(), readLineStderr() and readyReadStderr().

If you use one of the launch() functions the data you pass will be sent to the program's standard input which will be closed once all the data has been written. You should *not* use writeToStdin() or closeStdin() if you use launch(). If you need to send data to the program's standard input after it has started running use start() instead of launch().

Both start() and launch() can accept a string list of strings with the format, key=value, where the keys are the names of environment variables.

You can test to see if a program is running with isRunning(). The program's process identifier is available from processIdentifier(). If you want to terminate a running program use tryTerminate(), but note that the program may ignore this. If you *really* want to terminate the program, without it having any chance to clean up, you can use kill().

As an example, suppose we want to start the `uic` command (a Qt command line tool used with *Qt Designer*) and perform some operations on the output (the `uic` outputs the code it generates to standard output by default). Suppose

further that we want to run the program on the file "small_dialog.ui" with the command line options "-tr i18n". On the command line we would write:

```
uic -tr i18n small_dialog.ui
```

A code snippet for this with the QProcess class might look like this:

```
UicManager::UicManager()
{

    proc = new QProcess( this );

    proc->addArgument( "uic" );
    proc->addArgument( "-tr" );
    proc->addArgument( "i18n" );
    proc->addArgument( "small_dialog.ui" );

    connect( proc, SIGNAL(readyReadStdout()),
             this, SLOT(readFromStdout()) );

    if ( !proc->start() ) {
        // error handling

    }
}

void UicManager::readFromStdout()
{
    // Read and process the data.
    // Bear in mind that the data might be output in chunks.

}
```

Although you may need quotes for a file named on the command line (e.g. if it contains spaces) you shouldn't use extra quotes for arguments passed to addArgument() or setArguments().

The readyReadStdout() signal is emitted when there is new data on standard output. This happens asynchronously: you don't know if more data will arrive later. In the above example you could connect the processExited() signal to the slot UicManager::readFromStdout() instead. If you do so, you will be certain that all the data is available when the slot is called. On the other hand, you must wait until the process has finished before doing any processing.

See also QSocket [p. 126], Input/Output and Networking and Miscellaneous Classes.

Member Type Documentation

QProcess::Communication

This enum type defines the communication channels connected to the process.

- `QProcess::Stdin` - Data can be written to the process's standard input.

- `QProcess::Stdout` - Data can be read from the process's standard output.
- `QProcess::Stderr` - Data can be read from the process's standard error.
- `QProcess::DupStderr` - Duplicates standard error to standard output for new processes; i.e. everything that the process writes to standard error, is reported by QProcess on standard output instead. This is especially useful if your application requires that the output on standard output and standard error is read in the same order as the process output it. Please note that this is a binary flag, so if you want to activate this together with standard input, output and error redirection (the default), you have to specify `Stdin|Stdout|Stderr|DupStderr` for the setCommunication() call.

See also setCommunication() [p. 116] and communication() [p. 112].

Member Function Documentation

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

Constructs a QProcess object. The *parent* and *name* parameters are passed to the QObject constructor.

See also setArguments() [p. 116], addArgument() [p. 112] and start() [p. 116].

QProcess::QProcess (const QString & arg0, QObject * parent = 0, const char * name = 0)

Constructs a QProcess with *arg0* as the command to be executed. The *parent* and *name* parameters are passed to the QObject constructor.

The process is not started. You must call start() or launch() to start the process.

See also setArguments() [p. 116], addArgument() [p. 112] and start() [p. 116].

QProcess::QProcess (const QStringList & args, QObject * parent = 0, const char * name = 0)

Constructs a QProcess with *args* as the arguments of the process. The first element in the list is the command to be executed. The other elements in the list are the arguments to this command. The *parent* and *name* parameters are passed to the QObject constructor.

The process is not started. You must call start() or launch() to start the process.

See also setArguments() [p. 116], addArgument() [p. 112] and start() [p. 116].

QProcess::~QProcess ()

Destroys the class.

If the process is running, it is NOT terminated! Standard input, standard output and standard error of the process are closed.

You can connect the destroyed() signal to the kill() slot, if you want the process to be terminated automatically when the class is destroyed.

See also tryTerminate() [p. 117] and kill() [p. 113].

void QProcess::addArgument (const QString & arg) [virtual]

Adds *arg* to the end of the list of arguments.

The first element in the list of arguments is the command to be executed; the following elements are the arguments to the command.

See also arguments() [p. 112] and setArguments() [p. 116].

Example: process/process.cpp.

QStringList QProcess::arguments () const

Returns the list of arguments that are set for the process. Arguments can be specified with the constructor or with the functions setArguments() and addArgument().

See also setArguments() [p. 116] and addArgument() [p. 112].

bool QProcess::canReadLineStderr () const

Returns TRUE if it's possible to read an entire line of text from standard error at this time; otherwise returns FALSE.

See also readLineStderr() [p. 114] and canReadLineStdout() [p. 112].

bool QProcess::canReadLineStdout () const

Returns TRUE if it's possible to read an entire line of text from standard output at this time; otherwise returns FALSE.

See also readLineStdout() [p. 115] and canReadLineStderr() [p. 112].

void QProcess::clearArguments ()

Clears the list of arguments that are set for the process.

See also setArguments() [p. 116] and addArgument() [p. 112].

void QProcess::closeStdin () [virtual slot]

Closes standard input of the process.

This function also deletes pending data that is not written to standard input yet.

See also wroteToStdin() [p. 117].

int QProcess::communication () const

Returns the communication required with the process.

See also setCommunication() [p. 116].

int QProcess::exitStatus () const

Returns the exit status of the process or 0 if the process is still running. This function returns immediately and does not wait until the process is finished.

If normalExit() is FALSE (e.g. if the program was killed or crashed), this function returns 0, so you should check the return value of normalExit() before relying on this value.

See also normalExit() [p. 114] and processExited() [p. 114].

bool QProcess::isRunning () const

Returns TRUE if the process is running, otherwise FALSE.

See also normalExit() [p. 114], exitStatus() [p. 113] and processExited() [p. 114].

void QProcess::kill () const [slot]

Terminates the process. This is not a safe way to end a process since the process will not be able to do cleanup. tryTerminate() is a safer way to do it, but processes might ignore a tryTerminate().

The nice way to end a process and to be sure that it is finished, is doing something like this:

```
process->tryTerminate();
QTimer::singleShot( 5000, process, SLOT( kill() ) );
```

This tries to terminate the process the nice way. If the process is still running after 5 seconds, it terminates the process the hard way. The timeout should be chosen depending on the time the process needs to do all the cleanup: use a higher value if the process is likely to do heavy computation on cleanup.

The slot returns immediately: it does not wait until the process has finished. When the process really exited, the signal processExited() is emitted.

See also tryTerminate() [p. 117] and processExited() [p. 114].

bool QProcess::launch (const QByteArray & buf, QStringList * env = 0) [virtual]

Runs the process and writes the data *buf* to the process's standard input. If all the data is written to standard input, standard input is closed. The command is searched for in the path for executable programs; you can also use an absolute path in the command itself.

If *env* is null, then the process is started with the same environment as the starting process. If *env* is non-null, then the values in the stringlist are interpreted as environment setttings of the form `key=value` and the process is started with these environment settings. For convenience, there is a small exception to this rule under Unix: if *env* does not contain any settings for the environment variable `LD_LIBRARY_PATH`, then this variable is inherited from the starting process.

Returns TRUE if the process could be started; otherwise returns FALSE.

Note that you should not use the slots writeToStdin() and closeStdin() on processes started with launch(), since the result is not well-defined. If you need these slots, use start() instead.

The process may or may not read the *buf* data sent to its standard input.

You can call this function even when a process that was started with this instance is still running. Be aware that if you do this the standard input of the process that was launched first will be closed, with any pending data being deleted,

and the process will be left to run out of your control. Similarly, if the process could not be started the standard input will be closed and the pending data deleted. (On operating systems that have zombie processes, Qt will also wait() on the old process.)

The object emits the signal launchFinished() when this function call is finished. If the start was successful, this signal is emitted after all the data has been written to standard input. If the start failed, then this signal is emitted immediately.

See also start() [p. 116] and launchFinished() [p. 114].

bool QProcess::launch (const QString & buf, QStringList * env = 0) [virtual]

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

The data *buf* is written to standard input with writeToStdin() using the QString::local8Bit() representation of the strings.

void QProcess::launchFinished () [signal]

This signal is emitted when the process was started with launch(). If the start was successful, this signal is emitted after all the data has been written to standard input. If the start failed, then this signal is emitted immediately.

See also launch() [p. 113] and QObject::deleteLater() [Additional Functionality with Qt].

bool QProcess::normalExit () const

Returns TRUE if the process has exited normally; otherwise returns FALSE. This implies that this function returns FALSE if the process is still running.

See also isRunning() [p. 113], exitStatus() [p. 113] and processExited() [p. 114].

void QProcess::processExited () [signal]

This signal is emitted when the process has exited.

See also isRunning() [p. 113], normalExit() [p. 114], exitStatus() [p. 113], start() [p. 116] and launch() [p. 113].

Example: process/process.cpp.

PID QProcess::processIdentifier ()

Returns platform dependent information about the process. This can be used together with platform specific system calls.

Under Unix the return value is the PID of the process, or -1 if no process is belonging to this object.

Under Windows it is a pointer to the `PROCESS_INFORMATION` struct, or 0 if no process is belonging to this object.

QString QProcess::readLineStderr () [virtual]

Reads a line of text from standard error, excluding any trailing newline or carriage return characters and returns it. Returns QString::null if canReadLineStderr() returns FALSE.

See also canReadLineStderr() [p. 112], readyReadStderr() [p. 115], readStderr() [p. 115] and readLineStdout() [p. 115].

QString QProcess::readLineStdout () [virtual]

Reads a line of text from standard output, excluding any trailing newline or carriage return characters, and returns it. Returns QString::null if canReadLineStdout() returns FALSE.

See also canReadLineStdout() [p. 112], readyReadStdout() [p. 115], readStdout() [p. 115] and readLineStderr() [p. 114].

QByteArray QProcess::readStderr () [virtual]

Reads the data that the process has written to standard error. When new data is written to standard error, the class emits the signal readyReadStderr().

If there is no data to read, this function returns a QByteArray of size 0: it does not wait until there is something to read.

See also readyReadStderr() [p. 115], readLineStderr() [p. 114], readStdout() [p. 115] and writeToStdin() [p. 117].

QByteArray QProcess::readStdout () [virtual]

Reads the data that the process has written to standard output. When new data is written to standard output, the class emits the signal readyReadStdout().

If there is no data to read, this function returns a QByteArray of size 0: it does not wait until there is something to read.

See also readyReadStdout() [p. 115], readLineStdout() [p. 115], readStderr() [p. 115] and writeToStdin() [p. 117].

Example: process/process.cpp.

void QProcess::readyReadStderr () [signal]

This signal is emitted when the process has written data to standard error. You can read the data with readStderr().

Note that this signal is only emitted when there is new data and not when there is old, but unread data. In the slot connected to this signal, you should always read everything that is available at that moment to make sure that you don't lose any data.

See also readStderr() [p. 115], readLineStderr() [p. 114] and readyReadStdout() [p. 115].

void QProcess::readyReadStdout () [signal]

This signal is emitted when the process has written data to standard output. You can read the data with readStdout().

Note that this signal is only emitted when there is new data and not when there is old, but unread data. In the slot connected to this signal, you should always read everything that is available at that moment to make sure that you don't lose any data.

See also readStdout() [p. 115], readLineStdout() [p. 115] and readyReadStderr() [p. 115].

Example: process/process.cpp.

void QProcess::setArguments (const QStringList & args) [virtual]

Sets *args* as the arguments for the process. The first element in the list is the command to be executed. The other elements in the list are the arguments to the command. Any previous arguments are deleted.

See also arguments() [p. 112] and addArgument() [p. 112].

void QProcess::setCommunication (int commFlags)

Sets *commFlags* as the communication required with the process.

commFlags is a bitwise OR between the flags defined in Communication.

The default is `Stdin|Stdout|Stderr`.

See also communication() [p. 112].

void QProcess::setWorkingDirectory (const QDir & dir) [virtual]

Sets *dir* as the working directory for a process. This does not affect running processes; only processes that are started afterwards are affected.

Setting the working directory is especially useful for processes that try to access files with relative filenames.

See also workingDirectory() [p. 117] and start() [p. 116].

bool QProcess::start (QStringList * env = 0) [virtual]

Tries to run a process for the command and arguments that were specified with setArguments(), addArgument() or that were specified in the constructor. The command is searched in the path for executable programs; you can also use an absolute path to the command.

If *env* is null, then the process is started with the same environment as the starting process. If *env* is non-null, then the values in the stringlist are interpreted as environment setttings of the form `key=value` and the process is started in these environment settings. For convenience, there is a small exception to this rule: under Unix, if *env* does not contain any settings for the environment variable `LD_LIBRARY_PATH`, then this variable is inherited from the starting process; under Windows the same applies for the enverionment varialbe `PATH`.

Returns TRUE if the process could be started, otherwise FALSE.

You can write data to standard input of the process with writeToStdin(), you can close standard input with closeStdin() and you can terminate the process tryTerminate() resp. kill().

You can call this function even when there already is a running process in this object. In this case, QProcess closes standard input of the old process and deletes pending data, i.e., you loose all control over that process, but the process is not terminated. This applies also if the process could not be started. (On operating systems that have zombie processes, Qt will also wait() on the old process.)

See also launch() [p. 113] and closeStdin() [p. 112].

Example: process/process.cpp.

void QProcess::tryTerminate () const [slot]

Asks the process to terminate. Processes can ignore this wish. If you want to be sure that the process really terminates, you must use kill() instead.

The slot returns immediately: it does not wait until the process has finished. When the process really exited, the signal processExited() is emitted.

See also kill() [p. 113] and processExited() [p. 114].

QDir QProcess::workingDirectory () const

Returns the working directory that was set with setWorkingDirectory(), or the current directory if none has been set.

See also setWorkingDirectory() [p. 116] and QDir::current() [p. 36].

void QProcess::writeToStdin (const QByteArray & buf) [virtual slot]

Writes the data *buf* to the standard input of the process. The process may or may not read this data.

This function returns immediately; the QProcess class might write the data at a later point (you have to enter the event loop for that). When all the data is written to the process, the signal wroteToStdin() is emitted. This does not mean that the process really read the data, since this class only detects when it was able to write the data to the operating system.

See also wroteToStdin() [p. 117], closeStdin() [p. 112], readStdout() [p. 115] and readStderr() [p. 115].

void QProcess::writeToStdin (const QString & buf) [virtual slot]

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

The string *buf* is handled as text using the QString::local8Bit() representation.

void QProcess::wroteToStdin () [signal]

This signal is emitted if the data sent to standard input (via writeToStdin()) was actually written to the process. This does not imply that the process really read the data, since this class only detects when it was able to write the data to the operating system. But it is now safe to close standard input without losing pending data.

See also writeToStdin() [p. 117] and closeStdin() [p. 112].

QServerSocket Class Reference

The QServerSocket class provides a TCP-based server.

This class is part of the **network** module.

`#include <qserversocket.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QServerSocket** (Q_UINT16 port, int backlog = 1, QObject * parent = 0, const char * name = 0)
- **QServerSocket** (const QHostAddress & address, Q_UINT16 port, int backlog = 1, QObject * parent = 0, const char * name = 0)
- **QServerSocket** (QObject * parent = 0, const char * name = 0)
- virtual **~QServerSocket** ()
- bool **ok** () const
- Q_UINT16 **port** () const
- int **socket** () const
- virtual void **setSocket** (int socket)
- QHostAddress **address** () const
- virtual void **newConnection** (int socket)

Protected Members

- QSocketDevice * **socketDevice** ()

Detailed Description

The QServerSocket class provides a TCP-based server.

This class is a convenience class for accepting incoming TCP connections. You can specify the port or have QServerSocket pick one, and listen on just one address or on all the machine's addresses.

Using the API is very simple: subclass QServerSocket, call the constructor of your choice, and implement newConnection() to handle new incoming connections. There is nothing more to do.

(Note that due to lack of support in the underlying APIs, QServerSocket cannot accept or reject connections conditionally.)

See also QSocket [p. 126], QSocketDevice [p. 135], QHostAddress [p. 71], QSocketNotifier [p. 142] and Input/Output and Networking.

Member Function Documentation

QServerSocket::QServerSocket (Q_UINT16 port, int backlog = 1, QObject * parent = 0, const char * name = 0)

Creates a server socket object, that will serve the given *port* on all the addresses of this host. If *port* is 0, QServerSocket will pick a suitable port in a system-dependent manner. Use *backlog* to specify how many pending connections the server can have.

The *parent* and *name* arguments are passed on to the QObject constructor.

Warning: On Tru64 Unix systems a value of 0 for *backlog* means that you don't accept any connections at all; you should specify a value larger than 0.

QServerSocket::QServerSocket (const QHostAddress & address, Q_UINT16 port, int backlog = 1, QObject * parent = 0, const char * name = 0)

Creates a server socket object, that will serve the given *port* only on the given *address*. Use *backlog* to specify how many pending connections the server can have.

The *parent* and *name* arguments are passed on to the QObject constructor.

Warning: On Tru64 Unix systems a value of 0 for *backlog* means that you don't accept any connections at all; you should specify a value larger than 0.

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

Construct an empty server socket.

This constructor, in combination with setSocket(), allows us to use the QServerSocket class as a wrapper for other socket types (e.g. Unix Domain Sockets under Unix).

The *parent* and *name* arguments are passed on to the QObject constructor.

See also setSocket() [p. 120].

QServerSocket::~QServerSocket () [virtual]

Destroys the socket.

This causes any backlogged connections (connections that have reached the host, but not yet been completely set up by calling QSocketDevice::accept()) to be severed.

Existing connections continue to exist; this only affects the acceptance of new connections.

QHostAddress QServerSocket::address () const

Returns the address on which this object listens, or 0.0.0.0 if this object listens on more than one address. ok() must be TRUE before calling this function.

See also port() [p. 120] and QSocketDevice::address() [p. 137].

void QServerSocket::newConnection (int socket) [virtual]

This pure virtual function is responsible for setting up a new incoming connection. *socket* is the fd (file descripor) for the newly accepted connection.

bool QServerSocket::ok () const

Returns TRUE if the construction succeeded; otherwise returns FALSE.

Q_UINT16 QServerSocket::port () const

Returns the port number on which this server socket listens. This is always non-zero; if you specify 0 in the constructor, QServerSocket will pick a non-zero port itself. ok() must be TRUE before calling this function.

See also address() [p. 120] and QSocketDevice::port() [p. 139].

Example: network/httpd/httpd.cpp.

void QServerSocket::setSocket (int socket) [virtual]

Sets the socket to use *socket*. bind() and listen() should already have been called for *socket*.

This allows us to use the QServerSocket class as a wrapper for other socket types (e.g. Unix Domain Sockets under Unix).

int QServerSocket::socket () const

Returns the operating system socket.

QSocketDevice * QServerSocket::socketDevice () [protected]

Returns a pointer to the internal socket device. The returned pointer is null if there is no connection or pending connection.

There is normally no need to manipulate the socket device directly since this class does all the necessary setup for most client or server socket applications.

QSessionManager Class Reference

The QSessionManager class provides access to the session manager.

`#include <qsessionmanager.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- QString **sessionId** () const
- void * **handle** () const
- bool **allowsInteraction** ()
- bool **allowsErrorInteraction** ()
- void **release** ()
- void **cancel** ()
- enum **RestartHint** { RestartIfRunning, RestartAnyway, RestartImmediately, RestartNever }
- void **setRestartHint** (RestartHint hint)
- RestartHint **restartHint** () const
- void **setRestartCommand** (const QStringList & command)
- QStringList **restartCommand** () const
- void **setDiscardCommand** (const QStringList &)
- QStringList **discardCommand** () const
- void **setManagerProperty** (const QString & name, const QString & value)
- void **setManagerProperty** (const QString & name, const QStringList & value)
- bool **isPhase2** () const
- void **requestPhase2** ()

Detailed Description

The QSessionManager class provides access to the session manager.

The session manager is responsible for session management, most importantly for interruption and resumption. A "session" is a kind of record of the state of the system, e.g. which applications were run at start up and which applications are currently running. The session manager is used to save the session, e.g. when the machine is shut down; and to restore a session, e.g. when the machine is started up. Use QSettings to save and restore an individual application's settings, e.g. window positions, recently used files, etc.

QSessionManager provides an interface between the application and the session manager so that the program can work well with the session manager. In Qt, session management requests for action are handled by the two virtual functions QApplication::commitData() and QApplication::saveState(). Both provide a reference to a session manager object as argument, to allow the application to communicate with the session manager.

During a session management action (i.e. within commitData() and saveState()), no user interaction is possible *unless* the application got explicit permission from the session manager. You ask for permission by calling allowsInteraction() or, if it's really urgent, allowsErrorInteraction(). Qt does not enforce this, but the session manager may.

You can try to abort the shutdown process by calling cancel(). The default commitData() function does this if some top-level window rejected its closeEvent().

For sophisticated session managers provided on Unix/X11, QSessionManager offers further possibilites to fine-tune an application's session management behavior: setRestartCommand(), setDiscardCommand(), setRestartHint(), setProperty(), requestPhase2(). See the respective function descriptions for further details.

See also Main Window and Related Classes and Environment Classes.

Member Type Documentation

QSessionManager::RestartHint

This enum type defines the circumstances under which this application wants to be restarted by the session manager. The current values are

- `QSessionManager::RestartIfRunning` - if the application is still running when the session is shut down, it wants to be restarted at the start of the next session.
- `QSessionManager::RestartAnyway` - the application wants to be started at the start of the next session, no matter what. (This is useful for utilities that run just after startup and then quit.)
- `QSessionManager::RestartImmediately` - the application wants to be started immediately whenever it is not running.
- `QSessionManager::RestartNever` - the application does not want to be restarted automatically.

The default hint is RestartIfRunning.

Member Function Documentation

bool QSessionManager::allowsErrorInteraction ()

This is similar to allowsInteraction(), but also tells the session manager that an error occurred. Session managers may give error interaction request higher priority, which means that it is more likely that an error interaction is permitted. However, you are still not guaranteed that the session manager will allow interaction.

See also allowsInteraction() [p. 122], release() [p. 124] and cancel() [p. 123].

bool QSessionManager::allowsInteraction ()

Asks the session manager for permission to interact with the user. Returns TRUE if interaction is permitted; otherwise returns FALSE.

The rationale behind this mechanism is to make it possible to synchronize user interaction during a shutdown. Advanced session managers may ask all applications simultaneously to commit their data, resulting in a much faster shutdown.

When the interaction is completed we strongly recommend releasing the user interaction semaphore with a call to release(). This way, other applications may get the chance to interact with the user while your application is still busy saving data. (The semaphore is implicitly released when the application exits.)

If the user decides to cancel the shutdown process during the interaction phase, you must tell the session manager that this has happened by calling cancel().

Here's an example of how an application's QApplication::commitData() might be implemented:

```
void MyApplication::commitData( QSessionManager& sm ) {
    if ( sm.allowsInteraction() ) {
        switch ( QMessageBox::warning(
                    yourMainWindow,
                    tr("Application Name"),
                    tr("Save changes to document Foo?"),
                    tr("&Yes"),
                    tr("&No"),
                    tr("Cancel"),
                    0, 2) ) {
        case 0: // yes
            sm.release();
            // save document here; if saving fails, call sm.cancel()
            break;
        case 1: // continue without saving
            break;
        default: // cancel
            sm.cancel();
            break;
        }
    } else {
        // we did not get permission to interact, then
        // do something reasonable instead.
    }
}
```

If an error occurred within the application while saving its data, you may want to try allowsErrorInteraction() instead.

See also QApplication::commitData() [Additional Functionality with Qt], release() [p. 124] and cancel() [p. 123].

void QSessionManager::cancel ()

Tells the session manager to cancel the shutdown process. Applications should not call this function without first asking the user.

See also allowsInteraction() [p. 122] and allowsErrorInteraction() [p. 122].

QStringList QSessionManager::discardCommand () const

Returns the currently set discard command.

See also setDiscardCommand() [p. 125], restartCommand() [p. 124] and setRestartCommand() [p. 125].

void * QSessionManager::handle () const

X11 only: returns a handle to the current `SmcConnection`.

bool QSessionManager::isPhase2 () const

Returns TRUE if the session manager is currently performing a second session management phase; otherwise returns FALSE.

See also requestPhase2() [p. 124].

void QSessionManager::release ()

Releases the session manager's interaction semaphore after an interaction phase.

See also allowsInteraction() [p. 122] and allowsErrorInteraction() [p. 122].

void QSessionManager::requestPhase2 ()

Requests a second session management phase for the application. The application may then return immediately from the QApplication::commitData() or QApplication::saveState() function, and they will be called again once most or all other applications have finished their session management.

The two phases are useful for applications such as the X11 window manager that need to store information about another application's windows and therefore have to wait until these applications have completed their respective session management tasks.

Note that if another application has requested a second phase it may get called before, simultaneously with, or after your application's second phase.

See also isPhase2() [p. 124].

QStringList QSessionManager::restartCommand () const

Returns the currently set restart command.

See also setRestartCommand() [p. 125] and restartHint() [p. 124].

RestartHint QSessionManager::restartHint () const

Returns the application's current restart hint. The default is RestartIfRunning.

See also setRestartHint() [p. 125].

QString QSessionManager::sessionId () const

Returns the identifier of the current session.

If the application has been restored from an earlier session, this identifier is the same as it was in that earlier session.

See also QApplication::sessionId() [Additional Functionality with Qt].

void QSessionManager::setDiscardCommand (const QStringList &)

See also discardCommand() [p. 123] and setRestartCommand() [p. 125].

void QSessionManager::setManagerProperty (const QString & name, const QStringList & value)

Low-level write access to the application's identification and state record are kept in the session manager.

The property called *name* has its value set to the string list *value*.

void QSessionManager::setManagerProperty (const QString & name, const QString & value)

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

Low-level write access to the application's identification and state records are kept in the session manager.

The property called *name* has its value set to the string *value*.

void QSessionManager::setRestartCommand (const QStringList & command)

If the session manager is capable of restoring sessions it will execute *command* in order to restore the application. The command defaults to

```
appname -session id
```

The `-session` option is mandatory; otherwise QApplication cannot tell whether it has been restored or what the current session identifier is. See QApplication::isSessionRestored() and QApplication::sessionId() for details.

If your application is very simple, it may be possible to store the entire application state in additional command line options. This is usually a very bad idea because command lines are often limited to a few hundred bytes. Instead, use QSettings, or temporary files or a database for this purpose. By marking the data with the unique sessionId(), you will be able to restore the application in a future session.

See also restartCommand() [p. 124], setDiscardCommand() [p. 125] and setRestartHint() [p. 125].

void QSessionManager::setRestartHint (RestartHint hint)

Sets the application's restart hint to *hint*. On application startup the hint is set to RestartIfRunning.

Note that these flags are only hints, a session manager may or may not respect them.

We recommend setting the restart hint in QApplication::saveState() because most session managers perform a checkpoint shortly after an application's startup.

See also restartHint() [p. 124].

QSocket Class Reference

The QSocket class provides a buffered TCP connection.

This class is part of the **network** module.

`#include <qsocket.h>`

Inherits QObject [Additional Functionality with Qt] and QIODevice [p. 76].

Public Members

- enum **Error** { ErrConnectionRefused, ErrHostNotFound, ErrSocketRead }
- **QSocket** (QObject * parent = 0, const char * name = 0)
- virtual **~QSocket** ()
- enum **State** { Idle, HostLookup, Connecting, Connected, Closing, Connection = Connected }
- State **state** () const
- int **socket** () const
- virtual void **setSocket** (int socket)
- QSocketDevice * **socketDevice** ()
- virtual void **setSocketDevice** (QSocketDevice * device)
- virtual void **connectToHost** (const QString & host, Q_UINT16 port)
- QString **peerName** () const
- virtual bool **open** (int m)
- virtual void **close** ()
- virtual void **flush** ()
- virtual Offset **size** () const
- virtual Offset **at** () const
- virtual bool **at** (Offset index)
- virtual bool **atEnd** () const
- Q_ULONG **bytesAvailable** () const
- Q_ULONG **waitForMore** (int msecs) const
- Q_ULONG **bytesToWrite** () const
- virtual Q_LONG **readBlock** (char * data, Q_ULONG maxlen)
- virtual Q_LONG **writeBlock** (const char * data, Q_ULONG len)
- virtual int **getch** ()
- virtual int **putch** (int ch)
- virtual int **ungetch** (int ch)
- bool **canReadLine** () const

- virtual QString **readLine** ()
- Q_UINT16 **port** () const
- Q_UINT16 **peerPort** () const
- QHostAddress **address** () const
- QHostAddress **peerAddress** () const

Signals

- void **hostFound** ()
- void **connected** ()
- void **connectionClosed** ()
- void **delayedCloseFinished** ()
- void **readyRead** ()
- void **bytesWritten** (int nbytes)
- void **error** (int)

Protected Slots

- virtual void **sn_read** (bool force = FALSE)
- virtual void **sn_write** ()

Detailed Description

The QSocket class provides a buffered TCP connection.

It provides a totally non-blocking QIODevice, and modifies and extends the API of QIODevice with socket-specific code.

The functions you're likely to call most are connectToHost(), bytesAvailable(), canReadLine() and the ones it inherits from QIODevice.

connectToHost() is the most-used function. As its name implies, it opens a connection to a named host.

Most network protocols are either packet-oriented or line-oriented. canReadLine() indicates whether a connection contains an entire unread line or not, and bytesAvailable() returns the number of bytes available for reading.

The signals error(), connected(), readyRead() and connectionClosed() inform you of the progress of the connection. There are also some less commonly used signals. hostFound() is emitted when connectToHost() has finished its DNS lookup and is starting its TCP connection. delayedCloseFinished() is emitted when close() succeeds. bytesWritten() is emitted when QSocket moves data from its "to be written" queue into the TCP implementation.

There are several access functions for the socket: state() returns whether the object is idle, is doing a DNS lookup, is connecting, has an operational connection, etc. address() and port() return the IP address and port used for the connection. The peerAddress() and peerPort() functions return the IP address and port used by the peer, and peerName() returns the name of the peer (normally the name that was passed to connectToHost()). socket() returns a pointer to the QSocketDevice used for this socket.

QSocket inherits QIODevice, and reimplements some functions. In general, you can treat it as a QIODevice for writing, and mostly also for reading. The match isn't perfect, since the QIODevice API is designed for devices that are controlled by the same machine, and an asynchronous peer-to-peer network connection isn't quite like that. For example, there is

nothing that matches QIODevice::size() exactly. The documentation for open(), close(), flush(), size(), at(), atEnd(), readBlock(), writeBlock(), getch(), putch(), ungetch() and readLine() describes the differences in detail.

See also QSocketDevice [p. 135], QHostAddress [p. 71], QSocketNotifier [p. 142] and Input/Output and Networking.

Member Type Documentation

QSocket::Error

This enum specifies the possible errors:

- `QSocket::ErrConnectionRefused` - if the connection was refused
- `QSocket::ErrHostNotFound` - if the host was not found
- `QSocket::ErrSocketRead` - if a read from the socket failed

QSocket::State

This enum defines the connection states:

- `QSocket::Idle` - if there is no connection
- `QSocket::HostLookup` - during a DNS lookup
- `QSocket::Connecting` - during TCP connection establishment
- `QSocket::Connected` - when there is an operational connection
- `QSocket::Closing` - if the socket is closing down, but is not yet closed.

Member Function Documentation

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

Creates a QSocket object in QSocket::Idle state.

The *parent* and *name* arguments are passed on to the QObject constructor.

QSocket::~QSocket () [virtual]

Destroys the socket. Closes the connection if necessary.

See also close() [p. 130].

QHostAddress QSocket::address () const

Returns the host address of this socket. (This is normally the main IP address of the host, but can be e.g. 127.0.0.1 for connections to localhost.)

Offset QSocket::at () const [virtual]

Returns the current read index. Since QSocket is a sequential device, the current read index is always zero.

Reimplemented from QIODevice [p. 79].

bool QSocket::at (Offset index) [virtual]

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

Moves the read index forward to *index* and returns TRUE if the operation was successful. Moving the index forward means skipping incoming data.

Reimplemented from QIODevice [p. 79].

bool QSocket::atEnd () const [virtual]

Returns TRUE if there is no more data to read; otherwise returns FALSE.

Reimplemented from QIODevice [p. 79].

Q_ULONG QSocket::bytesAvailable () const

Returns the number of incoming bytes that can be read, i.e. the size of the input buffer. Equivalent to size().

See also bytesToWrite() [p. 129].

Example: network/networkprotocol/nntp.cpp.

Q_ULONG QSocket::bytesToWrite () const

Returns the number of bytes that are waiting to be written, i.e. the size of the output buffer.

See also bytesAvailable() [p. 129].

void QSocket::bytesWritten (int nbytes) [signal]

This signal is emitted when data has actually been written to the network. The *nbytes* parameter specifies how many bytes were written.

The bytesToWrite() function is often used in the same context, and it says how many buffered bytes there are left to write.

See also writeBlock() [p. 134] and bytesToWrite() [p. 129].

bool QSocket::canReadLine () const

Returns TRUE if it's possible to read an entire line of text from this socket at this time; otherwise returns FALSE.

Note that if the peer closes the connection unexpectedly, this function returns FALSE. This means that loops such as this won't work:

```
while( !socket->canReadLine() ) // Wrong.
    ...
```

See also readLine() [p. 132].

Examples: network/clientserver/client/client.cpp, network/httpd/httpd.cpp, network/mail/smtp.cpp and network/networkprotocol/nntp.cpp.

void QSocket::close () [virtual]

Closes the socket.

The read buffer is cleared.

If the output buffer is empty, the state is set to QSocket::Idle and the connection is terminated immediately. If the output buffer still contains data to be written, QSocket goes into the QSocket::Closing state and the rest of the data will be written. When all of the outgoing data have been written, the state is set to QSocket::Idle and the connection is terminated. At this point, the delayedCloseFinished() signal is emitted.

See also state() [p. 134] and bytesToWrite() [p. 129].

Examples: network/clientserver/client/client.cpp, network/httpd/httpd.cpp and network/networkprotocol/nntp.cpp.

Reimplemented from QIODevice [p. 79].

void QSocket::connectToHost (const QString & host, Q_UINT16 port) [virtual]

Attempts to make a connection to *host* on the specified *port* and return immediately.

Any connection or pending connection is closed immediately, and QSocket goes into the HostLookup state. When the lookup succeeds, it emits hostFound(), starts a TCP connection and goes into the Connecting state. Finally, when the connection succeeds, it emits connected() and goes into the Connected state. If there is an error at any point, it emits error().

host may be an IP address in string form, or it may be a DNS name. QSocket will do a normal DNS lookup if required. Note that *port* is in native byte order, unlike some other libraries.

See also state() [p. 134].

Examples: network/clientserver/client/client.cpp, network/mail/smtp.cpp and network/networkprotocol/nntp.cpp.

void QSocket::connected () [signal]

This signal is emitted after connectToHost() has been called and a connection has been successfully established.

See also connectToHost() [p. 130] and connectionClosed() [p. 130].

Examples: network/clientserver/client/client.cpp, network/mail/smtp.cpp and network/networkprotocol/nntp.cpp.

void QSocket::connectionClosed () [signal]

This signal is emitted when the other end has closed the connection. The read buffers may contain buffered input data which you can read after the connection was closed.

See also connectToHost() [p. 130] and close() [p. 130].

Examples: network/clientserver/client/client.cpp and network/networkprotocol/nntp.cpp.

void QSocket::delayedCloseFinished () [signal]

This signal is emitted when a delayed close is finished.

If you call close() and there is buffered output data to be written, QSocket goes into the QSocket::Closing state and returns immediately. It will then keep writing to the socket until all the data has been written. Then, the delayedCloseFinished() signal is emitted.

See also close() [p. 130].

Examples: network/clientserver/client/client.cpp and network/httpd/httpd.cpp.

void QSocket::error (int) [signal]

This signal is emitted after an error occurred. The parameter is the Error value.

Examples: network/clientserver/client/client.cpp and network/networkprotocol/nntp.cpp.

void QSocket::flush () [virtual]

Implementation of the abstract virtual QIODevice::flush() function.

Reimplemented from QIODevice [p. 80].

int QSocket::getch () [virtual]

Reads a single byte/character from the internal read buffer. Returns the byte/character read, or -1 if there is nothing to be read.

See also bytesAvailable() [p. 129] and putch() [p. 132].

Reimplemented from QIODevice [p. 80].

void QSocket::hostFound () [signal]

This signal is emitted after connectToHost() has been called and the host lookup has succeeded.

See also connected() [p. 130].

Example: network/networkprotocol/nntp.cpp.

bool QSocket::open (int m) [virtual]

Opens the socket using the specified QIODevice file mode *m*. This function is called automatically when needed and you should not call it yourself.

See also close() [p. 130].

Reimplemented from QIODevice [p. 82].

QHostAddress QSocket::peerAddress () const

Returns the host address as resolved from the name specified to the connectToHost() function.

QString QSocket::peerName () const

Returns the host name as specified to the connectToHost() function. An empty string is returned if none has been set.

Example: network/mail/smtp.cpp.

Q_UINT16 QSocket::peerPort () const

Returns the peer's host port number, normally as specified to the connectToHost() function. If none has been set, this function returns 0.

Note that Qt always uses native byte order, i.e. 67 is 67 in Qt, there is no need to call htons().

Q_UINT16 QSocket::port () const

Returns the host port number of this socket, in native byte order.

int QSocket::putch (int ch) [virtual]

Writes the character *ch* to the output buffer.

Returns *ch*, or -1 if some error occurred.

See also getch() [p. 131].

Reimplemented from QIODevice [p. 83].

Q_LONG QSocket::readBlock (char * data, Q_ULONG maxlen) [virtual]

Reads max *maxlen* bytes from the socket into *data* and returns the number of bytes read. Returns -1 if an error occurred.

Example: network/networkprotocol/nntp.cpp.

Reimplemented from QIODevice [p. 83].

QString QSocket::readLine () [virtual]

Returns a line of text including a terminating newline character (\n). Returns "" if canReadLine() returns FALSE.

See also canReadLine() [p. 129].

Examples: network/clientserver/client/client.cpp, network/httpd/httpd.cpp, network/mail/smtp.cpp and network/networkprotocol/nntp.cpp.

void QSocket::readyRead () [signal]

This signal is emitted when there is incoming data to be read.

Every time there is new incoming data this signal is emitted once. Bear in mind that new incoming data is only reported once; i.e. if you do not read all data, this signal is not emitted again unless new data arrives on the socket.

See also readBlock() [p. 132], readLine() [p. 132] and bytesAvailable() [p. 129].

Examples: network/clientserver/client/client.cpp, network/httpd/httpd.cpp, network/mail/smtp.cpp and network/networkprotocol/nntp.cpp.

void QSocket::setSocket (int socket) [virtual]

Sets the socket to use *socket* and the state() to Connected. The socket should already be connected.

This allows us to use the QSocket class as a wrapper for other socket types (e.g. Unix Domain Sockets under Unix).

Example: network/httpd/httpd.cpp.

void QSocket::setSocketDevice (QSocketDevice * device) [virtual]

Sets the internal socket device to *device*. Passing a *device* of 0 will cause the internal socket device to be used. Any existing connection will be disconnected before using the new *device*.

The new device should not be connected before being associated with a QSocket; after setting the socket call connectToHost() to make the connection.

This function is useful if you need to subclass QSocketDevice and want to use the QSocket API, for example, to implement Unix domain sockets.

Offset QSocket::size () const [virtual]

Returns the number of incoming bytes that can be read right now (like bytesAvailable()).

Reimplemented from QIODevice [p. 84].

void QSocket::sn_read (bool force = FALSE) [virtual protected slot]

Internal slot for handling socket read notifications.

This function has can usually be only entered once (i.e. no recursive calls). If the argument *force* is TRUE, the function is executed, but no readyRead() signals are emitted. This behaviour is useful for the waitForMore() function, so that it is possible to call waitForMore() in a slot connected to the readyRead() signal.

void QSocket::sn_write () [virtual protected slot]

Internal slot for handling socket write notifications.

int QSocket::socket () const

Returns the socket number, or -1 if there is no socket at the moment.

QSocketDevice * QSocket::socketDevice ()

Returns a pointer to the internal socket device.

There is normally no need to manipulate the socket device directly since this class does the necessary setup for most applications.

State QSocket::state () const

Returns the current state of the socket connection.

See also QSocket::State [p. 128].

Examples: network/clientserver/client/client.cpp and network/networkprotocol/nntp.cpp.

int QSocket::ungetch (int ch) [virtual]

This implementation of the virtual function QIODevice::ungetch() prepends the character *ch* to the read buffer so that the next read returns this character as the first character of the output.

Reimplemented from QIODevice [p. 84].

Q_ULONG QSocket::waitForMore (int msecs) const

Wait up to *msecs* milliseconds for more data to be available.

If *msecs* is -1 the call will block indefinitely.

This is a blocking call and should be avoided in event driven applications.

Returns the number of bytes available.

See also bytesAvailable() [p. 129].

Q_LONG QSocket::writeBlock (const char * data, Q_ULONG len) [virtual]

Writes *len* bytes to the socket from *data* and returns the number of bytes written. Returns -1 if an error occurred.

Example: network/networkprotocol/nntp.cpp.

Reimplemented from QIODevice [p. 85].

QSocketDevice Class Reference

The QSocketDevice class provides a platform-independent low-level socket API.

This class is part of the **network** module.

`#include <qsocketdevice.h>`

Inherits QIODevice [p. 76].

Public Members

- enum **Type** { Stream, Datagram }
- **QSocketDevice** (Type type = Stream)
- **QSocketDevice** (int socket, Type type)
- virtual **~QSocketDevice** ()
- bool **isValid** () const
- Type **type** () const
- int **socket** () const
- virtual void **setSocket** (int socket, Type type)
- bool **blocking** () const
- virtual void **setBlocking** (bool enable)
- bool **addressReusable** () const
- virtual void **setAddressReusable** (bool enable)
- int **receiveBufferSize** () const
- virtual void **setReceiveBufferSize** (uint size)
- int **sendBufferSize** () const
- virtual void **setSendBufferSize** (uint size)
- virtual bool **connect** (const QHostAddress & addr, Q_UINT16 port)
- virtual bool **bind** (const QHostAddress & address, Q_UINT16 port)
- virtual bool **listen** (int backlog)
- virtual int **accept** ()
- Q_LONG **bytesAvailable** () const
- Q_LONG **waitForMore** (int msecs, bool * timeout = 0) const
- virtual Q_LONG **readBlock** (char * data, Q_ULONG maxlen)
- virtual Q_LONG **writeBlock** (const char * data, Q_ULONG len)
- virtual Q_LONG **writeBlock** (const char * data, Q_ULONG len, const QHostAddress & host, Q_UINT16 port)
- Q_UINT16 **port** () const
- Q_UINT16 **peerPort** () const

- QHostAddress **address** () const
- QHostAddress **peerAddress** () const
- enum **Error** { NoError, AlreadyBound, Inaccessible, NoResources, Bug, Impossible, NoFiles, ConnectionRefused, NetworkFailure, UnknownError }
- Error **error** () const

Protected Members

- void **setError** (Error err)

Detailed Description

The QSocketDevice class provides a platform-independent low-level socket API.

This class is not really intended for use outside Qt. It can be used to achieve some things that QSocket does not provide, but it's not particularly easy to understand or use.

The essential purpose of the class is to provide a QIODevice that works on sockets, wrapped in a platform-independent API.

See also QSocket [p. 126], QSocketNotifier [p. 142], QHostAddress [p. 71] and Input/Output and Networking.

Member Type Documentation

QSocketDevice::Error

This enum type describes the error states of QSocketDevice. At present these errors are defined:

- `QSocketDevice::NoError` - all is fine.
- `QSocketDevice::AlreadyBound` - bind() said so.
- `QSocketDevice::Inaccessible` - the operating system or firewall prohibits something.
- `QSocketDevice::NoResources` - the operating system ran out of something.
- `QSocketDevice::Bug` - there seems to be a bug in QSocketDevice.
- `QSocketDevice::Impossible` - the impossible happened, usually because you confused QSocketDevice horribly. Simple example:

  ```
  ::close( sd->socket() );
  sd->writeBlock( someData, 42 );
  ```

  The libc ::close() closes the socket, but QSocketDevice is not aware of this. So when you call writeBlock(), the impossible happens.
- `QSocketDevice::NoFiles` - the operating system will not let QSocketDevice open another file.
- `QSocketDevice::ConnectionRefused` - a connection attempt was rejected by the peer.
- `QSocketDevice::NetworkFailure` - there is a network failure between this host and... and whatever.
- `QSocketDevice::UnknownError` - the operating system reacted in a way that the Qt developers did not foresee.

QSocketDevice::Type

This enum type describes the type of the socket:

- `QSocketDevice::Stream` - a stream socket (TCP, usually)
- `QSocketDevice::Datagram` - a datagram socket (UDP, usually)

Member Function Documentation

QSocketDevice::QSocketDevice (Type type = Stream)

Creates a QSocketDevice object for a stream or datagram socket.

The *type* argument must be either QSocketDevice::Stream for a reliable, connection-oriented TCP socket, or QSocketDevice::Datagram for an unreliable UDP socket.

See also blocking() [p. 138].

QSocketDevice::QSocketDevice (int socket, Type type)

Creates a QSocketDevice object for the existing socket *socket*.

The *type* argument must match the actual socket type; use QSocketDevice::Stream for a reliable, connection-oriented TCP socket, or QSocketDevice::Datagram for an unreliable, connectionless UDP socket.

QSocketDevice::~QSocketDevice () [virtual]

Destroys the socket device and closes the socket if it is open.

int QSocketDevice::accept () [virtual]

Extracts the first connection from the queue of pending connections for this socket and returns a new socket identifier. Returns -1 if the operation failed.

See also bind() [p. 138] and listen() [p. 138].

QHostAddress QSocketDevice::address () const

Returns the address of this socket device. This may be 0.0.0.0 for a while, but is set to something sensible when there is a sensible value it can have.

bool QSocketDevice::addressReusable () const

Returns TRUE if the address of this socket can be used by other sockets at the same time, and FALSE if this socket claims exclusive ownership.

See also setAddressReusable() [p. 139].

bool QSocketDevice::bind (const QHostAddress & address, Q_UINT16 port) [virtual]

Assigns a name to an unnamed socket. The name is the host address *address* and the port number *port*. If the operation succeeds, bind() returns TRUE. Otherwise, it returns FALSE without changing what port() and address() return.

bind() is used by servers for setting up incoming connections. Call bind() before listen().

bool QSocketDevice::blocking () const

Returns TRUE if the socket is in blocking mode, or FALSE if it is in nonblocking mode or if the socket is invalid.

Note that this function does not set error().

Warning: On Windows, this function always returns TRUE since the ioctlsocket() function is broken.

See also setBlocking() [p. 140] and isValid() [p. 138].

Q_LONG QSocketDevice::bytesAvailable () const

Returns the number of bytes available for reading, or -1 if an error occurred.

Warning: On Microsoft Windows, we use the ioctlsocket() function to determine the number of bytes queued on the socket. According to Microsoft (KB Q125486), ioctlsocket() sometimes return an incorrect number. The only safe way to determine the amount of data on the socket is to read it using readBlock(). QSocket has workarounds to deal with this problem.

bool QSocketDevice::connect (const QHostAddress & addr, Q_UINT16 port) [virtual]

Connects to the IP address and port specified by *addr* and *port*. Returns TRUE if it establishes a connection, and FALSE if not. error() explains why.

Note that error() commonly returns NoError for non-blocking sockets; this just means that you can call connect() again in a little while and it'll probably succeed.

Error QSocketDevice::error () const

Returns the first error seen.

bool QSocketDevice::isValid () const

Returns TRUE if this is a valid socket; otherwise returns FALSE.

See also socket() [p. 141].

bool QSocketDevice::listen (int backlog) [virtual]

Specifies how many pending connections a server socket can have. Returns TRUE if the operation was successful, otherwise FALSE.

The listen() call only applies to sockets where type() is Stream, not Datagram sockets. listen() must not be called before bind() or after accept(). It is common to use a *backlog* value of 50 on most Unix systems.

See also bind() [p. 138] and accept() [p. 137].

QHostAddress QSocketDevice::peerAddress () const

Returns the address of the port this socket device is connected to. This may be 0.0.0.0 for a while, but is set to something sensible when there is a sensible value it can have.

Note that for Datagram sockets, this is the source port of the last packet received, and that it is in native byte order.

Q_UINT16 QSocketDevice::peerPort () const

Returns the port number of the port this socket device is connected to. This may be 0 for a while, but is set to something sensible when there is a sensible value it can have.

Note that for Datagram sockets, this is the source port of the last packet received.

Q_UINT16 QSocketDevice::port () const

Returns the port number of this socket device. This may be 0 for a while, but is set to something sensible when there is a sensible value it can have.

Note that Qt always uses native byte order, i.e. 67 is 67 in Qt, there is no need to call htons().

Q_LONG QSocketDevice::readBlock (char * data, Q_ULONG maxlen) [virtual]

Reads max *maxlen* bytes from the socket into *data* and returns the number of bytes read. Returns -1 if an error occurred.

Reimplemented from QIODevice [p. 83].

int QSocketDevice::receiveBufferSize () const

Returns the size of the OS receive buffer.

See also setReceiveBufferSize() [p. 140].

int QSocketDevice::sendBufferSize () const

Returns the size of the OS send buffer.

See also setSendBufferSize() [p. 140].

void QSocketDevice::setAddressReusable (bool enable) [virtual]

Sets the address of this socket to be usable by other sockets too if *enable* is TRUE, and to be used exclusively by this socket if *enable* is FALSE.

When a socket is reusable, other sockets can use the same port number (and IP address), which is, in general, good. Of course other sockets cannot use the same (address,port,peer-address,peer-port) 4-tuple as this socket, so there is no risk of confusing the two TCP connections.

See also addressReusable() [p. 137].

void QSocketDevice::setBlocking (bool enable) [virtual]

Makes the socket blocking if *enable* is TRUE or nonblocking if *enable* is FALSE.

Sockets are blocking by default, but we recommend using nonblocking socket operations, especially for GUI programs that need to be responsive.

Warning: On Windows, this function does nothing since the ioctlsocket() function is broken.

Whenever you use a QSocketNotifier on Windows, the socket is immediately made nonblocking.

See also blocking() [p. 138] and isValid() [p. 138].

void QSocketDevice::setError (Error err) [protected]

Allows subclasses to set the error state to *err*.

void QSocketDevice::setReceiveBufferSize (uint size) [virtual]

Sets the size of the OS receive buffer to *size*.

The OS receive buffer size effectively limits two things: how much data can be in transit at any one moment, and how much data can be received in one iteration of the main event loop.

The default is OS-dependent. A socket that receives large amounts of data is probably best off with a buffer size of 49152.

void QSocketDevice::setSendBufferSize (uint size) [virtual]

Sets the size of the OS send buffer to *size*.

The OS send buffer size effectively limits how much data can be in transit at any one moment.

The default is OS-dependent. A socket that sends large amounts of data is probably best off with a buffer size of 49152.

void QSocketDevice::setSocket (int socket, Type type) [virtual]

Sets the socket device to operate on the existing socket *socket*.

The *type* argument must match the actual socket type; use QSocketDevice::Stream for a reliable, connection-oriented TCP socket, or QSocketDevice::Datagram for an unreliable, connectionless UDP socket.

Any existing socket is closed.

See also isValid() [p. 138] and close() [p. 79].

int QSocketDevice::socket () const

Returns the socket number, or -1 if it is an invalid socket.

See also isValid() [p. 138] and type() [p. 141].

Type QSocketDevice::type () const

Returns the socket type which is either QSocketDevice::Stream or QSocketDevice::Datagram.

See also socket() [p. 141].

Q_LONG QSocketDevice::waitForMore (int msecs, bool * timeout = 0) const

Wait up to *msecs* milliseconds for more data to be available. If *msecs* is -1 the call will block indefinitely.

This is a blocking call and should be avoided in event driven applications.

Returns the number of bytes available for reading, or -1 if an error occurred.

If *timeout* is non-null and no error occurred (i.e. it does not return -1), then this function sets *timeout* out to TRUE, if the reason for returning was that the timeout was reached, otherwise it sets *timeout* to FALSE. This is useful to find out if the peer closed the connection.

See also bytesAvailable() [p. 138].

Q_LONG QSocketDevice::writeBlock (const char * data, Q_ULONG len) [virtual]

Writes *len* bytes to the socket from *data* and returns the number of bytes written. Returns -1 if an error occurred.

This is used for QSocketDevice::Stream sockets.

Reimplemented from QIODevice [p. 85].

Q_LONG QSocketDevice::writeBlock (const char * data, Q_ULONG len, const QHostAddress & host, Q_UINT16 port) [virtual]

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

Writes *len* bytes to the socket from *data* and returns the number of bytes written. Returns -1 if an error occurred.

This is used for QSocketDevice::Datagram sockets. You have to specify the *host* and *port* of the destination of the data.

QSocketNotifier Class Reference

The QSocketNotifier class provides support for socket callbacks.

`#include <qsocketnotifier.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- enum **Type** { Read, Write, Exception }
- **QSocketNotifier** (int socket, Type type, QObject * parent = 0, const char * name = 0)
- **~QSocketNotifier** ()
- int **socket** () const
- Type **type** () const
- bool **isEnabled** () const
- virtual void **setEnabled** (bool enable)

Signals

- void **activated** (int socket)

Detailed Description

The QSocketNotifier class provides support for socket callbacks.

This class makes it possible to write asynchronous socket-based code in Qt. Using synchronous socket operations blocks the program, which is clearly not acceptable for an event-driven GUI program.

Once you have opened a non-blocking socket (whether for TCP, UDP, a UNIX-domain socket, or any other protocol family your operating system supports), you can create a socket notifier to monitor the socket. Then you connect the activated() signal to the slot you want to be called when a socket event occurs.

There are three types of socket notifiers (read, write and exception); you must specify one of these in the constructor.

The type specifies when the activated() signal is to be emitted:

1. QSocketNotifier::Read - There is data to be read (socket read event).
2. QSocketNotifier::Write - Data can be written (socket write event).

3. QSocketNofifier::Exception - An exception has occurred (socket exception event). We recommend against using this.

For example, if you need to monitor both reads and writes for the same socket you must create two socket notifiers.

Example:

```
int sockfd;                                 // socket identifier
struct sockaddr_in sa;                      // should contain host address
sockfd = socket( AF_INET, SOCK_STREAM, 0 ); // create TCP socket
// make the socket non-blocking here, usually using fcntl( O_NONBLOCK )
::connect( sockfd, (struct sockaddr*)&sa,   // connect to host
           sizeof(sa) );                    //   NOT QObject::connect()!
QSocketNotifier *sn;
sn = new QSocketNotifier( sockfd, QSocketNotifier::Read, parent );
QObject::connect( sn, SIGNAL(activated(int)),
                  myObject, SLOT(dataReceived()) );
```

The optional *parent* argument can be set to make the socket notifier a child of any QObject, e.g. a widget, thus being automatically destroyed when the widget is destroyed.

For read notifiers it makes little sense to connect the activated() signal to more than one slot because the data can be read from the socket only once.

Also observe that if you do not read all the available data when the read notifier fires, it fires again and again.

If you disable the read notifier your program may deadlock. (The same applies to exception notifiers if you have to use them, for instance if you *have* to use TCP urgent data.)

For write notifiers, immediately disable the notifier after the activated() signal has been received and you have sent the data to be written on the socket. When you have more data to be written, enable it again to get a new activated() signal. The exception is if the socket data writing operation (send() or equivalent) fails with a "would block" error, which means that some buffer is full and you must wait before sending more data. In that case you do not need to disable and re-enable the write notifier; it will fire again as soon as the system allows more data to be sent.

The behavior of a write notifier that is left in enabled state after having emitting the first activated() signal (and no "would block" error has occurred) is undefined. Depending on the operating system, it may fire on every pass of the event loop or not at all.

If you need a time-out for your sockets you can use either timer events or the QTimer class.

Socket action is detected in the main event loop of Qt. The X11 version of Qt has a single UNIX select() call that incorporates all socket notifiers and the X socket.

Note that on XFree86 for OS/2, select() works only in the thread in which main() is running; you should therefore use that thread for GUI operations.

See also QSocket [p. 126], QServerSocket [p. 118], QSocketDevice [p. 135] and Input/Output and Networking.

Member Type Documentation

QSocketNotifier::Type

- `QSocketNotifier::Read`
- `QSocketNotifier::Write`

- `QSocketNotifier::Exception`

Member Function Documentation

QSocketNotifier::QSocketNotifier (int socket, Type type, QObject * parent = 0, const char * name = 0)

Constructs a socket notifier with the *parent* and the *name* that watches *socket* for *type* events, and enables it.

It is generally advisable to explicitly enable or disable the socket notifier, especially for write notifiers.

See also setEnabled() [p. 144] and isEnabled() [p. 144].

QSocketNotifier::~QSocketNotifier ()

Destroys the socket notifier.

void QSocketNotifier::activated (int socket) [signal]

This signal is emitted under certain conditions specified by the notifier type():

1. QSocketNotifier::Read - There is data to be read (socket read event).
2. QSocketNotifier::Write - Data can be written (socket write event).
3. QSocketNofifier::Exception - An exception has occurred (socket exception event).

The *socket* argument is the socket identifier.

See also type() [p. 145] and socket() [p. 145].

bool QSocketNotifier::isEnabled () const

Returns TRUE if the notifier is enabled; otherwise returns FALSE.

See also setEnabled() [p. 144].

void QSocketNotifier::setEnabled (bool enable) [virtual]

Enables the notifier if *enable* is TRUE or disables it if *enable* is FALSE.

The notifier is enabled by default.

If the notifier is enabled, it emits the activated() signal whenever a socket event corresponding to its type occurs. If it is disabled, it ignores socket events (the same effect as not creating the socket notifier).

Write notifiers should normally be disabled immediately after the activated() signal has been emitted; see discussion of write notifiers in the class description above.

See also isEnabled() [p. 144] and activated() [p. 144].

int QSocketNotifier::socket () const

Returns the socket identifier specified to the constructor.

See also type() [p. 145].

Type QSocketNotifier::type () const

Returns the socket event type specified to the constructor: QSocketNotifier::Read, QSocketNotifier::Write, or QSocketNotifier::Exception.

See also socket() [p. 145].

QTextIStream Class Reference

The QTextIStream class is a convenience class for input streams.

`#include <qtextstream.h>`

Inherits QTextStream [p. 150].

Public Members

- **QTextIStream** (const QString * s)
- **QTextIStream** (QByteArray ba)
- **QTextIStream** (FILE * f)

Detailed Description

The QTextIStream class is a convenience class for input streams.

For simple tasks code should be simple, so this class is a shorthand to avoid passing the *mode* argument to the normal QTextStream constructors.

This class makes it easy, for example, to write things like this:

```
QString data = "123 456";
int a, b;
QTextIStream(&data) >> a >> b;
```

See also QTextOStream [p. 148], Input/Output and Networking and Text Related Classes.

Member Function Documentation

QTextIStream::QTextIStream (const QString * s)

Constructs a stream to read from the string *s*.

QTextIStream::QTextIStream (QByteArray ba)

Constructs a stream to read from the array *ba*.

QTextIStream::QTextIStream (FILE * f)

Constructs a stream to read from the file *f*.

QTextOStream Class Reference

The QTextOStream class is a convenience class for output streams.

`#include <qtextstream.h>`

Inherits QTextStream [p. 150].

Public Members

- **QTextOStream** (QString * s)
- **QTextOStream** (QByteArray ba)
- **QTextOStream** (FILE * f)

Detailed Description

The QTextOStream class is a convenience class for output streams.

For simple tasks, code should be simple, so this class is a shorthand to avoid passing the *mode* argument to the normal QTextStream constructors.

This makes it easy for example, to write things like this:

```
QString result;
QTextOStream(&result) << "pi = " << 3.14;
```

See also Input/Output and Networking and Text Related Classes.

Member Function Documentation

QTextOStream::QTextOStream (QString * s)

Constructs a stream to write to string *s*.

QTextOStream::QTextOStream (QByteArray ba)

Constructs a stream to write to the array *ba*.

QTextOStream::QTextOStream (FILE * f)

Constructs a stream to write to the file *f*.

QTextStream Class Reference

The QTextStream class provides basic functions for reading and writing text using a QIODevice.

`#include <qtextstream.h>`

Inherited by QTextIStream [p. 146] and QTextOStream [p. 148].

Public Members

- enum **Encoding** { Locale, Latin1, Unicode, UnicodeNetworkOrder, UnicodeReverse, RawUnicode, UnicodeUTF8 }
- void **setEncoding** (Encoding e)
- void **setCodec** (QTextCodec * codec)
- **QTextStream** ()
- **QTextStream** (QIODevice * iod)
- **QTextStream** (QString * str, int filemode)
- QTextStream (QString & str, int filemode) *(obsolete)*
- **QTextStream** (QByteArray a, int mode)
- **QTextStream** (FILE * fh, int mode)
- virtual **~QTextStream** ()
- QIODevice * **device** () const
- void **setDevice** (QIODevice * iod)
- void **unsetDevice** ()
- bool **atEnd** () const
- bool eof () const *(obsolete)*
- QTextStream & **operator>>** (QChar & c)
- QTextStream & **operator>>** (char & c)
- QTextStream & **operator>>** (signed short & i)
- QTextStream & **operator>>** (unsigned short & i)
- QTextStream & **operator>>** (signed int & i)
- QTextStream & **operator>>** (unsigned int & i)
- QTextStream & **operator>>** (signed long & i)
- QTextStream & **operator>>** (unsigned long & i)
- QTextStream & **operator>>** (float & f)
- QTextStream & **operator>>** (double & f)
- QTextStream & **operator>>** (char * s)
- QTextStream & **operator>>** (QString & str)

- QTextStream & **operator>>** (QCString & str)
- QTextStream & **operator<<** (QChar c)
- QTextStream & **operator<<** (char c)
- QTextStream & **operator<<** (signed short i)
- QTextStream & **operator<<** (unsigned short i)
- QTextStream & **operator<<** (signed int i)
- QTextStream & **operator<<** (unsigned int i)
- QTextStream & **operator<<** (signed long i)
- QTextStream & **operator<<** (unsigned long i)
- QTextStream & **operator<<** (float f)
- QTextStream & **operator<<** (double f)
- QTextStream & **operator<<** (const char * s)
- QTextStream & **operator<<** (const QString & s)
- QTextStream & **operator<<** (const QCString & s)
- QTextStream & **operator<<** (void * ptr)
- QTextStream & **readRawBytes** (char * s, uint len)
- QTextStream & **writeRawBytes** (const char * s, uint len)
- QString **readLine** ()
- QString **read** ()
- void **skipWhiteSpace** ()
- int **flags** () const
- int **flags** (int f)
- int **setf** (int bits)
- int **setf** (int bits, int mask)
- int **unsetf** (int bits)
- void **reset** ()
- int **width** () const
- int **width** (int w)
- int **fill** () const
- int **fill** (int f)
- int **precision** () const
- int **precision** (int p)

Detailed Description

The QTextStream class provides basic functions for reading and writing text using a QIODevice.

The text stream class has a functional interface that is very similar to that of the standard C++ iostream class. The difference between iostream and QTextStream is that our stream operates on a QIODevice which is easily subclassed, whereas iostream operates on FILE * pointers which cannot be subclassed.

Qt provides several global functions similar to the ones in iostream:

- bin sets the QTextStream to read/write binary numbers
- oct sets the QTextStream to read/write octal numbers
- dec sets the QTextStream to read/write decimal numbers

- hex sets the QTextStream to read/write hexadecimal numbers
- endl forces a line break
- flush forces the QIODevice to flush any buffered data
- ws eats any available whitespace (on input)
- reset resets the QTextStream to its default mode (see reset())
- qSetW(int) sets the field width as specified with the argument
- qSetFill(int) sets the fill character as specified with the argument
- qSetPrecision(int) sets the precision as specified with the argument

Warning: By default QTextStream will automatically detect whether integers in the stream are in decimal, octal, hexadecimal or binary format when reading from the stream. In particular, a leading '0' signifies octal, i.e. the sequence "0100" will be interpreted as 64.

The QTextStream class reads and writes text; it is not appropriate for dealing with binary data (but QDataStream is).

By default, output of Unicode text (i.e. QString) is done using the local 8-bit encoding. This can be changed using the setEncoding() method. For input, the QTextStream will auto-detect standard Unicode "byte order marked" text files; otherwise the local 8-bit encoding is used.

The QIODevice is set in the constructor, or later using setDevice(). If the end of the input is reached atEnd() returns TRUE. Data can be read into variables of the appropriate type using the operator>>() overloads, or read in its entirety into a single string using read(), or read a line at a time using readLine(). Whitespace can be skipped over using skipWhiteSpace(). You can set flags for the stream using flags() or setf(). The stream also supports width(), precision() and fill(); use reset() to reset the defaults.

See also QDataStream [p. 19], Input/Output and Networking and Text Related Classes.

Member Type Documentation

QTextStream::Encoding

- `QTextStream::Locale`
- `QTextStream::Latin1`
- `QTextStream::Unicode`
- `QTextStream::UnicodeNetworkOrder`
- `QTextStream::UnicodeReverse`
- `QTextStream::RawUnicode`
- `QTextStream::UnicodeUTF8`

Member Function Documentation

QTextStream::QTextStream ()

Constructs a data stream that has no IO device.

QTextStream::QTextStream (QIODevice * iod)

Constructs a text stream that uses the IO device *iod*.

QTextStream::QTextStream (QString * str, int filemode)

Constructs a text stream that operates on the Unicode QString, *str*, through an internal device. The *filemode* argument is passed to the device's open() function; see QIODevice::mode().

If you set an encoding or codec with setEncoding() or setCodec(), this setting is ignored for text streams that operate on QString.

Example:

```
QString str;
QTextStream ts( &str, IO_WriteOnly );
ts << "pi = " << 3.14; // str == "pi = 3.14"
```

Writing data to the text stream will modify the contents of the string. The string will be expanded when data is written beyond the end of the string. Note that the string will not be truncated:

```
QString str = "pi = 3.14";
QTextStream ts( &str, IO_WriteOnly );
ts <<  "2+2 = " << 2+2; // str == "2+2 = 414"
```

Note that because QString is Unicode, you should not use readRawBytes() or writeRawBytes() on such a stream.

QTextStream::QTextStream (QString & str, int filemode)

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

This constructor is equivalent to the constructor taking a QString* parameter.

QTextStream::QTextStream (QByteArray a, int mode)

Constructs a text stream that operates on the byte array, *a*, through an internal QBuffer device. The *mode* argument is passed to the device's open() function; see QIODevice::mode().

Example:

```
QByteArray array;
QTextStream ts( array, IO_WriteOnly );
ts << "pi = " << 3.14 << '\0'; // array == "pi = 3.14"
```

Writing data to the text stream will modify the contents of the array. The array will be expanded when data is written beyond the end of the string.

Same example, using a QBuffer:

```
QByteArray array;
QBuffer buf( array );
```

```
buf.open( IO_WriteOnly );
QTextStream ts( &buf );
ts << "pi = " << 3.14 << '\0'; // array == "pi = 3.14"
buf.close();
```

QTextStream::QTextStream (FILE * fh, int mode)

Constructs a text stream that operates on an existing file handle *fh* through an internal QFile device. The *mode* argument is passed to the device's open() function; see QIODevice::mode().

Note that if you create a QTextStream `cout` or another name that is also used for another variable of a different type, some linkers may confuse the two variables, which will often cause crashes.

QTextStream::~QTextStream () [virtual]

Destroys the text stream.

The destructor does not affect the current IO device.

bool QTextStream::atEnd () const

Returns TRUE if the IO device has reached the end position (end of the stream or file) or if there is no IO device set; otherwise returns FALSE.

See also QIODevice::atEnd() [p. 79].

Examples: addressbook/centralwidget.cpp and grapher/grapher.cpp.

QIODevice * QTextStream::device () const

Returns the IO device currently set.

See also setDevice() [p. 160] and unsetDevice() [p. 161].

bool QTextStream::eof () const

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

This function has been renamed to atEnd().

See also QIODevice::atEnd() [p. 79].

int QTextStream::fill () const

Returns the fill character. The default value is ' ' (space).

int QTextStream::fill (int f)

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

Sets the fill character to *f*. Returns the previous fill character.

int QTextStream::flags () const

Returns the current stream flags. The default value is 0.

The meanings of the flags are:

- *skipws* - Not currently used; whitespace always skipped
- *left* - Numeric fields are left-aligned
- *right* - Not currently used (by default, numerics are right-aligned)
- *internal* - Puts any padding spaces between +/- and value
- *bin* - Output *and* input only in binary
- *oct* - Output *and* input only in octal
- *dec* - Output *and* input only in decimal
- *hex* - Output *and* input only in hexadecimal
- *showbase* - Annotates numeric outputs with 0b, 0, or 0x if in *bin*, *oct*, or *hex* format
- *showpoint* - Not currently used
- *uppercase* - Uses 0B and 0X rather than 0b and 0x
- *showpos* - Shows + for positive numeric values
- *scientific* - Uses scientific notation for floating point values
- *fixed* - Uses fixed-point notation for floating point values

Note that unless *bin*, *oct*, *dec*, or *hex* is set, the input base is octal if the value starts with 0, hexadecimal if it starts with 0x, binary if it starts with 0b, and decimal otherwise.

See also setf() [p. 161] and unsetf() [p. 161].

int QTextStream::flags (int f)

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

Sets the stream flags to *f*. Returns the previous stream flags.

See also setf() [p. 161] and unsetf() [p. 161].

QTextStream & QTextStream::operator<< (QChar c)

Writes character `char` to the stream and returns a reference to the stream.

The character *c* is assumed to be Latin1 encoded independent of the Encoding set for the QTextStream.

QTextStream & QTextStream::operator<< (char c)

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

Writes character *c* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (signed short i)

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

Writes a `short` integer *i* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (unsigned short i)

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

Writes an `unsigned short` integer *i* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (signed int i)

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

Writes an `int` *i* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (unsigned int i)

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

Writes an `unsigned int` *i* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (signed long i)

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

Writes a `long int` *i* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (unsigned long i)

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

Writes an `unsigned long int` *i* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (float f)

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

Writes a `float` *f* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (double f)

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

Writes a `double` *f* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (const char * s)

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

Writes a string to the stream and returns a reference to the stream.

The string *s* is assumed to be Latin1 encoded independent of the Encoding set for the QTextStream.

QTextStream & QTextStream::operator<< (const QString & s)

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

Writes *s* to the stream and returns a reference to the stream.

QTextStream & QTextStream::operator<< (const QCString & s)

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

Writes *s* to the stream and returns a reference to the stream.

The string *s* is assumed to be Latin1 encoded independent of the Encoding set for the QTextStream.

QTextStream & QTextStream::operator<< (void * ptr)

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

Writes a pointer to the stream and returns a reference to the stream.

The *ptr* is output as an unsigned long hexadecimal integer.

QTextStream & QTextStream::operator>> (QChar & c)

Reads a char *c* from the stream and returns a reference to the stream. Note that whitespace is *not* skipped.

QTextStream & QTextStream::operator>> (char & c)

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

Reads a char *c* from the stream and returns a reference to the stream. Note that whitespace is skipped.

QTextStream & QTextStream::operator>> (signed short & i)

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

Reads a signed `short` integer *i* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (unsigned short & i)

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

Reads an unsigned `short` integer *i* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (signed int & i)

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

Reads a signed `int` *i* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (unsigned int & i)

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

Reads an unsigned `int` *i* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (signed long & i)

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

Reads a signed `long` int *i* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (unsigned long & i)

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

Reads an unsigned `long` int *i* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (float & f)

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

Reads a `float` *f* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (double & f)

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

Reads a `double` *f* from the stream and returns a reference to the stream. See flags() for an explanation of the expected input format.

QTextStream & QTextStream::operator>> (char * s)

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

Reads a "word" from the stream into *s* and returns a reference to the stream.

A word consists of characters for which isspace() returns FALSE.

QTextStream & QTextStream::operator>> (QString & str)

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

Reads a "word" from the stream into *str* and returns a reference to the stream.

A word consists of characters for which isspace() returns FALSE.

QTextStream & QTextStream::operator>> (QCString & str)

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

Reads a "word" from the stream into *str* and returns a reference to the stream.

A word consists of characters for which isspace() returns FALSE.

int QTextStream::precision () const

Returns the precision. The default value is 6.

int QTextStream::precision (int p)

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

Sets the precision to *p*. Returns the previous precision setting.

QString QTextStream::read ()

Reads the entire stream and returns a string containing the text.

See also QIODevice::readLine() [p. 83].

Examples: action/application.cpp, application/application.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp, qdir/qdir.cpp and qwerty/qwerty.cpp.

QString QTextStream::readLine ()

Reads a line from the stream and returns a string containing the text.

The returned string does not contain any trailing newline or carriage return. Note that this is different from QIODevice::readLine(), which does not strip the newline at the end of the line.

On EOF you will get a QString that is null. On reading an empty line the returned QString is empty but not null.

See also QIODevice::readLine() [p. 83].

Example: addressbook/centralwidget.cpp.

QTextStream & QTextStream::readRawBytes (char * s, uint len)

Reads *len* bytes from the stream into *s* and returns a reference to the stream.

The buffer *s* must be preallocated.

Note that no encoding is done by this function.

Warning: The behavior of this function is undefined unless the stream's encoding is set to Unicode or Latin1.

See also QIODevice::readBlock() [p. 83].

void QTextStream::reset ()

Resets the text stream.

- All flags are set to 0.
- The field width is set to 0.
- The fill character is set to ' ' (space).
- The precision is set to 6.

See also setf() [p. 161], width() [p. 162], fill() [p. 154] and precision() [p. 159].

void QTextStream::setCodec (QTextCodec * codec)

Sets the codec for this stream to *codec*. Will not try to autodetect Unicode.

Note that this function should be called before any data is read to/written from the stream.

See also setEncoding() [p. 160].

Example: qwerty/qwerty.cpp.

void QTextStream::setDevice (QIODevice * iod)

Sets the IO device to *iod*.

See also device() [p. 154] and unsetDevice() [p. 161].

void QTextStream::setEncoding (Encoding e)

Sets the encoding of this stream to *e*, where *e* is one of:

- Locale - Uses local file format (Latin1 if locale is not set), but autodetecting Unicode(utf16) on input.
- Unicode - Uses Unicode(utf16) for input and output. Output will be written in the order most efficient for the current platform (i.e. the order used internally in QString).
- UnicodeUTF8 Using Unicode(utf8) for input and output. If you use it for input it will autodetect utf16 and use it instead of utf8.
- Latin1 - ISO-8859-1. Will not autodetect utf16.

- UnicodeNetworkOrder - Uses network order Unicode(utf16) for input and output. Useful when reading Unicode data that does not start with the byte order marker.
- UnicodeReverse - Uses reverse network order Unicode(utf16) for input and output. Useful when reading Unicode data that does not start with the byte order marker or when writing data that should be read by buggy Windows applications.
- RawUnicode - Like Unicode, but does not write the byte order marker nor does it auto-detect the byte order. Useful only when writing to nonpersistent storage used by a single process.

Locale and all Unicode encodings, except RawUnicode, will look at the first two bytes in an input stream to determine the byte order. The initial byte order marker will be stripped off before data is read.

Note that this function should be called before any data is read to or written from the stream.

See also setCodec() [p. 160].

Examples: network/httpd/httpd.cpp and qwerty/qwerty.cpp.

int QTextStream::setf (int bits)

Sets the stream flag bits *bits*. Returns the previous stream flags.

Equivalent to `flags( flags() | bits )`.

See also unsetf() [p. 161].

int QTextStream::setf (int bits, int mask)

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

Sets the stream flag bits *bits* with a bit mask *mask*. Returns the previous stream flags.

Equivalent to `flags( (flags() & ~mask) | (bits & mask) )`.

See also unsetf() [p. 161].

void QTextStream::skipWhiteSpace ()

Positions the read pointer at the first non-whitespace character.

void QTextStream::unsetDevice ()

Unsets the IO device. Equivalent to setDevice(0).

See also device() [p. 154] and setDevice() [p. 160].

int QTextStream::unsetf (int bits)

Clears the stream flag bits *bits*. Returns the previous stream flags.

Equivalent to `flags( flags() & ~mask )`.

See also setf() [p. 161].

int QTextStream::width () const

Returns the field width. The default value is 0.

int QTextStream::width (int w)

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

Sets the field width to *w*. Returns the previous field width.

QTextStream & QTextStream::writeRawBytes (const char * s, uint len)

Writes the *len* bytes from *s* to the stream and returns a reference to the stream.

Note that no encoding is done by this function.

See also QIODevice::writeBlock() [p. 85].

QUrl Class Reference

The QUrl class provides a URL parser and simplifies working with URLs.

This class is part of the **network** module.

`#include <qurl.h>`

Inherited by QUrlOperator [p. 179].

Public Members

- **QUrl** ()
- **QUrl** (const QString & url)
- **QUrl** (const QUrl & url)
- **QUrl** (const QUrl & url, const QString & relUrl, bool checkSlash = FALSE)
- virtual **~QUrl** ()
- QString **protocol** () const
- virtual void **setProtocol** (const QString & protocol)
- QString **user** () const
- virtual void **setUser** (const QString & user)
- bool **hasUser** () const
- QString **password** () const
- virtual void **setPassword** (const QString & pass)
- bool **hasPassword** () const
- QString **host** () const
- virtual void **setHost** (const QString & host)
- bool **hasHost** () const
- int **port** () const
- virtual void **setPort** (int port)
- bool **hasPort** () const
- QString **path** (bool correct = TRUE) const
- virtual void **setPath** (const QString & path)
- bool **hasPath** () const
- virtual void **setEncodedPathAndQuery** (const QString & pathAndQuery)
- QString **encodedPathAndQuery** ()
- virtual void **setQuery** (const QString & txt)
- QString **query** () const
- QString **ref** () const

- virtual void **setRef** (const QString & txt)
- bool **hasRef** () const
- bool **isValid** () const
- bool **isLocalFile** () const
- virtual void **addPath** (const QString & pa)
- virtual void **setFileName** (const QString & name)
- QString **fileName** () const
- QString **dirPath** () const
- QUrl & **operator=** (const QUrl & url)
- QUrl & **operator=** (const QString & url)
- bool **operator==** (const QUrl & url) const
- bool **operator==** (const QString & url) const
- **operator QString** () const
- virtual QString **toString** (bool encodedPath = FALSE, bool forcePrependProtocol = TRUE) const
- virtual bool **cdUp** ()

Static Public Members

- void **decode** (QString & url)
- void **encode** (QString & url)
- bool **isRelativeUrl** (const QString & url)

Protected Members

- virtual void **reset** ()
- virtual bool **parse** (const QString & url)

Detailed Description

The QUrl class provides a URL parser and simplifies working with URLs.

The QUrl class is provided for simple work with URLs. It can parse, decode, encode, etc.

QUrl works with the decoded path and encoded query in turn.

Example:

```
http://www.trolltech.com:80/cgi-bin/test%20me.pl?cmd=Hello%20you
```

Function	Returns
protocol()	"http"
host()	"www.trolltech.com"
port()	80
path()	"cgi-bin"
fileName()	"test me.pl"
query()	"cmd=Hello%20you"

Example:

http://doc.trolltech.com/qdockarea.html#lines

Function	Returns
protocol()	"http"
host()	"doc.trolltech.com"
fileName()	"qdockarea.html"
ref()	"lines"

The individual parts of a URL can be set with setProtocol(), setHost(), setPort(), setPath(), setFileName(), setRef() and setQuery(). A URL could contain, for example, an ftp address which requires a user name and password; these can be set with setUser() and setPassword().

Because path is always encoded internally you must not use "%00" in the path, although this it is okay (but not recommended) for the query.

QUrl is normally used like this:

```
QUrl u( "http://www.trolltech.com" );
// or
QUrl u( "file:/home/myself/Mail", "Inbox" );
```

You can then access and manipulate the various parts of the URL.

To make it easy to work with QUrls and QStrings, QUrl implements the necessary cast and assignment operators so you can do following:

```
QUrl u( "http://www.trolltech.com" );
QString s = u;
// or
QString s( "http://www.trolltech.com" );
QUrl u( s );
```

Use the static functions, encode() and decode() to encode or decode a URL in a string. (They operate on the string in-place.) The isRelativeUrl() static function returns TRUE if the given string is a relative URL.

If you want to use an URL to work on a hierarchical structure (e.g. a local or remote filesystem), you might want to use the subclass QUrlOperator.

See also QUrlOperator [p. 179], Input/Output and Networking and Miscellaneous Classes.

Member Function Documentation

QUrl::QUrl ()

Constructs an empty URL that is invalid.

QUrl::QUrl (const QString & url)

Constructs a URL by parsing the string *url*.

If you pass a string like "/home/qt", the "file" protocol is assumed.

QUrl::QUrl (const QUrl & url)

Copy constructor. Copies the data of *url*.

QUrl::QUrl (const QUrl & url, const QString & relUrl, bool checkSlash = FALSE)

Constructs an URL taking *url* as the base (context) and *relUrl* as a relative URL to *url*. If *relUrl* is not relative, *relUrl* is taken as the new URL.

For example, the path of

```
QUrl u( "ftp://ftp.trolltech.com/qt/source", "qt-2.1.0.tar.gz" );
```

will be "/qt/srource/qt-2.1.0.tar.gz".

On the other hand,

```
QUrl u( "ftp://ftp.trolltech.com/qt/source", "/usr/local" );
```

will result in a new URL, "ftp://ftp.trolltech.com/usr/local", because "/usr/local" isn't relative.

Similarly,

```
QUrl u( "ftp://ftp.trolltech.com/qt/source", "file:/usr/local" );
```

will result in a new URL, with "/usr/local" as the path and "file" as the protocol.

Normally it is expected that the path of *url* points to a directory, even if the path has no slash at the end. But if you want the constructor to handle the last part of the path as a file name if there is no slash at the end, and to let it be replaced by the file name of *relUrl* (if it contains one), set *checkSlash* to TRUE.

QUrl::~QUrl () [virtual]

Destructor.

void QUrl::addPath (const QString & pa) [virtual]

Adds the path *pa* to the path of the URL.

See also setPath() [p. 170] and hasPath() [p. 167].

bool QUrl::cdUp () [virtual]

Changes the directory to one directory up.

See also setPath() [p. 170].

void QUrl::decode (QString & url) [static]

Decodes the string *url* in-place.

See also encode() [p. 167].

QString QUrl::dirPath () const

Returns the directory path of the URL. This is the part of the path of the URL without the fileName(). See the documentation of fileName() for a discussion of what is handled as file name and what is handled as directory path.

See also setPath() [p. 170] and hasPath() [p. 167].

Example: network/networkprotocol/nntp.cpp.

void QUrl::encode (QString & url) [static]

Encodes the string *url* in-place.

See also decode() [p. 166].

QString QUrl::encodedPathAndQuery ()

Returns the encoded path and query.

See also decode() [p. 166].

QString QUrl::fileName () const

Returns the file name of the URL. If the path of the URL doesn't have a slash at the end, the part between the last slash and the end of the path string is considered to be the file name. If the path has a slash at the end, an empty string is returned here.

See also setFileName() [p. 170].

Example: network/networkprotocol/nntp.cpp.

bool QUrl::hasHost () const

Returns TRUE if the URL contains a hostname; otherwise returns FALSE.

See also setHost() [p. 170].

bool QUrl::hasPassword () const

Returns TRUE if the URL contains a password; otherise returns FALSE.

Warning: Passwords passed in URLs are normally *insecure*; this is due to the mechanism, not because of Qt.

See also setPassword() [p. 170] and setUser() [p. 171].

bool QUrl::hasPath () const

Returns TRUE if the URL contains a path, otherwise FALSE.

See also path() [p. 169] and setPath() [p. 170].

bool QUrl::hasPort () const

Returns TRUE if the URL contains a port; otherwise returns FALSE.

See also setPort() [p. 171].

bool QUrl::hasRef () const

Returns TRUE if the URL has a reference; otherwise returns FALSE.

See also setRef() [p. 171].

bool QUrl::hasUser () const

Returns TRUE if the URL contains a username; otherwise returns FALSE.

See also setUser() [p. 171] and setPassword() [p. 170].

QString QUrl::host () const

Returns the hostname of the URL.

See also setHost() [p. 170] and hasHost() [p. 167].

bool QUrl::isLocalFile () const

Returns TRUE if the URL is a local file, otherwise FALSE.

Example: qdir/qdir.cpp.

bool QUrl::isRelativeUrl (const QString & url) [static]

Returns TRUE if *url* is relative; otherwise returns FALSE.

bool QUrl::isValid () const

Returns TRUE if the URL is valid; otherwise returns FALSE. A URL is invalid if it cannot be parsed, for example.

QUrl::operator QString () const

Composes a string version of the URL and returns it.

See also QUrl::toString() [p. 171].

QUrl & QUrl::operator= (const QUrl & url)

Assigns the data of *url* to this class.

QUrl & QUrl::operator= (const QString & url)

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

Parses *url* and assigns the resulting data to this class.

If you pass a string like "/home/qt" the "file" protocol will be assumed.

bool QUrl::operator== (const QUrl & url) const

Compares this URL with *url* and returns TRUE if they are equal; otherwise returns FALSE.

bool QUrl::operator== (const QString & url) const

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

Compares this URL with *url*. *url* is parsed first. Returns TRUE if *url* is equal to this url; otherwise returns FALSE.

bool QUrl::parse (const QString & url) [virtual protected]

Parses the *url*.

QString QUrl::password () const

Returns the password of the URL.

Warning: Passwords passed in URLs are normally *insecure*; this is due to the mechanism, not because of Qt.

See also setPassword() [p. 170] and setUser() [p. 171].

QString QUrl::path (bool correct = TRUE) const

Returns the path of the URL. If *correct* is TRUE, the path is cleaned (deals with too many or too few slashes, cleans things like "/../..", etc). Otherwise path() returns exactly the path that was parsed or set.

See also setPath() [p. 170] and hasPath() [p. 167].

Example: qdir/qdir.cpp.

int QUrl::port () const

Returns the port of the URL or -1 if no port has been set.

See also setPort() [p. 171].

QString QUrl::protocol () const

Returns the protocol of the URL. Typically, "file", "http", or "ftp", etc.

See also setProtocol() [p. 171].

QString QUrl::query () const

Returns the (encoded) query of the URL.

See also setQuery() [p. 171] and decode() [p. 166].

QString QUrl::ref () const

Returns the (encoded) reference of the URL.

See also setRef() [p. 171], hasRef() [p. 168] and decode() [p. 166].

void QUrl::reset () [virtual protected]

Resets all parts of the URL to their default values and invalidates it.

void QUrl::setEncodedPathAndQuery (const QString & pathAndQuery) [virtual]

Parses *pathAndQuery* for a path and query and sets those values. The whole string has to be encoded.

See also encode() [p. 167].

void QUrl::setFileName (const QString & name) [virtual]

Sets the file name of the URL to *name*. If this URL contains a fileName(), the original file name is replaced by *name*.

See the documentation of fileName() for a more detailed discussion of what is handled as file name and what is handled as a directory path.

See also fileName() [p. 167].

void QUrl::setHost (const QString & host) [virtual]

Sets the hostname of the URL to *host*.

See also host() [p. 168] and hasHost() [p. 167].

void QUrl::setPassword (const QString & pass) [virtual]

Sets the password of the URL to *pass*.

Warning: Passwords passed in URLs are normally *insecure*; this is due to the mechanism, not because of Qt.

See also password() [p. 169] and setUser() [p. 171].

void QUrl::setPath (const QString & path) [virtual]

Sets the path of the URL to *path*.

See also path() [p. 169] and hasPath() [p. 167].

void QUrl::setPort (int port) [virtual]

Sets the port of the URL to *port*.

See also port() [p. 169].

void QUrl::setProtocol (const QString & protocol) [virtual]

Sets the protocol of the URL to *protocol*. Typically, "file", "http", or "ftp", etc.

See also protocol() [p. 169].

void QUrl::setQuery (const QString & txt) [virtual]

Sets the query of the URL to *txt*. *txt* must be encoded.

See also query() [p. 170] and encode() [p. 167].

void QUrl::setRef (const QString & txt) [virtual]

Sets the reference of the URL to *txt*. *txt* must be encoded.

See also ref() [p. 170], hasRef() [p. 168] and encode() [p. 167].

void QUrl::setUser (const QString & user) [virtual]

Sets the username of the URL to *user*.

See also user() [p. 171] and setPassword() [p. 170].

QString QUrl::toString (bool encodedPath = FALSE, bool forcePrependProtocol = TRUE) const [virtual]

Composes a string version of the URL and returns it. If *encodedPath* is TRUE the path in the returned string is encoded. If *forcePrependProtocol* is TRUE and *encodedPath* looks like a local filename, the "file:/" protocol is also prepended.

See also encode() [p. 167] and decode() [p. 166].

QString QUrl::user () const

Returns the username of the URL.

See also setUser() [p. 171] and setPassword() [p. 170].

QUrlInfo Class Reference

The QUrlInfo class stores information about URLs.

`#include <qurlinfo.h>`

Public Members

- **QUrlInfo** ()
- **QUrlInfo** (const QUrlOperator & path, const QString & file)
- **QUrlInfo** (const QUrlInfo & ui)
- **QUrlInfo** (const QString & name, int permissions, const QString & owner, const QString & group, uint size, const QDateTime & lastModified, const QDateTime & lastRead, bool isDir, bool isFile, bool isSymLink, bool isWritable, bool isReadable, bool isExecutable)
- **QUrlInfo** (const QUrl & url, int permissions, const QString & owner, const QString & group, uint size, const QDateTime & lastModified, const QDateTime & lastRead, bool isDir, bool isFile, bool isSymLink, bool isWritable, bool isReadable, bool isExecutable)
- QUrlInfo & **operator=** (const QUrlInfo & ui)
- virtual **~QUrlInfo** ()
- virtual void **setName** (const QString & name)
- virtual void **setDir** (bool b)
- virtual void **setFile** (bool b)
- virtual void **setSymLink** (bool b)
- virtual void **setOwner** (const QString & s)
- virtual void **setGroup** (const QString & s)
- virtual void **setSize** (uint s)
- virtual void **setWritable** (bool b)
- virtual void **setReadable** (bool b)
- virtual void **setPermissions** (int p)
- virtual void **setLastModified** (const QDateTime & dt)
- bool **isValid** () const
- QString **name** () const
- int **permissions** () const
- QString **owner** () const
- QString **group** () const
- uint **size** () const
- QDateTime **lastModified** () const
- QDateTime **lastRead** () const

- bool **isDir** () const
- bool **isFile** () const
- bool **isSymLink** () const
- bool **isWritable** () const
- bool **isReadable** () const
- bool **isExecutable** () const
- bool **operator==** (const QUrlInfo & i) const

Static Public Members

- bool **greaterThan** (const QUrlInfo & i1, const QUrlInfo & i2, int sortBy)
- bool **lessThan** (const QUrlInfo & i1, const QUrlInfo & i2, int sortBy)
- bool **equal** (const QUrlInfo & i1, const QUrlInfo & i2, int sortBy)

Detailed Description

The QUrlInfo class stores information about URLs.

This class is just a container for storing information about URLs, which is why all information must be passed in the constructor.

Unless you're reimplementing a network protocol you're unlikely to create QUrlInfo objects yourself, but you may get QUrlInfo objects from functions, e.g. QUrlOperator::info().

See also Input/Output and Networking and Miscellaneous Classes.

Member Function Documentation

QUrlInfo::QUrlInfo ()

Constructs an invalid QUrlInfo object with default values.

See also isValid() [p. 175].

QUrlInfo::QUrlInfo (const QUrlOperator & path, const QString & file)

Constructs a QUrlInfo object with information about the file *file* in the *path*. It tries to find the information about the *file* in the QUrlOperator *path*.

If the information is not found, this constructor creates an invalid QUrlInfo, i.e. isValid() returns FALSE. You should always check if the URL info is valid before relying on the return values of any getter functions.

If *file* is empty, it defaults to the actual directory of the QUrlOperator *path*.

See also isValid() [p. 175] and QUrlOperator::info() [p. 184].

QUrlInfo::QUrlInfo (const QUrlInfo & ui)

Copy constructor, copies *ui* to this URL info object.

QUrlInfo::QUrlInfo (const QString & name, int permissions, const QString & owner, const QString & group, uint size, const QDateTime & lastModified, const QDateTime & lastRead, bool isDir, bool isFile, bool isSymLink, bool isWritable, bool isReadable, bool isExecutable)

Constructs a QUrlInfo object by specifying all the URL's information. The information that is passed is the *name*, the *permissions*, the *owner* and *group*, as well as the *size*, *lastModified* date/time and *lastRead* date/time. Flags are also passed, specifically, *isDir*, *isFile*, *isSymLink*, *isWritable*, *isReadable* and *isExecutable*.

QUrlInfo::QUrlInfo (const QUrl & url, int permissions, const QString & owner, const QString & group, uint size, const QDateTime & lastModified, const QDateTime & lastRead, bool isDir, bool isFile, bool isSymLink, bool isWritable, bool isReadable, bool isExecutable)

Constructs a QUrlInfo object by specifying all the URL's information. The information that is passed is the *url*, the *permissions*, the *owner* and *group*, as well as the *size*, *lastModified* date/time and *lastRead* date/time. Flags are also passed, specifically, *isDir*, *isFile*, *isSymLink*, *isWritable*, *isReadable* and *isExecutable*.

QUrlInfo::~QUrlInfo () [virtual]

Destroys the URL ifno object. The QUrlOperator object to which this URL referred is not affected.

bool QUrlInfo::equal (const QUrlInfo & i1, const QUrlInfo & i2, int sortBy) [static]

Returns TRUE if *i1* equals to *i2*; otherwise returns FALSE. The objects are compared by the value, which is specified by *sortBy*. This must be one of QDir::Name, QDir::Time or QDir::Size.

bool QUrlInfo::greaterThan (const QUrlInfo & i1, const QUrlInfo & i2, int sortBy) [static]

Returns TRUE if *i1* is greater than *i2*; otherwise returns FALSE. The objects are compared by the value, which is specified by *sortBy*. This must be one of QDir::Name, QDir::Time or QDir::Size.

QString QUrlInfo::group () const

Returns the group of the URL.

See also isValid() [p. 175].

bool QUrlInfo::isDir () const

Returns TRUE if the URL is a directory; otherwise returns FALSE.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

bool QUrlInfo::isExecutable () const

Returns TRUE if the URL is executable; otherwise returns FALSE.

See also isValid() [p. 175].

bool QUrlInfo::isFile () const

Returns TRUE if the URL is a file; otherwise returns FALSE.

See also isValid() [p. 175].

bool QUrlInfo::isReadable () const

Returns TRUE if the URL is readable; otherwise returns FALSE.

See also isValid() [p. 175].

bool QUrlInfo::isSymLink () const

Returns TRUE if the URL is a symbolic link; otherwise returns FALSE.

See also isValid() [p. 175].

bool QUrlInfo::isValid () const

Returns TRUE if the URL info is valid; otherwise returns FALSE. Valid means that the QUrlInfo contains real information. E.g., a call to QUrlOperator::info() might return a an invalid QUrlInfo, if no information about the requested entry is available.

You should always check if the URL info is valid before relying on the values.

bool QUrlInfo::isWritable () const

Returns TRUE if the URL is writable; otherwise returns FALSE.

See also isValid() [p. 175].

QDateTime QUrlInfo::lastModified () const

Returns the last modification date of the URL.

See also isValid() [p. 175].

QDateTime QUrlInfo::lastRead () const

Returns the date when the URL was read the last time.

See also isValid() [p. 175].

bool QUrlInfo::lessThan (const QUrlInfo & i1, const QUrlInfo & i2, int sortBy) [static]

Returns TRUE if *i1* is less than *i2*; otherwise returns FALSE. The objects are compared by the value, which is specified by *sortBy*. This must be one of QDir::Name, QDir::Time or QDir::Size.

QString QUrlInfo::name () const

Returns the file name of the URL.

See also isValid() [p. 175].

Examples: network/ftpclient/ftpmainwindow.cpp and network/ftpclient/ftpview.cpp.

QUrlInfo & QUrlInfo::operator= (const QUrlInfo & ui)

Assigns the values of *ui* to this QUrlInfo object.

bool QUrlInfo::operator== (const QUrlInfo & i) const

Compares this QUrlInfo with *i* and returns TRUE if they are equal; otherwise returns FALSE.

QString QUrlInfo::owner () const

Returns the owner of the URL.

See also isValid() [p. 175].

int QUrlInfo::permissions () const

Returns the permissions of the URL.

See also isValid() [p. 175].

void QUrlInfo::setDir (bool b) [virtual]

If *b* is TRUE then the URL is set to be a directory; if \b is FALSE then the URL is set not to be a directory (which normally means it is a file). (Note that a URL can refer both a file and a directory even though most file systems do not support this duality.)

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

void QUrlInfo::setFile (bool b) [virtual]

If *b* is TRUE then the URL is set to be a file; if \b is FALSE then the URL is set not to be a file (which normally means it is a directory). (Note that a URL can refer both a file and a directory even though most file systems do not support this duality.)

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

void QUrlInfo::setGroup (const QString & s) [virtual]

Specifies that the owning group of the URL is called *s*.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

void QUrlInfo::setLastModified (const QDateTime & dt) [virtual]

Specifies that the object the URL refers to was last modified at *dt*.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

void QUrlInfo::setName (const QString & name) [virtual]

Sets the name of the URL to *name*. The name is the full text, for example, "http://doc.trolltech.com/qurlinfo.html".

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

void QUrlInfo::setOwner (const QString & s) [virtual]

Specifies that the owner of the URL is called *s*.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

void QUrlInfo::setPermissions (int p) [virtual]

Specifies that the URL has access permision *p*.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

void QUrlInfo::setReadable (bool b) [virtual]

Specifies that the URL is readable if *b* is TRUE and not readable if *b* is FALSE.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

void QUrlInfo::setSize (uint s) [virtual]

Specifies that the URL has size *s*.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

void QUrlInfo::setSymLink (bool b) [virtual]

Specifies that the URL refers to a symbolic link if *b* is TRUE and that it does not if *b* is FALSE.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

void QUrlInfo::setWritable (bool b) [virtual]

Specifies that the URL is writable if *b* is TRUE and not writable if *b* is FALSE.

If you call this function for an invalid URL info, this function turns it into a valid one.

See also isValid() [p. 175].

Example: network/networkprotocol/nntp.cpp.

uint QUrlInfo::size () const

Returns the size of the URL.

See also isValid() [p. 175].

QUrlOperator Class Reference

The QUrlOperator class provides common operations on URLs.

This class is part of the **network** module.

`#include <qurloperator.h>`

Inherits QObject [Additional Functionality with Qt] and QUrl [p. 163].

Public Members

- **QUrlOperator** ()
- **QUrlOperator** (const QString & url)
- **QUrlOperator** (const QUrlOperator & url)
- **QUrlOperator** (const QUrlOperator & url, const QString & relUrl, bool checkSlash = FALSE)
- virtual **~QUrlOperator** ()
- virtual const QNetworkOperation * **listChildren** ()
- virtual const QNetworkOperation * **mkdir** (const QString & dirname)
- virtual const QNetworkOperation * **remove** (const QString & filename)
- virtual const QNetworkOperation * **rename** (const QString & oldname, const QString & newname)
- virtual const QNetworkOperation * **get** (const QString & location = QString::null)
- virtual const QNetworkOperation * **put** (const QByteArray & data, const QString & location = QString::null)
- virtual QPtrList<QNetworkOperation> **copy** (const QString & from, const QString & to, bool move = FALSE, bool toPath = TRUE)
- virtual void **copy** (const QStringList & files, const QString & dest, bool move = FALSE)
- virtual bool **isDir** (bool * ok = 0)
- virtual void **setNameFilter** (const QString & nameFilter)
- QString **nameFilter** () const
- virtual QUrlInfo **info** (const QString & entry) const
- virtual void **stop** ()

Signals

- void **newChildren** (const QValueList<QUrlInfo> & i, QNetworkOperation * op)
- void **finished** (QNetworkOperation * op)
- void **start** (QNetworkOperation * op)
- void **createdDirectory** (const QUrlInfo & i, QNetworkOperation * op)

- void **removed** (QNetworkOperation * op)
- void **itemChanged** (QNetworkOperation * op)
- void **data** (const QByteArray & data, QNetworkOperation * op)
- void **dataTransferProgress** (int bytesDone, int bytesTotal, QNetworkOperation * op)
- void **startedNextCopy** (const QPtrList<QNetworkOperation> & lst)
- void **connectionStateChanged** (int state, const QString & data)

Protected Members

- virtual void **clearEntries** ()
- void **getNetworkProtocol** ()
- void **deleteNetworkProtocol** ()

Related Functions

- void **qInitNetworkProtocols** ()

Detailed Description

The QUrlOperator class provides common operations on URLs.

This class operates on hierarchical structures (such as filesystems) using URLs. Its API allows all the common operations (listing children, removing children, renaming, etc.). The class uses the functionality of registered network protocols to perform these operations. Depending of the protocol of the URL, it uses an appropriate network protocol class for the operations. Each of the operation functions of QUrlOperator creates a QNetworkOperation object that describes the operation and puts it into the operation queue for the network protocol used. If no suitable protocol could be found (because no implementation of the necessary network protocol is registered), the URL operator emits errors. Not every protocol supports every operation, but error handling deals with this problem.

A QUrlOperator can be used like this e.g. for downloading a file:

```
QUrlOperator op;
op.copy( QString("ftp://ftp.trolltech.com/qt/source/qt-2.1.0.tar.gz"),
         "file:/tmp" );
```

You will also need to connect to some signals of the QUrlOperator to be informed of success, errors, progress and more things.

Of course an implementation for the FTP protocol has to be registered for this example to work, e.g. QFtp. You can use the function qInitNetworkProtocols() to register all the network protocols that are shipped with the Qt network extension (at the moment FTP, HTTP and local file system are supported).

For more information about the Qt Network Architecture see the Qt Network Documentation.

See also QNetworkProtocol [p. 99], QNetworkOperation [p. 96], Input/Output and Networking and Miscellaneous Classes.

Member Function Documentation

QUrlOperator::QUrlOperator ()

Constructs a QUrlOperator with an empty (i.e. invalid) URL.

QUrlOperator::QUrlOperator (const QString & url)

Constructs a QUrlOperator using *url* and parses this string.

If you pass strings like "/home/qt" the "file" protocol is assumed.

QUrlOperator::QUrlOperator (const QUrlOperator & url)

Constructs a copy of *url*.

QUrlOperator::QUrlOperator (const QUrlOperator & url, const QString & relUrl, bool checkSlash = FALSE)

Constructs a QUrlOperator. The URL on which this QUrlOperator operates is constructed out of the arguments *url*, *relUrl* and *checkSlash*: see the corresponding QUrl constructor for an explanation of these arguments.

QUrlOperator::~QUrlOperator () [virtual]

Destructor.

void QUrlOperator::clearEntries () [virtual protected]

Clears the cache of children.

void QUrlOperator::connectionStateChanged (int state, const QString & data) [signal]

This signal is emitted whenever the state of the connection of the network protocol of the URL operator changes. *state* describes the new state, which is a QNetworkProtocol::ConnectionState value.

data is a string that describes the change of the connection. This can be used to display a message to the user.

QPtrList<QNetworkOperation> QUrlOperator::copy (const QString & from, const QString & to, bool move = FALSE, bool toPath = TRUE) [virtual]

Copies the file *from* to *to*. If *move* is TRUE, the file is moved (copied and removed). *from* must point to a file and *to* points to a directory (into which *from* is copied) unless *toPath* is set to FALSE. If *toPath* is set to FALSE then the *to* variable is assumed to be the absolute file path (destination file path + file name). The copying is done using the get() and put() operations. If you want to be notified about the progress of the operation, connect to the dataTransferProgress() signal. Bear in mind that the get() and put() operations emit this signal through the QUrlOperator. The number of

transferred bytes and the total bytes that you receive as arguments in this signal do not relate to the the whole copy operation; they relate first to the get() and then to the put() operation. Always check what type of operation the signal comes from; this is given in the signal's last argument.

At the end, finished() (with success or failure) is emitted, so check the state of the network operation object to see whether or not the operation was successful.

Because a move or copy operation consists of multiple operations (get(), put() and maybe remove()), this function doesn't return a single QNetworkOperation, but rather a list of them. They are in the order: get(), put() and (if applicable) remove().

See also get() [p. 183] and put() [p. 185].

void QUrlOperator::copy (const QStringList & files, const QString & dest, bool move = FALSE) [virtual]

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

Copies *files* to the directory *dest*. If *move* is TRUE the files are moved, not copied. *dest* must point to a directory.

This function calls copy() for each entry in *files* one after the other. You don't get a result from this function; each time a new copy begins, startedNextCopy() is emitted, with a list of QNetworkOperations that describe the new copy operation.

void QUrlOperator::createdDirectory (const QUrlInfo & i, QNetworkOperation * op) [signal]

This signal is emitted when mkdir() succeeds and the directory has been created. *i* holds the information about the new directory. *op* is the pointer to the operation object, which contains all the information about the operation, including the state. Using op->arg(0) you also get the file name of the new directory.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

void QUrlOperator::data (const QByteArray & data, QNetworkOperation * op) [signal]

This signal is emitted when new *data* has been received after calling get() or put(). \op holds the name of the file whose data is retrieved in op->arg(0) and the (raw) data in op->rawArg(1).

op is the pointer to the operation object which contains all the information about the operation, including the state.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

void QUrlOperator::dataTransferProgress (int bytesDone, int bytesTotal, QNetworkOperation * op) [signal]

This signal is emitted during data transfer (using put() or get()). *bytesDone* specifies how many bytes of *bytesTotal* have been transferred. More information about the operation is stored in *op*, the pointer to the network operation that is processed. *bytesTotal* may be -1, which means that the total number of bytes is not known.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

void QUrlOperator::deleteNetworkProtocol () [protected]

Deletes the currently used network protocol.

void QUrlOperator::finished (QNetworkOperation * op) [signal]

This signal is emitted when an operation of some sort finishes, whether with success or failure. *op* is the pointer to the operation object, which contains all the information, including the state, of the operation which has been finished. Check the state and error code of the operation object to see whether or not the operation was successful.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

const QNetworkOperation * QUrlOperator::get (const QString & location = QString::null) [virtual]

Tells the network protocol to get data from *location* or, if this is QString::null, to get data from the location to which this URL points (see QUrl::fileName() and QUrl::encodedPathAndQuery()). What happens then depends on the network protocol. The data() signal is emitted when data comes in. Because it's unlikely that all data will come in at once, multiple data() signals will most likely be emitted. The dataTransferProgress() is emitted while processing the operation. At the end, finished() (with success or failure) is emitted, so check the state of the network operation object to see whether or not the operation was successful.

If *location* is QString::null, the path of this QUrlOperator should point to a file when you use this operation. If *location* is not empty, it can be a relative URL (a child of the path to which the QUrlOperator points) or an absolute URL.

For example, to get a web page you might do something like this:

```
QUrlOperator op( "http://www.whatever.org/cgi-bin/search.pl?cmd=Hello" );
op.get();
```

For most other operations, the path of the QUrlOperator must point to a directory. If you want to download a file you could do the following:

```
QUrlOperator op( "ftp://ftp.whatever.org/pub" );
// do some other stuff like op.listChildren() or op.mkdir( "new_dir" )
op.get( "a_file.txt" );
```

This will get the data of ftp://ftp.whatever.org/pub/a_file.txt.

Never do anything like this:

```
QUrlOperator op( "http://www.whatever.org/cgi-bin" );
op.get( "search.pl?cmd=Hello" ); // WRONG!
```

If *location* is not empty and relative it must not contain any queries or references, just the name of a child. So if you need to specify a query or reference, do it as shown in the first example or specify the full URL (such as http://www.whatever.org/cgi-bin/search.pl?cmd=Hello) as *location*.

See also copy() [p. 181].

void QUrlOperator::getNetworkProtocol () [protected]

Finds a network protocol for the URL and deletes the old network protocol.

QUrlInfo QUrlOperator::info (const QString & entry) const [virtual]

Returns the URL information for the child *entry*, or returns an empty QUrlInfo object if there is no information available about *entry*.

bool QUrlOperator::isDir (bool * ok = 0) [virtual]

Returns TRUE if the URL is a directory; otherwise returns FALSE. This may not always work correctly, if the protocol of the URL is something other than file (local filesystem). If you pass a bool pointer as the *ok* argument, **ok* is set to TRUE if the result of this function is known to be correct, and to FALSE otherwise.

void QUrlOperator::itemChanged (QNetworkOperation * op) [signal]

This signal is emitted whenever a file which is a child of the URL has been changed, for example by successfully calling rename(). *op* holds the original and new file names in the first and second arguments, respectively; they can be accessed with op->arg(0) and op->arg(1).

op is the pointer to the operation object which contains all the information about the operation, including the state.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

const QNetworkOperation * QUrlOperator::listChildren () [virtual]

Starts listing the children of this URL (e.g. of a directory). The signal start() is emitted before the first entry is listed and finished() is emitted after the last one. The newChildren() signal is emitted for each list of new entries. If an error occurs, the signal finished() is emitted, so be sure to check the state of the network operation pointer.

Because the operation may not be executed immediately, a pointer to the QNetworkOperation object created by this function is returned. This object contains all the data about the operation and is used to refer to this operation later (e.g. in the signals that are emitted by the QUrlOperator). The return value can also be 0 if the operation object couldn't be created.

The path of this QUrlOperator must to point to a directory (because the children of this directory will be listed), not to a file.

const QNetworkOperation * QUrlOperator::mkdir (const QString & dirname) [virtual]

Tries to create a directory (child) with the name *dirname*. If it is successful, a newChildren() signal with the new child is emitted, and the createdDirectory() signal with the information about the new child is emitted, too. finished() (with success or failure) is also emitted after the operation has been processed, so check the state of the network operation object to see whether or not the operation was successful.

Because the operation will not be executed immediately, a pointer to the QNetworkOperation object created by this function is returned. This object contains all the data about the operation and is used to refer to this operation later (e.g. in the signals that are emitted by the QUrlOperator). The return value can also be 0 if the operation object couldn't be created.

The path of this QUrlOperator must to point to a directory because the new directory will be created in this path, not to a file.

QString QUrlOperator::nameFilter () const

Returns the name filter of the URL.

See also QUrlOperator::setNameFilter() [p. 186] and QDir::nameFilter() [p. 41].

void QUrlOperator::newChildren (const QValueList<QUrlInfo> & i, QNetworkOperation * op) [signal]

This signal is emitted after listChildren() was called and new children (e.g. files) have been read from a list of files. *i* holds the information about the new children. *op* is the pointer to the operation object which contains all the information about the operation, including the state.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

const QNetworkOperation * QUrlOperator::put (const QByteArray & data, const QString & location = QString::null) [virtual]

This function tells the network protocol to put *data* in *location*. If *location* is empty (QString::null), it puts the *data* in the location to which the URL points. What happens depends on the network protocol. Depending on the network protocol, some data might come back after putting data, in which case the data() signal is emitted. The dataTransferProgress() is emitted during processing of the operation. At the end, finished() (with success or failure) is emitted, so check the state of the network operation object to see whether or not the operation was successful.

If *location* is QString::null, the path of this QUrlOperator should point to a file when you use this operation. If *location* is not empty, it can be a relative (a child of the path to which the QUrlOperator points) or an absolute URL.

For putting some data to a file you can do the following:

```
QUrlOperator op( "ftp://ftp.whatever.com/home/me/filename" );
op.put( data );
```

For most other operations, however, the path of the QUrlOperator must point to a directory. If you want to upload data to a file you could do the following:

```
QUrlOperator op( "ftp://ftp.whatever.com/home/me" );
// do some other stuff like op.listChildren() or op.mkdir( "new_dir" )
op.put( data, "filename.dat" );
```

This will upload the data to ftp://ftp.whatever.com/home/me/filename.dat.

See also copy() [p. 181].

const QNetworkOperation * QUrlOperator::remove (const QString & filename) [virtual]

Tries to remove the file (child) *filename*. If it succeeds the removed() signal is emitted. finished() (with success or failure) is also emitted after the operation has been processed, so check the state of the network operation object to see whether or not the operation was successful.

Because the operation will not be executed immediately, a pointer to the QNetworkOperation object created by this function is returned. This object contains all the data about the operation and is used to refer to this operation later (e.g. in the signals that are emitted by the QUrlOperator). The return value can also be 0 if the operation object couldn't be created.

The path of this QUrlOperator must point to a directory; because if *filename* is relative, it will try to remove it in this directory.

void QUrlOperator::removed (QNetworkOperation * op) [signal]

This signal is emitted when remove() has been succesful and the file has been removed. *op* holds the file name of the removed file in the first argument which can be accessed with op->arg(0).

op is the pointer to the operation object which contains all the information about the operation, including the state.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

const QNetworkOperation * QUrlOperator::rename (const QString & oldname, const QString & newname) [virtual]

Tries to rename the file (child) *oldname* to *newname*. If it succeeds, the signal itemChanged() is emitted. finished() (with success or failure) is also emitted after the operation has been processed, so check the state of the network operation object to see whether or not the operation was successful.

Because the operation may not be executed immediately, a pointer to the QNetworkOperation object created by this function is returned. This object contains all the data about the operation and is used to refer to this operation later (e.g. in the signals that are emitted by the QUrlOperator). The return value can also be 0 if the operation object couldn't be created.

This path of this QUrlOperator must to point to a directory because *oldname* and *newname* are handled relative to this directory.

void QUrlOperator::setNameFilter (const QString & nameFilter) [virtual]

Sets the name filter of the URL to *nameFilter*.

See also QDir::setNameFilter() [p. 43].

void QUrlOperator::start (QNetworkOperation * op) [signal]

Some operations (such as listChildren()) emit this signal when they start processing the operation. *op* is the pointer to the operation object which contains all the information about the operation, including the state.

See also QNetworkOperation [p. 96] and QNetworkProtocol [p. 99].

void QUrlOperator::startedNextCopy (const QPtrList<QNetworkOperation> & lst) [signal]

This signal is emitted if copy() starts a new copy operation. *lst* contains all QNetworkOperations related to this copy operation.

See also copy() [p. 181].

void QUrlOperator::stop () [virtual]

Stops the current network operation and removes all waiting network operations of this QUrlOperator.

Related Functions

void qInitNetworkProtocols ()

This function registers the network protocols for FTP and HTTP. You have to call this function before you use QUrlOperator for these protocols.

This function is declared in qnetwork.h.

QWindowsMime Class Reference

The QWindowsMime class maps open-standard MIME to Window Clipboard formats.

`#include <qmime.h>`

Public Members

- **QWindowsMime** ()
- virtual **~QWindowsMime** ()
- virtual const char * **convertorName** ()
- virtual int **countCf** ()
- virtual int **cf** (int index)
- virtual bool **canConvert** (const char * mime, int cf)
- virtual const char * **mimeFor** (int cf)
- virtual int **cfFor** (const char * mime)
- virtual QByteArray **convertToMime** (QByteArray data, const char * mime, int cf)
- virtual QByteArray **convertFromMime** (QByteArray data, const char * mime, int cf)

Static Public Members

- void **initialize** ()
- QPtrList<QWindowsMime> **all** ()
- QWindowsMime * **convertor** (const char * mime, int cf)
- const char * **cfToMime** (int cf)

Detailed Description

The QWindowsMime class maps open-standard MIME to Window Clipboard formats.

Qt's drag-and-drop and clipboard facilities use the MIME standard. On X11, this maps trivially to the Xdnd protocol, but on Windows although some applications use MIME types to describe clipboard formats, others use arbitrary non-standardized naming conventions, or unnamed built-in formats of Windows.

By instantiating subclasses of QWindowsMime that provide conversions between Windows Clipboard and MIME formats, you can convert proprietary clipboard formats to MIME formats.

Qt has predefined support for the following Windows Clipboard formats:

- CF_UNICODETEXT - converted to "text/plain;charset=ISO-10646-UCS-2" and supported by QTextDrag.
- CF_TEXT - converted to "text/plain;charset=system" or "text/plain" and supported by QTextDrag.
- CF_DIB - converted to "image/*", where * is a Qt image format, and supported by QImageDrag.
- CF_HDROP - converted to "text/uri-list", and supported by QUriDrag.

An example use of this class would be to map the Windows Metafile clipboard format (CF_METAFILEPICT) to and from the MIME type "image/x-wmf". This conversion might simply be adding or removing a header, or even just passing on the data. See the Drag-and-Drop documentation for more information on choosing and definition MIME types.

You can check if a MIME type is convertible using canConvert() and can perform conversions with convertToMime() and convertFromMime().

See also Drag And Drop Classes, Input/Output and Networking and Miscellaneous Classes.

Member Function Documentation

QWindowsMime::QWindowsMime ()

Constructs a new conversion object, adding it to the globally accessed list of available convertors.

QWindowsMime::~QWindowsMime () [virtual]

Destroys a conversion object, removing it from the global list of available convertors.

QPtrList<QWindowsMime> QWindowsMime::all () [static]

Returns a list of all currently defined QWindowsMime objects.

bool QWindowsMime::canConvert (const char * mime, int cf) [virtual]

Returns TRUE if the convertor can convert (both ways) between *mime* and *cf*; otherwise returns FALSE.

All subclasses must reimplement this pure virtual function.

int QWindowsMime::cf (int index) [virtual]

Returns the Windows Clipboard format supported by this convertor that is ordinarily at position *index*. This means that cf(0) returns the first Windows Clipboard format supported, and cf(countCf()-1) returns the last. If *index* is out of range the return value is undefined.

All subclasses must reimplement this pure virtual function.

int QWindowsMime::cfFor (const char * mime) [virtual]

Returns the Windows Clipboard type used for MIME type *mime*, or 0 if this convertor does not support *mime*.

All subclasses must reimplement this pure virtual function.

const char * QWindowsMime::cfToMime (int cf) [static]

Returns a MIME type for *cf*, or 0 if none exists.

QByteArray QWindowsMime::convertFromMime (QByteArray data, const char * mime, int cf) [virtual]

Returns *data* converted from MIME type *mime* to Windows Clipboard format *cf*.

Note that Windows Clipboard formats must all be self-terminating. The return value may contain trailing data.

All subclasses must reimplement this pure virtual function.

QByteArray QWindowsMime::convertToMime (QByteArray data, const char * mime, int cf) [virtual]

Returns *data* converted from Windows Clipboard format *cf* to MIME type *mime*.

Note that Windows Clipboard formats must all be self-terminating. The input *data* may contain trailing data.

All subclasses must reimplement this pure virtual function.

QWindowsMime * QWindowsMime::convertor (const char * mime, int cf) [static]

Returns the most-recently created QWindowsMime that can convert between the *mime* and *cf* formats. Returns 0 if no such convertor exists.

const char * QWindowsMime::convertorName () [virtual]

Returns a name for the convertor.

All subclasses must reimplement this pure virtual function.

int QWindowsMime::countCf () [virtual]

Returns the number of Windows Clipboard formats supported by this convertor.

All subclasses must reimplement this pure virtual function.

void QWindowsMime::initialize () [static]

This is an internal function.

const char * QWindowsMime::mimeFor (int cf) [virtual]

Returns the MIME type used for Windows Clipboard format *cf*, or 0 if this convertor does not support *cf*.

All subclasses must reimplement this pure virtual function.

Index

absFilePath()
    QDir, 34
    QFileInfo, 62
absPath()
    QDir, 34
accept()
    QSocketDevice, 137
activated()
    QSocketNotifier, 144
addArgument()
    QProcess, 112
addFactory()
    QMimeSourceFactory, 92
addFilePath()
    QMimeSourceFactory, 93
addOperation()
    QNetworkProtocol, 102
addPath()
    QUrl, 166
address()
    QServerSocket, 120
    QSocket, 128
    QSocketDevice, 137
addresses()
    QDns, 47
addressReusable()
    QSocketDevice, 137
all()
    QWindowsMime, 189
allowsErrorInteraction()
    QSessionManager, 122
allowsInteraction()
    QSessionManager, 122
arg()
    QNetworkOperation, 97
arguments()
    QProcess, 112
at()
    QFile, 52, 53
    QIODevice, 79
    QSocket, 129
atEnd()
    QDataStream, 23
    QFile, 53
    QIODevice, 79
    QSocket, 129
    QTextStream, 154
autoDelete()
    QNetworkProtocol, 102

baseName()
    QFileInfo, 62
bind()
    QSocketDevice, 138
blocking()
    QSocketDevice, 138
ByteOrder
    QDataStream, 22
byteOrder()
    QDataStream, 23
bytesAvailable()
    QSocket, 129
    QSocketDevice, 138
bytesToWrite()
    QSocket, 129
bytesWritten()
    QSocket, 129

caching()
    QFileInfo, 62
cancel()
    QSessionManager, 123
canConvert()
    QWindowsMime, 189
canonicalName()
    QDns, 47
canonicalPath()
    QDir, 34
canReadLine()
    QSocket, 129
canReadLineStderr()
    QProcess, 112
canReadLineStdout()
    QProcess, 112
cd()
    QDir, 35
cdUp()
    QDir, 35
    QUrl, 166
cf()
    QWindowsMime, 189
cfFor()
    QWindowsMime, 189
cfToMime()
    QWindowsMime, 190
checkConnection()
    QNetworkProtocol, 102
cleanDirPath()
    QDir, 35
clear()
    QClipboard, 16
clearArguments()
    QProcess, 112
clearEntries()
    QUrlOperator, 181
clearOperationQueue()
    QNetworkProtocol, 103
close()
    QFile, 53
    QIODevice, 79
    QSocket, 130
closeStdin()
    QProcess, 112
Communication
    QProcess, 110
communication()
    QProcess, 112
connect()
    QSocketDevice, 138
connected()
    QSocket, 130
connectionClosed()
    QSocket, 130
ConnectionState
    QNetworkProtocol, 101
connectionStateChanged()
    QNetworkProtocol, 103
    QUrlOperator, 181
connectToHost()
    QSocket, 130
convertFromMime()
    QWindowsMime, 190
convertor()
    QWindowsMime, 190
convertorName()
    QWindowsMime, 190
convertSeparators()
    QDir, 35

convertToAbs()
    QDir, 35
    QFileInfo, 62
convertToMime()
    QWindowsMime, 190
copy()
    QUrlOperator, 181, 182
count()
    QDir, 36
countCf()
    QWindowsMime, 190
created()
    QFileInfo, 63
createdDirectory()
    QNetworkProtocol, 103
    QUrlOperator, 182
current()
    QDir, 36
currentDirPath()
    QDir, 36

data()
    QClipboard, 16
    QMimeSourceFactory, 93
    QNetworkProtocol, 103
    QUrlOperator, 182
dataBytesWritten()
    QFtp, 70
dataChanged()
    QClipboard, 16
dataClosed()
    QFtp, 70
dataConnected()
    QFtp, 70
dataReadyRead()
    QFtp, 70
dataTransferProgress()
    QNetworkProtocol, 103
    QUrlOperator, 182
decode()
    QUrl, 166
decodeName()
    QFile, 53
DecoderFn
    QFile, 52
defaultFactory()
    QMimeSourceFactory, 93
delayedCloseFinished()
    QSocket, 131
deleteNetworkProtocol()
    QUrlOperator, 183
device()
    QDataStream, 23
    QTextStream, 154
dir()
    QFileInfo, 63
dirName()
    QDir, 36
dirPath()
    QFileInfo, 63
    QUrl, 167
discardCommand()
    QSessionManager, 123
drives()
    QDir, 36

encode()
    QUrl, 167
encodedData()
    QMimeSource, 90
encodedEntryList()
    QDir, 36, 37
encodedPathAndQuery()
    QUrl, 167
encodeName()
    QFile, 53
EncoderFn
    QFile, 52
Encoding
    QTextStream, 152
entryInfoList()
    QDir, 37
entryList()
    QDir, 38
eof()
    QDataStream, 23
    QTextStream, 154
equal()
    QUrlInfo, 174
Error
    QNetworkProtocol, 101
    QSocket, 128
    QSocketDevice, 136
error()
    QSocket, 131
    QSocketDevice, 138
errorCode()
    QNetworkOperation, 97
exists()
    QDir, 38
    QFile, 54
    QFileInfo, 63
exitStatus()
    QProcess, 113
extension()
    QFileInfo, 63

fileName()
    QFileInfo, 64
    QUrl, 167
filePath()
    QDir, 38
    QFileInfo, 64
    QMimeSourceFactory, 94
fill()
    QTextStream, 154
filter()
    QDir, 39
FilterSpec
    QDir, 32
finished()
    QNetworkProtocol, 103
    QUrlOperator, 183
flags()
    QIODevice, 80
    QTextStream, 155
flush()
    QFile, 54
    QIODevice, 80
    QSocket, 131
format()
    QMimeSource, 90
free()
    QNetworkOperation, 97

get()
    QUrlOperator, 183
getch()
    QFile, 54
    QIODevice, 80
    QSocket, 131
getNetworkProtocol()
    QNetworkProtocol, 104
    QUrlOperator, 184
greaterThan()
    QUrlInfo, 174
group()
    QFileInfo, 64
    QUrlInfo, 174
groupId()
    QFileInfo, 64

handle()
    QFile, 54
    QSessionManager, 124
hasHost()
    QUrl, 167
hasOnlyLocalFileSystem()
    QNetworkProtocol, 104
hasPassword()
    QUrl, 167
hasPath()
    QUrl, 167
hasPort()
    QUrl, 168
hasRef()
    QUrl, 168
hasUser()
    QUrl, 168
home()
    QDir, 39
homeDirPath()
    QDir, 39
host()
    QUrl, 168
hostFound()

QSocket, 131
hostNames()
QDns, 47

image()
QClipboard, 17
info()
QUrlOperator, 184
initialize()
QWindowsMime, 190
IO_AbortError, 84
IO_Append, 55
IO_ConnectError, 84
IO_FatalError, 84
IO_Ok, 84
IO_OpenError, 84
IO_Raw, 55
IO_ReadError, 84
IO_ReadOnly, 55
IO_ReadWrite, 55
IO_TimeOutError, 84
IO_Translate, 55
IO_Truncate, 55
IO_UnspecifiedError, 84
IO_WriteError, 84
IO_WriteOnly, 55
ip4Addr()
QHostAddress, 72
isAsynchronous()
QIODevice, 80
isBuffered()
QIODevice, 80
isCombinedAccess()
QIODevice, 80
isDir()
QFileInfo, 64
QUrlInfo, 174
QUrlOperator, 184
isDirectAccess()
QIODevice, 81
isEnabled()
QSocketNotifier, 144
isExecutable()
QFileInfo, 65
QUrlInfo, 175
isFile()
QFileInfo, 65
QUrlInfo, 175
isInactive()
QIODevice, 81
isIp4Addr()
QHostAddress, 72
isLocalFile()
QUrl, 168
isOpen()
QIODevice, 81
isPhase2()
QSessionManager, 124
isPrintableData()
QDataStream, 23
isRaw()
QIODevice, 81
isReadable()
QDir, 39
QFileInfo, 65
QIODevice, 81
QUrlInfo, 175
isReadWrite()
QIODevice, 81
isRelative()
QDir, 39
QFileInfo, 65
isRelativePath()
QDir, 39
isRelativeUrl()
QUrl, 168
isRoot()
QDir, 40
isRunning()
QProcess, 113
isSequentialAccess()
QIODevice, 81
isSymLink()
QFileInfo, 65
QUrlInfo, 175
isSynchronous()
QIODevice, 82
isTranslated()
QIODevice, 82
isValid()
QLock, 88
QSocketDevice, 138
QUrl, 168
QUrlInfo, 175
isWorking()
QDns, 47
isWritable()
QFileInfo, 65
QIODevice, 82
QUrlInfo, 175
itemChanged()
QNetworkProtocol, 104
QUrlOperator, 184

kill()
QProcess, 113

label()
QDns, 48
lastModified()
QFileInfo, 65
QUrlInfo, 175
lastRead()
QFileInfo, 66
QUrlInfo, 176
launch()
QProcess, 113, 114
launchFinished()
QProcess, 114
lessThan()
QUrlInfo, 176
listChildren()
QUrlOperator, 184
listen()
QSocketDevice, 138
lock()
QLock, 88
locked()
QLock, 88

mailServers()
QDns, 48
makeAbsolute()
QMimeSourceFactory, 94
match()
QDir, 40
matchAllDirs()
QDir, 40
mimeFor()
QWindowsMime, 190
mkdir()
QDir, 40
QUrlOperator, 184
mode()
QIODevice, 82

name()
QFile, 54
QUrlInfo, 176
nameFilter()
QDir, 41
QUrlOperator, 185
newChild()
QNetworkProtocol, 104
newChildren()
QNetworkProtocol, 104
QUrlOperator, 185
newConnection()
QServerSocket, 120
normalExit()
QProcess, 114

Offset
QIODevice, 79
ok()
QServerSocket, 120
open()
QFile, 55, 56
QIODevice, 82
QSocket, 131
Operation
QNetworkProtocol, 101
operation()
QNetworkOperation, 97
operationGet()
QNetworkProtocol, 105

operationInProgress()
QNetworkProtocol, 105
operationListChildren()
QNetworkProtocol, 105
operationMkDir()
QNetworkProtocol, 105
operationPut()
QNetworkProtocol, 106
operationRemove()
QNetworkProtocol, 106
operationRename()
QNetworkProtocol, 106
operator
=()
QDir, 41
operator QString()
QUrl, 168
operator=()
QDir, 41
QFileInfo, 66
QHostAddress, 72
QUrl, 168, 169
QUrlInfo, 176
operator==()
QDir, 41
QHostAddress, 72
QUrl, 169
QUrlInfo, 176
operator[]()
QDir, 41
operator<<()
QDataStream, 24, 25
QTextStream, 155–157
operator>>()
QDataStream, 25, 26
QTextStream, 157–159
owner()
QFileInfo, 66
QUrlInfo, 176
ownerId()
QFileInfo, 66
ownsClipboard()
QClipboard, 17
ownsSelection()
QClipboard, 17

parse()
QUrl, 169
parseDir()
QFtp, 70
password()
QUrl, 169
path()
QDir, 42
QUrl, 169
peerAddress()
QSocket, 132
QSocketDevice, 139
peerName()
QSocket, 132
peerPort()
QSocket, 132
QSocketDevice, 139
permission()
QFileInfo, 66
permissions()
QUrlInfo, 176
PermissionSpec
QFileInfo, 61
pixmap()
QClipboard, 17
port()
QServerSocket, 120
QSocket, 132
QSocketDevice, 139
QUrl, 169
precision()
QTextStream, 159
processExited()
QProcess, 114
processIdentifier()
QProcess, 114
protocol()
QUrl, 169
protocolDetail()
QNetworkOperation, 98
provides()
QMimeSource, 90
put()
QUrlOperator, 185
putch()
QFile, 56
QIODevice, 83
QSocket, 132

qualifiedNames()
QDns, 48
query()
QUrl, 170

rawArg()
QNetworkOperation, 98
read()
QTextStream, 159
readAll()
QIODevice, 56, 83
readBlock()
QFile, 56
QIODevice, 83
QSocket, 132
QSocketDevice, 139
readBytes()
QDataStream, 26
readLine()
QFile, 57
QIODevice, 83
QSocket, 132
QTextStream, 159
readLineStderr()
QProcess, 114
readLineStdout()
QProcess, 115
readLink()
QFileInfo, 66
readRawBytes()
QDataStream, 27
QTextStream, 160
readStderr()
QProcess, 115
readStdout()
QProcess, 115
readyRead()
QFtp, 70
QSocket, 133
readyReadStderr()
QProcess, 115
readyReadStdout()
QProcess, 115
receiveBufferSize()
QSocketDevice, 139
RecordType
QDns, 46
recordType()
QDns, 48
ref()
QUrl, 170
refresh()
QFileInfo, 67
registerNetworkProtocol()
QNetworkProtocol, 106
release()
QSessionManager, 124
remove()
QDir, 42
QFile, 57
QUrlOperator, 185
removed()
QNetworkProtocol, 107
QUrlOperator, 186
removeFactory()
QMimeSourceFactory, 94
rename()
QDir, 42
QUrlOperator, 186
requestPhase2()
QSessionManager, 124
reset()
QIODevice, 83
QTextStream, 160
QUrl, 170
resetStatus()
QIODevice, 83
restartCommand()
QSessionManager, 124
RestartHint

QSessionManager, 122
restartHint()
QSessionManager, 124
resultsReady()
QDns, 48
rmdir()
QDir, 42
root()
QDir, 42
rootDirPath()
QDir, 43

selectionChanged()
QClipboard, 17
selectionModeEnabled()
QClipboard, 17
sendBufferSize()
QSocketDevice, 139
separator()
QDir, 43
serialNumber()
QMimeSource, 90
servers()
QDns, 48
sessionId()
QSessionManager, 124
setAddress()
QHostAddress, 73
setAddressReusable()
QSocketDevice, 139
setArg()
QNetworkOperation, 98
setArguments()
QProcess, 116
setAutoDelete()
QNetworkProtocol, 107
setBlocking()
QSocketDevice, 140
setByteOrder()
QDataStream, 27
setCaching()
QFileInfo, 67
setCodec()
QTextStream, 160
setCommunication()
QProcess, 116
setCurrent()
QDir, 43
setData()
QClipboard, 17
QMimeSourceFactory, 94
setDecodingFunction()
QFile, 57
setDefaultFactory()
QMimeSourceFactory, 94
setDevice()
QDataStream, 27
QTextStream, 160
setDir()
QUrlInfo, 176
setDiscardCommand()
QSessionManager, 125
setEnabled()
QSocketNotifier, 144
setEncodedPathAndQuery()
QUrl, 170
setEncoding()
QTextStream, 160
setEncodingFunction()
QFile, 57
setError()
QSocketDevice, 140
setErrorCode()
QNetworkOperation, 98
setExtensionType()
QMimeSourceFactory, 94
setf()
QTextStream, 161
setFile()
QFileInfo, 67, 68
QUrlInfo, 177
setFileName()
QUrl, 170
setFilePath()
QMimeSourceFactory, 94
setFilter()
QDir, 43
setGroup()
QUrlInfo, 177
setHost()
QUrl, 170
setImage()
QClipboard, 17
QMimeSourceFactory, 95
setLabel()
QDns, 49
setLastModified()
QUrlInfo, 177
setManagerProperty()
QSessionManager, 125
setMatchAllDirs()
QDir, 43
setName()
QFile, 58
QUrlInfo, 177
setNameFilter()
QDir, 43
QUrlOperator, 186
setOwner()
QUrlInfo, 177
setPassword()
QUrl, 170
setPath()
QDir, 43
QUrl, 170
setPermissions()
QUrlInfo, 177
setPixmap()
QClipboard, 18
QMimeSourceFactory, 95
setPort()
QUrl, 171
setPrintableData()
QDataStream, 27
setProtocol()
QUrl, 171
setProtocolDetail()
QNetworkOperation, 98
setQuery()
QUrl, 171
setRawArg()
QNetworkOperation, 98
setReadable()
QUrlInfo, 178
setReceiveBufferSize()
QSocketDevice, 140
setRecordType()
QDns, 49
setRef()
QUrl, 171
setRestartCommand()
QSessionManager, 125
setRestartHint()
QSessionManager, 125
setSelectionMode()
QClipboard, 18
setSendBufferSize()
QSocketDevice, 140
setSize()
QUrlInfo, 178
setSocket()
QServerSocket, 120
QSocket, 133
QSocketDevice, 140
setSocketDevice()
QSocket, 133
setSorting()
QDir, 44
setState()
QNetworkOperation, 98
setSymLink()
QUrlInfo, 178
setText()
QClipboard, 18
QMimeSourceFactory, 95
setUrl()
QNetworkProtocol, 107
setUser()
QUrl, 171
setVersion()
QDataStream, 27
setWorkingDirectory()
QProcess, 116
setWritable()

QUrlInfo, 178
size()
QFile, 58
QFileInfo, 68
QIODevice, 84
QSocket, 133
QUrlInfo, 178
skipWhiteSpace()
QTextStream, 161
sn_read()
QSocket, 133
sn_write()
QSocket, 133
socket()
QServerSocket, 120
QSocket, 134
QSocketDevice, 141
QSocketNotifier, 145
socketDevice()
QServerSocket, 120
QSocket, 134
sorting()
QDir, 44
SortSpec
QDir, 33
start()
QNetworkProtocol, 107
QProcess, 116
QUrlOperator, 186
startedNextCopy()
QUrlOperator, 186
State
QNetworkProtocol, 102
QSocket, 128
state()
QIODevice, 84
QNetworkOperation, 98
QSocket, 134
status()
QIODevice, 84
stop()
QNetworkProtocol, 107
QUrlOperator, 187
supportedOperations()
QNetworkProtocol, 107
supportsSelection()
QClipboard, 18

takeDefaultFactory()
QMimeSourceFactory, 95
text()
QClipboard, 18
texts()
QDns, 49
toString()
QHostAddress, 73
QUrl, 171
tryTerminate()
QProcess, 117
Type
QLock, 87
QSocketDevice, 137
QSocketNotifier, 143
type()
QSocketDevice, 141
QSocketNotifier, 145

ungetch()
QFile, 58
QIODevice, 84
QSocket, 134
unlock()
QLock, 88
unsetDevice()
QDataStream, 28
QTextStream, 161
unsetf()
QTextStream, 161
url()
QNetworkProtocol, 107
user()
QUrl, 171

version()
QDataStream, 28

waitForMore()
QSocket, 134
QSocketDevice, 141
width()
QTextStream, 162
workingDirectory()
QProcess, 117
writeBlock()
QIODevice, 85
QSocket, 134
QSocketDevice, 141
writeBytes()
QDataStream, 28
writeRawBytes()
QDataStream, 28
QTextStream, 162
writeToStdin()
QProcess, 117
wroteToStdin()
QProcess, 117