QFile Class Reference

[QtCore module]

The QFile class provides an interface for reading from and writing to files. More...

```
#include <QFile>
```

Inherits QIODevice.

Inherited by QTemporaryFile.

Note: All functions in this class are reentrant, except for setEncodingFunction() and setDecodingFunction(), which are nonreentrant.

- List of all members, including inherited members
- Obsolete members
- Qt 3 support members

Public Types

typedef	**DecoderFn**
typedef	**EncoderFn**
enum	**FileError** { NoError, ReadError, WriteError, FatalError, ..., CopyError }
enum	**MemoryMapFlags** { NoOptions }
enum	**Permission** { ReadOwner, WriteOwner, ExeOwner, ReadUser, ..., ExeOther }
typedef	**PermissionSpec**
flags	**Permissions**

Public Functions

	QFile (const QString & *name*)
	QFile (QObject * *parent*)
	QFile (const QString & *name*, QObject * *parent*)
	~QFile ()
bool	**copy** (const QString & *newName*)
FileError	**error** () const
bool	**exists** () const
QString	**fileName** () const
bool	**flush** ()
int	**handle** () const

bool **link** (const QString & *linkName*)
uchar * **map** (qint64 *offset*, qint64 *size*, MemoryMapFlags *flags* = NoOptions)
bool **open** (FILE * *fh*, OpenMode *mode*)
bool **open** (int *fd*, OpenMode *mode*)
Permissions **permissions** () const
bool **remove** ()
bool **rename** (const QString & *newName*)
bool **resize** (qint64 *sz*)
void **setFileName** (const QString & *name*)
bool **setPermissions** (Permissions *permissions*)
QString **symLinkTarget** () const
bool **unmap** (uchar * *address*)
void **unsetError** ()

Reimplemented Public Functions

virtual bool **atEnd** () const
virtual void **close** ()
virtual bool **isSequential** () const
virtual bool **open** (OpenMode *mode*)
virtual qint64 **pos** () const
virtual bool **seek** (qint64 *off*)
virtual qint64 **size** () const

- 33 public functions inherited from QIODevice
- 29 public functions inherited from QObject

Static Public Members

bool **copy** (const QString & *fileName*, const QString & *newName*)
QString **decodeName** (const QByteArray & *localFileName*)
QString **decodeName** (const char * *localFileName*)
QByteArray **encodeName** (const QString & *fileName*)
bool **exists** (const QString & *fileName*)
bool **link** (const QString & *fileName*, const QString & *linkName*)
Permissions **permissions** (const QString & *fileName*)
bool **remove** (const QString & *fileName*)
bool **rename** (const QString & *oldName*, const QString & *newName*)
bool **resize** (const QString & *fileName*, qint64 *sz*)
void **setDecodingFunction** (DecoderFn *function*)
void **setEncodingFunction** (EncoderFn *function*)
bool **setPermissions** (const QString & *fileName*, Permissions

permissions)
QString **symLinkTarget** (const QString & *fileName*)

- 5 static public members inherited from QObject

Reimplemented Protected Functions

virtual qint64 **readData** (char * *data*, qint64 *len*)
virtual qint64 **readLineData** (char * *data*, qint64 *maxlen*)
virtual qint64 **writeData** (const char * *data*, qint64 *len*)

- 5 protected functions inherited from QIODevice
- 7 protected functions inherited from QObject

Additional Inherited Members

- 1 property inherited from QObject
- 1 public slot inherited from QObject
- 4 signals inherited from QIODevice
- 1 signal inherited from QObject
- 5 protected functions inherited from QIODevice
- 7 protected functions inherited from QObject

Detailed Description

The QFile class provides an interface for reading from and writing to files.

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

The file name is usually passed in the constructor, but it can be set at any time using setFileName(). QFile expects the file separator to be '/' regardless of operating system. The use of other separators (e.g., '\') is not supported.

You can check for a file's existence using exists(), and remove a file using remove(). (More advanced file system related operations are provided by QFileInfo and QDir.)

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 also call the QIODevice-inherited functions read(), readLine(), readAll(), write(). QFile also inherits getChar(), putChar(), and ungetChar(), which work one character at a time.

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

Reading Files Directly

The following example reads a text file line by line:

```
    QFile file("in.txt");
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
```

```
        return;

    while (!file.atEnd()) {
        QByteArray line = file.readLine();
        process_line(line);
    }
```

The QIODevice::Text flag passed to open() tells Qt to convert Windows-style line terminators ("\r\n") into C++-style terminators ("\n"). By default, QFile assumes binary, i.e. it doesn't perform any conversion on the bytes stored in the file.

Using Streams to Read Files

The next example uses QTextStream to read a text file line by line:

```
    QFile file("in.txt");
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        return;

    QTextStream in(&file);
    while (!in.atEnd()) {
        QString line = in.readLine();
        process_line(line);
    }
```

QTextStream takes care of converting the 8-bit data stored on disk into a 16-bit Unicode QString. By default, it assumes that the user system's local 8-bit encoding is used (e.g., ISO 8859-1 for most of Europe; see QTextCodec::codecForLocale() for details). This can be changed using setCodec().

To write text, we can use operator<<(), which is overloaded to take a QTextStream on the left and various data types (including QString) on the right:

```
    QFile file("out.txt");
    if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
        return;

    QTextStream out(&file);
    out << "The magic number is: " << 49 << "\n";
```

QDataStream is similar, in that you can use operator<<() to write data and operator>>() to read it back. See the class documentation for details.

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 use standard C++ APIs (<cstdio> or <iostream>) or platform-specific APIs to access files instead of QFile, you can use the encodeName() and decodeName() functions to convert between Unicode file names and 8-bit file names.

On Unix, there are some special system files (e.g. in /proc) for which size() will always return 0, yet you may still be able to read more data from such a file; the data is generated in direct response to you calling read(). In this case, however, you cannot use atEnd() to determine if there is more data to read (since atEnd() will return true for a file that claims to have size 0). Instead, you should either call readAll(), or call read() or readLine() repeatedly until no more data can be read. The next example uses QTextStream to read /proc/modules line by line:

```
    QFile file("/proc/modules");
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        return;
```

```
    QTextStream in(&file);
    QString line = in.readLine();
    while (!line.isNull()) {
        process_line(line);
        line = in.readLine();
    }
```

Signals

Unlike other QIODevice implementations, such as QTcpSocket, QFile does not emit the aboutToClose(), bytesWritten(), or readyRead() signals. This implementation detail means that QFile is not suitable for reading and writing certain types of files, such as device files on Unix platforms.

Platform Specific Issues

File permissions are handled differently on Linux/Mac OS X and Windows. In a non writable directory on Linux, files cannot be created. This is not always the case on Windows, where, for instance, the 'My Documents' directory usually is not writable, but it is still possible to create files in it.

See also QTextStream, QDataStream, QFileInfo, QDir, and The Qt Resource System.

Member Type Documentation

typedef QFile::DecoderFn

This is a typedef for a pointer to a function with the following signature:

```
QString myDecoderFunc(const QByteArray &localFileName);
```

See also setDecodingFunction().

typedef QFile::EncoderFn

This is a typedef for a pointer to a function with the following signature:

```
QByteArray myEncoderFunc(const QString &fileName);
```

See also setEncodingFunction() and encodeName().

enum QFile::FileError

This enum describes the errors that may be returned by the error() function.

Constant	Value	Description
QFile::NoError	0	No error occurred.

QFile::ReadError	1	An error occurred when reading from the file.
QFile::WriteError	2	An error occurred when writing to the file.
QFile::FatalError	3	A fatal error occurred.
QFile::ResourceError	4	
QFile::OpenError	5	The file could not be opened.
QFile::AbortError	6	The operation was aborted.
QFile::TimeOutError	7	A timeout occurred.
QFile::UnspecifiedError	8	An unspecified error occurred.
QFile::RemoveError	9	The file could not be removed.
QFile::RenameError	10	The file could not be renamed.
QFile::PositionError	11	The position in the file could not be changed.
QFile::ResizeError	12	The file could not be resized.
QFile::PermissionsError	13	The file could not be accessed.
QFile::CopyError	14	The file could not be copied.

enum QFile::MemoryMapFlags

This enum describes special options that may be used by the map() function.

Constant	Value	Description
QFile::NoOptions	0	No options.

This enum was introduced in Qt 4.4.

enum QFile::Permission
flags QFile::Permissions

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.

Constant	Value	Description
QFile::ReadOwner	0x4000	The file is readable by the owner of the file.
QFile::WriteOwner	0x2000	The file is writable by the owner of the file.
QFile::ExeOwner	0x1000	The file is executable by the owner of the file.
QFile::ReadUser	0x0400	The file is readable by the user.
QFile::WriteUser	0x0200	The file is writable by the user.
QFile::ExeUser	0x0100	The file is executable by the user.
QFile::ReadGroup	0x0040	The file is readable by the group.
QFile::WriteGroup	0x0020	The file is writable by the group.
QFile::ExeGroup	0x0010	The file is executable by the group.
QFile::ReadOther	0x0004	The file is readable by anyone.
QFile::WriteOther	0x0002	The file is writable by anyone.
QFile::ExeOther	0x0001	The file is executable by anyone.

Warning: Because of differences in the platforms supported by Qt, the semantics of ReadUser, WriteUser and ExeUser are platform-dependent: On Unix, the rights of the

owner of the file are returned and on Windows the rights of the current user are returned. This behavior might change in a future Qt version.

Note that Qt does not by default check for permissions on NTFS file systems, as this may decrease the performance of file handling considerably. It is possible to force permission checking on NTFS by including the following code in your source:

```
extern Q_CORE_EXPORT int qt_ntfs_permission_lookup;
```

Permission checking is then turned on and off by incrementing and decrementing `qt_ntfs_permission_lookup` by 1.

```
qt_ntfs_permission_lookup++; // turn checking on
qt_ntfs_permission_lookup--; // turn it off again
```

The Permissions type is a typedef for QFlags<Permission>. It stores an OR combination of Permission values.

typedef QFile::PermissionSpec

Use QFile::Permission instead.

Member Function Documentation

QFile::QFile (const QString & *name*)

Constructs a new file object to represent the file with the given *name*.

QFile::QFile (QObject * *parent*)

Constructs a new file object with the given *parent*.

QFile::QFile (const QString & *name*, QObject * *parent*)

Constructs a new file object with the given *parent* to represent the file with the specified *name*.

QFile::~QFile ()

Destroys the file object, closing it if necessary.

bool QFile::atEnd () const [virtual]

Reimplemented from QIODevice::atEnd().

Returns true if the end of the file has been reached; otherwise returns false.

For regular empty files on Unix (e.g. those in `/proc`), this function returns true, since the file system reports that the size of such a file is 0. Therefore, you should not depend on atEnd() when reading data from such a file, but rather call read() until no more data can be read.

void QFile::close () [virtual]

Reimplemented from QIODevice::close().

Calls QFile::flush() and closes the file. Errors from flush are ignored.

See also QIODevice::close().

bool QFile::copy (const QString & *newName*)

Copies the file currently specified by fileName() to a file called *newName*. Returns true if successful; otherwise returns false.

Note that if a file with the name *newName* already exists, copy() returns false (i.e. QFile will not overwrite it).

The source file is closed before it is copied.

See also setFileName().

bool QFile::copy (const QString & *fileName*, const QString & *newName*) [static]

This is an overloaded function.

Copies the file *fileName* to *newName*. Returns true if successful; otherwise returns false.

If a file with the name *newName* already exists, copy() returns false (i.e., QFile will not overwrite it).

See also rename().

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

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

See also setDecodingFunction() and encodeName().

QString QFile::decodeName (const char * *localFileName*) [static]

This is an overloaded function.

Returns the Unicode version of the given *localFileName*. See encodeName() for details.

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

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.

See also decodeName() and setEncodingFunction().

FileError QFile::error () const

Returns the file error status.

The I/O device status returns an error code. For example, if open() returns false, or a read/write operation returns -1, this function can be called to find out the reason why the operation failed.

See also unsetError().

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

Returns true if the file specified by *fileName* exists; otherwise returns false.

bool QFile::exists () const

This is an overloaded function.

Returns true if the file specified by fileName() exists; otherwise returns false.

See also fileName() and setFileName().

QString QFile::fileName () const

Returns the name set by setFileName() or to the QFile constructors.

See also setFileName() and QFileInfo::fileName().

bool QFile::flush ()

Flushes any buffered data to the file. Returns true if successful; otherwise returns false.

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(). On systems that use file descriptors for sockets (i.e. Unix systems, but not Windows) the handle can be used with QSocketNotifier as well.

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

This function is not supported on Windows CE.

See also QSocketNotifier.

bool QFile::isSequential () const [virtual]

Reimplemented from QIODevice::isSequential().

Returns true if the file can only be manipulated sequentially; otherwise returns false.

Most files support random-access, but some special files may not.

See also QIODevice::isSequential().

bool QFile::link (const QString & *linkName*)

Creates a link named *linkName* that points to the file currently specified by fileName(). What a link is depends on the underlying filesystem (be it a shortcut on Windows or a symbolic link on Unix). Returns true if successful; otherwise returns false.

This function will not overwrite an already existing entity in the file system; in this case, `link()` will return false and set error() to return RenameError.

Note: To create a valid link on Windows, *linkName* must have a `.lnk` file extension.

Note: On Symbian, no link is created and false is returned if fileName() currently specifies a directory.

See also setFileName().

bool QFile::link (const QString & *fileName*, const QString & *linkName*) [static]

This is an overloaded function.

Creates a link named *linkName* that points to the file *fileName*. What a link is depends on the underlying filesystem (be it a shortcut on Windows or a symbolic link on Unix). Returns true if successful; otherwise returns false.

See also link().

uchar * QFile::map (qint64 *offset*, qint64 *size*, MemoryMapFlags *flags* = NoOptions)

Maps *size* bytes of the file into memory starting at *offset*. A file should be open for a map to succeed but the file does not need to stay open after the memory has been mapped. When the QFile is destroyed or a new file is opened with this object, any maps that have not been unmapped will automatically be unmapped.

Any mapping options can be passed through *flags*.

Returns a pointer to the memory or 0 if there is an error.

Note: On Windows CE 5.0 the file will be closed before mapping occurs.

This function was introduced in Qt 4.4.

See also unmap() and QAbstractFileEngine::supportsExtension().

bool QFile::open (OpenMode *mode*) [virtual]

Reimplemented from QIODevice::open().

Opens the file using OpenMode *mode*, returning true if successful; otherwise false.

The *mode* must be QIODevice::ReadOnly, QIODevice::WriteOnly, or QIODevice::ReadWrite. It may also have additional flags, such as QIODevice::Text and QIODevice::Unbuffered.

Note: In WriteOnly or ReadWrite mode, if the relevant file does not already exist, this function will try to create a new file before opening it.

See also QIODevice::OpenMode and setFileName().

bool QFile::open (FILE * *fh*, OpenMode *mode*)

This is an overloaded function.

Opens the existing file handle *fh* in the given *mode*. Returns true if successful; otherwise returns false.

Example:

```
#include <stdio.h>

void printError(const char* msg)
{
    QFile file;
    file.open(stderr, QIODevice::WriteOnly);
    file.write(msg, qstrlen(msg));        // write to stderr
    file.close();
}
```

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

Warning:

1. If *fh* does not refer to a regular file, e.g., it is `stdin`, `stdout`, or `stderr`, you may not be able to seek(). size() returns `0` in those cases. See QIODevice::isSequential() for more information.
2. Since this function opens the file without specifying the file name, you cannot use this QFile with a QFileInfo.

Note: For Windows CE you may not be able to call resize().

Note for the Windows Platform

fh must be opened in binary mode (i.e., the mode string must contain 'b', as in "rb" or "wb") when accessing files and other random-access devices. Qt will translate the end-of-line characters if you pass QIODevice::Text to *mode*. Sequential devices, such as stdin and stdout, are unaffected by this limitation.

You need to enable support for console applications in order to use the stdin, stdout and stderr streams at the console. To do this, add the following declaration to your application's project file:

```
CONFIG += console
```

See also close() and qmake Variable Reference.

bool QFile::open (int *fd*, OpenMode *mode*)

This is an overloaded function.

Opens the existing file descriptor *fd* in the given *mode*. Returns true if successful; otherwise returns 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 *fd* is not a regular file, e.g, it is 0 (`stdin`), 1 (`stdout`), or 2 (`stderr`), you may not be able to seek(). In those cases, size() returns `0`. See QIODevice::isSequential() for more information.

Warning: For Windows CE you may not be able to call seek(), setSize(), fileTime(). size() returns `0`.

Warning: Since this function opens the file without specifying the file name, you cannot use this QFile with a QFileInfo.

See also close().

Permissions QFile::permissions () const

Returns the complete OR-ed together combination of QFile::Permission for the file.

See also setPermissions() and setFileName().

Permissions QFile::permissions (const QString & *fileName*) `[static]`

This is an overloaded function.

Returns the complete OR-ed together combination of QFile::Permission for *fileName*.

qint64 QFile::pos () const `[virtual]`

Reimplemented from QIODevice::pos().

qint64 QFile::readData (char * *data*, qint64 *len*) `[virtual protected]`

Reimplemented from QIODevice::readData().

qint64 QFile::readLineData (char * *data*, qint64 *maxlen*) `[virtual protected]`

Reimplemented from QIODevice::readLineData().

bool QFile::remove ()

Removes the file specified by fileName(). Returns true if successful; otherwise returns false.

The file is closed before it is removed.

See also setFileName().

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

This is an overloaded function.

Removes the file specified by the *fileName* given.

Returns true if successful; otherwise returns false.

See also remove().

bool QFile::rename (const QString & *newName*)

Renames the file currently specified by fileName() to *newName*. Returns true if successful; otherwise returns false.

If a file with the name *newName* already exists, rename() returns false (i.e., QFile will not overwrite it).

The file is closed before it is renamed.

See also setFileName().

bool QFile::rename (const QString & *oldName*, const QString & *newName*) [static]

This is an overloaded function.

Renames the file *oldName* to *newName*. Returns true if successful; otherwise returns false.

If a file with the name *newName* already exists, rename() returns false (i.e., QFile will not overwrite it).

See also rename().

bool QFile::resize (qint64 *sz*)

Sets the file size (in bytes) *sz*. Returns true if the file if the resize succeeds; false otherwise. If *sz* is larger than the file currently is the new bytes will be set to 0, if *sz* is smaller the file is simply truncated.

See also size() and setFileName().

bool QFile::resize (const QString & *fileName*, qint64 *sz*) [static]

This is an overloaded function.

Sets *fileName* to size (in bytes) *sz*. Returns true if the file if the resize succeeds; false otherwise. If *sz* is larger than *fileName* currently is the new bytes will be set to 0, if *sz* is smaller the file is simply truncated.

See also resize().

bool QFile::seek (qint64 *off*) [virtual]

Reimplemented from QIODevice::seek().

void QFile::setDecodingFunction (DecoderFn *function*) [static]

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

Warning: This function is not reentrant.

See also setEncodingFunction() and decodeName().

void QFile::setEncodingFunction (EncoderFn *function*) [static]

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

Warning: This function is not reentrant.

See also encodeName() and setDecodingFunction().

void QFile::setFileName (const QString & *name*)

Sets the *name* of the file. The name can have no path, a relative path, or an absolute 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 the application's current directory path *at the time of the open()* call.

Example:

```
QFile file;
QDir::setCurrent("/tmp");
file.setFileName("readme.txt");
QDir::setCurrent("/home");
file.open(QIODevice::ReadOnly);      // opens "/home/readme.txt" under Unix
```

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

See also fileName(), QFileInfo, and QDir.

bool QFile::setPermissions (Permissions *permissions*)

Sets the permissions for the file to the *permissions* specified. Returns true if successful, or

false if the permissions cannot be modified.

See also permissions() and setFileName().

bool QFile::setPermissions (const QString & *fileName*, Permissions *permissions*) [static]

This is an overloaded function.

Sets the permissions for *fileName* file to *permissions*.

qint64 QFile::size () const [virtual]

Reimplemented from QIODevice::size().

Returns the size of the file.

For regular empty files on Unix (e.g. those in `/proc`), this function returns 0; the contents of such a file are generated on demand in response to you calling read().

QString QFile::symLinkTarget (const QString & *fileName*) [static]

Returns the absolute path of the file or directory referred to by the symlink (or shortcut on Windows) specified by *fileName*, or returns an empty string if the *fileName* does not correspond to a symbolic link.

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

This function was introduced in Qt 4.2.

QString QFile::symLinkTarget () const

This is an overloaded function.

Returns the absolute path of the file or directory a symlink (or shortcut on Windows) points to, or a an empty string if the object isn't a symbolic link.

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

This function was introduced in Qt 4.2.

See also fileName() and setFileName().

bool QFile::unmap (uchar * *address*)

Unmaps the memory *address*.

Returns true if the unmap succeeds; false otherwise.

This function was introduced in Qt 4.4.

See also map() and QAbstractFileEngine::supportsExtension().

void QFile::unsetError ()

Sets the file's error to QFile::NoError.

See also error().

qint64 QFile::writeData (const char * *data*, qint64 *len*) [virtual protected]

Reimplemented from QIODevice::writeData().