qDoc

©nxSoftware.com

General.

qDoc allows Purebasic applications to create proprietary documents which act like an entire file system in that a single document can contain any number of disparate 'files', files which can be accessed in a 'random' fashion. In this way, qDoc can be said to create *compound documents*.

Each such document itself occupies just a single disc (or memory) file (regardless of how many '*files*' it contains!)

qDoc is functionally almost identical to our comDoc library, but, whereas comDoc is for Windows applications only, qDoc is **fully cross-platform** (it has been tested on Windows and Mac OSX).

For example, imagine a word processing application. Such an application may wish to save not just the text entered by the user within it's documents, but it may also need to save images, charts, spreadsheets and so on within the same document and in such a way that each separate image or chart etc. can easily be retrieved and altered etc. This is exactly what qDoc facilitates and in a fashion almost identical to Purebasic's file library.

Open a qDocument, open one of it's files and then proceed to read and/or write bytes, longs, strings… etc. from/to the file.

Technically, a qDocument is an SQLite database. Individual files are stored as individual tables (with names matching the individual file name) and data is stored in blob format. Each file is buffered in order to speed up read/writes etc. and writes are automatically rolled back if errors are detected.

This means of course that developers can augment a qDocument by simply adding additional tables and the like as with any SQLite database (just take care with the table names to avoid any clashes!)

The whole point of qDoc though is to offer developers access to files/tables, not through SQL statements, but through our regular file commands. A qDocument would be used when database tables with their rows and columns structures are not appropriate, but when direct access to individual file bytes is!

qDoc arose out of my own need for this facility (in a cross-platform way) and that is enough justification for me! ☺

Package, licence and terms of use.

This package contains all the qDoc source files, various demo programs (as appropriate) and this user guide.

The software contained within this package is free to use in any project (commercial or otherwise) or as a learning tool. I do, however, assert my moral right to be identified as the creator of this software (except where acknowledgements are given) and thus ask that due acknowledgement is given within any product/creation in which my source code forms a part. Use the software for any purpose whatsoever.

The qDoc software is provided on an *as is* basis, with no warranty either given or implied, meaning that I am not liable for any damage caused by its use (or misuse!) nor by damage caused by other programs based on its source code.

Acknowledgements.

Thanks to the following:

kiffi for the code allowing for the swift backing up of SQLite databases.

To use qDoc within a Purebasic program.

Simply ensure that the following command resides within your Purebasic source code file (in an appropriate form) before any attempt is made to use any of the libraries facilities :

XincludeFile *"qDoc.pbi"*

That's it.

Creating a qDocument.

The qDoc library offers up an OOP (Object Oriented Programming) interface and as such we must deal with 2 classes; documents and files.

The *qDoc_DocumentObject* class offers methods for administering individual documents.

The *qDoc_FileObject* class offers methods for administering individual files within an 'open' document.

There are 2 public functions for creating a *qDoc_DocumentObject* object :

qDoc_CreateDocument(fileName$="", pageSize = #qDoc_MINPAGESIZE)

and *qDoc_OpenDocument(fileName$)*

With the *qDoc_CreateDocument(fileName$="", pageSize = #qDoc_MINPAGESIZE)* function, leave the filename$ parameter empty to create a memory based document. Leave the pageSize parameter empty for a default of 1024 bytes (which is the minimum permitted). A document's page size doubles as the size of the disc page and also as an individual file's cache size.

Memory based documents are advisable if intending to write quite a lot of data to the document as they perform much faster than disc based documents. **See the 'Tips and Tricks' section below.**

Each of the 2 functions listed above, if successful, return a fully instantiated *qDoc_DocumentObject* object through which we can, amongst other things, create, open or delete individual files etc.

Methods of the qDoc_DocumentObject class.

Here we list the methods of the *qDoc_DocumentObject* class.

\Close()

Destroys the underlying *qDoc_DocumentObject* object.
Failure to issue this method call could result in data being lost.

No return value.

\CreateFile.i(fileName$)

Replaces any existing file of the same name.
Fails if a file with the same name is already open.

Returns, if successful, a fully instantiated *qDoc_FileObject* object.

\DeleteFile.i(filename$)

Deletes the named file.
Fails if a file with the same name is already open.

Returns non-zero if successful.

\GetDatabaseID()

Returns the underlying SQLite database#. Useful if wishing to add some SQL tables etc.

No return value.

\GetFileSize.i(filename$)

Returns -1 if the file is not found within the underlying document. Otherwise it returns the size of the file in bytes.

\OpenFile.i(fileName$)

Opens an **existing** file for read/write operations. Fails if the file is already open for write operations since any individual file can be opened multiple times for reading, but only once for writing. Use the \ReadFile() method to open the file for read only.

Returns, if successful, a fully instantiated *qDoc_FileObject* object.

\ReadFile.i(fileName$)

Opens an **existing** file for read only operations.

Returns, if successful, a fully instantiated *qDoc_FileObject* object.

\SaveDocumentAs.i(filename$)

Save the document to disc. Any existing file is deleted.

Returns non-zero if successful.

\SaveToMemoryBasedDocument.i()

Creates a new memory based qDocument which houses an exact copy of the underlying qDocument. Useful to speed up write operations **(see the 'Tips and Tricks' section below)**.

Returns, if successful, a fully instantiated *qDoc_DocumentObject* object representing the newly created memory based document.

\BeginFileList.i()

Use in order to examine all the files within the underlying qDocument.

Returns the number of files found.

\GetFileListEntry.s(index)

Use after a call to \BeginFileList()

Returns either the appropriate filename in the enumeration or an empty string.

\FinishFileList()

Use when finished enumerating the underlying qDocument's files.

Methods of the qDoc_FileObject class.

Here we list the methods of the *qDoc_FileObject* class. Many are self explanatory and, for these, no detailed explanations are given.

Note that all write methods will set the file's error flag if required.

\Close()

Destroys the underlying *qDoc_FileObject* object.
Not strictly required because all open files are automatically closed when the parent document is closed.

No return value.

\Eof()

Returns #True or #False.

\FileSeek(newPosition)

If the new position is beyond the end of the file then the file pointer is placed at the end of the file. No additional empty bytes are written etc.

\FlushBuffers()

Mostly for internal use, but we include it for 'advanced' use in cases where developers may wish to work directly with the underlying SQLite database etc.

\Loc()

Returns the current file pointer in the range 0 to 'file size'. A value of 'file size; means that we are at the end of the file.

\Lof()

Returns the length of the file in bytes.

\ReadAsciiCharacter.a()

\ReadByte.b()

\ReadCharacter.c()

\ReadData.i(*memoryBuffer, lengthToRead)

Returns the number of bytes actually read.

\ReadDouble.d()

\ReadFloat.f()

\ReadInteger.i()

\ReadLong.l()

\ReadOLEString.s()

Reads a string previously written with the \WriteOLEString() method **(see the 'Tips and Tricks' section below)**.

\ReadQuad.q()

\ReadString.s(format=0)

Leave format empty for a format matching the Unicode compiler switch etc. Otherwise set to #PB_Ascii or #PB_Unicode or #PB_UTF8.

\ReadUnicodeCharacter.u()

\ReadWord.w()

\WriteAsciiCharacter(value.a)

\WriteByte(value.b)

\WriteCharacter(value.c)

\WriteData(*memoryBuffer, length)

\WriteDouble(value.d)

\WriteFloat(value.f)

\WriteInteger(value.i)

\WriteLong(value.l)

\WriteOLEString(text$)

Writes a string in Unicode format which is prepended with 4-bytes containing the length (in bytes) of the string. There is no null terminator **(see the 'Tips and Tricks' section below)**.

Use the \ReadOLEString() method to retrieve such strings.

\WriteQuad(value.q)

\WriteString(text$, format=0)

A null terminator is added.

Leave format empty for a format matching the Unicode compiler switch etc. Otherwise set to #PB_Ascii or #PB_Unicode or #PB_UTF8.

\WriteUnicodeCharacter(value.u)

\WriteWord(value.w)

\ClearErrorFlag()

A file's error flag is not cleared automatically allowing for a single check for errors after multiple writes etc. Saves checking after each individual write operation.

\GetErrorFlag.i()

A value of #True means that there has been an error.

\GetUserData.i()

Each file can have a single integer value associated with it. These values are not saved however and persist only as long as the file instance is valid.

\SetUserData.i(value)

Returns the previous value.

Q&A : tips and tricks.

Tip 1.

When in need of writing more than a *little* data to a qDocument then we advise that you create a memory-based document in the first instance.

Writing to such documents is **much** faster than writing to disc based ones.

When done use the \SaveDocumentAs() method to save to disc.

If opening a disc based qDocument for writing then after opening the document, you can use the \SaveToMemoryBasedDocument() method to create a memory based copy of the document. After this close the disc based document and perform all write operations on the memory based one.

When done, use the \SaveDocumentAs() method to save to disc, over-writing the original document.

Tip 2.

Consider using OLE strings if writing a lot of string data via the methods \ReadOLEString() and \WriteOLEString().

These strings can be written and retrieved far faster than *regular* strings.

Don't worry, they work across all supported platforms!