JThread manual (v1.2.1)

Jori Liesenborgs
jori.liesenborgs@gmail.com

June 20, 2006

1 Introduction

A lot of projects on which I'm working use threads. To be able to use the same code on both unix and MS-Windows platforms, I decided to write some simple wrapper classes for the existing thread functions on those platforms.

The JThread package is very simple: currently, it only contains three classes, namely `JThread`, `JMutex` and `JMutexAutoLock`. As their names might suggest, `JThread` represents a thread and `JMutex` a mutex. The thread class only contains very basic functions, for example to start or kill a thread.

2 Copyright & disclaimer

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3 Usage

Here follows a description of the `JThread`, `JMutex` and `JMutexAutoLock` classes. Note that functions with return type `int` always return a value of zero or more on success and a negative value in case something went wrong.

3.1 `JMutex`

The class definition of `JMutex` is shown below. Before you can use an instance of this type, you must first call the `Init` function. You can check if the mutex was already initialized by checking the return value of `IsInitialized`. After the initialization, the mutex can be locked and unlocked by calling the functions `Lock` and `Unlock` respectively.

```
class JMutex
{
public:
    JMutex();
    ~JMutex();
    int Init();
    int Lock();
    int Unlock();
    bool IsInitialized();
};
```

3.2 `JMutexAutoLock`

The class definition of `JMutexAutoLock` is shown below. It is meant to make it easier to implement thread-safe functions, without having to worry about when to unlock a mutex.

```
class JMutexAutoLock
{
public:
    JMutexAutoLock(JMutex &m);
    ~JMutexAutoLock();
};
```

The code below illustrates the way this class can be used:

```
void MyClass::MyFunction()
{
    JMutexAutoLock autoLock(m_myMutex);

    // Do operations protected by mutex 'm_myMutex' here
}
```

When the `autoLock` variable is created, it automatically locks the mutex `m_myMutex` specified in the constructor. The destructor of the `autoLock` variable makes sure the lock is released again.

3.3 JThread

To create your own thread, you have to derive a class from `JThread`, which is depicted below. In your derived class, you have to implement a member function `Thread`, which will be executed in the new thread. Your own `Thread` implementation should call `ThreadStarted` immediately.

To start your thread, you simply have to call the `Start` function. This function finishes when your own `Thread` function has called `ThreadStarted`. This way, when the `Start` function finishes, you can be really sure that your own `Thread` implementation is really running.

You can check if the thread is still running by calling `IsRunning`. If the thread has finished, you can check its return value by calling `GetReturnValue`. Finally, in case your thread gets stuck, you can end it by using the `Kill` function.

You should be careful with this `Kill` function: if you call it when the thread is working with a mutex (for example an internal mutex), this mutex can be left in a locked state, which in turn can cause another thread to block. You should only use the `Kill` function when you're absolutely sure that the thread is stuck in some loop and cannot be ended otherwise.

```
class JThread
{
public:
    JThread();
    virtual ~JThread();
    int Start();
    int Kill();
    virtual void *Thread() = 0;
    bool IsRunning();
```

```
    void *GetReturnValue();
protected:
    void ThreadStarted();
};
```